Verily: A Web Framework for Creating More Reasonable Web Applications

John L. Singleton
University of Central Florida, Orlando, USA
jsinglet@acm.org

Gary T. Leavens
University of Central Florida, Orlando, USA
leavens@eecs.ucf.edu

ABSTRACT

The complexity of web application construction is increasing at an astounding rate. Developing for the web typically crosses multiple application tiers in a variety of languages, which can result in disjoint code bases. This lack of standardization introduces new challenges for reasoning.

In this paper we introduce Verily, a new web framework for Java that supports the development of verifiable web applications. Rather than requiring that programs be verified in separate a posteriori analysis, Verily supports construction via a series of Recipes, which are properties of an application that are enforced at compile time. In addition to introducing the Verily framework, we also present two Recipes: the Core Recipe, an application architecture for web applications designed to replace traditional server-side Model View Controller, and the Global Mutable State Recipe, which enables developers to use sessions within their applications without resorting to the use of unrestricted global mutable state. Demo Video: http://www.youtube.com/watch?v=TjRF7E4um3c

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—Frameworks; D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures

General Terms
Verification, Languages

Keywords
Model View Controller, Method Router Response, web frameworks, Verily framework

1. INTRODUCTION

Because of the way that they encapsulate domain-specific complexity, software frameworks have become indispensable [1, 15, 17]. This applies especially to web applications, where the general focus of framework design has centered on issues of productivity. While this has had favorable effects on productivity [14], little has been done about verification1 within web application frameworks.

However, the lack of focus on verification within web applications is not surprising. Fundamentally, web applications differ from traditional applications in several ways. Unlike traditional applications, the design of HTTP ensures that portions of the application may be accessed essentially at random. Thus, to encode the logic of having a definite start state and transitions, a programmer must manually encode the logic to thread application state throughout a sequence of URLs.

By contrast, in a typical graphical desktop application, a user’s actions may be limited simply by limiting the options visible. In a web application, no such restrictions may be made, as URLs are always accessible (e.g., using a browser’s “back” button). This introduces a complication that differentiates web applications from the traditional view of applications as finite automata [2, 5, 6].

An additional complication is that behaviors vary widely between frameworks. For example, many frameworks implement a form of Model View Controller (MVC) as the primary design pattern for application construction. But frameworks vary widely in their interpretation of MVC; this makes it difficult to create a general method for reasoning about web application code.

Verily diverges from traditional frameworks by providing more than a skeleton of an application. Instead, Verily supports verification by providing a set of Recipes that specify desirable properties. Implementations are statically checked for compliance with these recipes. Compliance is enforced before code is allowed to run.

2. THE CORE RECIPE

The MVC pattern, originally designed for the Smalltalk-80 system [3], is an ineffective pattern for designing web applications. While the MVC pattern has been successfully employed in desktop applications, web-based applications follow a different execution cycle. Most notably, web-based applications rely on a request-response model that does not easily translate to the typical desktop application’s finite automaton model. In a web application, the normal calling semantics of a program’s execution are replaced by a mapping of intents or method calls to URLs that are sent to the server in the form of plain text.

The foundation of our approach is the introduction of a new design we call Method Router Response (MRR). MRR’s design has three kinds of components: application requests are processed by Methods, control flow is directed by Routers, and the resulting content (e.g., HTML or JSON) is displayed using Responses. The high level design of MRR is shown in Figure 1.

1In this paper we use the term verification to mean any of several different reasoning activities and techniques designed to increase reliability, such as testing, formal verification, and static analysis.
The following points contrast the MVC approach with our MRR approach.

No assumptions about cross-tier calls. One assumption present in MVC is that there is little (if any) penalty for method calls that cross tiers. For example, a view can require many back-and-forth calls between the controllers and models in order to perform a given display operation. (In many cases, this allows for a higher degree of modularization. Indeed, this is a major strength of the MVC model.) However, while this modularity is desirable, it is often impractical in the client-server model due to non-locality and the time penalty for crossing from the web tier to the server tier. In MRR calls that cross tiers are regarded as inherently more costly in terms of application performance and therefore should be minimized.

Responses are not a manifestation of a model. In MVC, the view is presented as a manifestation of the state of a model [3, 7, 9]. In MRR, views are dependent on the evaluation of parameterized methods. The underlying data that the views represent need not be constructed from a single model.

Explicitly recognizes protocol. Since web applications are delivered over HTTP, a great deal of work must be done in transforming HTTP requests, which are essentially strings of encoded parameters, into meaningful program representations. MRR provides explicit support for transforming requests into program level objects and for mapping requests onto executable code.

The following subsections detail the building blocks of the MRR design.

2.1 Methods

Methods are the individual functions that are responsible for implementing the web-facing logic of an application. In Verily Methods are simple function members of a class with the type signature public static final void(...) (see, for example, Listing 1).

```java
public class UserManagement {
    public static final void addUser(String email, String name){
        // ...
        return new TemplateHTMLContent(...);
    }
}
```

Listing 1: Example Method addUser.

Since web applications are delivered over HTTP, all data is passed to the server in plain text query strings. Thus a great deal of work must be done in transforming HTTP requests (strings) into program data. Methods define explicit guarantees for transforming requests into data and for mapping requests onto executable code. These statically-checked guarantees are as follows:

Requests are Well-Formed The Methods component of the Core Recipe guarantees that function parameters are passed with the right type and arity. The idea of ensuring well-formed requests bears resemblance to the view taken by the Google Web Toolkit, but does not require client side applications to be written in JavaScript in order to take advantage of it.

URI/Class Isomorphism Methods are mapped onto the URI structure in a way that is isomorphic to the placement of Methods in classes. For example, if a Method createUser belongs to the class UserManagement, then the URI that maps to the UserManagement.createUser Method would be: http://myapp/UserManagement/createUser.

Router Parity Verily checks that each Method has a matching Router method defined.

To signal that a particular class should be scanned for Methods, a user must put the class’s source file in the methods subdirectory of their Verily project. (Classes may also contain other members, but they will not be accessible to incoming requests.)

2.2 Routers

While Methods are responsible for handling the computational logic of the application, Routers are responsible for an application’s control flow. As indicated in Figure 1, Methods and Routers exist in pairs, yet they do not strictly depend on one another. This design has the advantage that navigational logic and application logic are clearly separated and therefore may be tested independently. We show an example Router in Listing 2.

```java
public class UserManagement {
    public static final Content addUser(String email, String name){
        // ...
        return new TemplateHTMLContent(...);
    }
}
```

Listing 2: Example Router addUser.

Similar to Methods, Routers have some conventions that must be followed and statically-checked guarantees, which are as follows:

Method Parity For each Router, there must be a correspondingly named Method with a matching classname and method name.

Request/Call Equivalence The parameters for a Router must match those of the corresponding Method in type and name; however, Routers may declare a return type. A Router’s declared return type describes the content that it will render. This guarantee allows a static analysis that eliminates errors arising from mismatched content types. Note that Routers are never accessed directly; data always first passes through a Method.

No Mutation of Global State Routers may read from but not write to Global Mutable State, as described in Section 3. This ensures that Routers do not require state to be built up to operate and may thus be more easily tested.

To indicate that a class should be scanned for Routers, its source file is placed in the routers directory within a project.
public class UserManagement{
  
  public static final void visitSite(
      WritableValue<Long> lastTs
  ){  
    // update the last timestamp
    lastTs.setValue(System.currentTimeMillis());
  }
}

Listing 3: Using session to track the last time a user visited.

2.3 Responses

Responses, which are types of data returned by Routers, encode how content is rendered by a browser. A Response must subclass the Content class, or use one of several provided subclasses. In Listing 2, the addUser Router is returning an instance of the TemplateHTMLContent class.

The purpose of encoding rendering information in the data returned by Routers is twofold. First, rather than introducing domain-specific concerns like HTTP content-type headers, these concerns are instead abstracted into the different Content classes. Second, since rendering is based on data (the Response), unit tests may easily inspect these data values, instead of creating mock web browsers to interact with the application.

Since much work has been done in providing templating capabilities for HTML pages, Verily uses the Freemarker3 template engine and supports its use with the TemplateHTMLContent class.

3. GLOBAL MUTABLE STATE RECIPE

HTTP is a stateless protocol, but applications often need mutable state. For example, applications often need to associate a sequence of computations with a particular client. This requirement is present in many popular web sites; for example, when a user logs into their banking website, the information presented must be personalized.

Since HTTP is a stateless protocol, web applications must invent a method for maintaining some form of state between requests of a related nature. This conversational flow between client and server is called a session. In a web application, a session behaves exactly like a global variable, since it exists in the global namespace and may be manipulated anywhere throughout the application.

Global variables are an antipattern because their use increases module coupling. Worse, because requests in a traditional web application may essentially be executed in any order, any code paths that do not explicitly check for state modification will allow current and future states of an application to be erroneously or maliciously influenced by a client.

The focus of the Global Mutable State (GMS) Recipe is to limit such deleterious effects of sessions, while providing a means for conversational flow.

To use GMS in Verily, a user must declare the type of a session-bound parameter as an instance of either ReadableValue<T> or WritableValue<T>. These generic classes mark session values that will be either read-only or writable during a session. Listing 3 shows an example of how to use the WritableValue type within a Method.

The code in Listing 3 updates the value of the lastTs session variable. Subsequent requests to the application from a particular user will then be able to access this value in other methods via

```java
public class UserManagement{
  
  public static final void visitSite(
      WritableValue<Long> lastTs
  ){  
    // update the last timestamp
    lastTs.setValue(System.currentTimeMillis());
  }
}

Listing 3: Using session to track the last time a user visited.
```

4. RELATED WORK

Hemel and Groenewegen, et al. identify the problems that creating a web application using multiple languages creates for static analysis and propose a domain specific language for the creation of web applications, WebDSL [10]. However, this approach introduces the problem of having to adopt a new language, which may be problematic if it makes existing third-party libraries unavailable. Verily is implemented in Java to demonstrate the generality of our approach, and the Recipes and ideas in Verily should be portable to other languages.

In their paper on WebJinn [13], Kojarski and Lorenz introduce two programming models, DDD and XP, which address the problems that arise as a result of scattering cross-cutting concerns across code written in multiple languages. They say that MVC does not make a suitable pattern for web applications because it fails to address code scattering problems. Their proposal, like Verily, is a Java-based approach, but it focuses on moving dynamic behavior to the domain of JSP pages. Verily's approach instead enables static analysis, and yet remains agnostic to the display technology on the client side.

Since parameter types cannot be Java primitives, null suffices.
This makes Verily useful for integration with client-side JavaScript applications or “single page” applications.

Within web pages, interface controls are often influenced by the result of server-side operations or data. This introduces complications for unifying analysis across tiers. Jaspan and Aldrich propose a method of specifying constraints on such interactions by using a propositional logic of relationships between components [12].

Our approach is partially inspired in part by Test Driven Development (TDD), in which tests are written before application construction [8]. Just as TDD structures its workflow around the creation of tests, which must be passed before a product is shipped, our approach structures its development around Recipes, which must be satisfied before code is allowed to run.

Though not specifically related to web applications, Stoughton addresses problems in structuring command-oriented functional programs by introducing a pattern called functional Model View Controller [16]. This is of interest because it is similar to Verily’s view of Methods and shares an emphasis on avoiding mutable state, however it does not directly consider the problem of marshaling function parameters over a protocol, as is the case with Verily.

5. DEMONSTRATION PLAN

The objective of our demonstration is to show that Verily improves the reliability of problematic aspects of web application development. We will first explain how to install Verily and then demonstrate the construction of an online auction application. By the end of the demonstration the audience will have built a small but complete application in Verily. Additionally, we will cover the following supplemental topics:

The JavaScript Bridge. Verily can automatically export Methods to JavaScript. This enables client-side applications to call Methods as if they were natively defined in JavaScript. We will show the use of both the synchronous and asynchronous JavaScript Bridge functionality.

Verily in Single Page Applications. Single page applications rely on client-side JavaScript to manage user interaction. The demonstration will extend the sample application to take advantage of the Backbone.js JavaScript framework to demonstrate how Verily solves several problems endemic to connecting client-side JavaScript to server-side applications.

6. CONCLUSIONS AND FUTURE WORK

In this paper we introduced the Verily framework and demonstrated how it solves several design problems endemic to traditional web application applications. The primary focus of the demonstration is to explain two of Verily’s Recipes.

Verily is currently under active development. Our next areas of focus are handling issues of parameter validation as well as random execution defects. We expect to solve these problems with a combination of introducing contracts [11, 4] as well as constraint solving Recipes. Readers interested in using the Verily framework may access it on its development website at: http://goverily.org.

7. REFERENCES


