FORWARD: Data-Centric UIs using Declarative Templates that Efficiently Wrap Third-Party JavaScript Components*

Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, Erick Zamora
{y4fu,kianwin,yannis}@cs.ucsd.edu, ezamora@ucsd.edu
UC San Diego

ABSTRACT

While Ajax programming and the plethora of JavaScript component libraries enable high-quality UIs in web applications, integrating them with page data is laborious and error-prone as a developer has to handcode incremental modifications with trigger-based programming and manual coordination of data dependencies. The FORWARD web framework simplifies the development of Ajax applications through declarative, state-based templates. This declarative, data-centric approach is characterized by the principle of logical/physical independence, which the database community has often deployed successfully. It enables FORWARD to leverage database techniques, such as incremental view maintenance, updatable views, capability-based component wrappers and cost-based optimization to automate efficient live visualizations. We demonstrate an end-to-end system implementation, including a web-based IDE (itself built in FORWARD), academic and commercial applications built in FORWARD and a wide variety of JavaScript components supported by the declarative templates.

1. INTRODUCTION

Ajax programming and JavaScript component libraries have led to a new generation of modern web applications, which are characterized by user interfaces commensurate with desktop applications. This is achieved by the developer handcoding performance optimizations for *incremental modifications* from the old page to the new page [4].

Consider the running example in Figure 1, which shows a web application providing faceted browsing over earthquake data. The page shows a summary of total earthquakes displayed, a table of earthquake details, and a map where each marker represents the earthquake's location and magnitude. In response to the user selecting/deselecting checkboxes for magnitude ranges, the page displays only earthquakes that satisfy the magnitude filter.

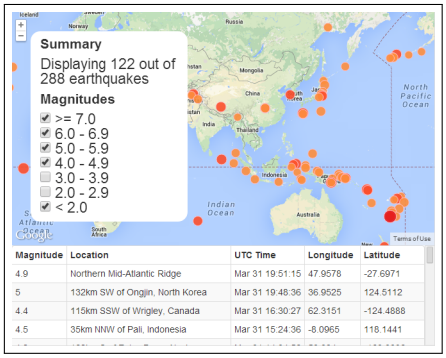

Figure 1: Running Example: Faceted Browsing of Earthquakes

```
{
 quakes: [
  {
   _key: 456,
   mag : 2.5,
   lat : -150.3,
   lng : 63.2,
   loc : '74km..',
   time: 'Mar..'
  }, ...
 ],
 filter: [
  {
   _key : 1,
   range: '>= 7.0',
   selected: true
  }, ...
 ]
}
```

Figure 2: Page State

Using mainstream web frameworks such as Ruby-on-Rails and Backbone, developers already modularize code following the Model-View-Controller (MVC) architectural pattern. The *model* (or *page state*) is an abstraction that represents the logical data used by user interfaces. For example, Figure 2 shows the example's page state in JSON format. Notice that the page state stores earthquake data once in `quakes`, even though the data is utilized thrice in the summary, table and map. Furthermore, the page state also contains the results of user interaction: `filter` stores user input on whether a range is selected, whereas `quakes` stores earthquakes that satisfy the filter.

Despite the page state abstraction, incremental modifications still occur in tedious and error-prone ways: (1) JavaScript components (maps, calendars, tabbed dialogs etc.) encapsulate their complex state by exporting programmatic APIs comprising methods and event handlers. When page state changes (e.g. displaying more earthquakes), calling a method (e.g. adding a particular marker on the map) incrementally re-renders the component as a side-effect. Conversely, when user interaction changes a component's state (e.g. deselecting a checkbox), an event handler fires and triggers code to change the page state. Effectively, a developer engages in event-driven / trigger-based programming to laboriously translate from page state changes to component state changes, and vice versa. (2) Due to trigger-based programming, the developer has to manually assess data flow dependencies on the page, in order to correctly transition the application from one consistent state to another. For example, when page state changes with the addition of earthquakes, calling methods to refresh the

*Supported by NSF III-1018961 and NSF III-1219263, PI'd by Prof Papakonstantinou who is a shareholder of App2you Inc, which commercializes outcomes of this research.

Proceedings of the VLDB Endowment, Vol. 7, No. 13

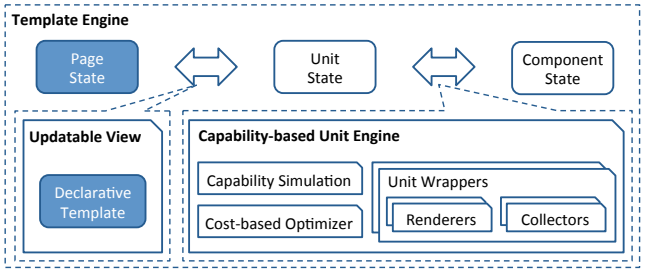

Figure 3: Architecture of Template Engine

```
<% unit google.map.Maps %>
 {
  markers : [
   <% for q in quakes %>
    {
     position: {
      lat: <%= q.lat %>,
      lng: <%= q.lng %>
     },
     size : <%= siz(q.mag) %>,
     color: <%= col(q.mag) %>
    }
   <% end for %>
  ]
 }
<% end unit %>
```

Figure 4: Template for Map

```
{
 markers : [
  {
   _key  : 456,
   position: {
    lat  : -150.3,
    lng  : 63.2,
   },
   size  : 3.5,
   color : '#FDD49E'
  }, ...
 ]
}
```

Figure 5: Unit State for Map

```
<% bind filter each f %>
 <% unit html.Checkbox %>
  {
   checked:
    <% bind f.selected %>,
   label: <%= f.range %>
  }
 <% end unit %>
<% end bind %>
```

Figure 6: Template for Checkboxes

```
{
 _key   : 456,
 checked: true,
 label  : '>= 7.0'
}
```

Figure 7: Unit State for Checkbox

map/table, but not the summary, results in a buggy page that is inconsistent. Such data dependencies cause incremental modifications to be very error-prone. (3) Vibrant activity among commercial vendors and open-source communities has led to numerous JavaScript libraries. As a data point, popularity tracker JSDB.io lists over 600 libraries. Since these libraries are varied and do not utilize the same API conventions (unlike HTML/DOM, which has been standardized), developers who integrate multiple libraries have to familiarize themselves with subtly different syntax and semantics for incremental modifications.

The database community has emphasized new opportunities in applying data-centric approaches, including data independence and declarative programming to other emerging language platforms [1]. In the same spirit, we present the FORWARD web framework [4, 5], which provides a declarative, state-based language for developers to create web applications. Declarative programming enables the system to leverage database techniques, such as *updatable views*, *capability-based wrappers* [6] and *cost-based optimization* to automatically propagate incremental modifications between page state and component state, and vice-versa. FORWARD thus extends prior declarative web frameworks such as Strudel [3] and Hilda [8], which have focused on modeling page state as queries, but do not perform incremental modifications. More recently, Model-View-ViewModel (MVVM) web frameworks such as AngularJS [2] and Knockout [7] have also employed declarativeness in utilizing HTML templates to automate incremental modifications of the browser DOM. FORWARD's novel template language supports HTML too, but goes beyond to also integrate the rich functionality of third-party JavaScript component libraries.

2. TEMPLATE ENGINE OVERVIEW

In this paper, we focus on the *template engine* of the FORWARD web framework, which automatically propagates changes between page state and component state, and vice-versa. The template engine is independent of the modules which output page state: FORWARD currently computes page state through a middleware query processor which evaluates SQL++ queries over a virtual database [5], but details of the query processor are beyond the scope of this paper.

Figure 3 presents the architecture of the template engine, where shaded blue boxes denote inputs by the developer, and white boxes denote data structures and modules provided by the template engine. In particular, a capability-based *unit wrapper* adapts a third-party JavaScript component into the template engine architecture, and is key in establishing independence between logical state and physical methods/event handlers. Section 2.1 shows examples of the declarative template language, and illustrates its updatable view semantics for bidirectional synchronization between the page state and unit state. Section 2.2 presents how capability-based unit wrappers perform bidirectional synchronization between unit state and component state. Section 2.3 presents how the logical/physical independence between the unit state and component state enables the template engine to simulate missing capabilities and perform cost-based optimizations. Section 2.4 illustrates how developers can specify renderers and collectors directly within a declarative template, thus enabling more lightweight integration than implementing fully-capable unit wrappers.

2.1 Declarative Templates as Updatable Views

Figure 3 shows that the template engine takes two inputs: (1) the page state, which is represented in the JSON data model. For efficiency of propagating changes bidirectionally across page state, unit state and component state, it is recommended (but not required) that each element of an array is identified by a *key* that remains immutable across page refreshes. A common and simple solution is to generate such keys from the primary keys of data stored within databases. For example, keys are encoded as special `_key` attributes in Figure 2. (2) the *declarative template*, which specifies how the page should be displayed in the browser. For example, Figure 4 and 6 show the declarative templates for the map and checkboxes respectively. Notably, a declarative template allows JavaScript components to be used without writing any integration JavaScript code.

The *unit state* represents the state of a unit wrapper. For example, Figure 5 shows the unit state instantiated by the declarative template of Figure 4, using the page state of Figure 2. Analogously, Figure 7 for Figure 6. Using the SQL++ query language which inputs and outputs JSON data [5], the semantics of the declarative template specifies the unit state as a SQL++ *updatable view* over the page state. Intuitively, instantiation of the unit state is easily expressed as a view: the `for` directive iterates over elements of an array (e.g. `quakes`), and for each element instantiates the value enclosed within the directive's body (e.g {...}); the `=`

directive instantiates the result of an expression. More subtly, a `bind` directive specifies a one-to-one mapping between a pair of values in the page state and unit state, such that an update on one value is propagated to the other. For example, deselecting a checkbox updates its `checked` attribute to `false` in the unit state (Figure 7), which in turn updates the corresponding `selected` attribute to `false` in the page state (Figure 2). A complete listing of template directives is presented as part of FORWARD's programming model [5].

2.2 Capability-based Unit Wrappers

Whereas a declarative template utilizes state, the programmatic API of a component utilizes incremental modifications. A unit wrapper translates between these two fundamentally different programming models through logical capabilities as follows.

Changes in the unit state are represented by a list of *diffs* on JSON data. Each diff is described by (1) an *op* which describes how data is changed: `insert`, `update` or `delete` (2) a *context path* which describes where the data is changed (3) a *payload* which describes the replacement data. For example, when an updated filter condition causes the map marker with key `456` to be removed from the map, the diff has op `delete`, context path `^.markers#456#`, and no payload. `^` denotes the root of the unit state, and `#...#` denotes keys. Notably, the context path navigates into array elements with keys if they are present, and ordinal positions otherwise.

A unit wrapper contains *renderers* and *collectors* (Figure 3). A renderer has (1) a *diff signature* comprising an op and context path. For example, the diff signature which matches the diff mentioned above has op `delete` and context path `^.markers#*#`, where `*` is a wildcard matched by any key value. Special ops are also allowed in diff signatures: `construct`/`destruct` to create/destroy a unit, and `attach`/`detach` for a parent unit to attach/detach a child unit. (2) a *rendering function* which inputs a diff matching the signature, and calls the incremental rendering method of the underlying component. For example, the renderer mentioned above has a rendering function that calls the map component's method to remove a particular marker. Thus, a renderer's rendering function propagates changes of the unit state into the component state. In addition to bridging the gap between the two programming models, a renderer effectively provides logical/physical independence: the diff signature describes its logical capabilities, while the function performs the physical rendering. Section 2.3 presents further optimizations arising from this logical/physical separation.

Conversely, a collector propagates changes of the component state into the unit state: it provides an event handler function, and when a component event fires, the function applies a corresponding diff on the unit state.

2.3 Simulating for Missing Capabilities and Cost-based Optimizations

A unit wrapper is required to handle all possible diffs on its unit state, but its underlying component may not support incremental rendering functions that are sufficiently fine-grained. For example, the Google Maps API offers a `setPosition` method for updating the position, but there is no method for updating only latitude or longitude.

As a result of renderer capabilities being described logically with diff signatures, the unit engine supports simulation of missing capabilities (Figure 3) as follows. Each renderer is required to declare the *maximally applicable* diff signature that is supported by its rendering function. Or equivalently, the diff signature with the fewest steps in its context path. For example, the renderer which calls `setPosition` has diff signature `update` and `^.markers#*#.position`. Given a diff, the unit engine uses *simulation rules* to find one or more renderers with the most specific diff signatures. Any renderer can be simulated by an `update` renderer of an ancestor attribute, while an `update` renderer of an array element can also be simulated by a combination of `insert` and `delete` renderers on the same array element. For example, an `update` diff on the `lat` attribute is simulated with the `setPosition` renderer. As another example, if the `setPosition` renderer were not supported by the unit wrapper, the diff is simulated with two renderers: removing the marker, and re-adding the marker. Since a unit wrapper supports at least two renderers for constructing and destructing the component, any diff can always be simulated by the base case of re-constructing the component.

Besides supporting missing capabilities, simulation also provides optimizations based on a cost model that considers the number of diffs and rendering time. For example, suppose many markers are added and removed from the map. Given a sufficiently large number of additions/removals, re-constructing the map is more efficient than separately adding/removing each marker.

For the cost model, each renderer in a unit wrapper is optionally specified with a *rendering cost* (defaults to 1), which indicates its cost relative to other renderers within the same unit wrapper. The implementor of a unit wrapper is responsible for empirically determining the respective costs.

```
1.  <script src="https://maps.googleapis.com/..” />
2.  <script src="my-maps.js" />
3.  <% unit custom
4.       render
5.         construct using constructMap,
6.         destruct  using destructMap
7.       collect using collectMap
8.  %>
9.   {
10.   <% render
11.        insert using insertMarker,
12.        update using updateMarker,
13.        delete using deleteMarker
14.   %>
15.     markers : [ ... ]
16.   <% end render %>
17.  }
18. <% end unit %>
```

Figure 11: Template for Lightweight Integration

2.4 Lightweight Integration

A unit wrapper encapsulates renderers and collectors for code reuse across applications. For cases where code reuse is not necessary, such as one-use infographics implemented using vector libraries such as D3 and Three.js, a template supports lightweight integration of custom JavaScript code. Suppose a unit wrapper for Google Maps has not been implemented. Figure 11 shows an example of integrating a map component inline within the template. The Google Maps library is included via a conventional HTML `script` tag (line 1). Custom JavaScript code for rendering and collecting functions are included via `my-maps.js` (line 2). The special `custom` unit wrapper (line 3) accepts a `render` parameter for specifying `construct`/`destruct` rendering functions (lines 5-6), and an optional `collect` parameter for specify-

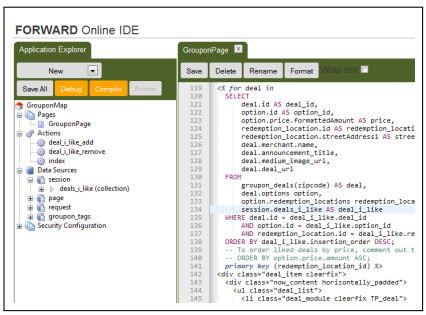

Figure 8: IDE Application for Cloud-based FORWARD

Figure 9: Example Application: Groupon + Google Maps

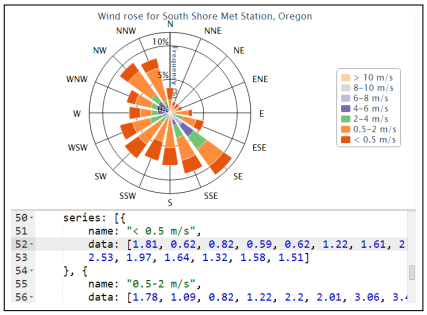

Figure 10: Interactive Debugger for Unit Wrappers

ing a single collecting function. The collecting function is invoked whenever user interaction occurs, and outputs the entirety of the unit state.

For ease of development, it is sufficient to specify only `construct`/`destruct` rendering functions, as the template engine simulates missing capabilities. For visualizing a large number of data points, a developer can optimize rendering time by specifying additional rendering functions using `render` directives. For example, lines 10-14 specify functions for inserting, updating and deleting a particular marker.

3. DEMONSTRATION

The demo showcases a full implementation of the template engine within the FORWARD web framework, which includes a library of 20 pre-built unit wrappers. 14 commercial and academic applications have also been built using the web framework, of which 3 representative ones will be shown. We present the demo proposal below, and highlight how the demo elucidates the template engine architecture.

Applications Figure 8 shows the IDE application (itself built in FORWARD), which is used to edit, compile and execute FORWARD applications that are hosted in a cloud-based service. The IDE application is used to present the declarative template (about 300 lines) needed to create the entirety of the example application in Figure 9. The example application retrieves current deals from Groupon's REST web service, and visualizes them on a Google Map component. We invite the audience to make changes to the example application, thereafter compile it and see the new application deployed immediately. We also explain the underlying updatable view semantics using the declarative template as a concrete example.

To validate the real-world practicality of the system, we also demo BioHeatMap, an analytics application currently utilized by two pharmateutical companies. BioHeatMap provides a dashboard for users to pose analytic queries over 20M PubMed articles, and displays results with rich visualizations such as time series charts and heat maps.

Unit Wrappers To demonstrate the wide applicability of unit wrappers, we present 20 unit wrappers from a variety of mainstream JavaScript libraries: (1) Standard UI component libraries, in particular Bootstrap, which provides general-purpose components such as dialogs, tabs, menus, progress bars and sliders. (2) Charting libraries, in particular Highcharts and Highstock, which provide chart components such as pie/line/bar/column charts, polar charts, scatter/bubble plots and box-and-whisker plots. (3) Map libraries, in particular Google Maps and Leaflet, which provide geographical and street maps. (4) Editor libraries, in particular Ace and CodeMirror, which provide customizable code editors supporting syntax highlighting, undo/redo, code folding and code completion.

The unit wrappers are presented using two interfaces: (i) a gallery of examples, which illustrates the wide range of unit wrappers (ii) an interactive debugger, as shown in Figure 10, which illustrates how renderers/collectors synchronize the component state and unit state. When the user changes in the editor a JSON value that represents the unit state (bottom pane), the component is refreshed (top pane). Conversely, when user input occurs in the component (e.g. showing/hiding a data series by clicking on the legend), the JSON value in the editor changes correspondingly.

Lightweight Integration To demonstrate the lightweight integration of arbitrary JavaScript components, we showcase declarative templates that incorporate custom D3 visualizations. An initial template contains only `construct` and `destruct` renderers, thus simulation results in the entire visualization being re-drawn whenever the page state changes. Then, we progressively enable other `insert`, `update` and `delete` renderers, thereby optimizing the rendering performance. Through this demo, we illustrate how simulation enables the developer to tradeoff coding time and runtime performance in a flexible manner.

4. REFERENCES

[1] R. Agrawal et al. The claremont report on database research. *Commun. ACM*, 52(6):56–65, 2009.

[2] Angularjs. `http://angularjs.org/`.

[3] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with Strudel. *VLDB J.*, 9(1):38–55, 2000.

[4] Y. Fu, K. Kowalczykowski, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Ajax-based report pages as incrementally rendered views. In *SIGMOD Conference*, 2010.

[5] Y. Fu, K. W. Ong, and Y. Papakonstantinou. Declarative ajax web applications through sql++ on a unified application state. In *DBPL*, 2013.

[6] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, pages 276–285, 1997.

[7] Knockout. `http://knockoutjs.com/`.

[8] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *ICDE*, page 32, 2006.