**WL++: a framework to build cross-platform mobile applications and RESTful back-ends**

Blerina Bazelli and Eleni Stroulia*

Department of Computing Science, University of Alberta, Edmonton, Canada
Email: bazelli@ualberta.ca
Email: stroulia@ualberta.ca
*Corresponding author

**Abstract:** About one in two adults and one in four teens own a smart phone in North America and use it to access online information and services. This, ever increasing, demand for mobile applications has given rise to the need for tools and methods to systematically support the design and construction of these applications. Responding to this need, we have developed WL++, a code-generation environment for mobile-application development. Using this tool, developers can create application-specific diagrams of the application’s logical model and annotate them with information about the user-interface widgets appropriate for interacting with the model elements. WL++ then produces a relational back-end for storing the model data, a set of RESTful APIs for accessing and updating the back-end, and a multi-platform mobile application that relies on the IBM Worklight framework to render, interact with and store the relevant data, through the chosen widgets and APIs. In addition, a general service monitors and records the usage of the APIs and the data exchange between the application and the back-end. In this paper, we describe the WL++ cross-mobile application generation framework and we illustrate its functionality with an example.

**Keywords:** cross-platform mobile development; model-driven engineering; MDE; eclipse plugin; RESTful API; code generation.


**Biographical notes:**

Blerina Bazelli is an MSc student in the Department of Computing Science at the University of Alberta, Canada under the supervision of Dr. Eleni Stroulia. She received her BSc degree from the Department of Applied Informatics at the University of Macedonia, Greece in 2011 under the supervision of Dr. A. Chatzigeorgiou. Her main research interests include web application development, software maintenance, mobile applications for e-health, model-driven engineering and code generation among others. She also participates in the Intelligent Diabetes Management project (http://idm.ualberta.ca/) developed at the University of Alberta aiming to keep track of patients’ blood glucose and provide suggestions to maintain it within an acceptable range.

Eleni Stroulia is a Professor and NSERC/AITF Industrial Research Chair on Service Systems Management (with support from IBM) with the Department of Computing Science at the University of Alberta. Her research addresses problems motivated by industrial and community needs with software systems designed to support people’s activities. Through her work, she has made substantial contributions to the areas of software analysis, migration and reengineering. More recently, she has been focusing on software services through web platforms (Web 2.0 collaborative tools, mobile devices and virtual worlds) as a way to offer innovative healthcare and education services. She enjoys an H-index of 31 with 260 publications and 3,200 references to the same and regularly serves on the programme committees of international software-engineering conferences.

This paper is a revised and expanded version of a paper entitled ‘Towards families of personalized mobile applications’ presented at the 1st International Workshop on Personalized Web Tasking, Santa Clara, CA, 27 June 2013.
1 Introduction

The shift from utilitarian cell phones for calling and texting to stylish devices with the capabilities of a personal computer happened in a very short time frame. The ubiquitous availability of these devices has made them the primary mode of accessing information and services on the internet. According to Pew (PewResearch, 2013) as of May 2013, 63% of adult cell owners (in North America) use their phones to go online and 34% of cell internet users go online mostly using their phones, and not using some other device such as desktop or laptop computers.

Given the broad usage of these devices, providers are highly motivated to deliver mobile applications to enable users to access their information and services. The question then becomes choosing a platform on which to deliver these applications. As of May 2013, Android and iPhone owners are equally common within the cell-owner population as a whole, although this ratio differs across various demographic groups (Smith, 2013). Given this market picture, application developers are highly motivated to deliver their applications on both these platforms. If one were to follow traditional software-engineering methodologies, mobile-application development would need to comply with the native programming languages of each target operating system. For instance, Android applications require the knowledge of the Java programming language whereas iPhone applications are developed in Objective C.

Parallel development on multiple platforms implies increased effort, higher cost, and more difficult maintenance and evolution, challenges that get exacerbated by the highly-competitive mobile-app market.

To mitigate this issue, technologies such as Phonegap (http://phonegap.com) and IBM Worklight (http://www-03.ibm.com/software/products/en/worklight) offer middleware services on which developers can create dynamic web-based applications that can be compiled into platform-specific code. These applications are in effect dynamic web applications that run on the device’s internal browser (web view); they look and feel like native applications due to the aforementioned technologies (Smutny, 2012) that enable the use of native user interface (UI) widgets and access to the devices’ hardware components (e.g., camera, accelerometer, GPS, etc.). Although native applications provide full access to the devices’ features and can potentially offer better performance and user experience (Charland and Leroux, 2011), cross-platform applications are a valid choice when the application follows a simple user-interaction model, it does not need to meet extremely fast response times, and development resources are at a premium.

In this work, we focus on a broad family of applications that fit the above profile, namely catalogue-style applications. Focusing on this class of applications enables us to develop a model of the application data and the services manipulating this data. Consequently, we assume that the data elements are:

a structured entities consisting of multimedia attributes
b collections of these entities.

We further postulate that the most important (and typically desired) features for these applications are:

a creation of new entities to be added to application repository
b retrieval of (potentially escalating) entity details given an entity identifier
c faceted search (given a combination of entity properties and desired value ranges for them return a collection that matches the input criteria)
d entity-state manipulation through RESTful APIs.

RESTful web services are broadly adopted especially when it comes in messaging frameworks for mobile web services; a REST mobile web server (MWS) outperforms a SOAP MWS, in terms of server utilisation and request waiting time (Aijaz et al., 2009), and in terms of efficiency and scalability (Mulligan and Gracanin, 2009). Furthermore, RESTful services allow the transmission of different data formats, including JSON, which is more lightweight than XML (usually transmitted through SOAP services).

To support the systematic development of such multiplatform applications, we have developed WL++, a code-generation environment as an Eclipse plugin. This framework assists developers in generating the frontend UI of the application, its local and backend storage, and the RESTful APIs implemented on the back-end server and invoked by the front-end to deliver the application functionalities. The plugin is composed of two main parts:

a the editing area
b the palette.

The editing area allows the creation of application diagrams through the available widgets included in the palette. Developers through drag and drop actions can create applications diagrams that can be later compiled into source code, in a model-driven engineering (MDE) method. The WL++ code-construction framework also includes a general monitoring service that observes the API calls and the data exchanged between the application and the back-end server. This data is optionally stored into a database in JSON format, in order to be consumed by a downstream domain-specific analysis-and-recommendation algorithm. The monitoring component can be used to forward recommendations as push notifications to the mobile application.

This work makes two important contributions. First, it proposes an abstract model, to represent the data entities commonly found in this family of applications (i.e., product catalogues). This model makes explicit a common data vocabulary for these applications and the relationships typically existing between the various data types. Second, it offers a software tool, WL++, as an extension of the Worklight IDE, to support the development of the broad family of catalogue-based mobile applications (e.g., a contact manager app that stores information regarding...
contacts in a list, a ToDo task list app that allows the addition/editing/deletion of ToDo task). As mentioned before, catalogue-style applications usually require operations such as entering and storing data, reviewing it later on and modifying it and/or deleting it. Also, since eventually there would be long lists created there is a need for a faceted search based on different criteria depending on the application data model.

The rest of this paper is structured as follows. Section 2 and Section 3 review the background of this work, in terms of the technologies that have given rise to the popularity of mobile web-based applications and the related research on systematising the development of mobile applications. In Section 4, we describe the major components of WL++ and its architecture. In Section 5, we present the available widgets of the application specification plugin and the process of constructing an example application by using these features. Finally, we conclude with a discussion about our future plans in Section 6.

2 Native vs. web-based mobile apps

In this section, we discuss some reasons why web-based mobile applications may be preferable to native applications, and we explain the relevance and usefulness of MDE techniques on the design of the WL++ application specification plugin. Large software providers, such as Google, Apple, and Nokia – to name a few, have been competing against each other in offering Software Development Kits (SDKs) to support developers building mobile applications on their platforms. Typically, each of these SDKs includes an integrated development environment (IDE) and a device emulator. However, an SDK assumes expertise with a different programming language that not only requires that developers invest some effort to become familiar with its underlying programming model, but also produces mobile applications that can only be deployed on the corresponding platform. Although, native applications are able to take full advantage of all the device features, and thus potentially offer a better user experience, they are fundamentally limited in their deployment scope and consequently in their adoption.

The operating system market share changes rapidly and therefore it is difficult, practically impossible, to predict which operating system(s) will become more prevalent even in the near future (Holzer and Ondrus, 2009). With that being said, in the case where an application aims to be used by a large population, ideally it should be developed to run on at least the top three or four dominant operating systems. However, developing an application for multiple operating systems can be extremely resource consuming; in fact, the application must be developed and tested separately for each operating system. Furthermore, it also requires either developers having multiple skills (i.e., able to programme in different programming languages) or several teams of developers working on each application separately. To make matters worse, multiple applications are more difficult to maintain: when there is a need for an update, changes should be implemented and applied to all the operating systems, whereas in the case of web-based applications, changes are implemented much faster and updates are effective to all the devices at the same time.

It is only relatively recently that cross-platform development tools, such as Phonegap, became able to access the hardware of mobile devices, such as accelerometer, camera and GPS. This ability has been instrumental in the adoption of the cross-platform mobile-application development practice. At the same time, the overall functionality and capabilities of these tools have been continuously improving. One dimension of improvement arises from the fact that the underlying technologies leveraged by these tools, i.e., HTML5, CSS3, and JavaScript, are continuously improving. In addition, most cross-platform tools, including Phonegap, can be extended with special purpose plugins, in the native language of each target platform, to leverage their specific features. The Phonegap community has been especially active in this regard and a wide variety of plugins have already been implemented. For example, there exists a plugin for Android devices which produces native notifications; the generation of these notifications and their context are controlled by Phonegap in JavaScript. Finally, substantial progress has been made in developing user-interaction extensions that imitate the look-and-feel of purely native applications, i.e., JQuery Mobile (http://jquerymobile.com/), Sencha Touch (http://www.sencha.com/products/touch), and app-UI (http://triceam.github.io/app-UI/), and techniques to improve the application responsiveness, i.e., with image preloading, and hardware acceleration.

Table 1 reviews several cross-platform mobile-application tools, with Phonegap, Rhomobile and Appcelerator being the most popular among them (Cowart, 2013). Phonegap and Rhomobile follow a similar development paradigm; on the other hand, Appcelerator differs significantly in terms of:

a the way it accesses native features (through a JavaScript API)

b the code it generates (web-based vs. native).

IBM Worklight is a full mobile application platform built on top of Phonegap. The advantages of Worklight over Phonegap include the application server, a mobile-browser simulator, a set of adapters used to communicate with back-ends, etc. The first three listed frameworks produce web-based applications, and offer the ability to extend them with additional features implemented in the programming languages supported by specific platforms. For example, Phonegap applications can take advantage of native features through plugins developed in the targeted platform language as mentioned before. On the other hand, Appcelerator Titanium, MoSync, and ifactr (based on Xamarin), produce as output native applications. In fact, although Appcelerator supports four platforms to date, the way it has been developed makes the support of more platforms harder.
Table 1  Comparison of popular cross-platform mobile frameworks based on the OS they support, developing language, IDE, licence and the type of the output application

<table>
<thead>
<tr>
<th>Framework</th>
<th>Mobile OS support</th>
<th>Language</th>
<th>IDE</th>
<th>Open source licence</th>
<th>Generated application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phonegap</td>
<td>iPhone, Android, Blackberry, WebOS, Windows Mobile, Symbian, Bada</td>
<td>HTML, CSS, JavaScript</td>
<td>Depending on the IDE supported by the OS</td>
<td>Apache Public, Licence v2</td>
<td>Web-based</td>
</tr>
<tr>
<td>Rhomobile</td>
<td>iPhone, Android, Windows Mobile</td>
<td>HTML, CSS, Java, Script, Ruby</td>
<td>RhoStudio</td>
<td>MIT</td>
<td>Web-based</td>
</tr>
<tr>
<td>IBM Worklight</td>
<td>iPhone, Android, Windows Mobile, Blackberry</td>
<td>HTML, CSS, JavaScript</td>
<td>Eclipse</td>
<td>No</td>
<td>Web-based</td>
</tr>
<tr>
<td>Appcelerator</td>
<td>iPhone, Android, Windows Mobile, Blackberry</td>
<td>JavaScript</td>
<td>Titanium Studio</td>
<td>Apache Public, Licence v2</td>
<td>Native</td>
</tr>
<tr>
<td>Titanium</td>
<td>iPhone, Android, Windows Mobile, Blackberry</td>
<td>JavaScript</td>
<td>Titanium Studio</td>
<td>Apache Public, Licence v2</td>
<td>Native</td>
</tr>
<tr>
<td>MoSync</td>
<td>iPhone, Android, Windows Mobile, Blackberry, Symbian</td>
<td>C/C++</td>
<td>Eclipse</td>
<td>GPL v2</td>
<td>Native</td>
</tr>
<tr>
<td>ifactr</td>
<td>iPhone, Android, Windows Mobile, Blackberry, Symbian</td>
<td>C#</td>
<td>Visual Studio</td>
<td>No</td>
<td>Native</td>
</tr>
</tbody>
</table>

Cross-platform applications require less effort and fewer resources than native applications for their development and maintenance and tend to be less expensive than their native counterparts. Furthermore, technologies such as HTML, CSS and JavaScript are widely used and therefore it is much easier to find skilled developers who are familiar with them than it is to find developers with expertise in each native technology stack. Consequently, multi-platform application development is beginning to emerge as the more popular mobile-application development paradigm.

3 Related research

Having reviewed the technologies currently available for mobile-application development, in this section, we discuss research projects aiming to support the development of these applications. Cabana is a multi-platform mobile development system intended for students in introductory computer-science courses (Dickson, 2012). Students are able to create cross-platform mobile applications, modelling them as a set of nodes connected with edges among them. The nodes represent screens, buttons and custom code modules responsible for the applications’ logic, which must be implemented in JavaScript. Although, Cabana meets its educational/pedagogical objectives, its features do not provide sufficient support for the development of real applications. First, its user-interaction model in terms of a screen-navigation graph is limited and makes the development of complex entities difficult. More importantly, however, it does not support the generation of back-ends, which is an essential part for the majority of mobile applications.

The work closest to ours is MD², a model-driven approach for cross-platform application development (Heitkötter et al., 2013). MD² defines a domain-specific language in terms of which the to-be-generated application is represented as a set of nested models. The language was developed in Xtext and Xtend is used to generate the code. Then, source code is generated for Android and iOS platforms through an Android and an iOS engine respectively. Furthermore, a back-end (Java service) is generated with capabilities of simple create, retrieve, update, delete (CRUD) operations. This gives the ability of communication between the application and the local/remote back-end. The MD² framework is clearly motivated by similar objectives as our work. However, it adopts a different code-construction methodology that involves two different compilers for the two target platforms. This decision implies that MD² covers fewer platforms than our framework, requires more complex application modelling, and can potentially result in better performing applications with more varied and versatile look-and-feel.

On the other hand, our WL++ framework focuses on a particular class of applications, with fairly simple interaction constraints that can be met by adopting a common middleware layer, compiled into multiple platforms. The end users of our tool are developers, and this is why WL++ aims to produce understandable and maintainable generated code. To that end, WL++ adopts Backbone.js (http://backbonejs.org/), a JavaScript framework that enables the separation (and smooth integration) of user-interface from (with) business-logic concerns. Also, the generated source code includes comments defined in the templates in order to assist developers in making further changes. Finally, we believe that the capabilities of cross-platform frameworks have been expanded and therefore applications generated by these tools are becoming more competitive to native ones. Besides, the code base that is shared among different
platforms (Android, iOS, and Windows) makes them popular by targeting larger groups of users.

Ribeiro and Silva (2014) developed a DSL for mobile applications, namely XISMobile. Following the MDD approach, they extended the XIS UML profile project through which developers can model web applications. According to the XMI Mobile approach, a native application can be generated by defining four major views: entities view (that represent the entities of the application), usecases view (responsible for the operations that could be performed on the data), architectural view (allows the connection of the application with external entities/services) and the user-interface view (where navigations and UI components are being defined). Through the XIS-Mobile language, simple native applications targeting the Android and the Windows Phone platforms are generated. In their work, they evaluated their language by developing a to-do list application. However, this work is still in its very initial steps.

Currently, there is a plethora of mobile applications available in the market. The available SDKs allow the community to develop applications at a rapid pace. Gasimov et al. (2010) examined the existing mobile applications and categorised them into four major categories based on their purpose:

- transaction (services offering communicate between several mobile phones)
- content dissemination (broadcasting a piece of information to a significant number of mobile phones)
- social networking (includes interaction among mobile phone users)
- personal productivity and leisure (performing personal activities) based.

They identified three main challenges in developing mobile applications (power-consumption minimisation, cross-platform compatibility, and software quality) and proposed the idea of standardisation of mobile hardware/software protocols (Höpfner et al., 2011). The WL++ framework is generally targeted towards personal productivity and leisure applications and has been inspired by two different mobile applications that our group had already developed, before embarking in the effort to systematise mobile-application development with a framework. More specifically, we have already implemented two multi-platform self-management mobile applications for e-health, namely EASI and Physitivity (Stroulia et al., 2013), and it is on the basis of this experience that we are developing WL++.

There are several tools for developing domain specific languages such as Xtend (http://www.eclipse.org/xtend/) and tools to generate code based on the DSL [e.g., Xtend (http://www.eclipse.org/xtend/), Acceleo (http://www.eclipse.org/acceleo/)]. Another toolkit similar to MD2 that use a DSL developed in Xtend is APPlause (APPlause, 2011), a cross-platform development toolkit that generates native mobile applications for Android, iOS, Windows Phone and the Google App Engine. This aforementioned frameworks/- toolkits use an editor that requires the user to insert several commands based on the DSL specified in order to generate the desired output. The translation of these commands to actual code (output) is done through tools such as Xtend and Acceleo. Xtend also offers a simplistic way to write Java code which later is transformed into actual Java code. Both Xtend and Acceleo are able to perform model to code transformations. More specifically, they rely on templates which a developer creates and then code is injected in these templates according to the application models. In our case, we use the StringTemplate (http://www.stringtemplate.org/) java template engine in order to generate the application code elements (Java, JavaScript and HTML code).

StringTemplate is a versatile and lightweight library that allows the developer to choose the delimiters of their choice in order to define their variables. These templates are rendered by the Java classes generated based on the Ecore metamodel and populated by extra methods where additional functionality have been manually added.

4 The WL++ application construction framework

In this section, we discuss the WL++ software architecture and the three major components it integrates. The first component is IBM Worklight, a comprehensive cross-platform mobile-application development and deployment environment. The second component is backbone, a framework that defines the organisation of the produced mobile applications, in a manner that should facilitate the process of extending and maintaining them in the future. Finally, the third component is our own WL++ application modelling plugin that relies on the graphical modelling framework (GMF) for designing and constructing mobile applications.

4.1 Mobile applications with IBM Worklight

We chose the Worklight IDE as it is a full mobile application platform leveraging a powerful JavaScript library (Phonegap) through which it accesses native features of the mobile devices. Phonegap has been broadly adopted by cross-platform mobile application developers as it supports a wide variety of platforms. Furthermore, it is well documented and there are already plenty of open source mobile applications developed in Phonegap. In addition, Worklight offers a ‘What you see is what you get’ (WYSIWYG) editor which allows us to preview the HTML pages before they are complied in a real browser, a built-in server which allows the testing on a number of mobile-device simulators, and several more built-in software components (discussed in detail in Section 5.1.1) that can be reused in individual mobile applications to easily support typically desired features.

The Worklight framework is built on top of Phonegap, a JavaScript library also known as Cordova. Resources common to all platforms exist in a common directory. Platform-specific optimisations and skinning can be done on
a per-platform basis, through environment-specific directories. The contents of the environment-specific directories are integrated with those of the general-purpose directories in order to compile platform-specific applications. Figure 1 depicts the directory organisation of Worklight application assets, for three environments (Android, Blackberry 10 and iPhone). By adding a new environment to a Worklight project, one can test the mobile application running on the simulator of their choice through the browser.

![Diagram of IBM Worklight project structure](image)

**Figure 1** The IBM worklight project structure (see online version for colours)

Note: Three environments have been added (Android, Blackberry 10 and iPhone.

At compilation time, platform-specific CSS and JavaScript assets are concatenated with their platform independent counterparts, and platform-specific HTML files and images replace similarly named platform-independent ones. This model is dictated by the interpretation rules of the corresponding languages. In CSS and JavaScript, subsequent definitions replace existing ones; therefore, concatenation ensures that platform-independent specifications will be applied as long as no platform-specific ones have been defined. This is not the case with HTML, and this is why asset replacement is required. This latter behaviour, however, implies that HTML templates need to be defined at both levels (in our case will be merged, not replaced). Only templates with the same name that are redefined will be replaced.

### 4.2 Modular UIs with backbone

Backbone belongs to a new breed of JavaScript libraries conceived to superimpose a conceptual structure, akin to the model-view-controller pattern, to web applications through **models**, **collections**, and **views**.

The **model** construct supports key-value bindings, manipulated through custom events; following the inversion-of-control pattern, when the model state (i.e., its key value) changes, the associated **views** get notified. **Collections** group models and provide a **API** for manipulating them. Backbone also provides the capability to map models and collections to a RESTful interface, thus supporting the archiving of the model state in back-end repositories. In addition, in **WL++** we have added functionality to map models directly to storage locally on the mobile device.

Essential to most non-trivial applications with complex data is the organisation of this data relative to each other. Each model contains domain-specific attributes and functionalities, which are easily modified while maintaining the same structure of relations. In this manner, **WL++** effectively supports the design and development of reusable logical models for application families, rather than reusable individual data structures, which is what the backbone models essentially are.

**Backbone views** are typically associated with **models** (or **collections**) and an HTML element, responsible for rendering the contents of the (collection of) model(s). Views also declare events and the logic to handle those events. Each view is typically associated with an HTML template, and may contain sub-views that independently handle their own content. In the **WL++** architecture, nested views are created for the existing models to be displayed, and typically for the members of a collection.

### 4.3 Application code construction with **WL++**

Backbone uses the underscore.js (http://underscorejs.org/) library to take advantage of its helper functions. The underscore.js library effectively functions as a templating engine that **WL++** uses to generate the HTML to be injected into the document object model (DOM). Typically, in each backbone **view** a template is compiled using the underscore **template** (‘template to be compiled’) command. The template variables (defined within the ‘%’ delimiter), are instantiated with information derived from the model that the developer creates in the **WL++** editor. Then, the data corresponding to models and collections is passed through the templates which are finally injected into the main DOM (Figure 2). **WL++** includes 12 main templates associated with each one of the following components:
• **four views**: dashboard, form view, list view, search view

The views correspond to the pages that a user may navigate to. These views are constructed based on the predefined templates and the application diagram. A template consists of the skeleton of this particular view which then is populated with UI components represented by the widgets specified in the application specification diagram.

• **two core**: client model, UI representation

A model is created based on the model’s attributes added during the construction process of the application diagram. Developers are responsible for defining their own models and specifying their attributes. As these attributes correspond to UI representations, we have defined a file that includes all the representations related to each attribute. For instance, a Text widget is represented by a text input form whereas an Image widget is composed of a box area to be accepted from the picture selected from the device’s library using the upload button.

• **one router**: the application’s router

The router plays the role of the controller; it associates URLs to views in order to allow end-users to navigate from one screen to another. This template specifies the router’s syntax according to backbone where the routes are added based on the navigation arrows among views created through the WL++ graphical editor.

• **two WL adapters**: adapter-impl, adapter-xml

These files are constructed by following the skeleton of a typical Worklight adapter. Besides the default procedures performing CRUD operations, specific parameters are replaced with the proper data retrieved from the application’s diagram. This data consists of the REST service’s URI, port, parameters passed to the REST service, etc.

• **three REST**: server model, REST, database

The server model reflects the client model (backbone model) and is implemented in Java. Therefore, a template representing a Java class is filled with attributes retrieved from the application’s model. Since each attribute is associated with a Java type (e.g., text widget corresponds to a string attribute, a GPS location to a set of two double attributes, etc.) we define these attributes and generate getters and setters methods for each one of them. The Database.java file focuses on the communication of the service with the external database and therefore, there reside the CRUD operations implemented. Finally, the REST URL paths are defined in the REST.java file.

There are also sub-templates related to specific modules such as the local-storage module. These sub-templates are injected into the main templates as needed. For instance, the local-storage template is used for data that needs to be stored into a web-SQL database on the device.

**Figure 2** An example of an underscore template, (a) template to be compiled (b) compiling and adding the template into the DOM while passing the data object as a parameter (c) the HTML added into the DOM (see online version for colours)

```html
<script type=template id="tmpl">
  <div>
    <h2>Name: %data.name%</h2>
    <h2>Age : %data.age%</h2>
  </div>
</script>

var template = 
  _.template($("#tmpl").html());
  $el.append(template(data));
</div>

<h2>Name: Tom </h2>
<h2>Age : 23 </h2>
</div>
```

The output of the WL++ framework is a ‘gen’ folder, which includes all the required application related files. The structure follows a typical Worklight project (Figure 1). This practice allows developers to quickly setup a Worklight project and transfer the files from the generated folder to the corresponding Worklight project, to be deployed.

A Worklight application consists of three types of files:

a. references required for the application development and deployment

b. the application and its adapters

c. server-customisation components.

When a new project is initiated, WL++ generates a generic ‘Hello World’ application, composed of several JavaScript, HTML and CSS files under the ‘common’ folder. The ‘common’ folder also includes files that initiate the Worklight server before any JavaScript code is executed. Finally, WL++ generates files specific to the category of application and its required adapters. In the end, this folder contains all the files shared among the targeted environments (Figure 1), in JavaScript, HTML and CSS. A user may add new environments such as Android or iOS when needed in order to be able to extend or override the common files so the application may look/behave differently while deployed in different platforms. Furthermore, a Worklight project has files responsible for the framework’s initialisation (e.g., the code within the initOptions.js file ensures that the Worklight JavaScript framework is initialised). These files are generated...
automatically by Worklight and imported into the HTML files by our framework, as it happens while creating a new Worklight project.

While Backbone does not impose any special structure, our experience through the development of several prototypes led us to the organisation of the generated files into four separate folders:

1. **libs**
2. **models**
3. **views**
4. **router** (Figure 1).

This practice assists the process of navigation and modification of the proper files when needed. The **libs** folder contains all the required libraries, i.e., the Backbone JavaScript files and the related underscore.js and relational libraries, JQuery/JQueryMobile, libraries supporting the implementation of the user-interface widgets, such as Mobiscroll.js (http://mobiscroll.com/) for the date and time picker component. The **models** folder contains the JavaScript files that describe all the application models. **Collections** are placed in the same file as their corresponding **models**. The **views** folder contains code responsible for the user-interface rendering. Typically, a view is associated with a (collection of) model(s) and renders the relevant information by using specific template components. Finally, the **router** folder contains the URL paths and the functions for creating the proper views according to the URLs. When users navigate from one view to another then the router destroys the previous views and creates new ones by passing the correct data to each view.

A Worklight project may have one or multiple adapters in order to communicate with existing back-ends, which are included in the **adapters** folder. An application that does not require any connection with a back-end, does not require any adapters. The need for a back-end is indicated by the developer when a Worklight Adapter widget is added in the application diagram, developed through the WL++ editor. In this case, two more folders are generated:

a. the **adapters** folder
b. the **restAPI** folder.

The **adapters** folder contains two files: **adapter.xml** and **adapter-impl.js** whereas the **restAPI** folder contains the RESTful web-service related files.

The **adapter.xml** file defines the URL and the port number of the web service, and the names of the procedures (functions) implemented in the **adapter-impl.js** file. Essentially, the **adapter.xml** file plays the role of the controller that invokes the right procedures in response to events triggered on the application side.

In addition to the adapter related files, WL++ generates the **restAPI** folder which has multiple Java files in order to be injected in a web service project. The code is responsible for the communication of the application (through the adapter) with a remote database through a set of RESTful APIs. The APIs correspond to basic CRUD and few filtering operations (search by specific attributes). More specifically, the first time the mobile application is launched, a remote database is created with two tables, one for storing the data entered and one for API monitoring purposes. When new data is entered, the corresponding procedure is called in order to transmit the data to the server; therefore, the web service stores this information in the database while the monitoring service keeps track of which API calls and what data was sent throughout the application – back-end communication.

### 4.4 The logical model of WL++ applications

The development of application families through the adoption and refinement of architecture frameworks in specific contexts is an established software-reuse practice. In principle, the objective in designing and developing application frameworks is:

a. to implement general behaviours that many application instances share in reusable components
b. to design a process for systematically and consistently specialising and integrating these components in the context of specific applications.

The MDE methodology advocates the definition of abstract models, capturing the behaviours of a family of related applications, and their systematic transformation into executable and deployable assets (Kent, 2002; Schmidt, 2006). Following this paradigm, the process of generating a software system is faster and less error-prone as the code generated has been previously tested. Furthermore, as models represent abstractions of parts of the systems, it is much easier to be understood by users/developers. According to Weigert and Weil (2006) the application of MDE methods has led to multiple benefits such as productivity and quality improvement. By concealing low-level code details through the use of models results into more stable and reusable models throughout the system. In addition, inspections on the generated code have become sparser due to the less error-prone auto-generated code. In this section, we review the logical model underlying the applications produced by our WL++ framework.

**WL++** supports the development of multi-featured catalogue-style applications. Users of this broad application family can record, and later update and/or delete, entries of different types. These entries are stored on the device, and can also be archived at the server side through a RESTful interface for remote storage (Christensen, 2009). Past entries can be summarily displayed as a list, filterable based on properties of the data model, with each list item linking to an entry editor. This architecture provides three variability points:

a. the attributes of the entry models
b. the view templates consisting of different interactive-widget combinations
c. the number of back-end services.
Limited coupling between the components of the system allows for views to be rearranged and customised as necessary. The logical model of the common architecture is presented in Figure 3. Logs are groupings for entries, and the collection of all logs capture the complete record of a user’s activity. Logs may contain multiple types of entries, a scenario typical in cases where there is a variability in the contents of the recorded entries.

Figure 3 The logical model for the application family

Source: Stroulia et al. (2013)

Figure 4 depicts the conceptual model of WL++ applications; instances of this model, corresponding to individual applications, are specified by the developer in the application specification diagram editor of the WL++ plugin. The application model consists of the main entry model (JSModel) which may be composed of zero or multiple attributes (of type attribute). A JSModel class is characterised by the model’s name, a list of attributes selected from the available widgets and the template that will generate the output JavaScript file [Figure 4(a)]. The output model file follows the Backbone model’s structure [Figure 4(b)].

A JSView entity is associated with a specific (collection of) model(s) and is responsible for the HTML-based templates which a particular view expects to be appended in the DOM element [Figure 4(c)]. The output view file follows the Backbone view’s structure and includes the template’s initialisation and the handling of the application events. For example, an event may be a button click or a change of an attribute value; these events are handled by the view through relevant functions. Also, the render() function renders the template by adding specific data depending on the model and the events [Figure 4(d)].

Typically a single-page application has only one router. Therefore, the router folder contains a single file, which is responsible for the navigations among the application’s views. The navigation between a FormView and a ListView happens as follows: when a new entry is added, i.e., when new data entered through the FormView, a navigation route that connects the FormView with the ListView is automatically generated. This route effectively enables navigation from a FormView to a ListView each time the ‘save’ button located in the header of a FormView is clicked. Furthermore, the Backbone library keeps track of the browsing history (hash changes). Therefore, a default ‘back’ button located on the left side of the header similarly to the back of a typical browser takes advantage of the Backbone hash-change functionality, supporting the navigation from a page to the previous visited one. For instance, since a FormView leads to a ListView, users can also navigate back to the FormView and/or the MainView (aka dashboard) through the ‘back’ button. The router class is composed of a list of routes and their corresponding functions [Figure 4(e)]. The output router according to backbone is generated following the structure in Figure 4(f).

The components of the WL++ conceptual model are mapped to the physical architecture (Figure 1) as follows: the JSModel is mapped to the files residing in the models folder whereas the JSView components in the views folder. In the future, we plan to extend the type of views besides the current ones (FormView, ListView).
5 Developing WL++ applications

WL++ comes with a simple and intuitive graphical UI that gives developers the ability to build their multi-platform applications through the creation of application specific diagrams. As the IBM Worklight tool is a framework installed within the Eclipse environment, we decided to use the GMF (http://www.eclipse.org/modeling/gmp) framework, which can also be installed within Eclipse in order to avoid any discrepancies among platforms while generating and building our applications. GMF supports the development of graphical editors based on two meta-models: Ecore and Genmodel. We first defined our components in the Emfatic language (http://www.eclipse.org/epsilon/doc/articles/emfatic/) and then generated the Ecore metamodel. Next we used the EuGENia tool (http://www.eclipse.org/epsilon/doc/eugenia/) that automatically generates the components needed to implement the GMF editor. The generated project is an Eclipse plugin project that can be exported and installed as such.

Figure 5  The Eclipse plugin’s UI (see online version for colours)

Notes: On the right is the palette containing the available widgets and connections. On the left is an example diagram designed by using the existing widgets.

The UI of the WL++ plugin consists of an editing canvas where developers are able to drag and drop components from a palette of widgets (Figure 5). The widgets were defined following the Backbone structure (Figure 6) and also based on HTML elements. As explained in Section 4.2, a model is the core data object in Backbone and therefore in our framework as well.

With that being said, there is a mapping between the models defined within the framework and the generated Backbone models. As a result, the developer starts by defining a model and then enriching it with attributes. Each widget, while dragged into the editing area, can be accompanied by a label defined by the developer. Essentially, most of the attributes correspond to HTML components in the final application UI. Currently, we have a basic set of attributes that are commonly used in mobile applications.

- The text widget can be added as an attribute to a model. It has a label property, which is defined by the user during the creation of the application diagram. This widget appears as a text field where end-users may enter their textual input.
- The numeric widget similarly appears as a label along with a numeric input field, accepting numerical values as input. This type of widget would be appropriate in cases of telephone numbers, product prices, age, etc.
- The password widget appears as an input box on the mobile application UI; however, the typed symbols are hidden and appear as stars. This widget may be useful in cases of user registration/login forms.
- The location widget can be used to indicate that an attribute is a location represented by a pair of longitude-latitude coordinates. This widget accesses the device’s GPS through the HTML5 Geolocation API and stores the latitude-longitude values in the mobile application local database (and external database if necessary).
- The image widget appears as an image box along with a button to upload images by selecting pictures from the mobile device’s picture library. It could be easily expanded into a widget accessing the device’s camera in order to capture picture/video instances.
- The timestamp widget results in a date-time picker in the application UI, implemented with the Mobiscroll (http://mobiscroll.com/) JavaScript library.

Figure 6  The structure of the mobile application based on the structure specified by Backbone

As the attributes are single components there is a need for form-like widgets to indicate how models composed of the above attributes should be displayed on the device’s screen. We have defined two such widgets: the FormView and the ListView.
WL++: a framework to build cross-platform mobile applications and RESTful back-ends

• FormViews are responsible for displaying the attributes contained into the model on the device screen. Through this view users are able to enter their entries and save them locally or remotely.

• A ListView serves as a separate screen where all the existing entries are displayed. A menu item is also added into the main screen from where users can navigate to the ListView and view their entries in a list form. Moreover, by clicking a list item, it gives the users the ability to edit/delete an entry.

Using the above widgets, a developer is able to construct a functional cross-platform application that would be able to run locally. The Worklight Adapter widget, when included in the application-model diagram, results in the generation of a back-end. The Worklight Adapter and its use will be explained in more detail later on.

Finally, there are three connection links namely HasAttribute, HasView and HasWLAdapter. These links represent connections among models and the rest of the diagram components. WL++ is aware of the link semantics and appropriately restricts what component can be linked to another. For example, the HasView link cannot have as starting point a Model component and ending point an Attribute. The HasView link is only able to connect a Model to a FormView or ListView component. Similarly, the HasAttribute can be drawn only between a Model and an Attribute.

Figure 7 An example of an XML file corresponding to the sample UML diagram depicted in Figure 5 (see online version for colours)

When the application developer saves the diagram, an XML file is automatically generated. As we can see in Figure 7, the XML file is a representation of the diagram. In particular, it includes all the information regarding the models, attributes, their representations, connections and labels depicted on the diagram. At this point, our framework parses the generated XML file in order to identify the components based on which the cross-platform application and the back-end will be generated.

The parsing starts when the developer clicks the ‘Parse XML file’ button located at the bottom of the palette. At the end of the parsing phase, then the ‘Generate Code’ button is clicked to generate the application files including the code necessary to produce the mobile application along with the back-end (Worklight Adapters and RESTful API in case where the Worklight Adapter has been added in the application diagram).

5.1 The generated back-end

Along with the mobile application that is deployed on a mobile phone and runs locally, the WL++ framework generates a RESTful API through which the mobile application can send/receive data to/from external databases. This communication happens through adapters within the Worklight framework. More specifically, Worklight offers developers the option to choose among four different kinds of adapters:

a. HTTP adapter
b. SQL adapter
c. Cast Iron adapter
d. JMS adapter.

The HTTP adapter is used in cases where RESTful or SOAP services need to be invoked whereas the SQL adapter is used when there is not a middleware between the application and the external database. Therefore, it is developed to execute SQL queries in order to perform operations such as insert, update, delete, etc. Finally, the Cast Iron adapter can be used to retrieve data from enterprise data sources and the JMS adapter allows the communication with a JMS-enabled messaging provider. WL++ uses the HTTP adapter to support the invocation of RESTful APIs. The output layers of our proposed framework are depicted in Figure 8. As mentioned before, only when the widget of the adapter is chosen during the creation of the application diagram, the back-end is generated. The web service that is created based on the generated back-end related files is a Java web service implemented using the Jersey (https://jersey.java.net/) framework.

5.1.1 Worklight adapters

The WL++ framework generates the files corresponding to the HTTP adapter since our goal is to invoke a RESTful service through a pre-defined set of APIs.

To that end, we developed a generic adapter through the Worklight interface which is customised based on the mobile application to be generated. The HTTP adapter is
deployed by the server that is built-in within Worklight (see top layer of Figure 8). The communication between the application and the back-end would be also possible by using AJAX requests. Although the Worklight adapters also use AJAX in the background, the adapters provide some advantages such as security and transparency of the retrieved data among others.

Figure 8 The different layers of the auto-generated components (see online version for colours)

5.1.2 RESTful service and supported operations

Upon generating the application related files, the next step involves the generation of the RESTful service, when the developer chooses the application to be generated along with a back-end. If the user selects an HTTP adapter then a RESTful service is automatically generated. Otherwise, a cross-platform mobile application is produced that runs locally without a supported backend. An HTTP adapter can of course be added manually later on. The REST APIs supported by our framework implement the CRUD operations and filtering operations relying on the attributes of the application models. Filtering is often useful when we have to deal with long lists of entries. For example, a shopping catalogue that contains a large number of different products would not be possible to be reviewed in order to find a specific product quickly.

5.1.3 The monitoring service

A fairly typical characteristic of the latest mobile applications is that of ‘push notifications’. As users interact with the application by storing/receiving data, the application often responds through notifications, typically containing information extracted based on the user’s interaction history or that of his/her peers. For this purpose, the WL++ framework includes a monitoring service which lives on the server side and keeps track of all the API calls that have been made by storing the API call and the data exchanged between the application and back-end. It also records the data exchanged between the mobile application and the back-end server in a JSON format. This data can be analysed and used in order to make recommendations by sending notifications from the server to the mobile device. Currently, this feature is under development.

5.2 An illustrative example

In this section, we explain in detail the steps to be followed in order to generate a Contact Manager application along with RESTful back-end service, by using our proposed framework. This type of application has been already built by developers multiple times as it is considered as an application that includes many features and illustrates successfully the use of new frameworks/technologies. We found plenty of implementations in public code repositories and personal blogs (OpenDJ-Contact-Manager, 2013; Mobile-Spot-D-Project, 2013). For this particular example, we were influenced by a similar application (Coenraets, 2011), which was developed on the Phonegap framework.

Before starting the designing of the application diagram (see Figure 5), the developer has to decide the topmost model that represents the application to be generated. In this case the main model is the contact manager. To indicate this, the developer selects the model widget from the palette, drags it to the editing area and labels it ‘contact manager’.

The second step is to decide which are the attributes of the ‘contact manager’ model. It should be noted that there is no limit on the number of attributes to be added. As a result, the developer adds several properties related to this application (i.e., name, last name, phone number, e-mail and picture). For the properties name, last name and e-mail the text widget was selected as it represents a text field on the application’s graphical UI whereas for the Phone Number attribute we selected the numeric widget. The latter appears as a numerical field on the UI (i.e., it only accepts numerical values). Finally, for the picture attribute the developer needs to drag and drop the image widget. This widget is composed of an image box and a button to select images from the mobile device’s image library as mentioned before. A birthday property was also added represented by a timestamp widget. The order in which the aforementioned attributes were added to the model is the same as the one they appear on the mobile device form view. In order to change the order of the properties (for example, to make the property e-mail appear before the
property phone number), the developer needs to modify the application diagram and regenerate the application. Alternatively, the generated code included in the HTML file could be modified to achieve the same results.

Next, the developer drags and drops a FormView widget and connects it to the model through the connection HasView. The stored contacts are to be displayed in a list view, so the developer also adds a ListView component to the model.

**Figure 9** Instances of the generated cross-platform mobile application (see online version for colours)

So far, we already have a cross-platform mobile application that can be deployed locally on the mobile device storing our contacts' information on a Web SQL database (local storage). As mentioned before, by adding a Worklight Adapter widget to the application diagram, a back-end service is generated. In fact, along with the adapter’s related files, a RESTful-service that not only supports CRUD and filtering operations but also creates the database’s structure is generated. The database structure is defined by our model’s attributes. For example, in our case the database’s table would be composed of seven columns namely id, name, last name, phone number, e-mail, birthday and picture.

Finally, to generate the source code, the developer first clicks the ‘Parse XML’ button and then the ‘generate code’ one. The ‘gen’ directory is then generated; this directory contains the files needed for the mobile application to be deployed on a mobile device and also the adapter’s and RESTful service’s files.

Currently, one needs to copy the application files to a Worklight project and the ones related to the RESTful web service to a web-service project. Since through the Worklight framework we can add as many mobile platforms as we desire, it is easy to export installation files in order to deploy the application on real mobile devices. For instance, by adding an Android environment, Worklight automatically generates an Android project (Android SDK must be installed before the environment is generated), and thus, we can export an installation file (apk) which we could install on a real device or upload it on the Android marketplace. In order to generate an iPhone application, the installation of the Xcode IDE is essential. Therefore, the effort of a developer that needs to run the generated application in different platforms has to do with the generation and extraction of the installation files specific to each platform. Upon the web service is running, we are able to access the external database through the APIs that we have defined. Each API call is monitored by the monitoring service that keeps a record of it on a table of the external database. The specific API and the data sent through the service are stored in a JSON format. This data may be used for recommendation purposes. For instance, by establishing a connection between the generated application and an existing shopping service, the birthday attribute could be used to send a notification of buying a specific gift that this person has favoured and which is currently on sale.

Figure 9 shows three different screenshots of the generated application. The first two represent the data entry screen (FormView) whereas the third one the ListView where all the contact entries are displayed. As can be seen from this example, generating a fully functional mobile application accompanied by a RESTful service can be accomplished through modelling the application’s entities by dragging and dropping actions of predefined widgets selected through a palette.

**Figure 10** Comparison of the employee directory app and contact manager app in terms of lines of code (LOC)

### 5.3 Evaluation

As we have already mentioned, the example Contact Manager application through which we illustrated WL++’s operation was inspired by a similar application we discovered in GitHub. In Figure 10, we compare the Contact Manager application generated by WL++ with the Employee Directory [a Phonegap application developed manually by Coenaets (2011)]. The employee directory inserts a list of employees in a list by executing multiple times the SQL insert command. Then it allows us to browse this list and/or search for employees based on their name/last name. Finally, it displays specific information associated with an employee (e.g., phone number, e-mail, etc.). On the other hand, the contact manager allows the insertion/update of contacts which then are displayed in a list. Although contact manager offers additional functionality, both applications have a high degree of similarity and share the same architecture (using Backbone.js as an MVC framework). Thus, we compare these two applications in terms of lines of code (LOC) for the different file categories. In total the
Employee Directory is composed of 1,344 LOC whereas the Contact Manager application 753 (excluding the LOC belonging to the back-end as the employee application does not have a back-end implementation). Below follows an explanation of the categories that we split the files in:

- **HTML**: This category includes the LOC of all the HTML files. Both applications are single page applications and therefore, there is one HTML file which contains the most essential templates/library imports that are always present in every application.

- **Models**: The employee directory application architecture is very similar to the contact manager as they both leverage the Backbone.js library to separate the UI from the application logic. Therefore, we compared the LOC of the model defined on the employee directory (model: employee) over the LOC defined on the contact manager (model: contact).

- **Views**: In the case of the employee directory app, there is a single file representing the views of the application which consists of 202 lines of code. On the other hand, contact manager has three separate files representing each view. WL++ generates each view in different file in order to make the maintenance task much easier. The LOC in the case of the contact manager application is in total 196.

- **Other JavaScript**: There are additional JavaScript files in both applications. The employee directory handles the communication with the devices database by doing a manual construction of SQL queries in order to fetch/store data in the database. However, for this purpose, WL++ uses the backbone-WebSQL (https://github.com/MarrLiss/backbone-websql) library, which is responsible for the retrieval/storage of the data. Furthermore, in this category, we have included the first script that instantiates all the models, collections, views and the local database.

- **Templates**: Although there are multiple templates defined, we counted the LOC of the templates that were used for this application. More specifically we computed the LOC by subtracting the LOC of the initial HTML file (before the code generation) and the HTML file (after the code generation). By doing this, we know how many LOC have been added in the initial HTML file and therefore, how many LOC of the templates have been used for this specific application. In our case there were 21 LOC added. On the other hand the employee directory has five templates (167 LOC in total), which are used when users navigate from one screen to another.

- **CSS**: We have defined our CSS file as a static file, which is the same for every application to be generated. This file is copied in the CSS folder of the generated application the same way the libraries are copied to the lib folder. This way, more CSS files can be added and switched according to the developer’s preferences. In our case, the CSS file has 272 LOC compared to the employee directory that has 449 LOC.

- **WL adapters and restful web service**: Although the employee directory does not have a Back-End, we have included the WL-adapters and the RESTful service back-end in the chart in order to compute the total LOC generated by WL++. Since the back-end is composed of the adapters and the web-service, in total there are 603 LOC.

### 6 Conclusions and future directions

In this paper, we presented the WL++ framework built as an Eclipse plugin that leverages MDE techniques to create cross-platform mobile applications that can be conceived as catalogues of structured descriptions. The major characteristics of the WL++ framework are as follows.

- Through a set of available widgets, developers are able to create application diagrams by using the WL++ editing canvas and generate mobile applications, without writing any code.

- The Backbone framework enforces a specific structure (by separating the UI from the business logic) to the generated code. Thus, developers are able to modify and maintain the code efficiently.

- Developers have the option of deciding whether they want their application to be connected to external databases or not. By selecting the Worklight Adapter widget, a RESTful API is automatically generated based on the model and its attributes. In addition to the basic CRUD operations, the API supports filtering by specific attributes, and the creation of the database’s structure.

In the future, we aim to add more essential widgets (e.g., checkboxes, radio buttons) and also integrate and allow the design of more complicated data models (nested models, models that inherit attributes from other models, etc.). Also, we plan to be able to allow developers to specify the navigation among the views and therefore construct the routing component based on developer-specified navigations as well as allow developers to define their own events and custom functions. Finally, we would like to be able to fill the gap between existing RESTful APIs and cross-platform applications. More specifically, we plan to systematically generate mobile applications based on existing API services by integrating several web services that fell under the same domain.

### Acknowledgements

This work has been funded by NSERC, AITF and IBM CAS Research Canada.
References


