KIRKE: Re-engineering of Web Applications to Mobile Apps

Rohit Mehra
Accenture Technology Labs, Bangalore, India
rohit.a.mehra@accenture.com

Vinayak Naik Rahul Purandare Kapish Malik
IIIT-Delhi, India
{naik, purandare, kapish15026}@iiitd.ac.in

ABSTRACT

A large number of web applications are written using server-side scripting languages. Although web browsers allow clients to run these applications, it is often cumbersome to depend on desktops for the services provided by the applications. Given the popularity and convenience of mobile devices, there is a clear need to have native mobile apps driving the applications along with web browsers. In this paper, we present a solution to re-engineer web applications developed using server-side scripting languages, into native mobile apps. The solution takes source code of the web application along with its test suite as input and produces corresponding cross-platform mobile apps. The entire re-engineering process is fully automatic requiring no manual intervention at any stage. Our solution is generic enough not only to handle popular server-side scripting languages, but also to output mobile apps that support diverse popular platforms including Android, iOS, and Windows Mobile. To showcase the capability and generality of our solution, we have developed a prototype tool KIRKE to handle applications developed using JSP, PHP, and ASP.NET. We present three case studies based on real-life codebases to evaluate the correctness, coverage, usability, and performance of our solution. The results indicate that KIRKE is capable of generating a mobile app that preserves the functionality of original web application and uses resources more efficiently when compared to the web application running on a mobile browser.

CCS Concepts

•General and reference → Cross-computing tools and techniques; •Information systems → Web applications; Web services; **•Software and its engineering → Software reverse engineering;**

Keywords

Software re-engineering, Dynamic analysis, Web applications, Mobile apps

MOBIQUITOUS '16, November 28-December 01, 2016, Hiroshima, Japan

DOI: http://dx.doi.org/10.1145/2994374.2994401

1 Introduction

The mobile phone is now the primary source of consuming online media with 60% of the average global mobile phone users using it every day to surf the web, leading to more than 60% of the total online traffic [11, 9]. The impact of mobile phones is more prominent in developing countries, as the first and primary access to the Internet for most users in these countries is from mobile phones. This unprecedented growth in mobile Internet user base has shifted the focus of developers from the web towards mobile. This surge of portability and ubiquity of mobile has forced the developers to lean towards a mobile-first approach and provide mechanisms to allow access to their existing web applications via mobile devices. Two possible strategies to move towards a mobile-first landscape are

- Develop server and mobile application from scratch. This involves high development cost and it takes time.
- Re-engineer the existing web applications to create mobile apps and its compatible server-side code. This would save development cost and time.

However, challenge is to design a re-engineering solution that can transform the existing server-side code with correctness and complete coverage, without compromising much with its usability. Further, it would be useful to have a solution that is generic enough to handle multiple server-side scripting languages and target a broad range of mobile platforms.

Extensive research in the domain of re-engineering has contributed towards various strategies that involve static analysis of the legacy application's source code and Unified Modeling Language (UML) diagrams. Though manual, sections of these strategies have been automated using existing code analyzers to expedite the entire process. One of the drawbacks of these strategies is the requirement of complex code parsers and analyzers, enabling relevant information extraction from large codebases. To increase efficiency and accuracy, they require extensive documentation and diagram support, in formats understandable by the supporting tools. High failure rates can be observed if prerequisites are either unavailable or not following the appropriate formats.

Dynamic analysis techniques overcome these drawbacks. They allow us to argue about the properties of an application by inserting probes into it and then executing it. Our approach leverages these techniques to extract information necessary to set up the client-side support for the target mobile apps. Advantages of this approach include easy migration to other languages and frameworks using lightweight codepoint weavers, no requirement of documentation or diagrams and complete automation without even having prior

knowledge of the application. To the best of our knowledge, our solution is the first attempt to automate the process of transforming web applications to mobile apps fully.

We present KIRKE, a prototype of our approach, to transform web applications that are based on popular server-side languages. As a proof of concept, we transformed three independent web applications, which are built on different server-side scripting languages, using KIRKE and conducted a study to understand the correctness, coverage, usability, and performance of the transformed apps. The salient features of our prototype tool KIRKE are as follows.

1. The solution only takes in source code and test cases and it does not require any manual interventions
2. The processing time is a function of the number of methods in the web application. Hence, it is scalable
3. If the web application follows Model-View-Controller (MVC) architectural pattern, which most large web applications do, the transformation is further expedited [18]
4. The approach is generic enough to work for all popular server-side scripting languages
5. It generates native mobile apps, with support to integrate model specific sensors like accelerometer, GPS and temperature for all popular mobile platforms
6. The code generated by our tool is human readable and documented, thereby easy to maintain and extend.

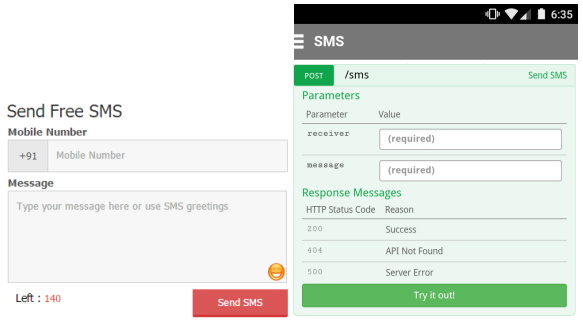
2 Problem and Approach

In this section, we provide the problem statement and give an overview of our approach.

2.1 Problem Statement

The problem under consideration is to transform an enterprise-class web application, written using any server-side scripting language, to a mobile app that preserves its functionality for all popular platforms with minimal time and effort. Given that the enterprise applications have a large number of Lines of Code(LOC), to reduce time and effort, the transformation should use the existing code with minimal changes and require minimal manual interventions. We assume that the application comes with its test suite, which we believe is a reasonable assumption to make for enterprise-class applications. Underlying are the major challenges which we need to address while designing a solution to the problem.

- Gathering extensive prior knowledge about web application's architecture, data flows, and UML diagrams will need extensive time and manual intervention. Therefore, our solution should not assume availability of any of these artifacts.
- Static analysis techniques require sophisticated, complex language parsers, and analyzers that are hard to construct and are not always openly available for all modern languages. Moreover, these techniques are often less scalable and imprecise. Therefore, any solution using static analysis techniques is specific to languages and will not support multiple languages.
- Different mobile platforms require an app to be built and modified using multiple languages and frameworks, which requires developers and designers with platform specific skills. Therefore, our solution should generate cross-platform mobile apps automatically.
- The generated code must be human readable so that it can be easily extended and maintained.



(a) (b)

Figure 1: (a) A screenshot of Send SMS page from way2sms.com. (b) Cross-platform mobile app to communicate with Send SMS web service, generated by KIRKE.

2.2 Approach

In this subsection, we give an overview of our approach using a simple example from a real-life application developed to build website `www.way2sms.com`. Figure 1(a) depicts a webpage, which is used to send an SMS to a desired mobile number. Our approach transforms the website's source code to allow sending SMS from a native mobile app.

To achieve the required transformation, we perform a sequence of steps starting with the dynamic analysis of the web application, in which the application code is automatically injected with synthetic method calls at pre-defined instrumentation points. If the web application is following MVC design pattern, the number of instrumentation points are fixed. MVC is the most popular and widely employed architectural pattern for developing web applications. The MVC pattern segregates presentation, business logic, and data. All server-side scripting languages support this pattern either out of the box (ASP.NET MVC and JSP-Servlet) or by using custom-built frameworks (CakePHP for PHP and Django for Python). Following are the components involved in MVC pattern.

- **Model** Responsible for providing logic and methods for accessing and updating the data layer.
- **View** Responsible for accepting data from the Model via intermediary Controller and presenting it using graphical user interfaces in HTML.
- **Controller** Every user interaction is converted to a HTTP request and handled by the Controller. It is the Controller's responsibility to delegate the request to appropriate Model and forward the corresponding response to be rendered in the View.

We take JSP to illustrate how MVC pattern is followed in a web application in Figure 2. For JSP, JavaBeans and Plain Old Java Objects (POJOs) represent the Model. JSPs represent the View and Servlets represent the Controller. Each of the user's interactions goes via HTTP methods in Controllers. Hence, controller methods act as a starting point for all the services making them candidate instrumentation points. There are seven such HTTP methods available in the JSP Controllers. They are doGet, doPost, doDelete, doPut, doTrace, doHead, and doOptions. In case a web application implements the MVC pattern, we instrument each of these controller methods to extract runtime information when these methods are invoked. Otherwise, we instrument

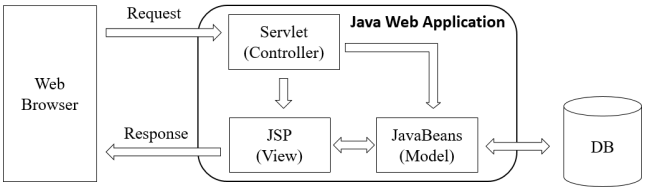


Figure 2: JSP's implementation of the MVC design pattern.

each method in the web application, which is still scalable.

We then rebuild the web application along with the injected code and execute it using the existing test suite. Execution of the injected code helps in the extraction of runtime parameters. This information is used to generate wrappers, around the legacy code using templates and code generators, acting as web services. The wrappers allow access to the internal legacy code via HTTP. Deployed web services are automatically annotated and described using a documentation framework [2], that provides a live documentation and testing environment for the generated web services. This documentation is then leveraged to generate client mobile applications that can run on multiple mobile platforms including Android, iOS, and Windows Mobile. All of the steps are explained in detail in Section 3.

We have developed a tool KIRKE that implements our approach and performs this transformation automatically. Figure 1(b) shows the output cross-platform mobile application generated by KIRKE for the "Send SMS" page.

3 Proposed Solution

Figure 3 depicts the schematic view of our approach. Our approach requires source code and test suite of a web application as input, and it generates corresponding mobile apps. All of the steps are fully automated.

For the context of this paper, we define a service as a functionality provided by the web application. A service is triggered directly through human interaction, i.e., submitting a form, clicking a button or invoking a URL, among others. A web application consists of multiple pages, each mapped with one or more services. Our target is to make all the services provided by the web application accessible via a native mobile app.

We will take "Send SMS" service, from `www.way2sms.com`, shown in Figure 1 as a running example to illustrate our approach. This service is used by millions of users for sending free SMS to any mobile number with a limit of 140 characters per message. Since `way2sms` is a closed source web application, we synthesized its background code as shown in Listing 1. The code accepts a receiver's mobile number and message as parameters bound inside a HttpServletRequest object. These parameters are extracted from the request object and passed to sendMessage() method. The code prints the appropriate status to screen, based upon success or failure of the message transmission.

In this section, we will describe the steps involved in the transformation. We use JSP to illustrate code snippets. The example is generic enough to illustrate the transformation process completely, yet simple enough not to get entangled in the complexity of the application logic and the syntax of JSP. Since a JSP page internally gets converted to a Servlet by the web container, we will demonstrate all the steps using the Servlet code.

```
public class SMS extends HttpServlet {
 @Override
 protected void doPost(HttpServletRequest request, HttpServletResponse
       response){
  Double receiver = request.getParameter("receiver");
  String message = request.getParameter("message");
  boolean sms_status=SMS.sendMessage(receiver,message);
  PrintWriter out = response.getWriter();
  out.println("<h1>" + sms_status + "</h1>"); }
```

Listing 1: Synthesized JSP code for Send SMS service offered by way2sms.com.

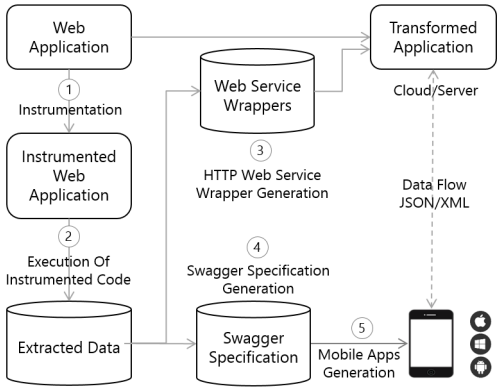


Figure 3: Proposed solution architecture depicting the steps required to transform a web application to mobile app.

3.1 Code Instrumentation and Execution

This composite step corresponds to the dynamic analysis that we perform. In the first step, KIRKE instruments source code of the web application with additional code, which at runtime extracts class names, interfaces, formal arguments, data types of the arguments, actual values passed to the arguments, and invoked methods.

To identify instrumentation points in the legacy web code, we exploit the separation of concerns property of the MVC design pattern. All controller methods act as candidate instrumentation points. In the particular case of JSP, Servlets acting as Controller contain seven pre-defined do*HTTP methods, that act as instrumentation points for the proposed approach. They are doGet, doPost, doDelete, doPut, doTrace, doHead, and doOptions. We instrument each of these methods to extract runtime information when these methods are invoked. In case of JSP, the work is least as the number of instrumentation points is just seven. In the case of "Send SMS" service, there is a doPost() method, to which we add a DataExtractor.extract() method, as shown in Listing 2.

After successful instrumentation of the code, it is compiled and executed. The objective, while execution, is to ensure that all the cases in the test suite are executed. For this example under consideration, we used Selenium, which is a suite of tools to automate web browsers across many platforms. The data retrieved by the instrumented code is saved for the next step. For the case of "Send SMS", the instrumented code extracts the following information:

- Class name: SMS
- HTTP method: doPost()
- Parameters: {receiver, message}
- Values passed to the parameters at runtime: {989968****,"Rs.500 withdrawn concerning transaction id: xyz"}
- Data type of parameters: {double, string}

```
protected void doPost(HttpServletRequest request, HttpServletResponse
      response){
 /* Instrumented Code */
 DataExtractor.extract(request,response,other_parameters);
 /* Legacy Code: Unchanged */ }
```

Listing 2: Instrumenting synthesized JSP code for Send SMS service offered by way2sms.com with DataExtractor.extract() method, in order to extract run time information.

```
/* ${*} signifies a placeholder */
public class ${class_name} {
 @POST
 @Path("${service_path}")
 @ApiOperation(value = "${service_description}")
 @ApiResponses(value = {${service_response}})
 public Response doPost(${service_parameters}){
  ${wrapper_code} }
```

Listing 3: JAX-RS template developed using Freemarker templates, required to develop HTTP Web Services.

The extracted information is used to create HTTP web service wrappers for the controller methods.

3.2 HTTP Web Service Wrapper Generation

Each of the controller methods is wrapped within a HTTP Web Service that contains code segment to

- Extract URL parameters from the incoming HTTP request
- Route an incoming request to appropriate legacy code
- Invoke legacy code and marshal parameters to Java objects to be passed to the legacy code
- Retrieve and unmarshall response generated by the legacy code
- Implement session management and security

Traditionally, web application code developed using server-side scripting languages is invoked by passing Java objects and it returns Java objects. For example, JSP Controller methods are invoked by passing parameters of type HttpServletRequest and they generate Java objects as a response. They do not use JSON or XML for passing data, which mobile apps use. The wrapper code containing web service function translates back and forth between Java objects and JSON or XML to communicate with the legacy code. The data collected in the previous step has all the necessary information, such as, the names of the classes and parameters to generate web service wrapper functions.

We prepared a template for JSP, so that collected data is substituted in the template to generate the wrapper functions. Listing 3 depicts a sample template for the web service wrapper based on Java API for REST-ful web services (JAX-RS). Similar templates can be generated for other server-side scripting languages.

Listing 4 shows substitution of the data extracted for "Send SMS" service in the template. The result is a web service function that can be called via HTTP. This web service function acts a wrapper code to the original JSP Controller method. Whenever the wrapper function is called, it marshals the passed values of the parameters to the original code, traps the response object by refactoring legacy application's return statements, unmarshall's response object, and forwards it to the client.

3.3 Swagger Specification Generation

The web application, with the added web service wrapper functions, is deployed on a server. The wrapper functions help to access legacy services over HTTP by referring to particular URI's, e.g. `https://shoppingstore.com/api/sms.json?receiver=989968&message=TransactionCompleted` relates to a synthesized HTTP Web Service, which sends an SMS to a mobile number specified as the receiver. The web service requires receiver's mobile number and message as query parameters. It returns a JSON object as a response. The later generated mobile app utilizes these HTTP web service URLs to talk to the legacy code. To automatically generate the mobile apps, we need a mechanism to translate the web service URIs to mobile application code.

```
public class SMS_Service {
 @POST
 @Path("/sms.json")
 @ApiOperation(value = "SMS Sending Web Service")
 @ApiResponses(value = {@ApiResponse(code = 404, message = "API Not Found"
       ),@ApiResponse(code = 500, message = "Server Error"),@ApiResponse(
       code = 200, message = "Success")})
 public Response doPost(@ApiParam(value = "Receiver Mobile Number",
       required = true) @QueryParam("receiver") Double receiver,
       @ApiParam(value = "Message To Be Sent", required = true)
       @QueryParam("message") String message ){
 /* Wrapper code to invoke doPost() method, under SMS Class with receiver
       and message as parameters */ }
}
```

Listing 4: HTTP Web Service wrapper based on JAX-RS specification required to communicate with synthesized JSP code for Send SMS service offered by way2sms.com.

```
/* ${*} signifies a placeholder
      */
"${path}" : {
   "${http-method}" : {
       "summary" :      ,
       "description" :  ,
       "operationId" :  ,
       "parameters" : [ {
          "name" :         ,
          "in" :           ,
          "description" :  ,
          "type" :
          }],
       "responses" : {
         "${response-code}" : {
              "description" :
                       }
         }
       }
```

Listing 5: Swagger specification template used to document a HTTP Web Service.

```
"/sms" : {
 "post" : {
   "summary" : "Send SMS",
   "description" : "SMS Sending Web
         Service",
   "operationId" : "doPost",
   "parameters" : [ {
    "name" : "receiver",
    "in" : "query",
    "description" : "Receiver Mobile
          Number",
    "type" : "double"
    },{
    "name" : "message",
    "in" : "query",
    "description" : "Message To Be
          Sent",
    "type" : "string" }],
   "responses" : {
    "404" : {
     "description" : "API Not Found"
           },
           "200" : {
     "description" : "Success"},
    "500" : {
        "description" : "Server Error"
     }
    }
  }
```

Listing 6: Swagger specification generated by Swagger-Codegen representing HTTP Web Service wrapper for Send SMS service offered by way2sms.com.

We use a tool called Swagger-Codegen that automatically converts web service wrapper to the corresponding Swagger specification. The Swagger specification is a formal structure widely used to document web services. The formalized JSON documentation contains all the information required to interface with the API such as access URL, parameter description and types, and response codes. Listing 5 shows a sample swagger specification template.

Web service wrappers contain annotations required to document a particular API. Before proceeding with the mobile apps generation, all wrappers are converted to corresponding Swagger specifications. Swagger-Codegen extracts specification from the server code, using these annotations and wrapper code. Listing 6 shows the extracted swagger specification for the Send SMS module. Here, @ApiOper-

ation specifies the functionality that the API is providing, @ApiResponses provides information about HTTP response codes that the API will generate while execution and @ApiParam describe parameters required to invoke the API.

3.4 Mobile Apps Generation

In this step, we use the Swagger specification of the HTTP web services to generate native mobile apps. We use Apache Cordova [4] to generate mobile apps from the Swagger specifications. Apache Cordova allows a native mobile app to be built using HTML, CSS, Javascript, and AJAX. It even allows access to mobile's native APIs, such as camera, accelerometer, and file manager, to enrich the functionality of the app. Apache Cordova takes in Swagger specification and build forms that use the newly generated web service to build a mobile app, using corresponding native mobile SDK. Swagger's live testing framework called Swagger-UI parses Swagger specification and generates a form-based user interface and necessary background code to communicate with the service depicted by the specification. We leverage this generated user interface and make it mobile friendly using Twitter Bootstrap, that provides CSS classes to make responsive interface elements. We pass this interface and code through Apache Cordova, which converts it to a cross-platform mobile app. Figure 1 shows the automatically generated mobile app for the Send SMS module. The generated mobile app supports a form-based user interface, which allows the user to easily type in parameters or data that is required to be sent to the server for processing. Clicking Try it out! button invokes mobile apps background code responsible for sending data to appropriate HTTP URL, getting and rendering the server's response on the user interface.

4 Discussion of the Solution

In this section, we discuss generalizing our approach to server-side scripting languages other than JSP and also the usability aspects of generated mobile apps.

4.1 Generalizability

JSP, PHP, and ASP.NET are the most widely used server-side programming languages for developing web applications. 78% of the websites using server side programming languages are based on PHP while it is 20% and 4% for ASP.NET and Java (sum is greater than 100% as many websites use more than one server-side programming language) [16]. Hence, we discuss generalizability from the point of view of these three languages. Our choice of using dynamic analysis was primarily driven by its advantages over static analysis, including easy language portability, higher precision, simplicity, scalability, and smaller development cost. This choice makes the techniques generalizable to other server-side scripting languages, such as PHP and ASP.NET. The only change required is the change of templates and instrumentation code, which is a one-time effort for any language.

4.1.1 ASP.NET

ASP.NET is a modern web development framework by Microsoft and supports three different patterns out of the box, web pages, web forms, and MVC. For ASP, C# files kept inside Models directory represent the Model. Web Forms represent the View, and C# files kept inside Controllers directory represents the Controller. Each of the user's interactions goes via methods defined under Controllers directory. Hence, these methods act as a starting point for all the services, which makes them candidate points for instrumentation. All Controller names end with "Controller" suffix, which makes it easy to locate those files.

```
public class SMSController : Controller{
    [HttpPost]
    public async Task<ActionResult> SendSMS(SMS model){
        boolean sms_status=SMS.sendMessage(model);
        ViewBag.status = sms_status;
        return View();  }   }
```

Listing 7: Synthesized ASP.NET code for Send SMS service offered by way2sms.com.

```
public class SMSController : Controller{
    [HttpPost]
    public async Task<ActionResult> SendSMS(SMS model)
    {   /* Instrumented Code */
        Hashtable request = new Hashtable();
        request[0] = model;
        DataExtractor.extract(request, this.GetType().Name, ViewContext.
              RouteData.Values["action"]);
        /* Legacy Code */   }
}
```

Listing 8: Instrumenting synthesized ASP.NET code for Send SMS service offered by way2sms.com with DataExtractor.extract(), to extract run time information.

Each method in the Controller maps to a unique URI in the web application, along with a corresponding View. There exists one-to-one mapping between the method in Controller and URI. Kirke uses these methods to extract data. As each user request goes via these Controller methods, it provides complete coverage to the Kirke. Every Controller method acts as an instrumentation point to instrument code for data extraction. Listing 9 shows the wrapper template for ASP.NET. Listings (7, 8, 10) demonstrates Send SMS service transformation for ASP.NET.

Though the proposed approach is applied in similar ways to both JSP and ASP.NET applications, we notice some fundamental differences between both the languages. In JSP, a Controller method can adopt only seven names as described earlier. On the other hand, methods in ASP.NET Controller can assume any developer defined custom name. Controller in ASP.NET can accept multiple parameters as opposed to a single HttpServletRequest object in JSP. Instrumentation codes need to be customized depending upon parameters for the Controller methods. ASP.NET even accepts Model objects as parameters. Hence, instrumentation code is modified to extract information from model classes about data members passed inside a Model. ASP.NET uses Web API in place of JAX-RS for wrapper generation. This is achieved by changing the template to support C# Web APIs. Rest of the steps, namely swagger specification and mobile app generation, are similar.

4.1.2 PHP

PHP is the most widely used server-side programming language on the web. Unlike JSP and ASP.NET, PHP supports MVC pattern via use of frameworks, such as CakePHP and Laravel. PHP applications developed using CakePHP or any other MVC framework follow a concrete MVC structure instilled by the framework. For PHP, files kept inside models directory represent the Model. HTML represents the View and PHP files kept inside controllers directory represents the Controller. Each of the user's interactions, goes via methods defined under controllers directory, hence these methods act as a starting point for all the services, which makes them candidate points for instrumentation. Similar to ASP, names of the Controller methods are developer-specific, as no naming restrictions are enforced by the framework. Due

```
/* ${*} signifies a placeholder */
public class ${class_name} : ApiController
{   /// <summary> ${service_description} </summary>
    /// <response code="${service_response_code}">${service_response_desc
        }</response>
    public async Task<HttpResponseMessage> Get(${service_parameters})
    {   ${wrapper_code}   }
}
```

Listing 9: ASP.NET Web API template developed using Freemarker templates, needed to develop C# Web Services.

```
public class SMS_Service : ApiController
{   // POST api/<controller>
    /// <summary> SMS Sending Web Service </summary>
    /// <param name="receiver">Receiver Mobile Number</param>
    /// <param name="message">Message To Be Sent</param>
    /// <response code="200">Success</response>
    /// <response code="500"">Server Error</response>
    public async Task<HttpResponseMessage> POST(Double receiver, String
        message){
    /* Calling Code To Invoke SendSMS() method, under SMSController Class
        with parameters being Receiver and Message converted to SMS
        Model */ }
}
```

Listing 10: HTTP Web Service wrapper based on ASP.NET Web API specification required to communicate with synthesized ASP.NET code for Send SMS service.

to generalizable nature of our approach, steps required for JSP and ASP.NET transformation will be followed out of the box in case of PHP. Only changes required are modifications in instrumentation code and web service template. As an exception for PHP, CakePHP provides methods to produce web services without creating wrappers, where legacy code is routed to a JSON view, rather than application defined HTML views. View automatically fetches the relevant data and forwards a JSON representation of it. This eliminates the need for wrapper templates in case of CakePHP, further expediting the re-engineering process.

4.2 Usability of the Mobile App

The re-engineered mobile app displays user interface generated using Swagger-UI and is composed of interactive forms to communicate with HTTP web service URLs. Mobile app users fill the forms as per the presented documentation extracted from the web application. Upon submitting the forms, the mobile app sends a HTTP request to the web service URL and renders the response in the user interface of the mobile app. The generated code for the mobile apps is such that the interface adapts itself to orientation changes of portrait and landscape modes. Look and feel of the interface remains identical across various platforms due to the usage of Twitter Bootstrap, that helps generate responsive user interfaces. It provides CSS classes to build additional user interface elements. In future versions of Kirke, we intend to improve the UI, such as, by porting images and design elements from the web applications to mobile apps. This way, the mobile app will have a similar look and feel as that of the web application.

5 Evaluation

Our evaluation of Kirke focuses on the coverage, correctness, ease of use, and efficiency of the execution of a re-engineered application. We define coverage as the degree of the functionality supported by the generated mobile app in comparison with the original application, and correctness as the degree of the semantical correctness of the functionality that was covered by the mobile app. We define the ease of use as the comparative effort of the users who interact with a web application as well as the transformed mobile app and efficiency as the resource utilization in terms of CPU, memory, and battery consumption by the mobile app while performing a task. Kirke's coverage depends on the quality of the test suite used in terms of the statement coverage. We expect that for an enterprise web application, a good test suite with a high statement coverage will always be available as one of the essential artifact's developed as a part of the application development process.

5.1 Selection of the Web Application

According to mobile analytics firm Flurry, from 2013-2014 shopping app usage recorded the highest growth than any other category of apps [1], which depicts huge demand of e-commerce websites turned into mobile apps. Hence, we selected three open source shopping cart applications based on JSP, PHP and ASP.NET for the evaluation. All three applications provide common functionalities of catalog management, customer management, online retail management, and customer order processing system.

Table 1 displays the web applications used for evaluation. Total LOC is an indicative of the project size. Server-side LOC depicts LOC written using respective server-side scripting languages only. It does not take into account code written using other languages, such as HTML, CSS, Javascript among others, in the same project. Transformation time refers to the time consumed by Kirke to transform web application to corresponding mobile app, excluding time taken for test suite execution on instrumented code, that varies with the number of test cases in the test suites. We conducted transformations on a laptop running on an Intel dual core 64-bit processor with 4GB of RAM.

5.2 Methodology

We selected ten important functionalities for all three web applications. The web application is composed of several user interaction screens spanning these ten functionalities from a customer's point of view, where customer being a mobile app user in an e-commerce scenario. These functionalities represent all the steps required to complete a transaction on an e-commerce platform. Since all three applications belong to shopping cart category, they all possess mentioned functionalities. Some of the functionalities include viewing product catalog, adding a product to shopping cart, searching for a product, among others.

Instrumented Methods denote the number of methods identified and instrumented using Kirke. These methods refer to Controller methods in the web application. Total Methods denote the number of methods that need to be instrumented in the absence of MVC controller methods, thereby increasing the time for code instrumentation. The amount of work would still be less than rewriting entire project into code that follows service oriented architecture and then coding the mobile apps. To check correctness and ease of use, we conducted a survey and asked participants to use the ten functionalities on web applications and transformed mobile apps. We then compared the numbers from both the surveys. To check ease of use, we asked the participants to give a subjective rating of their difference in experience.

We conducted a survey of sixty participants, majoring/majored in Computer Science and have a basic familiarity with mobile technology. Participants were divided into groups of twenty, to conduct the survey for the three case studies. We refrained from repeating candidates for different case studies as that could have resulted in functionality level comparisons depending upon legacy web application, and may have lead

Language	Web Application	Description	Total LOC	Server-side LOC	Total Methods	Instrumented Methods	Transformation Time
JSP	Saikiran Bookstore	Marketplace for books and stationary products [7]	14 K	10 K	121	80	84 sec
PHP	CakePHP Shopping Cart	Marketplace for Fashion Apparels [10]	50 K	15 K	76*	76	90 sec
ASP.NET	Open Order Framework	Order processing system for small business owners [6]	37 K	6 K	300	143	104 sec

Table 1: List of applications used for evaluation and corresponding transformation metrics.*For the PHP application, Total Methods equals Instrumented Methods, as entire business logic resides inside Controllers only, which is not a good practice.

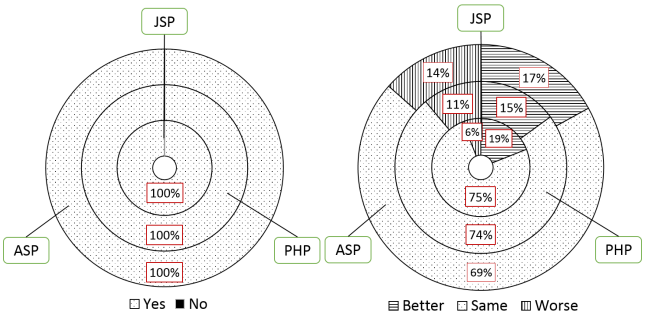


Figure 4: Survey analysis for JSP, PHP and ASP.NET case studies. (a) Percentage of functionalities successfully executed on the mobile app. (b) User experience with the mobile app in comparison with that of web application.

to biased feedbacks. Each participant first completed the ten functionalities on web application and then performed the same on the re-engineered mobile app. In particular, we asked the following questions to the participants.

1. Whether you were able to execute functionalities successfully via both web and mobile versions?
2. If not, what problem/s they experienced?
3. Whether responses on both usages matched or not? For example, ViewCatalog should display exactly same products and information on both the systems.
4. How was their experience using the mobile application, better, same, or worse?

5.3 Results

Figure 4 provides a compiled analysis. According to the survey results, 100% of the participants were successfully able to execute the functionalities and get same responses using the web applications and mobile apps. This ensures correctness for the chosen functionalities for all three applications. At least 72% candidates marked their responses to be "same" for all three case studies, regarding how they felt about using the mobile app. This measures ease of use, given already existing familiarity with the web app. 17% responses were marked "worse" while using the mobile app. This is because the mobile app, coming out of the tool, provides basic UI sans the images and icons in the web application. The messages from the mobile app are in JSON or XML format and not hidden from the user. Many participants were already familiar with such messages, but some were are not. As evident from the survey analysis, results for different case studies are synchronized, with no significant contrast, ensuring generalizability of the proposed approach. The resulting mobile apps are platform independent. We successfully deployed and tested them on Android (5.1.1), iOS (version 8), and Windows (version 8.1) mobile platforms. Though native, mobile apps project similar user interface for all the devices. Android-based mobile application executables along with a video demonstration representing mobile app usage on all three mobile platforms are available for download at https://github.com/rohitmehra/legacymodernization.

	JSP	
Metrics	Web Application	Mobile App
Battery Consumption	333 mW	205 mW
CPU Utilization	26%	10%
Memory Usage	184 MB	98 MB
	PHP	
Metrics	Web Application	Mobile App
Battery Consumption	304 mW	221 mW
CPU Utilization	9%	6%
Memory Usage	254 MB	112 MB
	ASP.NET	
Metrics	Web Application	Mobile App
Battery Consumption	251 mW	211 mW
CPU Utilization	14%	7%
Memory Usage	220 MB	101 MB

Table 2: Performance comparison between web application running in a mobile browser and generated mobile application for all three case studies (JSP, PHP and ASP.NET).

5.4 Performance

We conducted the performance evaluation of generated mobile applications in comparison to the web applications. While the web applications were executed on Mozilla Firefox mobile browser (ver. 45.0.1), the mobile applications were executed natively on the same Android (ver. 5.1.1) device with Snapdragon 810 processor and 4GB RAM. We profiled both systems for one minute of usage while performing the same tasks. We used Power Tutor (ver. 1.4) for measuring battery consumption and GameBench (ver. 3.2.2p) profiler for measuring CPU and memory consumption.

Table 2 shows the collected data. Overall, the generated mobile apps performed better in terms of resource utilization as they are native apps with only essential libraries. Both Apache Cordova applications and Web Browser use similar HTML rendering engines under the hood. However, web browsers provide far more features, including extensions and plugins, navigation controls, background services, and complex user interface that bloats its resource utilization. This is evident from the mobile app sizes where our mobile apps are 2.5 MB in size as compared to 39MB for the Mozilla Firefox android browser.

6 Related Work

Our approach addresses the problem of re-engineering legacy applications and generating code which targets multiple mobile platforms. Accordingly, we split the related literature into these two categories.

Re-engineering of Applications Liu et al. [12] presents motivation and key issues in performing migration of non-web applications to web services. Sneed [15] proposes a way

to create a wrapper around the non-web application to expose the internal business processes using the web services. Major steps in this process includes salvaging the legacy code to extract out business processes, which can then be linked to a wrapper. The wrapper must have interfaces and arguments similar to the legacy business process to forward the control to the legacy system and retrieve processed response. This process is semi-automated, requires static data flow analysis and is based on languages such as C and COBOL that are not typically used in modern web application architectures. In comparison, our approach is fully automated, works for all modern languages, and eliminates the need for computationally intensive and often imprecise static data flow analysis.

Matos et al. [13] leverage the architecture proposed by Sneed [15] but instead of plain code refactoring, they propose annotating the source code using specific pattern matching rules, reverse engineering the code to its corresponding graph representation, and then applying graph transformation rules to obtain the re-engineered application. Their approach applies to traditional desktop applications instead of web applications which our proposed approach supports. All of the methods discussed above are primarily based on static code analysis. Following paragraph discusses transformations where some operations are performed manually. Researchers at Microfocus defined manual steps to transform a legacy Cobol application to a Java web application, with the help of cross compilers and visual cobol IDE plugin [3]. In their approach, operations including candidate service identification, knowledge gain, and wrapper generation are performed manually with the support of provided IDE, contrast to proposed approach's automated way. Marchetto et al. [14] describes the steps and tools required to perform manual migration and emphasizes on testing for correctness. In comparison, our approach is fully automatic.

Generation of Mobile Apps for Web Applications Gonative.io [8] and Appypie.com [5] provide an automated approach to transform web application to mobile apps. Instead of re-engineering a web application, they render a snapshot of web application in a web view embedded within a native mobile app. It is similar to running a web application inside a mobile web browser. This approach does not provide access to the web application via web services. This limits its usability on mobile platforms, as a legacy web application designed for desktop environment is compacted and projected on mobile devices. In addition, since web browsers run slow on mobile devices, there is a need to develop native mobile apps, than developing mobile websites [17].

7 Conclusion and Future Work

In this paper, we looked at the problem of transforming enterprise-class web applications, written using server-side scripting languages, into mobile apps. For broad applicability, the approach for the transformation is generic and scalable enough to consider multiple popular scripting languages and mobile platforms. In our knowledge, the tools that are currently available either require some manual interventions or they provide a small subset of original application's functionalities.

We propose an approach based on dynamic analysis to provide a solution that suits out needs. The only assumption that we make is the availability of test suite, which is sound for the case of enterprise-class applications. We built a tool Kirke that implements our approach for three most popular server-side scripting languages. We evaluate our tool in terms of coverage, correctness, ease of use, and efficiency in execution. For the evaluation, we consider three open source codebases developed by third-party developers. We consistently find Kirke to perform satisfactorily for the case studies. While we leverage the use of MVC architectural pattern in the web applications to expedite the transformation, we don't depend on the use of any pattern. Kirke takes in web application's server code to build web services out of it. It then exposes these services through automatically generated mobile app. Kirke does not transform web application's client code. It does not matter what web application does, be it shopping site or even an online game.

In the future, we intend to improve the UI, such as, by porting images and design elements from the web applications to mobile apps. This way, the mobile app will have a similar look and feel as that of the web application. Our current implementation of Kirke supports CakePHP framework for PHP. We will support other frameworks, such as Laravel and Zend, in the future. Generated mobile app's user interface displays JSON or XML responses, which shall be replaced by incorporating interface elements, in future.

References

[1] Shopping, productivity and messaging give mobile another stunning growth year.

[2] Swagger: Rest documentation framework.

[3] Take your step-by-step journey from cobol to mobile.

[4] Apache. Apache cordova, learn more about the project.

[5] Appypie.com. Convert website to mobile apps, 2015.

[6] L. Bacaj. A lightweight shopping cart web application in asp.net mvc 5.

[7] Github. Saikiran bookstore shopping cart.

[8] GoNative.io. Convert your web application into native android and ios, 2015.

[9] InMobi. Global mobile media consumption wave 3 report.

[10] A. Kende. Shopping cart built with cakephp php framework.

[11] A. Lipsman. Major mobile milestones in may: Apps now drive half of all time spent on digital, 2014.

[12] Y. Liu et al. Reengineering legacy systems with restful web service. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 785–790. IEEE, 2008.

[13] C. Matos et al. Migrating legacy systems to service-oriented architectures. *Electronic Communications of the EASST*, 16, 2009.

[14] F. Ricca et al. From objects to services: toward a stepwise migration approach for java applications. *International journal on software tools for technology transfer*, 2009.

[15] H. M. Sneed et al. Wrapping legacy software for reuse in a soa. In *Multikonferenz Wirtschaftsinformatik*, volume 2, pages 345–360, 2006.

[16] Y. Sonmez et al. Performance comparison of php-asp web applications via database queries. In *Proceedings of the The International Conference on Engineering & MIS 2015*, page 45. ACM, 2015.

[17] Z. Wang et al. Why are web browsers slow on smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 91–96. ACM, 2011.

[18] K. Watanabe et al. A web application development framework using code generation from mvc-based ui model. In *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, pages 404–411. Springer, 2009.