Swan

A Web Application Framework for Stage

Final Report

Fred van den Driessche

Imperial College

June 14, 2010

Supervisor: Susan Eisenbach

Second Marker: Tony Field

♠

Abstract

Last year's release of Google Wave enforced an increasingly strong trend of feature-filled, highly interactive Web applications with Web browsers acting as a thin client. The enhanced execution speed of JavaScript allows the creation of browser-based applications which rival traditional desktop applications in their responsiveness and usability. Such applications eschew rendering the entire page beyond the initial download instead relying on partial interface updates using data from background requests. However, the development of such applications requires a very different style compared to traditional Web applications and varying implementations of JavaScript across browsers makes it hard for developers to create them.

Established Web frameworks such as DJANGO tend to focus on the creation of traditional Web applications and often rely on third-party JavaScript libraries for the provision of highly interactive features. GOOGLE WEB TOOLKIT concentrates on the creation of an interactive user interface but leaves the implementation of data storage to the user, while LIFT provides an Actor-based solution to the problem of scaling to meet the server requirements for such interactivity.

This report introduces SWAN, a Web application framework based on STAGE, a scalable, distributed, Actor language with clean, concise and expressive syntax. SWAN uses these features to create a full MVC framework through which users can develop highly responsive, scalable Web applications in a single high-level language.

Acknowledgements

First and foremost I would like to thank Susan Eisenbach, without whom SWAN would simply be an ugly duckling. Secondly, I would like to thank John Ayres and Chris Zetter for creating such an elegant and stable language on which SWAN stands. Finally I would like to thank my parents, whose continued support has got me this far, and my friends for putting up with me throughout this project.

Contents

List of Figures

Listings

Chapter 1

Introduction

The World Wide Web, as described in the original proposal [13], was initially conceived as a means for sharing and navigating information through a set of inter-linked documents. At first the documents, hand-crafted by their authors, were inherently static. Soon after, with the introduction of forms, users were able to interact with servers and with server-side scripting languages allowing the dynamic generation of HTML documents, the first Web applications were created. Nowadays Web technologies and architectures have matured to a point where it is possible for Web applications to mimic the abilities of fully-fledged desktop applications.

The Web application as a composition of a number of dynamically-generated Web pages is by far the most common. Examples include web-fora, social networking and video-sharing sites. In contrast are applications such as Google Wave which, following the initial page-load, do not reload the entire page at all — all interaction with the server takes place through background requests, creating a continuity interaction similar to that of a desktop application. There is currently a strong trend, championed by Google, for websites that feature the same levels of interactivity and seamlessness of experience that is provided by desktop applications. In bold terms it is a progression towards the browser as the universal thin client for applications, extending the primary vision for a universal means for information dispersal.

Web frameworks evolved out of the common problems facing Web application developers, simplifying and accelerating the task of application creation. Typically Web applications can be represented abstractly in a similar way to each other and frameworks often include toolkits for managing these common aspects. There is a variety of persistent storage: a file system but more regularly a relational database, which is used for data storage between sessions of user interaction. There are sets of routines for manipulation and control of the data stored in the application — the "business logic" of the application. Finally, there are variety of representations for displaying the data to users. Web applications are also widely required to handle a larger number of concurrent users. The implications of the last point are what often cause the largest headaches for the designers of Web applications: the management of scalability, security, authenticity and user session state. Of course, despite these similarities, each application has its own unique aspects so Web frameworks must tread a fine line between supporting the user whilst not restricting them.

Instead of requiring knowledge of a multitude of different Web technologies, frameworks allows the application creator to work using fewer languages. Existing Web frameworks such as the Java-based Google Web Toolkit [32] and its Python port Pyjamas [4] provide developers with programming environment which can be used to create dynamic front-ends for their Web applications through cross-compilation to JavaScript. Puder's XML11 framework [49] takes a similar approach for the client-side through an intermediate XML representation whilst keeping the server in Java. The more frequent approach, used by the Python-based Django framework amongst others, is to push data through templates to create pages as they are requested. The templates are created using domain-specific languages [26] which extend HTML. The template language of the Scala-based Lift [22] Web framework uses additional HTML tags to hook into business logic.

The client-server architecture of the Web, based around the request/response style of HTTP, the protocol on which it is built, makes the creation of highly interactive Web applications challenging. LIFT makes use of SCALA's Actors to provide enhanced interactivity whilst maintaining scalability. Comparatively, GWT uses an RPC mechanism to perform background requests creating an almost seamless user-interaction similar to desktop applications.

This project introduces SWAN, a Web framework built on top of Ayres' Actor-based STAGE language. It is designed to take advantage of the qualities of STAGE and its syntax to create a scalable Web application framework emphasising clarity of language while providing a coherent environment for the production of highly-interactive websites. SWAN is intended to outfit a developer with the tools to create Web applications using a single language in a style which will be familiar to those that have created regular desktop applications. In addition SWAN makes it effortless to define how application data can be exposed using a REST interface, for consumption of an application's data in many different settings.

Proposed by Hewitt, Bishop and Steiger [36], the Actor model creates a safe environment for concurrency through message-passing and the absence of any shared state. These properties are advantageous for Web applications since at any one time there can be numerous simultaneous requests for data. A request and its response along with any intermediate internal communication can all be modelled as messages. STAGE [9, 10], created by John Ayres, is an Actor language with its foundations in PYTHON. It promotes clean communication syntax and mobility features. STAGE was extended by Christopher Zetter to create STAGE# [58], providing a more scalable Actor distribution system. The STAGE language and the benefits of its use in a Web environment are discussed in more detail in Chapter 2, which also presents common features of Web applications and their architecture.

Chapter 3 explores the approaches of other Web frameworks for simplifying application development and additional features that they include. I investigate five state of the art frameworks: DJANGO, LIFT and GWT, which are frequently used to create enterprise Web applications including `washingtonpost.com`, `foursquare.com` and the previously mentioned Google Wave, along with the XML11 and BULLET. Many aspects of these different frameworks and their varying techniques for solving the same problems shaped SWAN throughout its development, though SWAN's fundamental mantra always kept the potential user and streamlining their workflow as the central consideration.

In Chapter 4 I present the SWAN framework itself, the server on which SWAN applications run and its components for creating the server-side and client-side elements of an application. I aim to confer to a prospective user a sense of how SWAN and its various features aids application development. I also discuss a number of additions to the core STAGE language which were added to support SWAN but will also be useful for the general STAGE programmer.

To evaluate SWAN, in Chapter 5 I present a sample application, a single-user blogging platform, as well as the server-side implementation of a multi-user chat program. I also present some benchmarks of the Web server on which SWAN applications are based, relative to the Apache HTTP Server and Apache Tomcat.

In Chapter 6, I outline conclusions drawn from the development of SWAN and suggest a number of improvements and extensions for SWAN.

Chapter 2

Web Application Architecture

The advances in Web technologies and the increase in power of client machines has allowed Web developers to create increasingly feature-filled websites which feature interactivity on a par with traditional desktop applications. However the more open environment of the Web and the differing expectations of its users puts different pressures on Web application developers. For desktop applications the focus is solely on time-to-market while for the Web the emphasis is on reliability, usability and security with consideration for availability and scalability as well [45]. These constraints have heavily influenced the architectural style of Web applications. This chapter discusses the architecture of Web applications, how they can meet scalability demands using REST interfaces and create responsive, desktop-like user interfaces through AJAX and COMET interactions.

2.1 Model-View-Controller

Commonly Web applications can be described as using a *tiered* architecture [47]. The standard desktop application is a 1-tier solution while the early Web usually featured 2-tier solutions — the client for display and server for processing and storage. The 3-tier solution, that separates the application's business logic from storage evolved from increasing pressures to resolve problems of scalability. The n-tier solution is a generalisation of 3-tier, promoting the internal separation of the application logic for increased security and reliability, and facilitating the use of third-party Web services.

Although the desktop application is described as a 1-tier solution its design is still relevant to the general n-tier architecture of web applications. The Model-View-Controller (MVC) design pattern is the most popular paradigm for creating desktop applications with graphical user interfaces. As formalised by Burbeck in [16], specifically for Smalltalk, a core triad of elements are defined:

Model — Contains purely application data. In a photo-editing application this could be a two dimensional array of pixel values. In a simple spreadsheet application it is the array of cell values.

View — Represents the Model to the application user in some form. There can be many Views for a single model. For the photo-editing application a View could be simply the photo itself, a crop of the photo, or the photo's histogram. In the spreadsheet, it could the standard cell representation or a graphical alternative: a bar, pie or scatter chart.

Controller — Interprets keyboard and mouse commands from the user and manipulates the Model accordingly. In the photo-editing application it might respond to a user command to make the photo grayscale and modify the pixel values of the model accordingly. In the spreadsheet, modifying values in the cell view updates the values in the model.

It is clear from the above description that the Model can be the basis for a number of Views and so it follows that the Model should not be dependent on a View in anyway. However, the View must stay current with the Model's state such that when the state is modified the View is updated. The solution is to use the Observer pattern [29], or if a fine-grained approach is necessary a light-weight

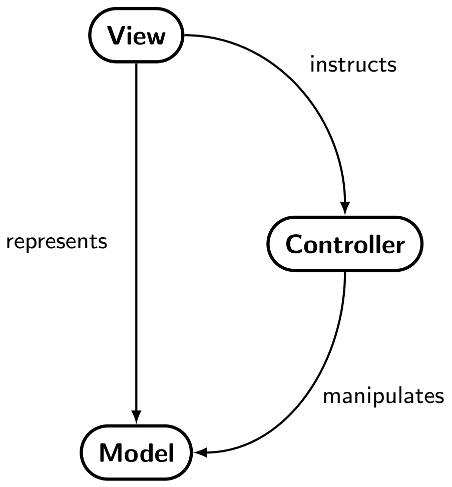

Figure 2.1: **The MVC Dependency Relationship**: *The Model encapsulates the application data. The Controller contains routines for modifying the Model. The View represents the Model to the user and captures user commands and passes them on to the Controller.*

Publish/Subscribe mechanism. These patterns also enable elegant handling of temporal changes to the Model.

The View and Controller are more dependent on each other, so much so that according to Burbeck's definition they are tightly coupled. Such a coupling is understandable if it is strictly the Controller that responds to user commands as the View must be informed to make any necessary changes. However, since commands to the Controller come through the View itself, any operations that change the appearance of a View directly can be short-circuited and captured by the View itself. A decoupled View and Controller, supported by a separate Model, as described in Figure 2.1, relates directly to the 3-tier model for web applications. The separation of the triad over the client-server divide is simple: The Model is stored on a database server and the View rendered by the browser for the user. The Controller makes up the application layer or layers in an n-tier application.

If the Model is *passive*, i.e. it can only be modified through user interaction, then in a Web application any request to the server (the Controller) modifies the Model if necessary and responds with an updated View. This flow of control mirrors that of a desktop application but, while it is satisfactory on the desktop, it is relatively slow and unresponsive on the Web. When a request is made to the server user control is suspended until the response is received. Even if this interruption is only a fraction of a second it disrupts the user's workflow. The following section describes a technique to provide improved responsiveness for Web applications.

2.2 Ajax

AJAX[1] is the popular name for the arrangement of technologies, Asynchronous JavaScript and XML amongst others [30], that obviate the need for a complete page-load for the client to retrieve data from the server. Its use provides the higher levels of interactivity and responsiveness on a par with regular desktop applications.

The common functionality of AJAX centres round the `XMLHttpRequest` object that is common to the JavaScript implementation of most browsers. Such a request object can be used to issue background HTTP requests to the server, without the users explicit instruction. The reply from the server contains

[1]Coined by Jesse James Garrett, its preferred use as a shorthand name, Ajax, rather than an acronym, which it may appear as at first. It bears no relation to any Greek hero, Dutch football team or cleaning products.

Listing 1 An AJAX Request

```
    request = new XMLHttpRequest(); //the request object

    //call-back method
    request.onreadystatechange = function(){

        //check the request finished successfully.
        if(request.readystate == 4 && request.status == 200){

            //display "Howdy" from <greeting>Howdy</greeting>
            alert(request.responseXML.documentElement.nodeValue)
        }
    }
    request.open("GET", "greeting", true); //asynchronous GET to "greeting"
    request.send(null);                    //send the request
```

the relevant data structured in XML. The XML is then interpreted on the client side using a callback function which can perform the necessary actions to update the user interface. A simple Ajax request is shown in Listing 1.

As with Representational State Transfer (Section 2.4 on page 7), Ajax is more of a style rather than a strict specification. Both synchronous and asynchronous requests can be made through `XMLHtppRequest` objects but asynchronous is preferred since synchronous requests block the execution of client-side code until the reply is received, causing the user interface to freeze. There is also no need to use XML for structuring the server reply. A good alternative is JavaScript Object Notation (JSON[2]) which is evaluated on the client-side. JSON provides the same structured information and can also include additional functionality as evaluation results in standard JavaScript objects that can contain functions. It avoids the need for the somewhat clunky access of data from an XML structure but relies on the use of the JavaScript `eval` function which is a potential security risk. A sample comparison of XML and JSON response processing is shown in Listing 2 and Listing 3 on the next page.

Listing 2 XML Response Processing

```
request.onreadystatechange = function(){
  if (request.readyState == 4 && request.status == 200) {
    table = document.getElementById('table');
    people = request.responseXML.documentElement.getElementsByTagName('person');
    val = ""
    for(i = 0; i < people.length; i++){
      val += "<tr><td>";
      val += people.item(i).getElementsByTagName('name')[0].textContent;
      val += "</td><td>";
      val +=people.item(i).getElementsByTagName('salary')[0].textContent;
      val += "</td></tr>";
    }
    table.innerHTML = val;
  }
}
```

Ajax can be used in a large range of ways. The photo-sharing website Flickr uses it in subtle ways: to allow users to title their photos and add descriptions while still using discrete page-loads for most interaction. At the other end of the spectrum are sites like Google Mail which relies totally on background requests. Instead of using discrete pages they download a JavaScript engine when the site is first accessed

[2]Pronounced as the name Jason, again no relation to any Greek hero.

Listing 3 JSON Response Processing

```
request.onreadystatechange = function(){
    if(request.readyState == 4 && request.status == 200){
        table = document.getElementById('table');
        people = eval("(" + xmlhttp.responseText + ")");
        val = "";
        for(x in people){
            person = people[x];
            val += "<tr><td>"+person.name+"</td><td>"+person.salary+"</td></tr>";
        }
        table.innerHTML = val;
    }
}
```

which is then used to construct the user interface, make requests and interpret the replies.

Developing a website that uses a JavaScript engine is still similar to the development of a desktop GUI application. The use of Ajax and how it relates to the MVC design pattern and *n*-tiered architecture is shown in Figure 2.2 on the facing page. The Controller is either split and partially implemented in JavaScript and partly through server-side routines or delivered in its entirety to the client as JavaScript. If the entire Controller is in JavaScript then computation is offloaded to the client, reducing load on the server and consequently improving scalability. The engine is responsible for issuing background requests to the server as needed and marshalling the response data before calling the relevant functions to modify the user interface.

Through Ajax it is possible to make background HTTP requests to update the data on the user's screen, a technique that is valid if the Model only changes through user interaction. Models are not necessarily passive — they can change over time or, as the Web is inherently multi-user, through interaction with a second user. On the desktop the Observable Model marks itself as modified and informs it's Observers. There is no parallel on the Web because only the client can initiate communication due to the client-server request/response cycle. The next section describes a technique to overcome this limitation and create the illusion of server-initiated communication.

2.3 Comet

The Web was designed using the client-server architectural style and this affects the scope of communication since servers cannot dispatch messages to clients without a prior request. This affects the responsiveness of applications since the server must rely on user interaction to refresh data on the client-side.

Comet, sometimes referred to as 'reverse Ajax', is the term used to describe techniques which allow servers to 'push' data out to the client [52]. This is an extremely useful feature when there are multiple users viewing and manipulating the same model. It allows the server to inform the user of changes as they happen, rather than waiting for the user to make a request to the server, achieving even more interactivity than plain Ajax.

There are a number of methods for implementing Comet or Comet-like effects:

Synchronous — Similar to standard Ajax but when a request is made to the server all changes to the model are tagged on to the end of the response [49]. This can have an odd effect on user experience as the updates may not be expected and the data may not be fresh.

Polling — Regular background requests are made to the server. If there is a change to the Model then it is transmitted, otherwise the response is empty. This is similar to an email client regularly

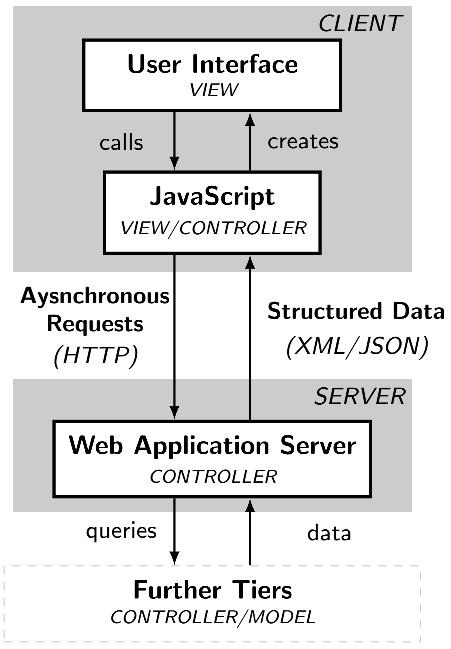

Figure 2.2: AJAX: *An overview of Ajax architecture and how it relates to the MVC design pattern and n-tier architecture. Adapted from* [30].

checking the server for mail. It is a good trade-off between scalability and latency.

Long-Polling — This is Comet as defined in [52]. An asynchronous request is opened to the server by the client. The server keeps the request alive until it has a response to transmit. When the client receives the reply it immediately opens another request to the server. This minimises the latency of data for the user but increases the load on the server as a large number of requests must be maintained simultaneously. In a regular web server, where a thread is blocked per request, this can cause major performance problems.

Forever Frame — Through using a hidden, infinite-length `iframe` HTML element which is sent `script` elements by the server. Due to the incremental nature of HTML rendering by web browsers each `script` element is executed as it is received and can be used to open a background request which eventually will result in a further `script` element being sent. The major downside is the difficulty in error-handling. [40]

Creating interactive, low-latency, *multi-user* web applications is possible through Comet though usually the impact on the scalability due to the long requests prevents such techniques on a large scale. The next section describes how constraints on the design of the server can amortise scalability issues.

2.4 Representational State Transfer

The previous sections described the general structure of a web application, how it can be related to the MVC triad and techniques augment Web application interactivity. This section focusses on the design and implementation of the Model component of MVC and how it supports the other application components.

The information stored as the persistent data of a web application should be accessible in a manner that is independent of the method of accessing it. This allows for easy expansion to support different client types that consume the data and also the evolution of data-storage structures. Essentially, having an open, uniform data platform is beneficial for the extensibility of the platform as a whole as well as the longevity of data that is stored in it. Additionally it can be made available, and possibly monetised, as data source for third-parties.

Representational State Transfer (REST) is a software architecture style developed since 1994 and defined by Roy Fielding in [25]. The Web itself can be regarded as a large implementation of a REST system, because the formalisation of REST is based on the design decisions taken during the Web's inception. REST refers to how a Web application should behave, Fielding explains:

> *A network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user.*

Although a full website is described above, REST can also be used more abstractly to provide a Web data service or data application programming interface (API). Such services are based around *resources*, they make use of the semantics of HTTP request methods and Uniform Resources Identifiers (URIs) to simplify its interface and means of access. The following sections define the REST style more formally and how it can it used to create a data service that can be used as the Model component of an MVC triad.

2.4.1 REST Contraints

REST is an architectural style rather than a well-defined specification or protocol. As defined in [25] there are six constraints on the implementation of a system:

Client-Server The separation of concerns through adopting the client-server style improves user interface portability and server scalability. For example, the main functionality of the service provided by a website is contained within the server; meaning the browser can be used to visit multiple websites. The separation can be blurred through Code on Demand.

Stateless Each request by the client to the server must be self-contained such that the server need not store session context to understand a request from a client. This improves the reliability and scalability of the server but increases network usage due to the need to transmit the session state for each request. This has a huge effect on the simplicity of a web service and its resource usage since each request can be serviced individually and independently of any other request. This permits processing of requests from the same client simultaneously.

Cacheable The response data for a request is marked as cacheable or non-cacheable in some manner. Clients are able to use data that is marked as cacheable without needing to re-request it from the server in the case of a second similar request. This mediates the effects of the previous constraint by reducing the number of requests where possible.

Uniform Interface Decoupling the clients and servers through a uniform interface simplifies the architecture and allows the implementation of the service to evolve independently of the means of accessing the service. This is typically achieved using URIs. There are a further set of constraints that describe a REST interface, described on the next page.

Layered System Layers can be used in the system to provide mediating components such as adapters and load-balancers. Each layer in the system is strictly dependent only on the layer directly below it. This aids scalability but impacts network performance as a request must go through a number of connections to be serviced. The layering refers to proxies and also the tiered structure described on page 3.

Code on Demand Client functionality can be extended by code such as JavaScript or Java applets downloaded from the server as required. This is the foundation for the rise of the highly interactive Web application. Without the ability to specialise functionality of the Web browser for a specific website the client would be *too* universal. Code on Demand bridges the gap between the pre-Web specific client application and the unvisersality of the browser.

Method	**Set**	**Entity**
`GET`	Retrieve the resource list	Retrieve specific entity details
`PUT`	Create or replace the set	Update the entity
`POST`	Extend the set with a new entity	Replace the entity or turn it into a set
`DELETE`	Remove the set	Remove the entity
`HEAD`	Get just the response headers	
`OPTIONS`	Get the available HTTP methods for a specified resource	
`TRACE`	Instructs the final server to echo the request, allowing the client to inspect what the server receives.	
`CONNECT`	Reserved for use with a proxy that can become a tunnel	

Table 2.1: **HTTP Request Methods**: *The first four methods are most relevant to RESTful web services, allowing CRUD operations, while* `HEAD` *and* `OPTIONS` *are convenient for testing server functionality.*

REST Interface Constraints

Fielding [25] defines further constraints on the interface of a REST environment. These emphasise the importance of conceptual resources and their identification over methods for their manipulation.

Identification of Resources URIs are used to name a specific resource for a request. Resources are not coupled in any way to their representation, there can be many representations for a single resource — the required representation is specified in the request. Thus a specific element in a database table can be requested using an identifier but its representation could be XML, JSON, CSV and more.

Manipulation of Resources through Representations A resource can only be modified by sending a representation of that resource to the server. Thus a representation can be both the *current* state of a resource or a *desired* state.

Self-descriptive messages Messages contain metadata, in headers or otherwise, that describe the content of the message. The metadata can describe both the representation being transmitted and the state of the represented resource. For example, when updating a resource, the format of the representation used should be transmitted so the server knows how to process the message.

Hypermedia as the Engine of Application State The state of an application can be changed by accessing resources, through URIs, identified in representations. As described in the quote on the preceding page, a website, or web application, can be represented as a state machine and following links through a website and submitting forms, changes the state of the application. For instance, the representation of a list of resources would contain identifiers for each of the individual resources in the list.

2.4.2 RESTful Web Services

REST Web services create a data platform that acts an ideal Model for a Web application. Rather than defining operations that can be performed on the application data it exposes the data itself, and therefore its state, as resources. Emphasis is placed on *nouns* for identifying resources rather than a wide range of *verbs* for accessing and manipulating them [21].

A RESTful Web service makes use solely of the eight HTTP methods for data control, these are listed in Table 2.1 on this page [23] . These methods can have different semantics depending on whether the resource is a set of entities or a single entity [51]. `PUT`, `GET`, `POST` and `DELETE` together allow standard

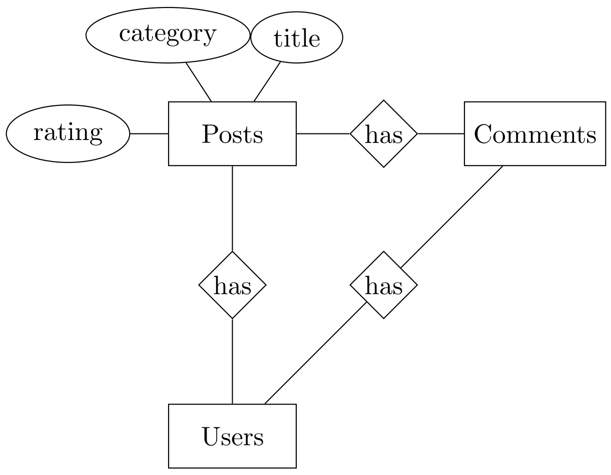

Figure 2.3: **Entity Relationships for a simple, multi-user blog**

CRUD[3] operations on application data. These operations, and examples of their use, are described in more detail in Section 2.4.4.

The basic HTTP operations provide a simple, adaptable, data API. This is extremely useful as it facilitates the creation of further means of consumption and manipulation of data in specific clients on a variety of different platforms. For example, an iPhone application could make straightforward use of a REST data service that is also the foundation for a web application. This is a strategy that the Twitter information network uses to great effect. It also makes the data accessible for machine consumption if suitable representations of the data are available.

Due to its basis in resources, REST is suitable for resource-oriented services but less well-aligned with activity-oriented services such as online checkout or banking systems [54]. However, a RESTful checkout service is still possible: users *get* a product listing resource, from which they can *get* specific item details which can be *put* in a basket resource. When the user wants to buy their selected items they *get* a representation of their basket and *post* it to the checkout resource along with a representation of their payment card. One could argue that this is a *stateful* system because state is stored in the basket resource but the **Stateless** REST constraint only declares that each request is self-contained and, in this case, they are. The slight issue with this system is that the items in the basket need to be deleted at some point. Still, it remains that it is not necessarily the most optimal use of a REST service mainly because a checkout service generally works with *verbs* rather than *nouns*.

The manner in which resources are addressed and accessed in a system is highly important and has a large effect on resource usability. The next section describes how URIs can be used to provide a structured way of specifying access to resources.

2.4.3 URI Design

A REST system is resource-oriented so it is important to have a suitable scheme for identification of resources. Hierarchical identification through URIs allows the formation of logical identifiers for resources. The structure of identifiers need have no relevance to the structure of the data that is accessed. If we consider the simple database schema represented in the Figure 2.3 for a mulit-user blog. **User**s write **Post**s, **Post**s have **Comment**s, **User**s can make **Comment**s but so can un-registered visitors. A URI scheme for the site could be as follows:

- `/users`: all the blog users.

[3]Create, Retrieve, Update, Delete.

- `/posts`: all the blog posts.
- `/comments`: all the comments on all the posts on the blog.
- `/post/123/`: a specific post.
- `/post/123/comments`: all the comments on a specific post.
- `/post/123/comment/01`: a specific comment on a specific post.
- `/post/123/comment/01/user`: the user that wrote a specific comment on a post, if they exist.
- `/post/123/user`: the user that wrote post 123.
- `/user/42/posts`: all the posts by a specific user.
- `/user/42/comments`: all the comments by a specific user.

Additionally, when taking attributes into account, we can add more 'meta'-URIs:

- `/categories`: all the post categories.
- `/category/Fruit`: all the posts in the `Fruit` category.
- `/search/String`: all posts related to `String`s.
- `/posts/latest`: the most recent post. Useful for the front page, perhaps.
- `/posts/toprated/5`: the five top-rated posts.

It's evident that the list of possible URIs is endless and there can be any number of URIs pointing to the same resource. What is also important is that the structure created through the URI schema is entirely conceptual — it's how the designer wants the resources to be seen, not how they are implemented. Theoretically the design of the database holding the data could be totally re-arranged, or even the whole dataset converted into a filesystem structure, without having to change the structure of the means of accessing it. It is important then to design URIs to stand test of time so it is preferable to avoid ephemeral data in URIs including data generation mechanisms and formats and even classification solely by topic [12].

If the URI structure *needs* to be changed for some reason then using HTTP response codes such as `301 Moved Permanently` can be used to maintain the integrity of the old URI design. This method retains the identifiers of the old resources but maps them on to the new identifiers.

Broken links are extremely damaging for user experience of a website so careful design is needed. However with the ability to delete resources (see `DELETE` on page 13) it's definitely possible to **intentionally** break links. For instance, a user may decide to remove a post. In this case it is important to inform the user that the resource does not exist any longer.

This section looked at how resources can be located in a uniform, structured manner. The following section enumerates the four most important methods for manipulating those resources.

2.4.4 HTTP Methods for REST

This section explains the four common HTTP request methods, `GET`, `PUT`, `POST` and `DELETE`, and gives examples of their use. As the basis for the example a hypothetical fridge will be used, it is located at `http://www.fridge.com/`. For brevity request and response headers are included only where necessary.

`GET`

The `GET` method retrieves a representation of a resource. If the resource is a set it will typically fetch a list of resources in the set, while if it is a specific resource it will return details pertaining to that resource. The `GET` method is defined as *safe* [23] — it should have no side-effect on the state of the application. Listing 4 shows sample request to find the contents of the fridge. The XML response shown has a very basic schema, it is just one of many possible ways of structuring the response. Often XML responses can get highly complex through the use of namespaces. Importantly, locators for elements in the collection are given so that the application can move state by requesting a listed resource.

Listing 4 A Set `GET` Request

```
Request:
 GET /contents
 Host: www.fridge.com

Response: 200 OK

    <contents>
        <item id="01" url="http://www.fridge.com/contents/01">Cheese</item>
        <item id="02" url="http://www.fridge.com/contents/02">Milk</item>
        <item id="03" url="http://www.fridge.com/contents/03">Butter</item>
        <item id="04" url="http://www.fridge.com/contents/04">Eggs</item>
    </contents>
```

If the resource specified is a single entity then the response might be similar to that shown in Listing 5. Again the response is shown in a basic XML schema. It is worth noting that the desired format of the response can be specified in the HTTP request headers, specifically the `Accept` header. If the content type requested is not available then the server responds with `415 Unsupported Media Type` [23].

Listing 5 An Entity `GET` Request

```
Request:
 GET /contents/02
 Host: www.fridge.com

Response: 200 OK

    <item id="02">
        <type>semi-skimmed</type>
        <size unit="ml">568</size>
        <remaining>200</remaining>
        <useby>01/02/03</useby>
    </item>
```

`PUT`

The `PUT` method is used for creating a resource at the specified location or updating it if it already exists. As shown in Listing 6, the indented version of the resource is contained in the message body in some representation, as described in the **Manipulation of Resources Constraint** on page 9. In this case the representation is in JSON, as specified by the header `Content-Type`. The response from the server depends on whether the resource is being created, in which case it should be **201** `Created`, otherwise it should be `200 OK` or `204 No Content`.

Listing 6 `PUT`ting an Entity

```
Request:
 PUT /contents/02
 Host: www.fridge.com
 Content-Type: application/json

 {
    "id":02,
    "type":"semi-skimmed",
    "size":{
        "unit":"ml"
        "value":568
    },
    "remaining":100,
    "useby":"01/02/03"
 }

Response: 200 OK
```

PUT is inherently not *safe* but it is *idempotent*, submitting the same request n times, where $n > 0$, should have the same effect as submitting it once.

POST

The effect of POST methods is the least well specified HTTP method. Its intended use is for modifying the state of an existing resource in some way [56]. It often used to extend a set with an additional member, such as in Listing 7 where another item has been put in the fridge. Arguably POST is similar to PUT, especially when you consider PUTting an item in a fridge, but with PUT the entire contents of the list would need to be included. POST is neither *safe* nor *idempotent* — the legality and sanity of submitting the same POST request multiple times is determined by the server. If the request is successful the server can optionally return a representation of a resource in its content.

Listing 7 POSTing an Entity

```
Request:
 POST /contents/
 Host: www.fridge.com
 Content-Type: application/xml

 <item name="Orange Juice">
    <size unit="ml">1000</size>
    <useby>04/05/06</useby>
 </item>

Response: 200 OK

<contents>
    <item id="01" url="http://www.fridge.com/contents/01">Cheese</item>
    <item id="02" url="http://www.fridge.com/contents/02">Milk</item>
    <item id="03" url="http://www.fridge.com/contents/03">Butter</item>
    <item id="04" url="http://www.fridge.com/contents/04">Eggs</item>
    <item id="05" url="http://www.fridge.com/contents/04">Orange Juice</item>
</contents>
```

DELETE

DELETE should remove the resource at the specified location, mark it for deletion, or move it to an inaccessible place. The response from the server can optionally include a representation of an entity representing the new state, usually the container that the resource was in, as in Listing 8. The DELETE method is also idempotent.

Listing 8 `DELETE` an Entity

```
Request:
 DELETE /contents/05
 Host: www.fridge.com
Response: 200 OK

<contents>
    <item id="01" url="http://www.fridge.com/contents/01">Cheese</item>
    <item id="02" url="http://www.fridge.com/contents/02">Milk</item>
    <item id="03" url="http://www.fridge.com/contents/03">Butter</item>
    <item id="04" url="http://www.fridge.com/contents/04">Eggs</item>
</contents>
```

2.4.5 Conclusion

There are two main alternatives to REST that can be used for a Web application. The first is to define a set of access and manipulation routines on the server-side which can be used through parameterised requests. This is essentially exposing the Controller component of the application rather than the Model. On one hand this means that the application need no longer be stateless but it does mean that the uses of the data are more restricted. One of the biggest advantages of a REST Web service is that the means of its resource consumption is not restricted in any form.

The second alternative is to use a Web service stack based on the Simple Object Access Protocol (SOAP). Using SOAP, an XML-format language, the consumer and producer correspond using envelopes, containing a header and body for metadata and content respectively. SOAP services are often seen as more complex and costly [46] and the messaging format overly verbose and unwieldy — making SOAP requests through the JavaScript `XMLHttpRequest` would indeed be time-consuming and computationally expensive. There has been a lot of debate over REST and SOAP Web Services and their pros and cons [33, 54] however it is the simplicity of REST that makes it more suitable as a data service for a Web application.

This section described how RESTful Web services can be used to provide a powerful data platform for a Web application that focusses on the data itself rather than ways of modifying and accessing it. The next section discusses the manners in which such a data service can be used by the View and Controller aspects to create the user interface for a Web application.

2.5 Templates and Cross-Compilation

Section 2.4 showed that it was possible to create an abstract Model component for a Web application using Representational State Transfer (REST) Web service. This section deals with the creation of the View and Controller components of the MVC triad with regards to Web applications and the different techniques for their creation.

The are two main approaches to creating the View and Controller aspects of Web applications. Templating systems mix regular HTML mark-up with specialised elements to create a domain-specific mark-up language for creating layouts. The specialised elements can denote variables, control structures and even perform database lookup [27]. The control of the application is still generally maintained through page transitions: HTTP requests sent to the server change the application state, the response is another page created using the template system. Asynchronous background requests through Ajax are available through templating systems though often via a third-party JavaScript library such as Prototype, MooTools or jQuery.

The second approach is JavaScript cross-compilation. This style allows developers to write in a high-level, typically object-oriented, language which is used to generate the Web application front-end using JavaScript as an assembly language. Due to varying JavaScript implementations across browsers the compilation process involves creating browser-specific compilations with the suitable script being served when a user accesses the site. The JavaScript downloaded to run the application contains:

- Bootstrap code to set browser-specific properties for the environment.
- Required API constructs from the high-level language implemented in JavaScript.
- The application itself in JavaScript form [34].

There are strengths and weaknesses for both approaches. The biggest drawback for templating systems is the use of a second language. Firstly the language must be learnt but more importantly it means the implementation of the application is split over two languages. Arguably the split is also a strength: it means the user interface is separated from the other application components explicitly. The cross-compilation technique allows web application developers to write their applications in a single, high-level programming language. The benefit of this is extended when IDE support is taken into account. On the downside the compilation process means that testing and debugging the application challenging without a specialised debug environment.

Both methods can be used on top of a RESTful Web service. Imagine REST resources are available through the `/resources` path on a Web server, further paths can be created for the actual Web application which contain the scripts of state control and the templates for display to the user. In this case careful control is needed such that the Web service is not confused, or merged, with the Web application. For cross-compilation the resources are available for requests through `Ajax` routines in the compiled JavaScript engine.

The processes for both techniques vary from framework to framework. The more specific details for a set of frameworks are described in Frameworks . If Swan is to use a JavaScript cross-compilation technique then it needs support from Stage, the language in which it is based, for clear, concise and expressive definition of the user interface, as discussed in the next section.

2.6 Stage

This section discusses the Stage language and the features that makes it suitable as a language for the creation of Web applications.

Stage [10, 9][4] is derived from Python and based on Hewitt's Actor model [36]. It focusses on concurrency through message passing, process mobility and distribution. These attributes make Stage well qualified for use on the server-side of a Web application where scalability is derived from the capacity to process client requests simultaneously. Stage also has clear, concise syntax which will be familiar to those that have written Python and make it suitable for use in creating user interfaces.

2.6.1 The Actor Model

The Actor model provides a general framework for concurrent computation. In simple terms concurrency allows a system or application to 'two or more things at the same time' through multiple processes executing simultaneously or multi-threading. Concurrent programming is regarded as extremely difficult to write and even harder to debug. There are a number of common pitfalls, as follows [55]:

Race Conditions mean that the result of execution is dependent on the timing and interleaving of events that occur during non-deterministic execution.

[4]Stage was originally created by John Ayres and extended by Christopher Zetter as Stage#. For brevity I refer simply to Stage.

Deadlock causes the system to freeze. It can occur when there are processes competing for exclusive access to shared resources whilst already having an unrelinquishing hold on some of the shared resources [19].

Livelock is similar to deadlock but where concurrent processes are still executing but making no genuine progress.

Starvation where a process is continually denied access to a shared resource.

The Actor model allows for concurrent execution in a environment where it is easier to avoid these problems. Actors can be seen as a constrained combination of objects and threads. Much like humans, Actors communicate solely through message passing and there is no shared state. Actors have a message queue where they receive messages from other Actors. An Actor's *behaviour* defines how it responds to messages from other Actors. On receipt of message an Actor can [8]:

- Send messages to other actors.
- Change its behaviour.
- Create new Actors.

In Stage, messages are received in order that they are sent but messages from two Actors are interleaved non-deterministically. Actors are named and can communicate only with other Actors they 'know'. Actors can learn of others by receiving a message containing the their name or by creating them.

2.6.2 Syntax

The syntax of an Actor in Stage is similar to that of a class definition in Python, an example Actor and usage is given in Listing 9. The Actors behaviour or *script* is given on lines 1–16. Lines 5–16 define the Actors external interface, messages to which it will respond, and a `Students` actions on receiving such messages. The `birth` message is the equivalent of an object constructor — it is guaranteed to be the called first. Sample usage is shown on lines 18–22.

Listing 10 demonstrates Actor communication through the canonical infinite Ping/Pong example. As you can see, sending a message to another Actor looks identical to calling a method on a object in regular Python. Stage also implements lazy synchronisation inspired by Futures: a message such as `v = a.value()` is only waited on when `v` is actually referenced, `print v` for instance. Other features include polling of variable through the `ready` keyword, binding callback methods through the `callback` primitive and through passing methods. These features make for a very powerful concurrent language which maintain the clarity and expressiveness of Python.

2.6.3 Mobility & Distribution

Actors in Stage execute in an environment known as a *Theatre*. Theatres can be run on multiple hosts across a network and Actors are free to migrate between them and communicate with Actors in other Theatres. Migration is performed through the `migrate_to` primitive which can be exposed to the external interface to allow migration on command from another Actor. Sending messages to an Actor in another Theatre, a *remote* Actor, is orders of magnitude slower than sending to a *local* Actor. This discrepancy motivates Actor *Friends*: if two Actors are friends they will migrate together where possible.

Due to distribution the Theatre network must handle the task of making sure the messages reach the intended Actor. Stage is backed by a highly-scalable distributed hash table (DHT) spread across *Managers* attached to each Theatre. This enables efficient location of remote Actors in a manner that minimises resource usage.

The ability for Actors to migrate across Theatres allows for load balancing. Through monitoring the load levels on the Theatres throughout the network, special load balancing Actors can instruct other conforming Actors on heavily-loaded Theatres to migrate to more lightly-loaded Theatres. Actors can also be addressed by type as well as by name through DHT using the following primitives:

Listing 9 A sample Stage Actor

```
class Student(MobileActor):
    def birth(self):
        self.energy = 10

    def write_report(self, hours):
        self.energy -= hours

    def drink_coffee(self, cup):
        self.energy += 1
        cup.empty()

    def is_tired():
        return energy < 0

    def sleep():
        self.energy = 10

s = Student()
s.write_report(5)
s.drink_coffee(acup)
s.write_report(5)
s.sleep if s.is_tired() else s.write_report(5)
```

Listing 10 Infinite Ping/Pong

```
class Ponger(LocalActor):

    def pong(self, pinger):
        print ‘‘pong’’
        pinger.ping(self)

class Pinger(LocalActor):

    def birth(self):
        m1 = Ponger()
        m1.pong(self)

    def ping(self, ponger):
        print ‘ping’
        ponger.pong(self)

Pinger() # prints pong, ping, pong, ping, pong...
```

- `find_subtype(t)` — find a random Actor of type `t` or a subtype of `t`.
- `find_type(t)` — find a random Actor specifically of type `t`.
- `find_all_types(t)` — find all of the Actors of the given type `t`.

The first two primitives can be used to distribute load amongst a class of Actor since a random instance is chosen. The third enables the explicit distribution of a set of tasks over a breed of Actor in the Theatre network. If the work can be performed locally then an Actor *Pool* can be used. Actor pools expose the interface of the underlying type in list form and maps each element of the list as a message to a specific Actor instance.

If Actor Pool for type `t` is requested and there are no suitable Actors then if `t` is a type of `CreateableActor` new Actor instances can be created automatically. The feature could help mediate flash-crowds,[5] a sudden, extreme spike in request frequency (shown in Figure 2.4 on the current page), which can have extremely harmful effects on Web servers. Dynamically *createable* Actors could allow the server to automatically scale to service the increased load.

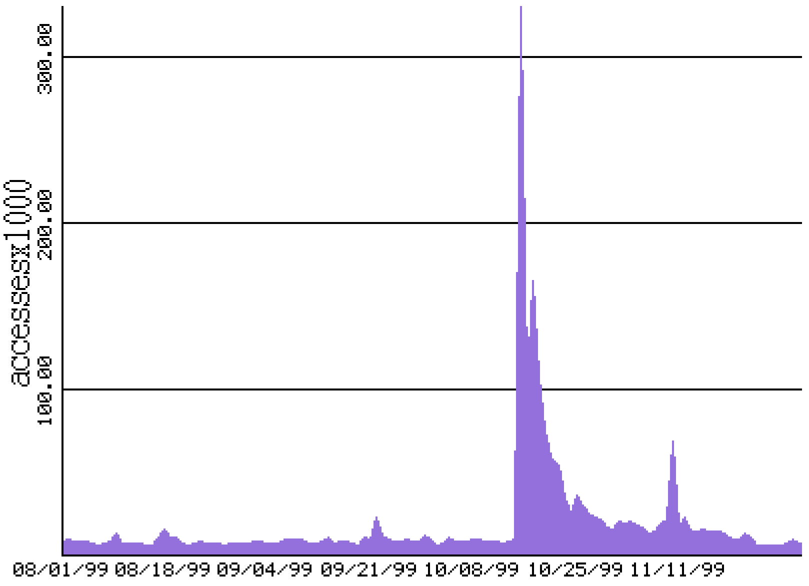

Figure 2.4: **Visualisation of a Flash-Crowd** on the US Geological Survey Pasadena Office Website after an earthquake in California on October 16, 1999 [53].

The concept of a dynamically *createable* Actor is extremely powerful when considering Web servers and the effect that flash-crowds can have on them. If the server can dynamically scale to service the increased load due to a sudden spike in visitors then the impact on response times can be reduced.

2.6.4 Conclusion

The language features described in the previous sections allow the simple distribution of workload across Actors and across Theatres making Stage a suitable candidate for a Web server environment. The clean syntax of Stage, demonstrated in Section 2.6.2, make it appropriate for use as high-level language for creating Web applications.

[5] Also known as the Slashdot effect.

Chapter 3

Frameworks

The following chapter gives an overview of a range of existing Web frameworks to compare and contrast their features and their approaches for solving problems common to Web application development.

3.1 Django

Django is a Python-based framework that describes itself as the 'Web framework for perfectionists with deadlines'. It emphasises rapid development and to that end provides a set of tools which perform to common Web application tasks. Development is based around the MVC triad that uses a template system for user interface creation and a database for persistent storage. Django's hierarchy is occasionally referred to MTV, Model-Template-View, due Django itself handling much of the Controllers work [38]. The following sections explore Django's design of each MTV component.

3.1.1 Model

The Model component of the MVC triad in a Django applicaiton is defined using Python classes which extend the `Model` class, such as in Listing 11. From the class definition SQL is generated using `Django`'s toolkit with each class mapping to a table.

Listing 11 A Django Model

```
class Todo(models.Model):
    name = models.CharField(max_length=50)
    desc = models.CharField(max_length=500)
    priority = models.IntegerField(choices=((1, 'High'),(2,'Medium'),(3,'Low')))
    due = models.DateTimeField('due date')
    parent = models.ForeignKey('self') #items are nestable
```

Member variables in the class that are of type `Field` denote attributes that should be stored in the database, each variable is mapped to a column name and type. There is a wide variety of pre-built field types ranging from the basic `CharField` for text to the more specific, such as `URLField` and `EmailField`, as well as fields for referencing other Model types. The SQL for Listing 11 is shown in Listing 12. In turn, the SQL is used to create database tables, again using the toolkit.

Listing 12 SQL for Django Model

```
    CREATE TABLE "todo_todo" (
        "id" integer NOT NULL PRIMARY KEY,
        "name" varchar(50) NOT NULL,
        "desc" varchar(500) NOT NULL,
        "priority" integer NOT NULL,
        "due" datetime NOT NULL,
        "parent_id" integer NOT NULL
    );
```

Model objects possess a CRUD interface for database modification through inheriting the `Model` class[1] which is demonstrated in Listing 13. The `save` method is used on instances for create and update operations, while the `delete` method unsurprisingly deletes the instance's record from the database. There is a wide range of retrieval operations available through methods such as `all`, `filter` and `exclude` of the static `objects` property. The methods result in a `QuerySet` object which, when evaluated, results in a query to the database and produces a list of the relevant objects. Detailed control over the datasets selected from the database is available through specially formatted keyword arguments (as on line 7 of Listing 13) and through chaining of method calls. Method chaining is similar to Futures in Stage — they are only evaluated when they are used, the chained query on line 8 only actually queries the database on line 10.

Listing 13 Django database queries

```
1     t = Todo(name="Outsourcing Report",desc="For project",priority=3,due=...
2     t.save()                               #create
3     t.name = "The Outsourcing Report"
4     t.save()                               #update
5     a = Todo.models.all()                  #all todo items.
6     f = a.filter(priority=3)               #high priority only.
7     e = a.filter(due__gte=datetime.now())  #unexpired todos only.
8     c = e.exclude(priority__lt=3)          #high priority, unexpired only.
      ⋮
9
10    for t in c:                            #database query
```

Through Django's Model layer the user is supplied with an easy way to define application data structures and a straightforward way of creating, modifying and retrieving data stored in the application's database.

3.1.2 Templates

In Django, Templates are the equivalent of Views. Through the template system presentation logic is strictly separated from business logic. Templates define representations of data for any form of textual output in an abstract mark-up language totally separate of Python. A sample template which could be used to list all todos is shown in Listing 14. Built-in template tags are used as control structures for conditional and loop operations. Variable output can be controlled through the use of filters which can be chained together, each piping it's output on to the next similarly to shell scripting. Templates are also composable and extendable promoting template re-use where possible.

Listing 14 A Django Template

```
<ul>{% for todo in todos %}
  <li class="priortiy{{ todo.priority }}">
    <b>{{ todo.name }}</b><em>{{ todo.desc|truncatewords=15 }}</em>
    <span class="timeleft">{{ todo.due|timeuntil }}</span>{#time left for todo#}
  </li>
{% endfor %}</ul>
```

Template output is produced by evaluating the template in a *context*. A context is esentially a dictionary, it defines the environment in which template evaluation takes place and specifies the values the template should use when referring to variables. A simple context for the previous listing is shown in Listing 15.

Listing 15 A Django Context

```
    c = Context({'todos' : Todo.objects.all() });
```

`Django`'s template system is extremely verbose compared to other template systems with similar features [35]. Possibly the most useful aspect of its design is that the output format is not limited — templates

[1]Or "for free" as it is put in the documentation [3]

can be created for JSON, CSV and all varieties of XML including XHTML, RSS and Atom. It also has some valuable features through its block and filter system but these have some frustrating limitations such as the lack of boolean comparison for `if` blocks.

3.1.3 Views

The previous two sections discussed how Django manages data model definition, and its persistent storage, and also the creation of data representations. This section discusses Views which, in simple terms, are responsible for responding to HTTP requests with HTTP responses. As in Listing 16, Views are typically defined as functions in which any necessary work is carried out and the response created for return to the user.

Listing 16 A Django View

```
    def listtodos(request): #lists the todos
        t = loader.get_template('template/todos.tpl')
        c = Context({'todos' : Todo.objects.all() })
        return HttpResponse(t.render(c)) #evaluate template in context
```

Views are bound to a particular URL through setting the `urlpatterns` variable for the site. The variable is map of the form:

$$\textit{url regular expression} \longrightarrow \textit{View function}$$

The use of regular expressions to specify the URL enables variables to be created from their elements which can then be passed on as parameters to View functions. For example, the url `/todo/123/` could be parsed to call a View function which selects and displays a specific todo, number 123, from the database. This approach is limited when the full range of HTTP methods is taken into consideration because the View function must inspect its `request` parameter to determine the method. A better method would be to specify the required View function in `urlpatterns` for each method implemented. Better still would be to map to a class, rather than a function, which implements functions for the needed HTTP request methods.

3.2 Lift

Lift is a framework that emphasises “security, maintainability, scalability and performance” [22]. It is based on the multi-paradigm Scala, a JVM language that combines aspects of object-oriented programming and functional programming along with event-based Actors [44]. Like Django, Lift focusses strictly on the MVC paradigm to separate business and presentation logic. It takes advantage of `Scala`’s Actors to provide scalable easy to use Comet mechanism and encourages their use to model user interactions asynchronously as described in the next section [31].

3.2.1 Actors in Lift

Lift’s main use of Actors is to provide easy Comet (see Section 2.3 on page 6) support through extension of the `CometActor` class. The canonical example [18], reproduced in Listing 17, shows how a `CometActor` is defined. Lines 2 to 4 are relevant only for template binding and the clock’s initial representation. On line 6 the Actor schedules the sending of a `Tick` message to itself in ten seconds.

Listing 17 A Lift Comet Actor

```
1 class Clock extends CometActor {
2     override def defaultPrefix = Full("clk")
3     def render = bind("time" -> timeSpan)
4     def timeSpan = (<span id="time">{timeNow}</span>)
5
6     ActorPing.schedule(this, Tick, 10000L)
7
```

```
 8     override def lowPriority :
 9     PartialFunction[Any, Unit] = {
10         case Tick => {
11             partialUpdate(SetHtml("time", Text(timeNow.toString)))
12             ActorPing.schedule(this, Tick, 10000L)
13         }
14     }
15 }
16 case object Tick
```

Lines 8 to 14 defines how the Actor responds to the message. The `partialUpdate` function does the leg-work of changing the HTML content of the `time` element to the current time. The Actor is used in Lift's XML template system as shown in Listing 18. When the template is processed the `lift:comet` tag produces the necessary code to perform the Comet requests to the specified `Clock` Actor. The `clk:time` tag refers to the binding code on lines 2 to 4 of Listing 17.

Listing 18 A Lift Comet XML Template [18]

```
<lift:surround with="default" at="content">
    <lift:comet type="Clock" name="Other">
        Current Time: <clk:time>Missing Clock</clk:time>
    </lift:comet>
</lift:surround>
```

The example above demonstrates only simple Comet usage. Full multi-user exploitation of Comet can be accomplished through the use of standard Actors to coordinate messages between instances of `CometActors`. For example, an `Auctioneer` Actor could coordinate the highest bid for a group of bidding `AuctionActor` Comet Actors, as in [28].

3.3 Google Web Toolkit

The Java-based Google Web Toolkit (GWT[2]) employs JavaScript cross-compilation techniques to create what Google terms "complex browser-based applications" such as Google Wave [32]. GWT supports most Java 1.5 syntax, with a few caveats due to incompatibilities with the implementation of JavaScript, and maintains a JRE Emulation library which contains a subset of the Java API which can be translated automatically into JavaScript. User interface creation is accomplished through a widget library similar to that of AWT/Swing or SWT. It is also possible to jump into native JavaScript where necessary through the JavaScript Native Interface (JSNI) and make server calls through an RPC mechanism or through standard HTTP requests. These features, along with Pyjamas, a Python port of GWT, are discussed in the following sections.

3.3.1 User Interface Creation

GWT's widget library includes standard HTML elements such as `Buttons`, `TextBoxes` and `Forms` as well as more advanced composite widgets including `Trees`, `SuggestBoxes`. Composites such as the basic `HorizontalPanel` and more advanced `TabPanels` and `SplitLayoutPanels` provide fine-grained and advanced management of the user interface layout. The control mechanism is managed through `EventHandlers`, an arrangement similar to AWT's `Listeners`. An example is shown in Listing 19 with the result shown in Figure 3.1. GWT's libraries lead to a user interface creation process that will be familiar to any Java developer who has created GUI applications using the standard Swing/AWT libraries.

The output shown in Figure 3.1 is plain un-styled HTML as rendered by the browser. Rather than allowing the developer to style the user interface programmatically GWT prefers the use of Cascading Style Sheets (CSS). Widgets therefore only expose methods for setting style names which translate to

[2]Pronounced 'gwit'

Listing 19 GWT User Interface Creation

```
public class Hello implements EntryPoint {

  public void onModuleLoad() {
    Label l = new Label("Your name: ");
    final TextBox f = new TextBox();
    final Label r = new Label("");

    Button b = new Button("Greet", new ClickHandler() {
      public void onClick(ClickEvent event) {//attach the event handler
        r.setText("Hello, " + f.getText() + "!");
      }
    });

    Panel p = RootPanel.get();//add the widgets to the root panel.
    p.add(l); p.add(f); p.add(b); p.add(r);
  }
}
```

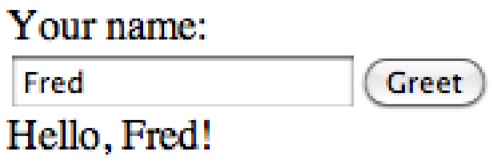

Figure 3.1: **GWT User Interface**: *Result of* Listing 19

HTML `class` attributes in the compiled code. Often a designer may want a specific style for a specific element which done through setting the element's `id` attribute. In GWT setting an element's `id` is done through the `DOM` object as in Listing 20. The `DOM` object can be used for lower-level manipulation of the Document Object Model of the page.

Listing 20 Setting an element id in GWT

```
DOM.setElementAttribute(f.getElement(), "id", "name-text-box");
```

User interface construction can also be controlled declaratively using the UiBinder library through which the user interface structure can be defined in an XML file. The elements defined in the XML can then be bound to fields inside the `Java` code allowing function to be controlled programmatically.

3.3.2 Cross-Compilation Process

The Java to JavaScript cross-compilation process in GWT is a complex process. There is no direct mapping of source code to output file. To provide for different locales and the varying implementations of JavaScript across popular modern browsers there are a number of different files output from the compilation process. The output files for an application called `Todo` include:

- `Todo.html` — the root application HTML file. This is the document served when the user first visits the site.
- `todo.nocache.js` — boiler-plate JavaScript included in the root application HTML.
- A number of files which take the form `{hash}.cache.html` — these contain localised, browser-specific JavaScript application implementations.

The range of generated implementations allow *deferred binding* — only the relevant, localised implementation for the current user is downloaded. The browser-specific files contain the JavaScript implementations of the required portions of the JAVA API along with the JavaScript implementations of the developer written code. There are three compilation options:

- `OBFUSCATED` — the default option intended for production use, output is not human-readable but the size is highly reduced for fast downloading.
- `PRETTY` — a relatively more human-readable option.
- `DETAILED` — a more verbose version of `PRETTY` with long, fully-qualified variable and function names [34].

The compressed `OBFUSCATED` version of Listing 19 runs to 354 lines while the `PRETTY` and `DETAILED` versions run to 2473 and 2703 lines respectively. The process that the GWT compiler goes through to create the JavaScript is slightly more clear when the `DETAILED` option is used. The basic buffer class in Listing 21 compiles to the JavaScript shown in Listing 22.

Listing 21 A Very Simple JAVA Buffer

```
public class Buffer {
    private Object obj;
    private boolean full = false;

    public void put(Object o){
        full = true;
        obj = o;
    }

    public Object get(){
        return obj;
    }

    public void empty(){
        full = false;
        obj = null;
    }
}
```

Listing 22 Buffer Compiled to Detailed JavaScript

```
function gwttest_Buffer(){
}

_ = gwttest_Buffer.prototype = new java_lang_Object;
_.java_lang_Object_typeId$ = 0;
_.gwttest_Buffer_obj = null;
```

The first detail is that the name of the class in the JavaScript is taken directly from the fully qualified `Java` name for the class, and the `obj` field name is appended to this to make the field in JavaScript. The more subtle aspect is that there is no implementation of the `put` and `get` methods. In fact, when the buffer is used the `obj` and `full` fields are accessed directly.

3.3.3 JavaScript Native Interface

At points it can be useful to jump into native JavaScript just as at points it can be convenient to use inline assembler in C. GWT achieves this through what it calls the JavaScript Native Interface which hijacks the `native` keyword to denote JavaScript functions, as in Listing 23.

Listing 23 A JSNI Method

```
public native int getHouseNumber(String name) /*-{
    //JavaScript goes here...
}-*/;
```

Through such functions the developer is able to use third-party JavaScript libraries with extra features such as animation or expose a JavaScript API for their GWT application which other page elements can use. JSNI methods are able to take JAVA objects as parameters and also return them . It is also possible to call parameterised JAVA methods and reference fields from the parent class, as shown in the augmented buffer in Listing 24 (which follows on from Listing 21 on the preceding page).

Listing 24 Extended Buffer with JSNI

```
public native void passObject() /*-{
    if(this.@gwttest.Buffer::full){
        someObject.give(this.@gwttest.Buffer::obj);
        this.@gwttest.Buffer::empty()();
    }
}-*/;
```

One downside of the JSNI is that static type checking is not possible: a JSNI method which claims to return an `int` but actually returns a `float` will only cause an error at runtime [32]. Through the JSNI it is possible to closely intertwine JAVA and JavaScript allowing developers to create an interface between a GWT application and other JavaScripts. It's also possible to wrap full third-party JavaScript libraries for high-level use in a GWT application [20].

3.3.4 Server Calls

GWT provides a RPC mechanism for communicating with the server. Since JavaScript execution is single-threaded, only asynchronous calls are permitted to avoid locking up the user interface while waiting for a response. An RPC mechanism is made up of three main components:

- **Service Interface** — an extension of the `RemoteService` interface that defines the procedures provided by the remote service.
- **Service Implementation** — the server-side implementation of the Service Interface which extends RemoteServiceServlet
- **Asynchronous Service Interface** — the interface actually used by the client-side JAVA. It must implement the same methods as the Service Interface but with asynchronous signatures as described below.

Listing 25 shows how the asynchronous interface differs from the service interface, asynchronous interface methods must have the same signature as their service counter parts but they must return `void` and take an additional argument of type `AsyncCallback` which handles the response from the server. For compilation reasons the asynchronous interface must take the format of {`ServiceInterfaceName`}`Async`.

Listing 25 GWT RPC Interfaces

```
public interface AddressService extends RemoteService{
    public Address getAddress(User user);
}

public interface AddressServiceAsync{
    public void getAddress(User user, AsyncCallback<Address> callback)
}
```

Actually making the RPC call is performed as in Listing 26. The service must be created through a call to `GWT.create` because the JavaScript RPC implementation is loaded through deferred binding due to the different implementations of `XMLHttpRequest` across browsers.

Listing 26 A GWT RPC

```
...
AddressServiceAsync service =
                (AddressServiceAsync) GWT.create(AddressService.class);

service.getAddress(user, new AsyncCallback<Address>{
    onSuccess(Address a){ //Hooray
        ...
    }
    onFailure(Throwable caught){ // :-(
        ...
    }
});
...
```

GWT's RPC approach is verbose but promotes a clean separation of service implementation and service utilisation and enables statically typed communication. Furthermore, it imposes no constraints on the implementation and architecture of the service — the servlets can be the access point for a large multi-tiered environment or the server itself in a more simple setting [32, 20].

Alternatively to the RPC mechanism there is the `RequestBuilder` object which can be used to create and send 'manual' HTTP requests. Again only asynchronous calls can be made so callback objects must be used to handle the servers response. The `RequestBuilder` object allows detailed control over the nature of the request made through standard HTTP request headers but currently only the HTTP `GET` and `POST` methods are supported.[3] The response is handled by a `RequestCallback` object which, similarly to the `AsyncCallback` object, has methods to manage request success and failure. The `onResponseRecieved` method, which handles request success, is passed a `Response` object through which the HTTP response details are accessible. The `RequestBuilder` approach is much more lightweight than the RPC mechanism with the obvious inconvenience of the loss of static type-checking. It also allows much more fine-grained control over the HTTP request sent to the server meaning that integration with a RESTful Web service would be possible if not for the limitation on the requests methods available.

There is no built-in support for COMET in GWT but this can be remedied in a very simple manner for both RPC and `RequestBuilder` server calls. The calls are wrapped in a method and the last action of the `Callback` handlers is to call the method recursively. This can be done both if the call succeeds and if it fails, so long as special attention is paid to the response status code.

3.3.5 Pyjamas

PYJAMAS is a PYTHON port of GWT with support for PYTHON's language features such as variable and keyword arguments and slicing [4]. It also supplies a widget library equivalent to GWT's. The 'Hello' example of Listing 19 is shown in PYJAMAS in Listing 27. The code is clearly very similar and the rendered output is not perceivably different from that shown in Figure 3.1. Looking closer at the JavaScript implementation show's a large difference in compilation technique — the JavaScript output is not obfuscated in any way and runs to about 16,000 lines.

Pyjamas also provides a method for jumping to raw JavaScript through the `JS()` function. The string argument to the function reproduced directly in the compiled JavaScript. This implementation varies from widely GWT's and is much less verbose — there's no need to define a new function for each snippet of JavaScript required [41].

[3]According to the GWT documentation this is intentional due to a bug in the `XMLHttpRequest` object in Apple's Safari

Listing 27 Pyjamas User Interface Creation (ported from Listing 19)

```
class Greeter:
    def onModuleLoad(self):
        self.l = Label("Your name: ")
        self.f = TextBox()
        self.r = Label("")

        self.b = Button("Greet", Callback(self.f,self.r))
        RootPanel().add(self.l)
        RootPanel().add(self.f)
        RootPanel().add(self.b)
        RootPanel().add(self.r)

class Callback:
    def __init__(self, src, dst):
        self.src = src
        self.dst = dst

    def onClick(self, sender):
        self.dst.setText("Hello, " + self.src.getText() + "!")
```

3.4 XML11

The XML11 framework, created by Arno Puder, enables developers to create Web applications in pure JAVA [49]. It emphasises a pluggable architecture where plugins can be attached at runtime. The client-side sections of XML11 applications are cross-compiled into JavaScript for execution in the browser. Bi-directional communication is supported by the XMLOB component, the XML Object Broker, through which it is possible for an object to communicate with any other despite its position on the client-server divide. Despite the component name, messages can be in both XML and JSON formats. Communication is performed through one of three COMET techniques: *a*) asynchronous (true COMET long-polling) *b*) synchronous *c*) polling — as described in Section 2.3 on page 6. It is the developer's decision to choose the most suitable trade-off between scalability and data latency.

3.4.1 XML11 Cross-compilation Process

The JavaScript generation process that XML11 uses, described in Figure 3.2 on the next page, is very different to that used in GWT and PYJAMAS. The relevant parts of the JAVA code are compiled into standard bytecode as usual. From the bytecode an XML representation of the code is created, each instruction mapping to an XML tag such that:

- `iload 1` $\longrightarrow$ `<jvm:iload label="1"/>`.
- `irem` $\longrightarrow$ `<jvm:irem />`.
- `iconst 3` $\longrightarrow$ `<jvm:iconst type="int"value="3"/>`.

The use of the `jvm` namespace is due to the use of same process for C# code as well. The XML namespaces `jvm` and `clr` are used to differentiate between XML representations of Java and C# respectively. Once the XML representation has been produced a further translation is performed through XSLT to produce the JavaScript. The JavaScript operates as simple stack-machine such that the translation is done through a one-to-one mapping between an XML tag and its JavaScript equivalent. The resultant JavaScript is correct but neither human readable nor particularly fast.

Since XML11 applications are developed entirely in JAVA there is no *enforced* application structure, simply the developer's best practices. When a class references a resource that can't be migrated to the

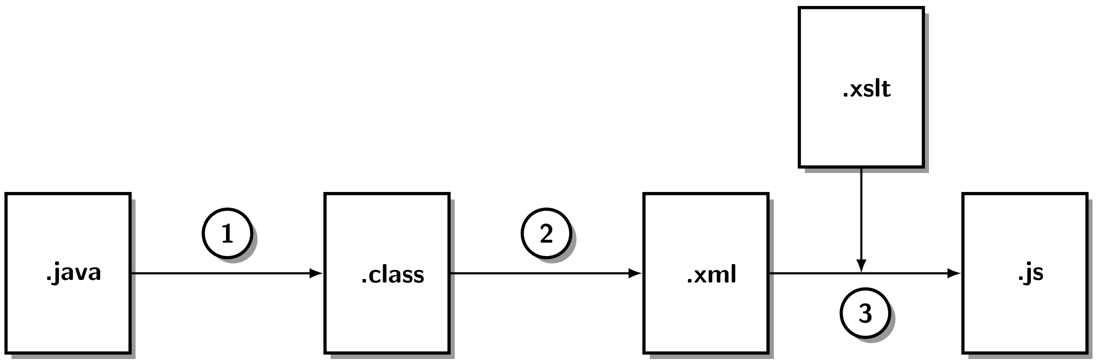

Figure 3.2: **XML11 Client Build Process**:
1) Java source code is compiled to bytecode as usual.
2) The bytecode is translated instruction for instruction to an XML representation.
3) Finally, the XML representation is transformed to JavaScript through XSL transformations.

client, a database for example, it must be kept on the server. The developer must explicitly define which classes are to be migrated to the client and which must stay on the server. When a client-side object references a server-side resource the invocation must be transported over HTTP from client to server and vice versa when server-side objects call a client-side object. The XML bytecode representations of the required classes are used to create a local Proxy object which in turn uses the Object Broker to marshall and transmit the message to the server.

It is the developer's decision whether program logic is migrated to the client or kept server-side. If the application is computationally intensive then it is preferable to keep the implementation on the server for faster execution in JAVA rather than JavaScript. If the application is highly interactive then it is beneficial to migrate relevant code to the client to increase responsiveness for the user and reduce the message load on the server. It's worth noting that XML11 supports the use of Java's AWT, Swing and SWT libraries for creating user interfaces for Web applications. This means that an application can be compiled for desktop and Web using the same source code [48].

3.5 Bullet

Created by Namit Chadha, BULLET is a 'minimal' Web framework based on AIR 2, Actor inspired Ruby 2, an extension of RUBY with Actors [17]. Actors are used throughout BULLET's MVC triad along with *HTTP Streaming*, a COMET-type technique, featuring prominently to create an architectural style coined as *Streaming Model-View-Controller*.

3.5.1 Streaming Model-View-Controller

The Streaming MVC style maintains the same separation of concerns as traditional MVC but when presenting data the models are rendered on demand and sent to the user as they become available. For example, if a request involves all the entries in a database table then each row is rendered and streamed to the user separately rather than rendering them all and sending them together as a batch. The benefit of this is emphasised when taking third-party tiers into account. The latency for third-party tiers will be much higher than local tiers so responding to a request is limited by the speed of the third-party. Through responding with local data immediately and relaying external data as it becomes available the responsiveness of the server, as perceived by the user, is increased.

The Streaming MVC style is achieved using HTTP Streaming: the client requests data and the server responds as usual but never closes the connection. The server is able to forward supplementary data on to the client as it becomes available through this persistent connection. The novel data can be processed and added to the user interface using hand-coded JavaScript. This approach means that the data the client receives is as fresh as possible since there is minimal data latency compared to the traditional request/response cycle. While indefinite connections are acceptable for a small number of users there is a scalability problem when large numbers of users are involved in long sessions. A Web server using a thread per connection can quickly run out of resources when connection longevity is potentially infinite. AiR 2 uses event-based Actors rather than thread-based Actors so this issue is less critical.

3.5.2 Use of Actors

Actors are used in all three components of MVC. Model actors are responsible for servicing database requests and querying external resources through *Adapters*. Adapters are responsible for the marshalling of data for external requests and un-marshalling responses to create useable Model objects. Eschewing asynchronous message passing to extract data, synchronous requests to Model Actors result in plain old objects. Bullet's Controller Actors are influenced by those of Camping, a featherweight Ruby Web application framework. They define `get` and `post` functions for specific request paths or *routes*. The definition of each function details the actions to be taken in the event of an HTTP request with the relevant method being received [6]. View Actors are responsible for compiling HTML representations for dispatch to users. Views make use of Haml[4] template system to generate HTML from Model objects.

3.6 Conclusion

Throughout this chapter we have reviewed a number of existing Web frameworks and their approaches to solving common Web application development problems such as persistent data storage and retrieval, user interface creation, and their support for background requests. An overview of their salient features is shown in Table 3.1.

Framework	**Language**	**User Interface**	**Storage Support**	**Ajax/Comet**
Django	Python	Template (Custom)	Database	3rd party only
Lift	Scala	Template (XML)	Database	Through Actors
GWT	Java	Java	User-supplied	Through RPC
XML11	Java	Java	User-supplied	Through XMLOB
Bullet	AiR2	Template (HAML)	Database/Adapters	Through Actors

Table 3.1: Comparison of Web Framework features

[4]HTML Abstraction Mark-up Language

Chapter 4

The Swan Framework

In this chapter we discuss the components that comprise the Swan Web application framework, their roles and implementation. I hope to provide a prospective Swan programmer with a comprehensive understanding of the stages in the creation of an application and an appreciation of Swan's design decisions. Throughout this chapter, where I refer to 'the user' the intended meaning is the user of the framework itself, i.e. the application programmer, rather than the end-user of a Swan application. As Swan is built on Stage, basic knowledge of its language features and execution model may help with understanding though it is not essential.

Mirroring the client-server architecture of the Web, the framework is formed of two distinct modules. The server-side environment, underpinned by an Actor-based Web server, features:

- Customisable handling of HTTP requests to the server and response generation.
- Database and File interaction for data persistence.
- An API for Database model definition and access.
- Access to third-party data sources through HTTP.
- Straightforward binding of resources to URLs.

The client-side of an application is programmed in Stage using the Swan JavaScript module. It includes a library for creating user interfaces and communicating asynchronously with the server using standard Stage language features.

Similarly to any framework, Swan endeavors to offer its users with extensive support throughout the application workflow without imposing unnecessary restrictions. The core tenet of "Should the user have to do this?" provided the main guidance through the design decisions that arose during the implementation of Swan. The application design follows an adaption of the Model-View-Controller pattern, it should be easy to follow for any programmer that has created an application with a user interface. Specialised Actors known as *Handlers* define the Model and the server-side part of the Controller of an application while the client-side Controller and View behaviour are specified through standard Stage Actors. A typical application creation workflow begins with the definition of data models and their methods of access through Handlers. It continues through the externalisation of the resources through URL definition and ends with the creation of the user interface and procedures for interacting with the model.

4.1 The Swan Server

Every Web application needs a stable Web server at its foundation. The server is generally responsible for accepting and managing client connections and pre-processing client requests into a format that is intelligible to the application. The Swan Web server, known as SWS, forms the foundation layer of Swan's server-side module and is a fully-fledged Stage application in it's own right.

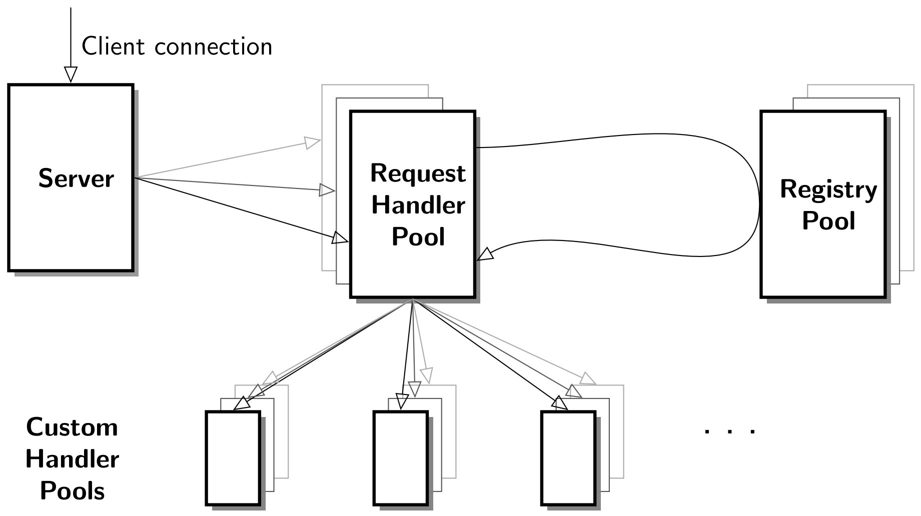

Figure 4.1: **The SWS Structure**: The **Server** receives the connection from the client and passes it to a **Request Handler** which parses the request and its headers before looking up the relevant custom handler through the **Registry** and forwarding on the request.

Client requests are passed through a tree-like chain of responsibility structure composed of Actor pools, as illustrated in Figure 4.1. This structure is similar at its core to that described in the Apache server's `worker` multi-processing module [7], though it is more directly customisable using further Actor pools, rather than passing on the request to external files and scripts.

The central `Server` Actor's sole task is to wait for connections from clients and pass them on down the next link in the chain. The second link in the chain, the `RequestHandlers`, perform the initial triage of requests and forward them down relevant paths in the tree, found through the `Registry`. Further processing of requests and response compilation and transmission are performed by user-defined Handlers, described in Section 4.2, which are registered and bound to specific URL patterns, as detailed in Section 4.3.

The launching of SWS performs the necessary steps to create and initialise the application's handlers. By default a pool of each handler is created along with a pool of primary `RequestHandler` Actors and `Registry` Actors. This results in a set of structures similar to thread pools, the main difference being that an Actor in the pool does not need to be idle to be chosen to process a request. Instead the request is queued up in the Actor's message queue, just as any other message is, and processed when it reaches the front of the queue. The use of pools of each Handler permits the concurrent servicing of requests to the same Handler type.

4.1.1 Request & Response

Typically, rather than interacting directly with the Webserver, the user instead specifies the actions to perform on receipt of a request and how the response to return to the client is compiled. In that sense the processing of a request is one-way, starting at the server and concluding at user-defined functionality with the sending of a response. However, data from a request will often still be needed by the user to properly service that request.

For each HTTP request SWS creates a corresponding `Request` object which lasts through the lifetime of the request and encapsulates the request parameters including the HTTP method, path and headers. It is created by the `RequestHandler` during the initial processing of the request and passed down through the chain as handling continues. This enables the user to interact directly with request data without having to parse the request themselves.

Listing 28 Using `Response` objects to generate responses to client requests

```
# Sends a '200 OK' HTTP response, with body 'content' in JSON encoding
response.send(200,content,'application/json')

# As above, with extra 'cache-control' header.
response.with_headers(cache-control:'no-cache').with_status(200).with_content(content)
response.send()

# Send a '404 Not Found' error.
response.send_error(404, 'Resource not found')
```

From a `Request` object a complementary `Response` object is automatically generated. It is only through Response objects that user can formulate suitable HTTP responses for a request and send data back to the client. `Response` objects have a flexible interface to allow users to compile responses in a variety of different ways depending on the circumstances in which the response is required, examples of which are shown in Listing 28 on this page. The user has full control over the variables set in the response but the framework will in many cases attempt to reduce the user's workload and assign the relevant settings for them. For example, when the user sets the content of the response body, the `Content-length` header is automatically set for them. The final message to a `Response` object is `send`, which takes all of the response parameters that have been set by the user and formats them in to an HTTP response before transmitting the data to the user.

4.1.2 Content-Type Encoding

Data on the Web can be communicated in an array of different formats depending on the format requested and the formats available. SWAN includes an extensible set of encoders and decoders for translating request bodies from and response bodies to format for transmission to and from the client. HTTP requests such as `GET` contain a sorted list of content types in the `Accept` header which designate the preferred format of the response body that the client would like to receive. For example, `text/html, text/plain, */*` implies that HTML is preferred but plain text is also acceptable and if neither of those are possible then anything format will suffice. Unless the user explicitly sets a content-type for the response SWAN will attempt to encode the response content in one of the types specified in the `Accept` header, given that the relevant encoder for that type is present. If no encoder can be found then a plain-text encoder is used and if that encoding fails a `406 Not Acceptable` response is issued, as recommended by the HTTP/1.1 specification (Section 10.4.7 of [24]).

Equivalently for `POST` and `PUT` requests, the encoding of the incoming request body is defined in the `Content-type` header. SWAN will attempt to decode the body into objects that can be used in servicing the request. If no suitable decoder is found then a `415 Unsupported Media Type` reponse is returned to the client.

As mentioned, the set of encoders and decoders is extensible, so the user can add support for a specific content type or encode certain responses in a particular way. They are able to store their custom codecs in their application directory and SWAN will load them into the system at launch time. Given the open-ended nature of the formats that can be requested by the client and used on the server-side the interface for encoders and decoders is unrestrictive. Decoding is initiated using the `get_decoding` method whilst encoding starts with `get_encoding`.

4.2 Handlers

Handlers are SWAN's technique for encapsulation of user-defined functionality. They are specialised Actors used to define access to resources in a SWAN application. They are closely aligned with REST resources described in Section 2.4, specifying the URL for a resource and the HTTP methods that can be

Listing 29 A simple SWAN Handler

```
 1 class Kennel(Handler):
 2     bindings = {'default':'/dogs/?'}
 3 
 4     def birth(self):
 5         self.dogs = list()
 6 
 7     #list the dogs in the kennel
 8     def get(self):
 9         response.send(200, self.dogs)
10 
11     #add a dog to the kennel
12     def post(self, body):
13         self.dogs.extend(body)
14         response.send(204)
```

used to manipulate and query it. When a SWAN application is launched pools for each custom Handler type are created. On delivery of a request for a particular resource it is forwarded to one member of the relevant Handler pool for further processing.

Listing 29 on the current page shows the definition a simple Handler which represents a kennel. Line 2 specifies the Handler's URL binding, explained in more detail in Section 4.3. Methods defined on a Handler that are intended for access through HTTP requests must begin with one of **`get`**, **`put`**, **`post`** or **`delete`**. These methods are called in response to the reception of a corresponding HTTP request. Thus the kennel has the capability to respond to `GET` and `POST` requests: **`get`** method on line 8 will be called on receipt of a request matching `GET /dogs/` whilst the **`post`** method on line 12 specifies the behaviour for a request of the form `POST /dogs/`.

Handlers will automatically respond to `OPTIONS` requests with a list of methods that are available for that Handler, as described in Table 2.1 on page 9. If a Handler receives a request for a HTTP method that it does not support, i.e. for which it does not define a method, then a `405 Method Not Allowed` error is automatically generated and sent.

In the body of a Handler method the user has implicit access to **`request`** and **`response`** objects, as defined in Section 4.1.1, for that particular request. This practice is arguably unintuitive for a new user as it is not obvious where the variables are defined nor that they are even available. It is hoped that it is a convention to which the user will become accustomed and perhaps thankful for, since it avoids the increased verbosity of having to explicitly declare **`request`** and **`response`** parameters for each method of the Handler. In contrast, for methods handling `PUT` and `POST` request, the user must explicitly list the **`body`** parameter as a method argument, it is automatically defined and provides access to the content of the request.

Handlers and their state persist between requests although no request-specific state is retained unless explicitly done so as in Listing 29. The use of such a Handlers must be carefully controlled however as the above would actually result in inconsistent and unexpected behaviour due to the use of pools of each Handler. When a dog is added to the list only the handler servicing that request is modified so requests which are forwarded to other Handlers will result in unpredictable responses. In such cases a single central Actor can be used to maintain the state however this may have performance implications due to its affects on concurrency. A preferred solution would be to store the list in a database table as described in Section 4.2.4.

Listing 30 Handling a long operation

```
class Invoker(Handler):
    bindings = {'default':'/invoke/?'}

    def get(self): #invoke the task
        callback('send_get_response', long_task.get_result(), response)

    def send_get_response(self, result, response):
        response.send(200, result)
```

4.2.1 Long Operations

Often operations involved in compiling a response for a particular request can take some time to complete. For example, imagine a website which determines the optimum route between two locations and the methods of public transport available. Such calculation may take seconds to complete, which, for a Website under high load, will be too long to tie up a Handler for.

In such cases, it is beneficial to delegate that long running task to an auxiliary Actor for execution. As shown in Listing 30, the user can bind a `callback`, a Stage language feature, to send a response when the result is ready. An alternative is to delegate further handling of the request entirely to the support Actor. The benefit of this is that the Handler is released temporarily for that request allowing it to service further requests.

4.2.2 Background Processes

We can also use Handlers to poll continuous processes to check for progress or change in state. For example, in Listing 31 we have an Actor that calculates the procession of prime numbers and a Handler which queries it to find the highest calculated so far. The prime calculator could be started when the server is launched or could be running is a separate Theatre. This concept can be expanded to add a Web user-interface for an existing Stage application using Handlers to hook into the Actors already present in the Theatre environment. This provides the ability to create large tiered architectures as described in Section 2.1.

Listing 31 Handling a continuous process

```
class PrimeHandler(Handler):
    bindings = {'default':'/primes/highest/?'}

    def birth(self):
        # finds the prime calculator in the theatre environment
        self.prime = find_type('PrimeCalculator')

    def get(self):
        response.send(200, .get_highest())

class PrimeCalculator(LocalActor):

    def birth(self):
        self.highest = 2
        while self.highest > 0:
            print "%s, " % self.highest,
            self.highest = self.nextPrime(self.highest+1)
```

Listing 32 Handling files and directories

```
class RootHandler(FileHandler):
    root = "/path/to/public_html/"
    bindings = {"default":"/files/`path`?"}
```

4.2.3 Files

Frequently users will need to be able to serve static files to client such as cascading style-sheets, images and other media. SWAN includes the `FileHandler` type which is used to provide access to the files in a specified directory and its sub folders, allowing them to be supplied as part of an application. As shown in Listing 32, it is extremely straight forward to use: the user must simply give the path to the directory which is to be server and, as usual, the handler's URL binding. When the application is launched, requests such as `GET /files/text/example.txt` will fetch the content of the file at `/path/to/public_html/text/example.txt`.

`FileHandler`s are supported by pool of workers that perform all the file interactions on behalf of the Handlers. Interactions are initiated using callbacks as described in Section 4.2.1 since they are relatively long operations.

4.2.4 Databases

For storage of data between sessions of end-user interaction, applications commonly use relational databases. Rather than provide a `DatabaseHandler` type SWAN provides the `db` package with a lower-level API for database interaction, inspired by DJANGO's Object-relational Mapper (ORM[1]). Using the `db` package the user can define their database tables in STAGE and access them through Handlers without having to write or execute any SQL.

Models & Fields

Database tables in SWAN are represented through subclasses of `Model` while their columns are properties which are subclasses of `Field`. Just as with Handlers, Models are descendants of Actors and during launch of the server a pool of each Model type is created. Listing 33 demonstrates the definition of two models in the context of a discussion board where each discussion thread has a title and a number of posts. The model definitions are used both to create the database tables and for querying and manipulating them once the server is launched.

Listing 33 Defining Database Models

```
class Thread(Model):
    title=TextField()

# Table name
class Post(Model):
# Table properties
    thread = ForeignKey('Thread','posts')
    poster = TextField()
    content= TextField()
    timestamp = TimeField()
```

The basic types of `Field` that are available by default are as follows:

`IntegerField` — for storing integers. Each model automatically has an `id` column of type `IntegerField` included.

[1]Usually pronounced to rhyme with 'form'

Listing 34 Querying Database Models

```
1 #prints the title of each thread that has 'Swan' in the title, and the number of
2 #posts in it.
3 swan_threads = Threads.filter(title=like("%Swan%"))
4 #...
5 for thread in swan_threads:
6     print thread.title, len(thread.posts)
```

`RealField` — for storing floating point numbers.

`TextField` — for text. `TextField`s take a numeric parameter defining the maximum number of characters for text in that column.

`TimeField` — for date and time values.

`ForeignKey` — are used to reference rows in other models. For example, line 7 of Listing 33 indicates that each `Post` references a particular `Thread`. The second parameter, '`posts`', denotes the property name for the set of rows that each foreign row references. That is, each `Thread` has a 'posts' property that can be used to access the `Posts` belonging to that discussion.

Manipulating and Querying Models

Models can be queried using three class methods listed below. It's worth noting that while the methods appear static in their use, as in Listing 34, they are treated as standard Actor messages at runtime: they pass through an Actor's message queue before execution.

`all` — creates and returns a list of Model instances representing each row in the database table.

`filter` — takes a list of constraints and returns a list of Model instances representing each row that fits the defined constraints.

`get` — takes a list of constraints that identify exactly one row from the table and returns a Model instance for it.

Constraints are passed as a dictionary whose keys must be properties of the Model, as shown on line 3 of Listing 34. Constraints available by default are `equals`, `lessthan`, `greaterthan`, `like`, `before` and `after` (for use with `TimeFields`), and `between`, though as with most aspects of Swan, the user can add their own if necessary.

A new row in a Model can be created using the class method `create` which takes a set of properties and creates a Model instance from them. If the properties of a Model instance are changed then it's `save` method must be called to update the database. Properties of an instance can be accessed as standard `Python` object properties, see line 6 of Listing 34.

Model instances and sets of Model instances are evaluated lazily — until the value of a property is actually required the instances remain as proxy objects. The result of the laziness is that the execution of queries to the database is delayed until unavoidable, at line 5 of Listing 34 rather than line 3, and the number of queries executed is partially reduced. This allows the chaining of `filter`ings and the creation of *potential* Model instances which are only realised as necessary.

The convenience of such a database interface can often result in suboptimal database utilisation, even with the lazy execution of queries. If we refer once more to Listing 34, the number of queries that will be executed is one more than the number of `Threads` matched by the filter on line 3. One query evaluates the filter, while there is one query for each `len(thread.posts)`. In this case the same results can be retrieved in one, more complex, query. If the user is comfortable with SQL then in such instances they

Listing 35 Using Models in Handlers

```
class ThreadHandler(Handler):
bindings = {
    'default':'/threads/'
}

#get a list of thread titles
def get(self):
    titles = [thread.title for thread in Threads.all()]
    response.send(200, titles, 'application/json')
```

are able to gain access to a low-level Database Actor, described in Section 4.5.2, from a Model, through which they can execute custom queries.

Listing 35 on the current page illustrates the typical use-case for Models: their utilisation within Handlers to service database requests. Although it is not entirely evident in such a short example, SWAN's provision of the database interface below the Handler level rather in the Handler level itself, increases flexibility since the user has access to multiple Models when defining a Handler.

4.2.5 Third-Party Sources

For security reasons, asynchronous requests from the client-side are currently restricted to the *same-origin policy*: requests must be to the same server using the same protocol and port as the original request. To side-step this limitation and allow users to combine foreign data into responses, SWAN provides access to a pool of `HTTP` Actors which can make requests to external sources on behalf of a Handler.

Handlers that need to make external requests should extend the basic `ExternalHandler` Actor which provides four functions for one-shot requests for the four main HTTP methods. For cases where more flexibility is needed the `get_connection` method is available. It returns an connection object with an interface which will be familiar to those with a knowledge of PYTHON's `httplib` but also includes accelerator functions for the primary HTTP methods.

Listing 36 demonstrates the use of an `ExternalHandler` to make a request to `flickr.com`, the photo sharing site. In this case we have used a callback recommended in Section 4.2.1 since HTTP requests can be quite lengthy. Lines 7 and 8 illustrate the alternative approaches for making external requests, the results are identical and both could replace the request on line 9.

Listing 36 Accessing External Resources

```
1  class FlickrHandler(ExternalHandler):
2      bindings = {'default':'/flickr/`user`'}
3
4      def get(self, user):
5          server = 'api.flickr.com'
6          url = '/services/rest/?method=flickr.people.getPublicPhotos&username=' + user
7          # self.get_connection(server).get(url).getresponse().read()
8          # self.get_connection(server).request('GET',url).getresponse().read()
9          callback('send_get_response', self.get_request(server, url), response)
10
11     def send_get_response(self, body, response)
12         #custom processing here.
13         response.send(200, resp, 'text/xml')
```

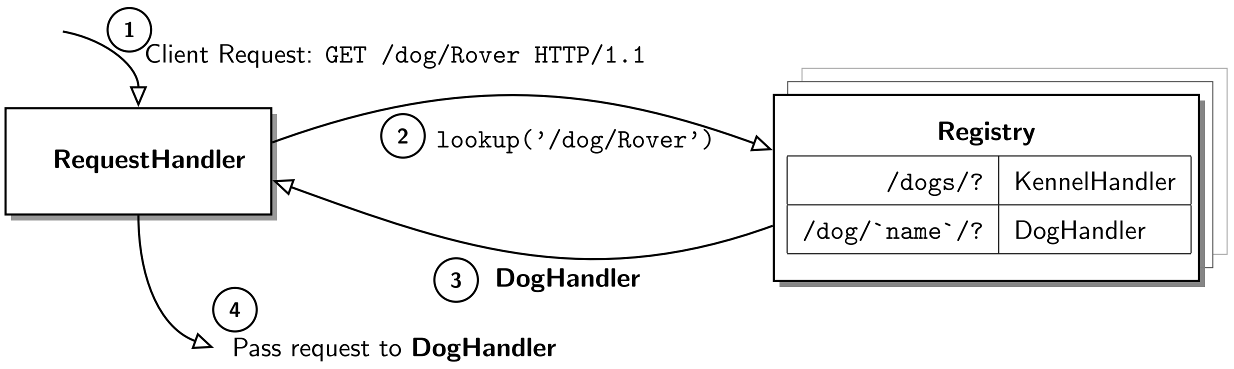

Figure 4.2: **Handler Lookup**:
1) A `GET` request from the client is forwarded to a `RequestHandler`.
2) The `RequestHandler` looks up the the request path in the **Registry**.
3) The **Registry** scans its bindings dictionary for a pattern matching the request path and returns a pool of the corresponding Handler type.
4) The `RequestHandler` forwards the request on to a member of the received Handler pool.

Since the data retrieved from external sources is somewhat unpredictable in content and format, direct support from Swan for intermediate processing is hard to provide. Of course, since Swan is based on Stage the user has access to all of Python's processing libraries such as `json`, `htmllib` and its XML parsers.

4.3 Handler Bindings and Registration

The previous section described how the user can use Handlers to specify the actions performed when reacting to a request from the client. An equally important aspect of a Web framework is the means through which custom server-side functionality is exposed to the client-side using Uniform Resource Locators. In this section we discuss how access to Swan Handlers is made available through the Registry and Handler *bindings*.

As mentioned in Section 4.2, each Handler definition includes a `bindings` parameter which designates the URL patterns for requests that should be directed to that Handler type. At application launch time, Swan creates a pool of each Handler type and makes an entry in the server Registry, a pool of `Registry` Actors, according to the Handler's `bindings` directive.

As portrayed in Figure 4.2, when a request is received the initial `RequestHandler` queries the Registry with the request path. If the request path matches a binding pattern in the Registry it will respond with a reference to the pool for the corresponding Handler type. The request is forwarded to a member of the pool.

4.3.1 Binding Patterns

Binding patterns in Swan are built on top of regular expressions, in a similar fashion to Django. They are composed of three types of component:

static — Specific text that must appear in the pattern, e.g. `/dog/` in the pattern `/dog/`name`/?`.

dynamic — For example `` `name` `` in `/dog/`name`/?`. Any text can appear in place of `` `name` `` and will be passed to the Handler function as a named argument.

optional — Any character, static component enclosed in parentheses or dynamic component immediately followed by a question mark, `?`, is optional — the pattern will match a given URL whether or not the component is present. This is especially useful for trailing forward-slashes: `/dogs/?` matches requests for both `/dogs` and `/dogs/`.

Dynamic components permit the user to define flexible patterns from which they can easily access request parameters. As shown in Listing 37, Handler methods that are bound to patterns with dynamic components list those components' names as method parameters. If the component is optional then a default value should be given for it.

Listing 37 Accessing URL Parameters

```
class DogHandler(Handler):
    bindings = {'default':'/dog/`name`/?`surname`?/?'}

    def get(self, name, surname = None):
        # process request
```

4.3.2 Binding Specifiers

It is normally desirable to group similar functionality together. Swan users can employ *binding specifiers* to extend a single Handler to cope with all aspects of one logical concept rather than having multiple Handlers.

Handler bindings themselves are maps of specifier to binding pattern. The `default` specifier directs requests to the basic `get`, `post`, `put`, `delete` functions. Further specifiers can be added to a binding, requests which match these additional patterns are routed through to the same Handler type, but to functions of the form `<<HTTP method>>_<<specifier name>>`. So if the specifier `'owner':'/dog/`name`/owner/?` were added to the `DogHandler` in Listing 37 then requests such as `GET /dog/Rover/owner HTTP/1.1` would be forwarded the function `get_owner` for handling.

4.3.3 Default Handlers

Almost all Web users are familiar with the `404 Not Found` error status, the standard response if the resource identified by request URL cannot be found by the server. If the Registry is queried with a URL that doesn't match any registered Handler binding pattern then the request is passed on to a member of the Default Handler pool.

`DefaultHandler`'s act as a catch-all for client requests to resources that cannot be located, their standard behaviour is to send a `404` response to the client. The user can override the default behaviour by defining their own Handler which extends the `DefaultHandler` type.

4.3.4 REST

It may not be immediately pertinent why Swan requires the user to provide binding patterns for the applications Handlers. It could theoretically generate the required URLs for the application itself. However, making the URLs patterns explicit means that REST interface, as detailed in Section 2.4, is immediately obvious. Additionally, because the patterns are defined by the user, they have full control over the structure of the resource identification hierarchy as discussed in Section 2.4.3.

4.4 Programming the Client-Side

As discussed in Chapter 2 and demonstrated in Chapter 3 there are a number of different approaches for the creation user-interface and control procedures for the client-side of a Web application. As opposed to the template languages of Lift and Django, Swan's uses a JavaScript cross-compilation technique similar to GWT and Pyjamas, permitting the user to create the client-side in Stage.

4.4.1 The SwanJS Compiler

In a process reflecting GWT's compilation from JAVA to JavaScript, the SWANJS compiler makes it possible for the user to create the front-end of their application in STAGE. The compiler processes the STAGE source, converts it to JavaScript and embeds it in an HTML stub. When the application is loaded the front-end is downloaded to the client and executed in the browser.

The cross-compilation procedure uses a syntactic translation process rather than a byte-code conversion equivalent to XML11. The compiler walks the abstract syntax tree of the application code and creates equivalent JavaScript constructs. The majority of STAGE syntax is supported in conversion, including operator overloading. The most apparent PYTHON language features which are not translated are `exec` statements and features related to generator expressions, though STAGE itself does not support their use either (see Section 5.1 of [9]).

The translated JavaScript is encapsulated in a base HTML file which is served to the client when the application is loaded. The HTML file also includes JavaScript common to all SWAN applications such as JavaScript implementations of PYTHON's built-in functions. Not all of PYTHON's built-in functions and types are supported. Notably functions related to dynamic code import and exection, the `compile` and `__import__` functions, are not supported, and also the `file` type.

Occasionally it can be necessary to write directly in JavaScript. Rather using an approach comparable to GWT, enforcing that an entire function is written in JavaScript, SWAN users can call the `native` method, passing a string of JavaScript as an argument, similar to PYJAMAS. At compile time, the JavaScript is embedded directly into compilation without any translation.

4.4.2 UI Library

A major part of application creation is the production of the user interface, through which the user interacts with the application. As previously discussed, GWT and PYJAMAS both supply a library with a large range of components that the user can compose to create their interface. SWAN's UI library contains a variety of basic components that parallel the standard HTML elements.

The UI Library wraps the low-level manipulations provided by the HTML/XML Document Object Model (DOM) and enables the user to manipulate components through an interface at a similar level to regular desktop UI libraries such as JAVA's Swing. In line with the Composite pattern, a user-interface can be represented as a tree of components that can be grouped into two major categories:

Composite — These components can contain other components. Components in this category include the base `Container` type, `Forms` and `Lists`.

Leaf — Individual components that are the end of a branch in the tree. These include input elements such as `Button`, `RadioButton`, `CheckBox`, `TextBox` and `TextArea`.

Listening for and responding to user input is controlled through STAGE method passing as described in section 5.3.7 of [9]. This strategy of using a standard STAGE language feature avoids the need for the application developer to learn any novel mechanism for handling user input.

4.4.3 Asynchronous HTTP Requests

To create a more seamless interaction experience and avoid disruptive page loads, SWAN uses asynchronous, background HTTP, or AJAX, requests to communicate with the server-side of the application. Requests can be made to any of the Handlers that the application defines. To the user, the requests appear as standard Actor messaging but they must be wrapped in a STAGE `callback`. This is a reasonable restriction as HTTP requests can be very lengthy and allows the user interface to remain responsive during the request.

Listing 38 Calling a Handler from the Client-side

```
    callback( DogHandler.get('Rover'), got_dog )
```

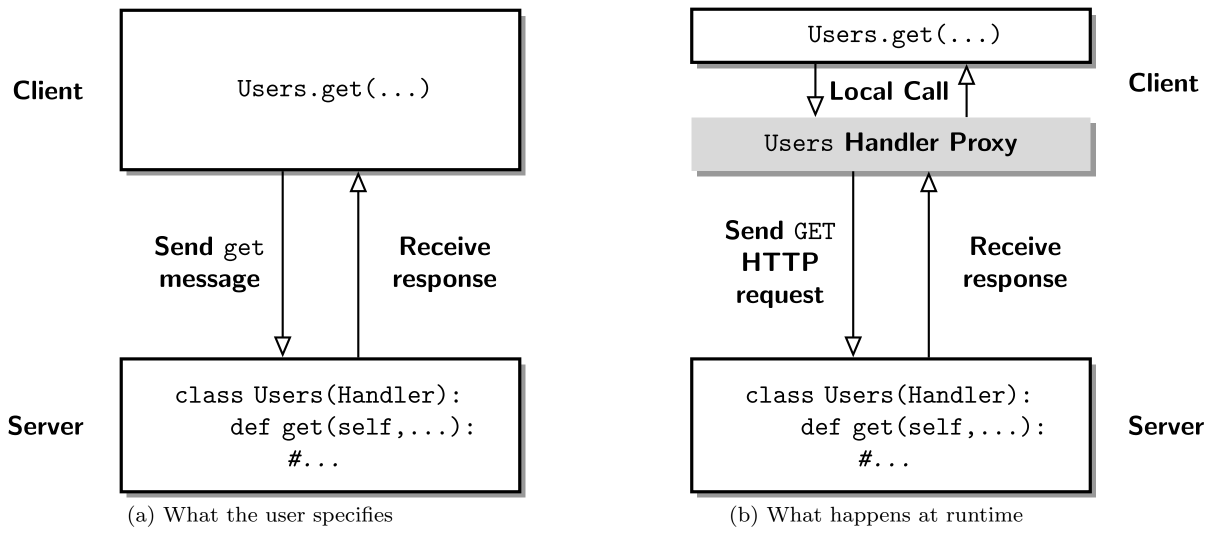

Figure 4.3: **Swan Asynchronous Requests**: The use of a Handler proxy on the client-side obfuscates the asynchronous HTTP request that is made between client and server. Thus the SWAN developer need only write the standard Actor message passing depicted in 4.3a while at runtime the process shown in 4.3b is followed.

During the compilation process, the compile embeds Handler proxies inside the client-side application module. These proxy objects have an identical interface to their server-side counterparts and, when queried, perform the necessary operations to create and send an asynchronous request to server, specifically to the Handler that was requested. Figure 4.3 on this page illustrates this difference between what the user writes and what is executed at runtime.

Comet

As outlined in Section 2.3, there are a number of different ways of achieving COMET effects, providing the illusion of the server pushing data to the client. SWAN uses the *long-polling* technique to create this appearance — when a response from the server is received for one request the client immediately makes a subsequent request to the server.

COMET is available in SWAN on the client-side through the `bind` function which takes the same parameters as `callback`. Each time a response from the server is received the provided callback function is executed while a further asynchronous request is made to the server. This technique requires support from the server-side since it must maintain the connection to the client until there is relevant information to send back.

4.5 Additions to Stage

This section describes extensions to the core STAGE language and runtime that were added to support the server-side implementation of SWAN. These features have been incorporated directly into STAGE since the programmer of any STAGE application would find them useful.

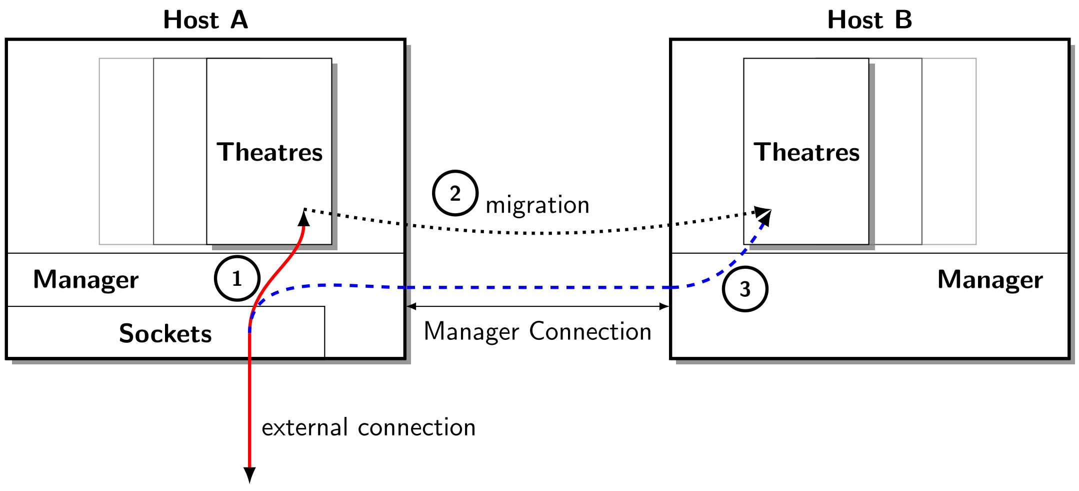

Figure 4.4: **Socket References**:
1) An Actor communicates with a external client through the Host A Manager's socket store via a socket reference.
2) The Actor migrates to a Host B in the STAGE network, taking its socket reference with it.
3) The Actor can still communicate with the external client, which remains connected to Host A, through the same socket reference. The communication is redirected through the Manager connection between the hosts A and B

4.5.1 Sockets

The implementation of the SWAN Web server, as described in Section 4.1, fundamentally relies on the use of network sockets for communicating with clients and the capacity to move those sockets from Actor to Actor.

SWAN's distributed runtime capabilities provide support for network communication between Managers and Theatres but the capacity to communicate with non-STAGE resources was severely limited. While full access to the PYTHON `socket` library is available to Actors, its use critically impedes distributed execution because it is not possible to transfer socket's between Theatres due to their low-level nature. Any Actor that directly uses PYTHON sockets becomes 'stuck' in the Theatre in which it is currently hosted.

To allow Actors to retain full mobility in STAGE's multi-host execution environment support for sockets is built directly into Managers. In addition to their original stores, each Manager maintains a further store of sockets from which an Actor can retrieve a socket reference with the standard `Python` socket interface. As described in Figure 4.4 , since each Manager is directly accessible from any other and there is one Manager per host, communication with a socket can continue after a socket reference is transferred between Actors or hosts as the underlying socket is maintained at the original host. Since Actors no longer retain direct access to sockets they are able to move between hosts whilst maintaining contact with any sockets they are using.

The Manager's interface for socket management provides three methods:

- `open_socket(`*`port`*`)` – Creates a new server socket in the Manager's store, listening on the specified port.
- `connect_socket(`*`address`*`)` – Creates a new client socket in the Manager's store, connected to the specified address.
- `close_socket(`*`reference`*`)` – Removes socket from the store and closes it.

Each newly created socket runs in its own thread to aid concurrency — this should not affect performance greatly as Managers have no other long running threads. The thread runs until either of its socket's peers close the connection.

For long-running tasks the overhead of communication redirection after Actor migration can be very high. In these cases connection delegation, where the client reconnects to the new host, may be a better approach [39]. This protocol requires support from the client and would be more suitable to implement at the Actor level.

4.5.2 Databases

SWAN's data persistence package needs an interface for executing SQL queries against databases. Access to databases is a common requirement for many programs so `DatabaseActors` were added to the Device package. PYTHON's database packages share a common interface specification [42] which suffers from the same problem as PYTHON sockets: the main objects for executing queries, `Cursors`, are not transferrable between processes. To sidestep this issue a similar solution as for sockets was created.

Database Actors create a connection to the specified database when they are created. The programmer can then use the `get_cursor` method to get a reference to a new `Cursor` for the connection. Just as with socket references, the reference can be passed freely between Actors and has an identical interface to standard PYTHON cursor objects.

It's worth noting at this point why STAGE follows PYTHON's database interface and doesn't provide a novel 'Actor-oriented' interface. The main reasons for this are familiarity and flexibility. Just as STAGE Actor definitions are identical to PYTHON class definitions the equivalence of the two database interfaces makes STAGE's interface easier to adopt. A simpler suggestion maybe an interface comparable to `execute_query` and `get_results` but this is inflexible and error prone: if two Actors use the same `DatabaseActor` and their messages interleave they may get the wrong results.

4.5.3 Minor Changes

Actor Pools

STAGE# added Actor pools to the language allowing the creation and simultaneous messaging of multiple actors of a given type (Section 5.4 of [58]). For SWAN this concept was extended through three functions:

- `all(`*`pool`*`)` — this allows the broadcasting of a message to all Actors in the pool. This is in contrast to a standard pool messaging which is equivalent to a parallel map, taking a list of messages and distributing one to each pool Actor. This is used in SWAN when registering a set of handlers with the Registry.
- `one(`*`pool`*`)` — this selects a single actor from the pool according to a defined selection algorithm. By default the algorithm is a simple round-robin procedure.
- `pool(`*`id_list`*`)` — allows ad-hoc creation of a pool from a set of known actors. This is in addition to the standard pool-creation function `get_pool` which also creates the Actors in the pool. This also enables the merging of two or more pools, possibly of different types but its main use is for creating of a pool of Actors which are not 'createable'.

Callbacks

The `callback` function in STAGE is used to install a handler — a function called when a specific result is ready. This construct has been extended so additional arguments can be passed to the handler function as demonstrated in Listing 39. A programmer is therefore able to use polyadic functions as handlers, rather than only monadic functions, provided that the additional parameters are known at the time the handler in bound.

Listing 39 Callbacks with additional parameters

```
#Calls 'generate_response' when the result of 'read_file' if ready with
#'content_type' as an additional argument.
callback('generate_repsonse', read_file(file_path), content_type)
```

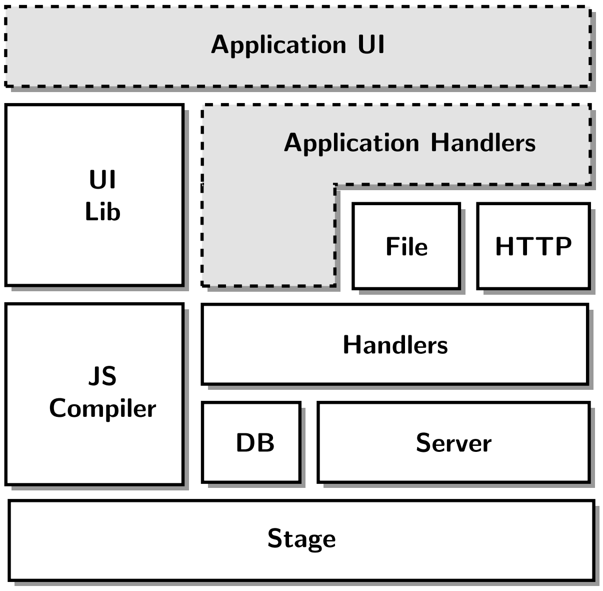

Figure 4.5: **The Major Components of the Swan Framework**:*Elements with a dashed border are provided by the user.*

4.6 Summary

The Swan Framework, outlined in Figure 4.5 on the current page, includes the necessary components to create both the client-side and server-side of highly-interactive Web applications in Stage. The server-side of an application is created using custom Handlers which plug in to the Server and have access to an easy-to-use Database API, the local filesystem as well as external HTTP servers. On the client-side the application creator has the use of a comprehensive UI Library through which they can create dynamic user-interfaces in Stage and cross-compile them to JavaScript to run in the browser.

Chapter 5

Evaluation

This chapter focusses on the evaluation of Swan, considering a number of differenct aspects of the Web framework. For Swan, as for any Web framework, there are two distinct, though connected, groups of interest. Firstly, there are the website developers — the users of the framework itself. Their expectations will be related to the ease of creation of websites using the framework and also satisfying the needs of the websites' end-users, the second group of interest. The expectations of Web users are related to their experience when using a website. If a website is slow, poorly designed and hard to navigate then the site will quickly lose the user's attention. Of course, a framework can only realistically directly influence the performance of the site but can indirectly aid design and navigation through supporting the developers by providing them with well-crafted tools.

This evaluation concentrates of these differing sets of expectations separately. Firstly, we look at Swan's ease-of-use and the complexity of creating a Web application using it. These criteria are intrinsically subjective in nature, attempting to evaluate them in a quantitative manner would unnecessarily force objectivity on them and produce a set of ultimately meaningless results. Instead we focus on a number of example applications and their relative implementation in other frameworks. Secondly, we investigate the performance of SWS, the Swan Web Server, and how quickly it responds to client requests in under varying conditions.

5.1 Framework

This section concerns a subjective evaluation of Swan's simplicity and clarity. This evaluation is carried out through a comparison two example Web applications implemented in Swan and other Web frameworks. A direct comparison between Swan and one other framework is not always possible because of fundamental differences in feature sets. For example, Django uses a template system to create static pages, where as Swan creates dynamic pages through cross-compilation to JavaScript.

The examples compared in this evaluation are intended to highlight Swan's salient features and compare them with those of other frameworks. We consider a simple blogging platform which demonstrates Swan application architecture in relation to the MVC pattern. Additionally we discuss a simple chat service, illustrating how server-side push effects can be achieved.

5.1.1 A Blogging Platform

This example is an implementation of a simple, single-user blogging platform. The user can create posts on which visitors can post comments with all data being stored in a relational database. We will consider this system from back to front, starting with the definition of data models, working through the data access layer to the font-end. Additionally we also consider automatically announcing a new blog post through the Twitter information network.

Listing 40 SWAN Model Definition for a Blog

```
class Post(Model):
    title = TextField(256)
    body = TextField(65536)
    timestamp = TimeField()

class Comment(Model):
    post_id = ForeignKey('Post', "comments")
    name = TextField(16)
    comment = TextField(65536)
    timestamp = TimeField()
```

Listing 41 LIFT Model Definition for a Blog

```
class Post extends LongKeyedMapper[Post] with IdPK{
    def getSingleton = Post
    object title extends MappedPoliteString(this, 256)
    object body extends MappedTextArea(this, 65536)
    object timestamp extends MappedDateTime(this)
}

object Post extends Post with LongKeyedMetaMapper[Post] {
    override def fieldOrder = List(title, content, timestamp)
}

class Comment extends LongKeyedMapper[Post] with IdPK{
    def getSingleton = Comment
    object post_id extends MappedLongForeignKey(this, Post)
    object name extends MappedPoliteString(this, 16)
    object comment extends MappedTextArea(this, 65536)
    object timestamp extends MappedDateTime(this)
}

object Comment extends Post with LongKeyedMetaMapper[Comment] {
    override def fieldOrder = List(post_id,name,comment,timestamp)
}
```

Listing 42 DJANGO Model Definition for a Blog

```
class Post(models.Model):
    title = models.CharField(max_length=256)
    content = models.TextField()
    timestamp = models.DateTimeField()

class Comment(models.Model):
    post_id = models.ForeignKey(Post)
    name = models.CharField(max_length=16)
    comment = models.TextField()
    timestamp = models.DateTimeField()
```

Server-side

This section discusses the Model and server-side portion of the Controller.

Listings 40, 41 and 42 on the facing page show the Model definition for the implementation of this system in SWAN, LIFT and DJANGO respectively. From this simple example, we can see that:

- Defining the Model for an application is similar in all three frameworks considered.
- The handling of raw database queries is abstracted so the user simply gives the table name and field names and types. Both DJANGO and LIFT validate the Models to check for integrity with respect to field types and relations - these types of checks are not performed by SWAN.
- DJANGO and SWAN are much more succinct than LIFT and use a much clearer syntax inherited from PYTHON

The approach is also very similar to the form of a Model definition in a strict MVC desktop application — there is no mention of how the data can be manipulated or displayed.

Moving forward to the server-side Controller component, we can compare the implementations for SWAN and DJANGO displayed in Listing 43 and Listing 44 on the next page. Here we can see some fundamental differences between the two frameworks:

Binding Patterns — SWAN locates the Handler's binding patterns with the Handler rather than in a separate file as DJANGO does. This practice co-locates this closely related information rather forcing the user to look between files to check patterns — something I found myself doing regularly during development of the Django implementation. SWAN's binding patterns are much more user-friendly than the DJANGO equivalents since much of the unnecessary syntax related to regular expressions is hidden from the user.

Handler Method Grouping — DJANGO's handling methods are discrete functions that are grouped only at the module level. In SWAN, users can collect related request handling methods into cohesive Handlers that manage one aspect of the application.

Request Methods — DJANGO leaves the differentiation of request HTTP method to the user through control structures. SWAN performs the sorting on the user's behalf, forwarding `POST` requests to `post_*` Handler methods and `GET` requests to `get_*` Handler methods. Thus Handler methods need only focus on one request type, increasing the clarity of the code.

Responses — In DJANGO request handling methods must result in the HTTP response that needs to be sent back to the user. It follows that any extra processing that is necessary for that request must be performed, or explicitly dispatched to a secondary thread or process, *before* the response is sent. There is no such restriction in SWAN and furthermore, due to SWAN's basis on STAGE dispatching additional work is no more than a method call away.

We can also see from this example how straightforward it is to create a REST interface for the application data: the binding patterns defined also specify URLs for accessing the application data via any device capable of making HTTP requests. This opens up the possibility of further applications consuming and modifying the data.

Client-side

Simple implementations for the client-side of the Blog application in both SWAN and GWT are given in Appendix B since they are both fairly sizeable, as is common with user interface code.

Listing 43 SWAN Handlers for a Blog

```
class RootHandler(FileHandler):
    root = "/files/"
    bindings = {"default":"/blog/`path`?"}

class Posts(ExternalHandler):
    bindings = {'default':'/posts/?','post' :'/post/`id`',
      'comments':'/post/`id`/comments'}

    def get(self):#get all posts
        response.send(200, Post.all())

    def post(self, body): #adds a new post
        Post.add(title=body['title'],body=body['body'],timestamp=now()).save()
        response.send(204)

    def get_post(self, id): #get a single post
        response.send(200, Post.get(id=equals(id)))

    def get_comments(self, id): #get comments for a post
        response.send(200, Post.get(id=equals(id)).comments)

    def post_comments(self, id, body): #add a comment to a post
        comments = Post.get(id=equals(id))).comments
        comments.add(name=body['name'],comment=body['comment'],timestamp=now()).save()
        response.send(204)
```

Listing 44 DJANGO Handling Methods for a Blog

```
# urls.py
urlpatterns = patterns('',
    (r'^blog/$', 'thesite.blog.views.core'),
    (r'^posts/$', 'thesite.blog.views.posts'),
    (r'^post/(?P<id>.*)/$', 'thesite.blog.view.post'),
    (r'^post/(?P<id>.*)/comments$', 'thesite.blog.view.comment')
)
# view.py
def core(request):
    return HttpResponse(loader.get_template('index.html').render({}))

def posts(request):
    if(request.method == 'GET'):
        posts = serializers.serialize('json',Post.objects.all())
        return HttpResponse(posts)
    if(request.method == 'POST'):
        body = request.BODY
        Post(title=body['title'],body=['body'],timestamp=now()).save()
        return HttpResponse(status=204)

def post(request, id):
    post = serializers.serialize('json',Post.objects.get(id=id))
    return HttpResponse(loader.get_template('post.html').render(post))

def comments(request, id):
    if(request.method == 'GET'):
        posts = Post.objects.get(id=id).comment_set
        data = serializers.serialize('json',posts)
        return HttpResponse(data)
    if(request.method == 'POST'):
        comments = Post.objects.get(id=id).comment_set
        comments.create(name=body['name'],comment=body['comment'],timestamp=now())
        return HttpResponse(status=204)
```

We can see that requests to the server are dealt with in similar ways in both frameworks, using callback functions to handle the response from the server. SWAN's requests to the server are made through a function call which appears as simple Actor message passing and the `callback` mechanism used to handle response from the server will be familiar to any STAGE programmer. SWAN is perhaps slightly more clear since simple functions can be used as callbacks where as in GWT an entirely new class, with success and failure methods, is required.

One huge advantage that GWT has is its use of an explicitly typed language and extremely strong integration with the Eclipse IDE. I found these tools provide extremely good support for the programmer and make development time incredibly fast, even for those who are not familiar with the framework. While SWAN applications are also developed in Eclipse, tool support is not nearly as strong.

GWT's RPC mechanism effectively hides the URLs for the data being requested. This obfuscation creates a much cleaner implementation for the intended application but makes the re-use of the data interface much less obvious compared to SWAN's binding patterns.

Announcing Posts on Twitter

The blog user may want to send an announcement to the Twitter information network when a new post is created, to inform their followers of the new article. Listing 45 shows that SWAN has external data sources as a core feature: we simply add the following lines the end of the `post` method in Listing 43:

Listing 45 Announcing on a post on Twitter

```
    url = '1/statuses/update.json'
    status = {status:"New post: " + body['title']}
    headers = {authorization:"Basic " + "user:password".encode("base64")}
    self.post('api.twitter.com', url, status, headers)
```

The equivalent of this in DJANGO would require explicit inclusion of PYTHON's `httplib`, the creation of an HTTP connection and making of request. This would also have to be done before the response is returned to the client, which could be delayed considerably.

5.1.2 Chat Service

This section discusses an implementation of the server-side of a basic chat service. In this case it is essential that the server supports COMET, i.e. that it is able to 'push' messages out to clients, since regular polling for new messages by the client would lead to stilted conversations.

Listing 46 on the following page demonstrates how this effect can be produced on the server using a central actor as a hub for all messages. When a request for messages arrives at a Handler it is forwarded to the central hub. The hub checks to see if there any new messages and if there are returns them to the user otherwise it holds on to the request. When a message is posted it is also forwarded to the central hub, which adds it to the message queue and sends it as a response to any requests that are waiting for a message.

While this implementation is not complex it is not an immediately obvious solution to the problem. The need for a central hub may not be entirely apparent to the developer at first. Furthermore, it is left to the developer to determine if there are any new messages that it needs to send out when a request arrives. This is an aspect that could be supported more directly by the framework since it common to all problems of this type.

5.1.3 Framework Conclusion

This evaluation has shown that SWAN has many of the required features of a Web framework. It allows the user to create applications in a style similar to the familiar MVC pattern with clean definition of an

Listing 46 A Chat Hub

```
class Messages(Handler):
    bindings = { 'default':'/messages/`name`/?`time`?'}

    def birth(self):
        self.hub = find_type('MessageHub')

    def get(self, name, time=None):#retrieve messages
        self.hub.get_messages(response, time)


    def post(self, name, body, time=None):#send a message
        self.hub.put_message(time(),name,body)
        response.send(204)

class MessageHub(MobileActor):
    def birth(self):
        self.msgs = []
        self.responses = []

    def get_messages(self, response, time=None):
        if time and self.msgs:
            msgs = filter(lambda msg:msg['time']>float(time),self.msgs)
            response.send(200,msgs)
        else:
            self.responses.append(response)

    def put_message(self, time, name, message):
        msg = {'time':time,'name':name,'message':message}
        self.msgs.insert(0,msg)
        while self.responses:
            response = self.responses.pop().send(200,[msg])
```

application's Model through the database interface. Handling of client requests is performed by concise, modular Handlers which automatically run in a concurrent environment. Additionally creation of a REST interface for external consumption and manipulation of an application's data is straightforward. We have also seen that SWAN has good support for features such as asynchronous and external HTTP requests to third party servers using regular STAGE language contstructs and can address more advanced concepts such as COMET.

5.2 Performance

This section serves as a quantitative evaluation of SWAN's Web server, SWS. The server that a Website runs on is of vital importance. Different websites can put varying pressures on a server depending on the nature of the application. A basic website built around a number of static pages will have user sessions composed of relatively infrequent requests from page downloads. In contrast, highly interactive Web applications may expect more frequent request involving less data. Either way the server's ability to respond to requests in a timely fashion is of vital importance as it can a have a huge impact on user interaction and perception of the website.

The latency of Web pages, i.e. the time between the user making the request and the response being received and rendered, has been shown to have a critical effect on the usability of a Website as perceived by the user and also how interesting they find it [50]. The consistency of the Website is also very important: it is almost unacceptable for latency to increase throughout a user's session on a Website. It is preferable for latency to decrease for the user's perception of responsiveness to remain constant [14]. Furthermore, even the smallest increase in latency has an the effect of reducing the numbers of visitors returning to a Website [15].

Since SWAN applications run on the custom SWS it is imperative that it is able to perform at a comparable level to other commonly used Web servers. This evaluation compares the performance of SWS against that of:

Apache HTTP Server — The Apache server has been in development since 1995 and is the most popular Web server in use with over 100 million websites supported by it [1, 43]. The DJANGO project recommends that Apache is used to serve DJANGO applications in production settings.

Apache Tomcat — Apache Tomcat, referred to as Tomcat to avoid confusion, is a widely-used implementation of the Java Servlet and JavaServer Pages technologies, providing an HTTP server execution environment for Java [2]. The server-side of GWT applications can be run in any Java Servlet container such as Tomcat.

Testing was performed using `ab`, the Apache Bench utility, a standard tool for gauging the performance of HTTP servers. Apache Bench is simply an HTTP client which sends a specified number of HTTP requests to a given server. Along with many other parameters, the concurrency level of those requests, i.e. the number of requests to perform simultaneously, can be varied. The tests revolved around a simple GET request to the server which resolved to a standard HTML file. Apache Bench was run from a separate client machine connected to the same local network as the server.

5.2.1 Tests

Response Times

Figure 5.1 shows the results of the first test: response times for 5000 requests at concurrency level 1 for SWS, Apache and Tomcat. We can draw the following conclusions:

- SWS takes on average 29 milliseconds to respond to a request.
- SWS is roughly 3 times slower than Apache and Tomcat, which take on average 11 milliseconds and 9 milliseconds to respond to requests, respectively.

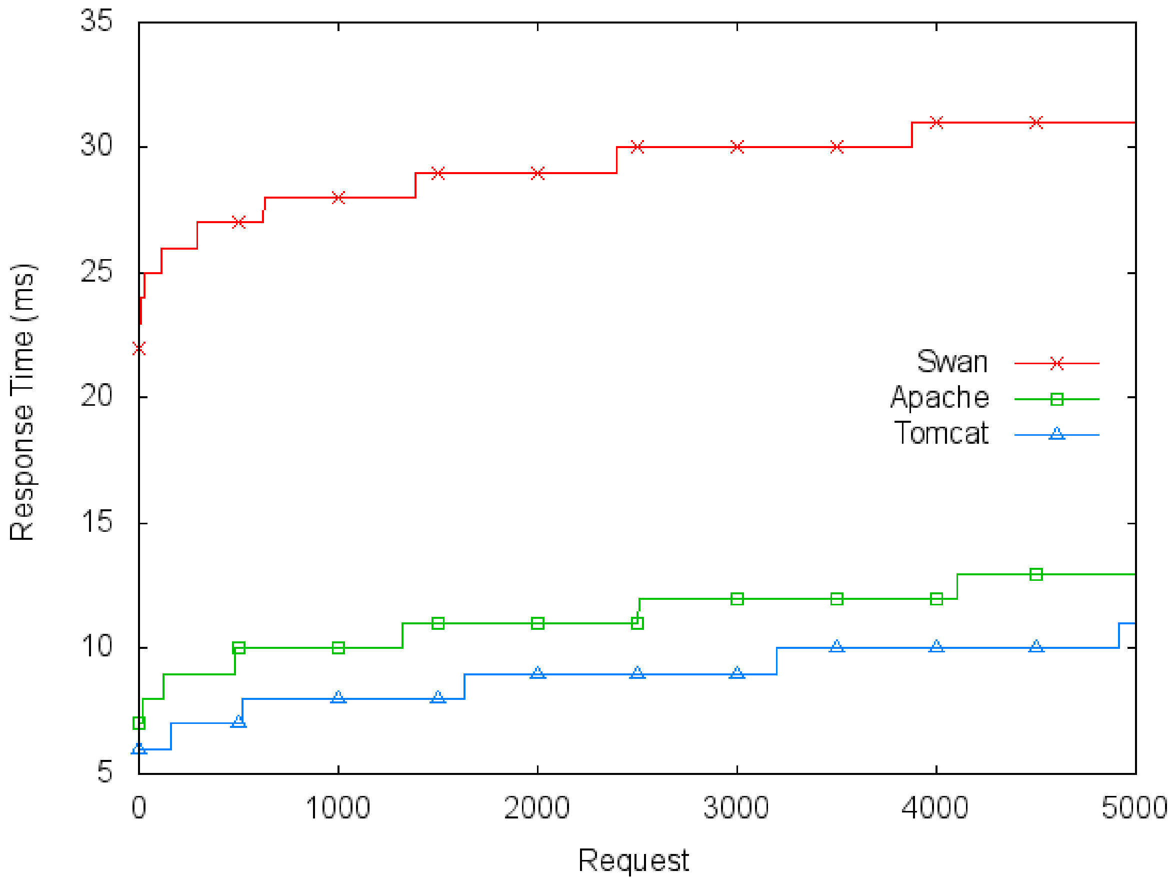

Figure 5.1: **Response Time for 5000 Requests**: *SWS is about 3 times slower than Apache and Tomcat.*

The performance difference can be attributed to at least two factors.

1. The abstraction of sockets required by the Stage runtime results in many inter-process calls to the Manager process. Since Python sockets cannot be moved between processes this issue is unavoidable on multicore systems.

2. Secondly, according to Ayres' measurements [9], message passing in Stage takes about half a millisecond. Thus aspects of the structure of SWS, such as the dynamic look up of Handlers in the Registry as described in Section 4.3, are extremely costly relative to the performance of Apache and Tomcat.

While SWS is distinctly slower than the other Web servers it performs consistently and responds in a time that is fast enough to avoid any truly adverse effects on application usability.

Concurrency

The second set of tests run investigate various aspects of the servers whilst handling 1000 requests at varying concurrency levels, the number of simultaneous requests made to server. These tests show the servers in a more realistic environment where more than one user is interacting with an application at once. The concurrency levels ranged from 0 to 250. Data for these tests can be found in Appendix A.

Figure 5.2 plots the mean time to respond to a request for each concurrency level. We observe the following from the chart:

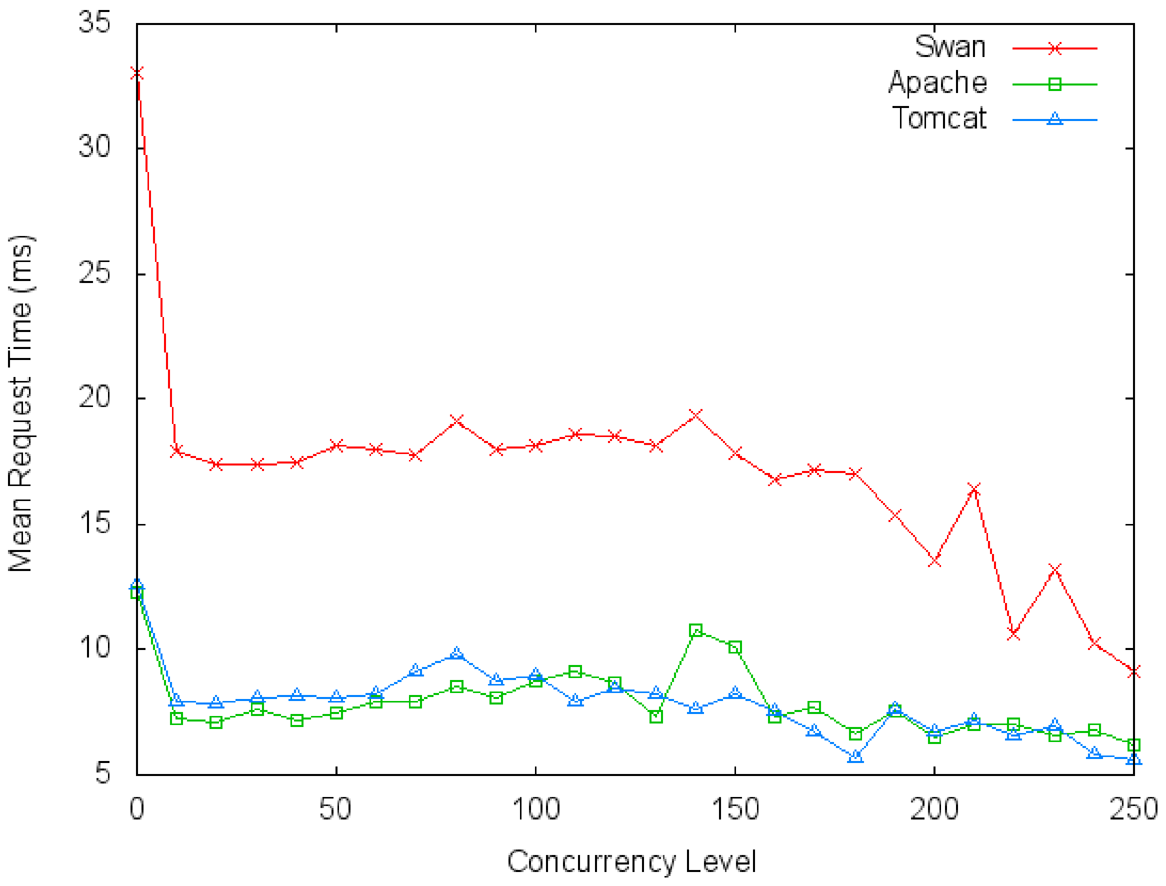

Figure 5.2: **Mean Request Time (ms)**

- At concurrency levels between 10 and 150, SWS takes about 18 milliseconds to respond, producing a rate of 55 requests handled per second.
- SWS takes, on average, a over twice as long to respond to requests compared to Apache and Tomcat.
- At concurrency levels greater than 150, mean request time falls sharply for SWS, as well as dropping for Apache and Tomcat.

The performance of SWS under varying levels, while not competitive with Apache or Tomcat, is at least consistent. The mean request time for SWS stays well within acceptable boundaries of responsiveness necessary for interactive web applications. The mean request time drops beyond 150 simultaneous requests for the reason illustrated in Figure 5.3 on the next page: for the higher concurrency levels, those over 150 an increasing proportion of requests fail. Indeed for one test run on Tomcat, 948 of the 1000 requests failed. Because these requests failed to connect they fail more quickly, resulting in a lower average request time being recorded.

5.2.2 Performance Conclusion

SWAN's transitive reliance on PYTHON through STAGE means that SWS will never be as fast as the C-based Apache or JAVA-based Tomcat. Having said that, SWS doesn't not perform too poorly to support the requirements of Web applications. SWS is also able to cope with a large number of simultaneous requests. Both Ayres and Zetter note STAGE's problems with PYTHON's Global Interpreter Lock which affects concurrency in PYTHON and those problems will heavily effect a system such as a Web server [58, 9].

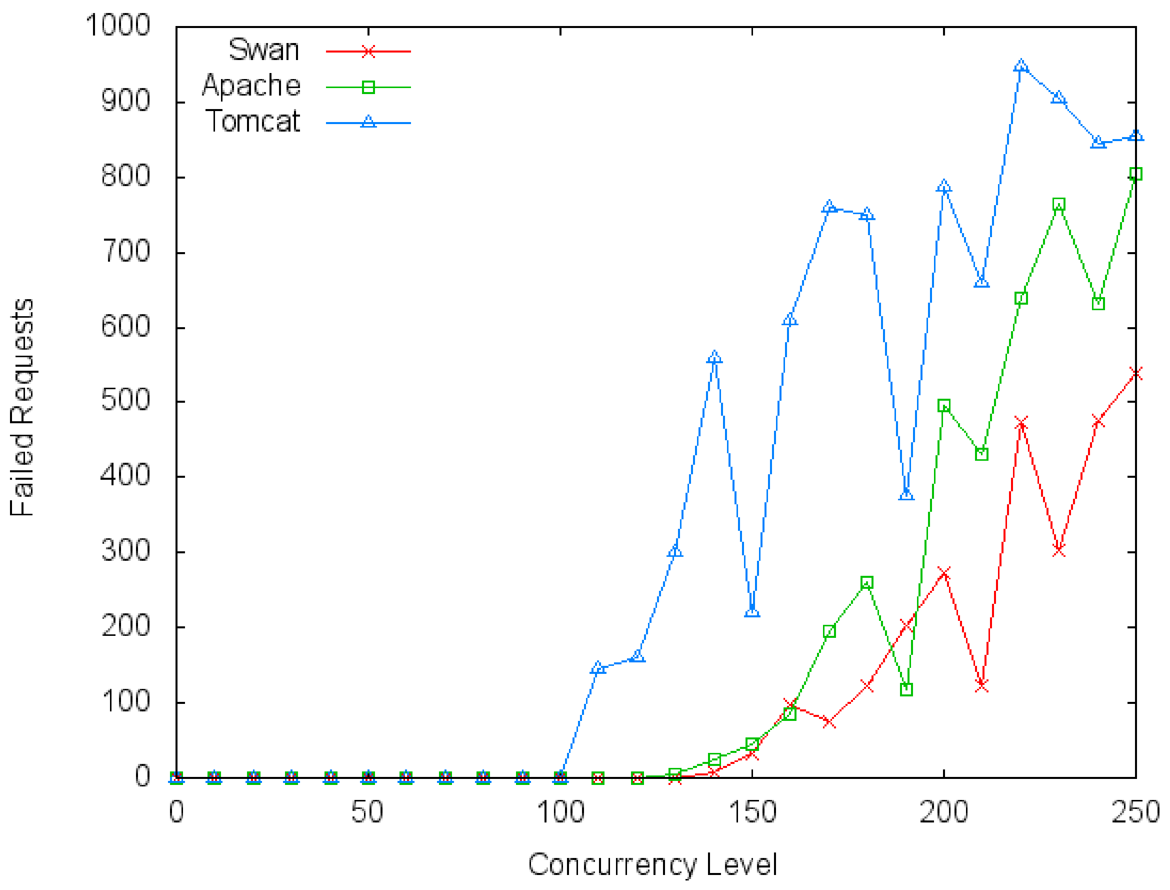

Figure 5.3: **Failed Requests**

Chapter 6

Conclusions & Further Work

This chapter discusses some insights I have gleaned into the architecture of Web applications and the requirements of Web frameworks as a result of this project.

We have shown that using a pure Actor-based language such as Stage to create desktop-style Model-View-Controller applications for the Web is a viable method, despite fundamental differences between the underlying architecture of Web and desktop environments. In fact, the request/response model of the Web fits well with the message-passing convention used by Actor languages.

I believe the use of Stage as the basis for Swan was a valid selection. Stage is a language with concurrency and distribution as its core features and since applications on the Web are by nature both concurrent and distributed Swan is able to talk full advantage of these characteristics. On the server-side Swan is able to take full advantage of the safe, concurrent execution environment to parallelise simultaneous handling of requests from the client with no onus on the application programmer. Regular Stage language features are also used on the client-side to handle asynchronous requests to the server. This permits the creation of highly-interactive Web applications with little interruption when communicating with the server, thus interaction is more comparable to a desktop application rather than a traditional website. Despite these advantages, Swan's Web sever suffers in terms of performance due to the relative slowness of Stage's execution environment. This handicap is slight though in terms of the requirements of Web applications.

Swan's adoption of the MVC pattern for application structure creates a logical separation of concerns, even over the client-server divide of the Web. Database Model definition and manipulation in Swan is controlled through an intuitive interface which lets the programmer avoid writing database queries as desired. The programmer also has simple access to the filesystem and data on external servers. Swan's Handlers provide a concise, modular approach to handling requests from the client. Through the use of hidden proxies, the application developer is able to write in a standard message-passing style, leaving the framework to manage the client-server communication.

The role of Web frameworks is to reduce the conceptual overhead associated with creating Web applications, streamline application creation and perform common tasks on behalf of the user. The fundamental question with Swan is clear: "Would a Web application developer actually use Swan?" I believe Swan simplifies Web application development through the decision to base it entirely on Stage avoiding the implicit need for knowledge of an additional template language or other technologies such as HTML and JavaScript. It uses concepts that will be familiar to desktop application developers and in particular Stage developers, rather than requiring the developer to learn how to use mechanisms such as the `XMLHttpRequest` object. Additionally, Swan also enables the developer to create a REST interface for their application opening up further possibilities for use of the application's data.

An intrinsic difficultly with Web applications is coping with the separation between the client and server. Google Web Toolkit copes with this problem admirably though in a rather verbose manner (detailed in

Section 3.3.4). I found this separation conceptually difficult while considering it as a client-server divide. Once I started thinking of it as simply two separate modules of a regular application it became easier to manage.

6.1 Further Work

This section discusses a number of the many aspects of SWAN which are suitable for further investigation or could be expanded on.

SWAN's server uses pools of Handlers to manage responding to client requests. The size of the pools does not change beyond the initial configuration at the launch of the server. The pools should scale to match the load on the server to optimise resource usage. This would free up system resources at times of low usage and improve the handling of requests at peak times. Pools for a specific type of Handler could scale independently of others if that Handler type is subject to a larger number of requests. The configuration of Handler pools could also be performed manually through a special SWAN application which would also provide monitoring of the server.

The execution of STAGE in the browser is currently fundamentally different execution inside a standard Theatre. Investigation into a full JavaScript Theatre for STAGE would definitely benefit SWAN. This would allow migration of Actors between client and server in the background using standard HTTP requests permitting SWAN applications to take advantage of even more STAGE language features. Such an implementation could be created using the Web Worker API, which is currently in draft state, through which each Actor could be run as a separate Worker [57].

Many Web frameworks automatically include resources for handling users and user authentication for a Web application. This is a feature which SWAN lacks entirely. I believe this would have to be implemented quite carefully to maintain the RESTful nature of SWAN's Handlers. It may be worth investigating Open Authentication[1] for this feature.

The rapidly progressing HTML5 specification includes a number of features, including client-side databases, which may be of interest for SWAN applications [37]. Apple are currently exhibiting a showcase of HTML5 applications including an offline Calendar and a Checkers Game[2]. This could result in a much more complete user interface library for SWAN application developers to use.

STAGE currently uses PYTHON 2.6 and there maybe some performance gains from moving the implementation to PYTHON 3.2 when it is released. 3.2 while it will still feature a Global Interpreter Lock, a new implementation is provided which no longer creates performance loss for using threads, though there is no gain either [11]. Ideally the total removal of the GIL would be ideal — removing STAGE's need for a PYTHON process per CPU core and thus the need to send data between processes. Additionally problems encountered with non-serialisable types would be mediated. Unfortunately, the Unladen Swallow project, which was planning to provide a PYTHON runtime without a GIL, no longer intends to do so [5].

6.2 Closing Remarks

Using SWAN developers can create immersive Web applications without the need for knowledge of a multitude of different languages and technologies. SWAN provides intuitive methods for performing some of the more intricate parts of application development, reusing core features of its underlying language, STAGE, to promote creation of Web applications in a common MVC pattern.

[1]http://www.openauthentication.org/
[2]http://www.apple.com/html5/

Appendix A

Web Server Benchmarking

This appendix details the results from the server benchmarking tests. Test were performed using Apache Bench 2.3. The server for each test was run on the following hardware:

Operating System	Mac OS X 10.6.3
Processor	Intel$^{®}$ Core i7 2.8GHz
Memory	4GB 1067MHz DDR3

Software Versions:

Apache HTTP Server	2.2.14
Apache Tomcat	6.0.26

Test Command: `ab -n 1000 -c` *c* `-r http://<server>/test`
where *c* defines the concurrency level.

A.1 Mean Response Time

The following table shows the mean response time for each of Apache, Tomcat and Swan for increasing concurrency levels.

Concurrency	Apache	Tomcat	Swan
1	12.317	12.602	33.075
5	8.288	8.759	17.474
10	7.289	7.931	17.904
20	7.137	7.844	17.430
30	7.655	8.090	17.421
40	7.150	8.149	17.461
50	7.480	8.063	18.190
60	7.968	8.227	18.010
70	7.932	9.114	17.802
80	8.539	9.844	19.106
90	8.089	8.750	17.974
100	8.785	8.956	18.161
110	9.168	7.917	18.591
120	8.686	8.459	18.504
130	7.308	8.259	18.170
140	10.787	7.605	19.329
150	10.099	8.240	17.836
160	7.327	7.550	16.787
170	7.724	6.711	17.166
180	6.656	5.653	17.001
190	7.592	7.641	15.359
200	6.530	6.703	13.575
210	7.039	7.152	16.401
220	7.055	6.577	10.614
230	6.552	6.972	13.159
240	6.804	5.859	10.263
250	6.167	5.622	9.136

A.2 Failed Requests

The following table shows the mean response time for each of Apache, Tomcat and Swan for increasing concurrency levels.

Concurrency	Apache	Tomcat	Swan	Concurrency	Apache	Tomcat	Swan
1	0	0	0	130	2	150	0
5	0	0	0	140	12	280	4
10	0	0	0	150	22	110	16
20	0	0	0	160	42	304	49
30	0	0	0	170	98	380	37
40	0	0	0	180	130	375	61
50	0	0	0	190	59	188	101
60	0	0	0	200	248	393	137
70	0	0	0	210	216	330	62
80	0	0	0	220	320	474	237
90	0	0	0	230	382	452	152
100	0	0	0	240	316	422	238
110	0	73	0	250	402	427	269
120	0	80	0				

Appendix B

Blog View Listings

Listing 47 SWAN View for a Blog

```python
def failed(self, status):
    print status

class PostsList(List):
    def __init__(self, control):
        super(PostsList, self).__init__()
        callback(Posts().get(), self.displayPosts, failed)

    def displayPosts(self,ps):
        for p in ps:
            self.addItem(Post(p))

class Post(P):
    def __init__(self, post):
        super(Post, self).__init__()
        self.post = post
        self.showing = False;
        self.title = P(post.title).on_click(self.show)
        self.content = P(post.content).setVisible(False)
        self.timestamp = P(post.timestamp).setVisible(False)
        self.add(self.title,self.content,self.timestamp)

    def show(self):
        self.showing = not self.showing
        self.content.setVisible(self.showing)
        self.timestamp.setVisible(self.showing)
        if self.comments:
            self.comments.setVisible(self.showing)
        else:
            self.comments = CommentsList(self.post.id)

class CommentsList(List):
    def __init__(self, id):
        super(CommentsList, self).__init__()
        callback(Posts().get_comments(id), self.displayComments, failed)

    def displayComments(self,cs):
        for c in cs:
            self.add(P(c.name),P(c.comment),P(c.timestamp))

def launch():
    body.add(PostsList())
```

Listing 48 GWT UI for a Blog

```java
public class Blog implements EntryPoint {

    private final BlogServiceAsync blogService = GWT.create(BlogService.class);

    public void onModuleLoad() {
        final VerticalPanel container = new VerticalPanel();
        RootPanel.get().add(container);
        blogService.getPosts(
                new AsyncCallback<List<Post>>() {
                    public void onFailure(Throwable caught) {
                        caught.printStackTrace();
                    }
                    public void onSuccess(List<Post> result) {
                        for(Post post : result){
                            container.add( new PostWidget(post) );
                        }
                    }
                } );
    }

    private class PostWidget extends VerticalPanel{

        private Label title;
        private Label content;
        private Label timestamp;
        private VerticalPanel comments;

        public PostWidget(final Post post){
            timestamp = new Label(""+post.getTimestamp());
            content = new Label(post.getContent());
            title = new Label(post.getTitle());
            content.setVisible( false );
            timestamp.setVisible( false );
            this.add( title );
            this.add( timestamp );
            this.add( content );
            title.addClickHandler( new ClickHandler() {
                public void onClick(ClickEvent event) {
                    content.setVisible( !content.isVisible() );
                    timestamp.setVisible( !timestamp.isVisible() );
                    if(comments == null){
                        comments = new VerticalPanel();
                        add(comments);
                        loadComments(post.getId());
                        comments.setVisible( false );
                    }
                    comments.setVisible( !comments.isVisible() );
                }
            });
            this.add(title);
        }

        private void loadComments(int id) {
            blogService.getComments(id,
                    new AsyncCallback<List<Comment>>() {
                        public void onFailure(Throwable caught) {
                            caught.printStackTrace();
                        }
                        public void onSuccess(List<Comment> result) {
                            for(Comment comment : result){
                                add( new CommentWidget(comment) );
                            }
                        }
                    } );
        }
    }

    public class CommentWidget extends FlowPanel {

        public CommentWidget(Comment comment) {
            this.add( new Label(comment.getName()));
            this.add( new Label(comment.getComment()) );
            this.add( new Label(""+comment.getTimestamp()));
        }

    }
}
```

Bibliography

[1] Apache http server project. http://httpd.apache.org/ABOUT_APACHE.html Accessed 14/06/2010 Accessed 14/06/2010. 53

[2] Apache tomcat. http://tomcat.apache.org/ Accessed 14/06/2010. 53

[3] Django at a glance. http://docs.djangoproject.com/en/dev/intro/overview/ Accessed 14/06/2010. 20

[4] Pyjamas framework. http://pyjs.org/ Accessed 14/06/2010. 1, 26

[5] Unladen swallow project plan. http://code.google.com/p/unladen-swallow/wiki/ProjectPlan Accessed 14/06/2010. 58

[6] Camping, a microframework. http://camping.rubyforge.org/files/README.html Accessed 14/06/2010, October 2006. 29

[7] Apache mpm worker. http://httpd.apache.org/docs/2.0/mod/worker.html. The Apache Software Foundation Accessed 14/06/2010, June 2010. 32

[8] Gul A Agha. *Actors: A Model Of Concurrent Computation In Distributed Systems*. PhD thesis, 1985. 16

[9] John Ayres. Implementing stage: the actor based language. Master's thesis, Imperial College London, June 2007. 2, 15, 41, 54, 55

[10] J.W. Ayres and Susan Eisenbach. Stage: Python with actors. In *International Workshop on Multicore Software Engineering (IWMSE)*, April 2009. 2, 15

[11] David M. Beazley. Inside the new gil. http://www.dabeaz.com/python/NewGIL.pdf Accessed 14/06/2010, January 2010. 58

[12] T. Berners-Lee. Cool uris don't change. http://www.w3.org/Provider/Style/URI Accessed 14/06/2010. 11

[13] Tim Berners-Lee and Robert Cailliau. Worldwideweb: Proposal for a hypertext project. http://www.w3.org/Proposal.html Accessed 14/06/2010, November 1990. 1

[14] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. *Computer Networks*, 33(1-6):1 – 16, 2000. 53

[15] Jake Brutlag. Speed matters for google web search, June 2009. 53

[16] Steve Burbeck. Applications programming in smalltalk-80(tm): Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). 1992. 3

[17] Namit Chadha. Air 2 (actor inspired ruby 2) and a minimal actor based web framework. Master's thesis, Imperial College London, June 2009. 28

[18] Derek Chen-Becker, Marius Danciu, and Tyler Weir. *The Definitive Guide to Lift: A Scala-based Web Framework*. Apress, 1st edition, May 2009. 21, 22

[19] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971. 16

[20] Robert Cooper and Charles Collins. *GWT in Practice.* Manning Publications, 1st edition, 2008. 25, 26

[21] M. Elkstien. Learn rest: A tutorial. `http://rest.elkstein.org/` Accessed 14/06/2010, February 2008. 9

[22] David Pollak et al. Lift webframework. `http://liftweb.net/` Accessed 14/06/2010. 1, 21

[23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. `http://www.w3.org/Protocols/rfc2616/rfc2616.html` Accessed 14/06/2010, June 1999. 9, 12

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999. 33

[25] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002. 8, 9

[26] Django Software Foundation. Django template language. `http://docs.djangoproject.com/en/dev/topics/templates/` Accessed 14/06/2010. 1

[27] Django Software Foundation. The django template language: For python programmers. `http://docs.djangoproject.com/en/dev/ref/templates/api/` Accessed 14/06/2010. 14

[28] Michael Galpin. Build comet applications using scala, lift, and jquery. IBM developerWorks Accessed 14/06/2010, March 2009. 22

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994. 3

[30] Jesse James Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php` Accessed 14/06/2010, February 2005. 4, 7

[31] Debasish Ghosh and Steve Vinoski. Scala and lift functional recipes for the web. *IEEE Internet Computing*, 13(3):88–92, 2009. 21

[32] Google. Google web toolkit. `http://code.google.com/webtoolkit/` Accessed 14/06/2010. 1, 22, 25, 26

[33] G Goth. Critics say web services need a rest. *IEEE distributed systems online*, 5(12), 2004. 14

[34] Robert Hanson and Adam Tacy. *GWT In Action: Easy Ajax with the Google Web Toolkit.* Manning Publications, Manning Publications Co. Sound View Court 3B Greenwich, CT 06830, 2007. 15, 24

[35] Cal Henderson. Why i hate django. Video `http://www.youtube.com/watch?v=i6Fr65PFqfk` Accessed 14/06/2010, September 2008. 20

[36] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 2, 15

[37] Ian Hickson. Html5 a vocabulary and associated apis for html and xhtml. editor's draft. `http://dev.w3.org/html5/spec/Overview.html` Accessed 14/06/2010, June 2010. 58

[38] Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right.* Springer, 2nd edition, 2009. 19

[39] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag. 44

[40] Anthony T Holdener III. *Ajax: The Definitive Guide*. O'Reilly Media, 1st edition, January 2008. 7

[41] Luke Kenneth Casson Leighton. Pyjamas book. `http://pyjs.org/book/output/Bookreader.html` Accessed 14/06/2010, 2009. 26

[42] Marc-André Lemburg. Python database api specification v2.0. `http://www.python.org/dev/peps/pep-0249/` Accessed 14/06/2010. 44

[43] Netcraft. February 2009 web server survey. `http://news.netcraft.com/archives/2009/02/18/february_2009_web_server_survey.html` Accessed 14/06/2010, February 2009. 53

[44] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 1st edition, 2008. 21

[45] Jeff Offutt. Quality attributes of web software applications. *IEEE Software*, 19:25–32, 2002. 3

[46] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM. 14

[47] Jeremy Petersen. Benefits of using the n-tiered approach for web applications. `http://www.adobe.com/devnet/coldfusion/articles/ntier.html` Accessed 14/06/2010. 3

[48] Arno Puder. Xml11 documentation. `http://www.xml11.org/documentation/` Accessed 14/06/2010. 28

[49] Arno Puder. A cross-language framework for developing ajax applications. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 105–112, New York, NY, USA, 2007. ACM. 1, 6, 27

[50] Judith Ramsay, Alessandro Barbesi, and Jenny Preece. A psychological investigation of long retrieval times on the world wide web. *Interacting with Computers*, 10(1):77 – 86, 1998. HCI and Information Retrieval. 53

[51] Leonard Richardson and Sam Ruby. *RESTful web services*, chapter 3, pages 52–53. O'Reilly Media, 1st edition, 2007. 9

[52] Alex Russell. Comet: Low latency data for browsers. `http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/` Accessed 14/06/2010, March 2006. 6, 7

[53] Stan Schwarz. Web servers, earthquakes, and the slashdot effect. `http://pasadena.wr.usgs.gov/office/stans/slashdot.html` Accessed 14/06/2010, August 2000. 18

[54] James Snell. Resource-oriented vs. activity-oriented web services. `http://www.ibm.com/developerworks/webservices/library/ws-restvsoap/` Accessed 14/06/2010, 2004 October. 10, 14

[55] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005. 15

[56] W3C. Uris, addressability, and the use of http get and post. `http://www.w3.org/2001/tag/doc/whenToUseGet.html` Accessed 14/06/2010, March 2004. 13

[57] WHATWG. Web workers draft recommendation. `http://www.whatwg.org/specs/web-workers/current-work/` Accessed 14/06/2010. 58

[58] Chris Zetter. Stage: A truly distributed scalable language. Master's thesis, Imperial College London, June 2009. 2, 44, 55

Colophon

This report was compiled using pdfTex 3.1415926 with the following packages:

`fullpage`	`color`
`varioref`	`hyperref`
`hypcap`	`todonotes`
`verbatim`	`paralist`
`textcomp`	`tocloft`
`subfig`	`rotating`
`tikz`	`fancyhdr`
`xspace`	`listings`
`caption`	