# The SQL-based All-Declarative FORWARD Web Application Development Framework * †


Yupeng Fu
UC San Diego
yupeng@cs.ucsd.edu

Kian Win Ong
app2you Inc.
kianwin@app2you.com

Yannis Papakonstantinou
app2you Inc. / UC San Diego
yannis@cs.ucsd.edu

Michalis Petropoulos
SUNY Buffalo / UC San Diego
mpetropo@buffalo.edu


## 1. INTRODUCTION

The vast majority of database-driven web applications perform, at a logical level, fundamentally simple INSERT / UPDATE / DELETE commands. In response to a user action on the browser, the web application executes a program that transitions the old state to a new state. The state is primarily persistent and often captured in a single database. Additional state, which is transient, is maintained in the session (e.g., the identity of the currently logged-in user, her shopping cart, etc.) and the pages. The programs perform a series of simple SQL queries and updates, and decide the next step using simple if-then-else conditions over the state. The changes made on the transient state, though technically not expressed in SQL, are also computationally as simple as basic SQL updates.

Despite their fundamental simplicity, creating web applications takes a disproportionate amount of time, which is expended in mundane data integration and coordination across the three layers of the application: (a) the visual layer on the browser, (b) the application logic layer on the server, and (c) the data layer in the database.

**Challenge 1: Language heterogeneities.** Each layer uses different and heterogeneous languages. The visual layer is coded in HTML / JavaScript; the application logic layer utilizes Java (or some other language, such as PHP); and the data layer utilizes SQL. Even for pure server-side / pure HTML-based applications, the heterogeneities cause impedance mismatch between the layers. They are resolved by mundane code that translates the SQL data into Java objects and then into HTML. When the front end issues a request, code is again needed to combine memory-residing objects of the session and the request with database data. Consequently, developers write a lot of "plumbing" code.

As a data point, in a UCSD web development class taught by the authors, the students built a web application of their choice. A class project averages 4700 lines of code and configuration, but for 1 line of SQL, there is a modest 1.5 lines of Java used for business logic, and 61 lines of Java used for plumbing.[1] That is a lot of plumbing code to write for the computationally simple functionality that is typically required by the majority of web applications!

**Challenge 2: Updating Ajax pages with event-driven imperative code.** Since 2005, Ajax led to a new generation of web applications characterized by user experience commensurate to desktop applications. The heavy usage of JavaScript code for browser-side computation and browser / server communication leads to superior user experience over pure server-side applications, comprising

- performance gains through partial updates of the page
- more responsive user interfaces through asynchronous requests, and
- rich functionality through various JavaScript component libraries, such as maps, calendars and tabbed dialogs.

For example, consider the web application page of Figure 1, where each user submits proposal reviews, reads the reviews provided by the other reviewers and also views a bar chart of the average grades for each proposal. In a pure server-side model, submitting a review for a single proposal will cause the entire page to be recomputed. Indeed, queries will be issued for the reviews and average grades of all proposals, not only for the reviewed proposal. Furthermore, the browser will block synchronously and blank out while


*Supported by NSF awards IIS 1018961, IIS 0917379 and a Google gift.


†The license grant at the bottom of the first column does not confer by implication, estoppel or otherwise any license or rights under any patents of authors or The Regents of the University of California.




5th *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.


[1] We count as business logic Java lines that decide the control flow of the application. We count as plumbing the Java lines responsible for binding SQL to Java and Javascript and vice versa, plus all code responsible for managing language and data structure heterogeneities.
A contributing factor to the huge plumbing to business logic ratio is the best practice usage of MVC frameworks such as Struts, which is encouraged by the class and promotes code modularity and reuse by separating the data model, business logic and user interface, but in turn creates more layers to copy data between.

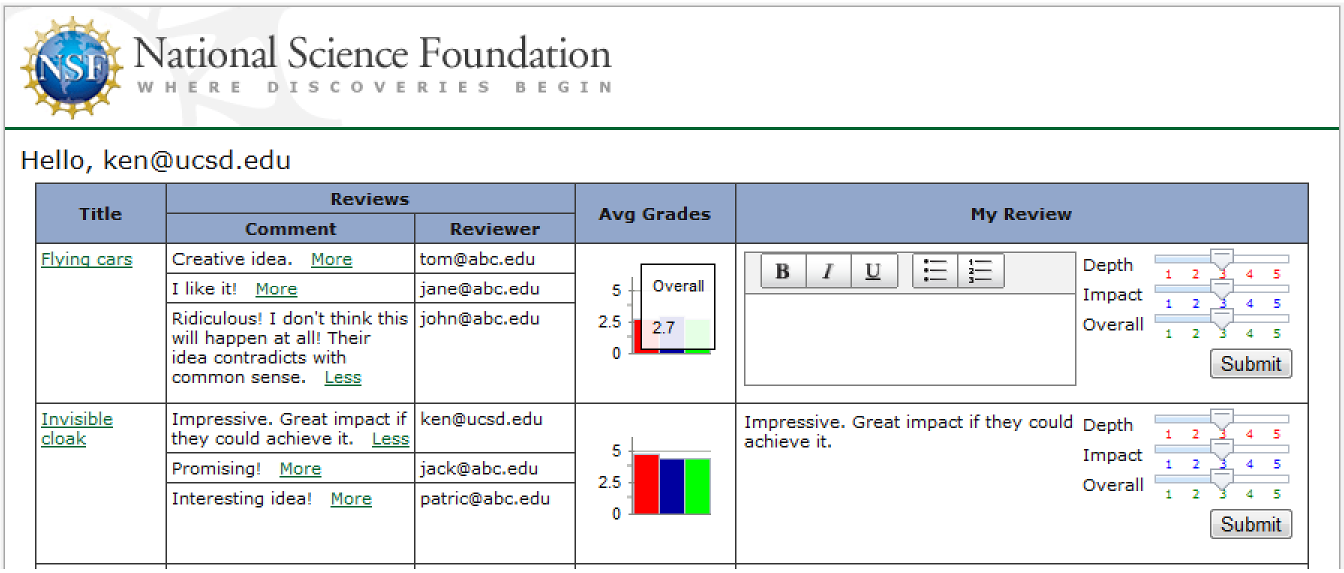


Figure 1: The review page of the running example

it waits for the new page from the server. Finally, various aspects of the browser state, including the data of non-submitted form elements, cursor positions, scroll bar positions and the state of JavaScript components will be lost and re-drawn, thus disorienting the user.

For an Ajax page, however, a developer will typically optimize his code to realize the benefits of Ajax and solve the pure server-side problems listed in the previous paragraph. The same user action (e.g., the review submission) causes the browser to run an *event handling* JavaScript function collecting data from the page's components relevant to the action (i.e. the proposal id, the review text and grades), and send an asynchronous *Xml Http Request (XHR)* with a *response handler* callback function specified. On the server, the developer implements queries that only compute the changed data (i.e. the newly inserted review and the average grade of the corresponding proposal), to take advantage of more efficient queries, as well as to conserve memory and bandwidth. While the asynchronous request is being processed, the browser keeps showing the old page instead of blanking out, and even allows additional user actions and consequent requests to be issued. When the browser receives the response, the response handler uses it to partially update the page's state. The partial update retains non-submitted forms, scroll bar positions, etc, therefore allowing the user to retain his visual anchors on the page.

The page state primarily consists of the browser DOM, which captures the state of HTML form components such as text boxes and radio buttons, and the state of the JavaScript variables, which are often parts of third-party JavaScript components. Therefore, the developer implements the response handler by writing imperative code that navigates the DOM and JavaScript components, and invokes JavaScript methods causing the DOM and components to incrementally render to the browser.

The Ajax optimizations demand a serious amount of additional development effort. For one, realizing the benefits of partial update requires the developer to program custom logic for each action that partially updates the page, which was not the case in pure server-side programming. In particular, in a pure server-side implementation, the developer needs to write code for the effect of each individual action on the database, but writes only one piece of code that generates the page according to the database state and session state. For example, suppose that the page of Figure 1 also provides a "Delete" (Review) button. The developer will have to write two pieces of code that modify the database when "Submit" is clicked and when "Delete" is clicked, respectively. The former issues an INSERT or UPDATE command while the latter issues a DELETE. Both of them share the same piece of code that generates the page showing the list of proposals, their reviews and the average grades. This piece of code is independent of what user action caused the re-generation of the page.

In contrast, in an Ajax application, each user action needs its own code to partially update the page. This piece of code consists of server-side code that retrieves a subset of the data needed for the page update, and browser-side JavaScript code that receives the data and rerenders a sub-region of the page. In the running example of Figure 1, a different piece of code would be needed for the "Submit" and the "Delete" (Review) buttons.

Such event-driven programming (which also occurs in Flash etc.) is well-known to be both error-prone and laborious [12], since it requires the developer to correctly assess the data flow dependencies on the page, and write code that correctly transitions the application from one consistent state to another. Moreover, in a time-sensitive collaborative application (similar to Google Spreadsheet) where many users work concurrently, these dependencies may extend beyond the page of the reviewer who submitted the review, and into the pages of other reviewers who are viewing the same proposal on their browsers. For example, if the developer had issued a query for `Average Grade`, but not `Reviews`, the page will display inconsistent data if another review had been concurrently inserted into the database.

Further compounding the custom logic required for each action is the amount of imperative code that needs to be implemented on the browser. Whereas the developer of a pure server-side application needs to understand only HTML, the developer of an Ajax application needs to integrate JavaScript as yet another language, understand the DOM in order to update the displayed HTML, and write code that refreshes the JavaScript components' state based on the nature of each partial update. Since there is no standardization between the component interfaces between different third-party libraries, the developer is left to manually integrate across these disparate component interfaces.

**Challenge 3: Distributed computations over both browser-side and server-side state.** In response to a user action, the browser sends a HTTP request to the server and activates a program, which needs access to relevant data on the page in order to perform computations that involve such page data and the database. Writing code that involves both page data and server-side data was already mundane and time consuming in pure server-side applications and became even more so in Ajax and Flash.

In a pure server-side application, the browser is essentially stateless since all state is lost when the new page is loaded. Using HTML markup such as `<input type="text" name="review" />`, the developer declaratively specifies the value collected by the textbox will appear as the parameter `review` in the HTTP request. The good news about pure server-side programming is that when a user action causes an HTTP request, the browser is responsible for navigating the DOM, and marshalling the request parameters according to the HTML specification. The bad news is that, on the server-side, the application first unmarshalls the request parameters by using Java (or PHP, etc.), and then typically issues SQL queries where the request parameters become parameters of an SQL statement. Overall, lines of Java code are expended in such trivial "extract parameter from the request, plug it in the query" tasks. With Ajax it gets much worse, as discussed next, since the marshalling of request parameters is no longer automatic.

In particular, in an Ajax application the browser maintains state across HTTP requests. Consequently, the state of the web application becomes distributed between the browser and the server. The developer is responsible for defining a custom marshalling format for the XHR request, typically in XML or JSON, and for writing imperative code to navigate over the DOM and marshall the relevant page data that must be sent to the server along with each HTTP request. The usage of JavaScript components (calendars, etc.) on the page further complicates the issue by requiring custom code that converts between the state of the component and the marshalling format. On the server-side, the developer writes custom code to unmarshall the request parameters, and then continues along the usual path, plugging such parameters into SQL statements.

A select few web application frameworks, such as Echo2 [7] and Microsoft's ASP.NET [2], mitigate the issue of distributed application state by automatically maintaining a mirror of the browser state on the server. Such synchronization can occur efficiently, using reduced bandwidth, by having the server send to the browser only the difference between the previous page state and the new page state. Yet, mirroring is only a half-solution, since the mirror made available on the server contains the exact and full state of the browser, despite the fact that each request cares about a different subset of page data. For example, the submission of a review for the first proposal of Figure 1 requires the data collected by the top editor and sliders, while the submission of a review for the second proposal requires the data of the bottom editor and sliders. Furthermore, the mirrored browser state include visual styling details, which do not matter when form data are collected, yet they trouble collection. For example, to read the values of the three sliders Depth, Impact and Overall in Figure 1, using a mirror-based Ajax framework, the developer will need to navigate through its ancestor `<div>` and `<table>` HTML elements that are used purely to specify visual layout. The net effect of these issues is that the developer's code includes many mundane lines that navigate around the extraneous information in order to obtain the relevant data for the request.

## 1.1 FORWARD's Declarative Solution

The data management field has recently applied with success and great promise declarative data-centric techniques in network management and games. In a similar fashion, FORWARD adopts an SQL-based, declarative approach to Ajax web application implementations, going beyond prior approaches such as Strudel [8] and WebML [3] that focused on pure server-side data publishing applications. In particular, FORWARD removes the great amount of Java and JavaScript code, which is written to address the challenges above, and replaces them with the use of SQL-based languages to facilitate integration and enable automatic optimization. The objective is to "make easy things easy and difficult things possible".[2]

FORWARD is a rapid web application development framework. The web application's pages are declaratively specified using *page configurations.* The programs that run when a request is issued are also declaratively specified, using *program configurations.* Both page configurations and program configurations are based on a minimally enhanced version of SQL, called *SQL++*, which provides access to the *unified application state* virtual database that, besides the persistent database of the application, includes transient memory-based data (notably session data and the page's data). The application runs in a continuous program / page cycle: An HTTP request triggers the FORWARD interpreter to execute a program specification. The program reads and writes the application's unified state and possibly invokes services that have side effects beyond the unified application state. (e.g., send an email). The program typically ends by identifying which page will be displayed next. FORWARD's interpreter creates a new page according to the respective page configuration. The page specification also specifies programs that are invoked upon specified user events. The invocation of a program restarts the program / page cycle.

The key contributions of FORWARD are:

- The use of SQL++ allows unified access to browser data and server-side data, including both the database and application server main memory (e.g., session). In conventional web application programming, such access would require Java and Javascript. FORWARD

[2] This objective is not followed by today's web application development frameworks. Paradoxically, powerful Turing-complete low-level imperative languages (such as Java and PHP) accomplish tasks that can be easily accomplished by appropriate SQL-based declarative languages.

eliminates Java and JavaScript from the majority of web applications, therefore resolving Challenges 1 and 3, in the same spirit as Hilda [13].

- The *page configurations* are essentially rendered views that visualize dynamic data generated by SQL++ and are automatically kept up-to-date by FORWARD. The specifications enable Ajax pages that feature arbitrary HTML and (pre-packaged) Ajax / JavaScript visual units (e.g. maps, calendars, tabbed windows), simply by tagging the data with tags such as `<map>`, `calendar`, etc. The AJAX pages are automatically and efficiently updated by the FORWARD interpreter by appropriately extended use of incremental view maintenance. The FORWARD developer need not worry about coordinating data from the server to the page's Ajax components, which resolves a "Challenge 2" problem.

- The business logic layer is specified by *program configurations*, where business logic decisions and the transfer of data between the database and services are expressed in SQL++, or, even simpler, in mappings, which are translated to SQL++. The program configurations have easy unified access to both the browser data and the database, because FORWARD guarantees that the browser's page data are correctly reflected into the unified application state's page data before they are used by the programs. The automatic reflection resolves Challenge 3.

- The page and program configurations have unified access to the persisted database, the page and the session via a single language (SQL++), therefore resolving Challenge 1. JavaScript needs to be written only if one needs to create a custom visual unit. Java needs to be written only for computations not easily expressible in SQL. For example, one can build the entire Microsoft CMT in the SQL++ based page and program configurations, except for the reviewer/paper matching step, which requires a Java-coded stable matching algorithm to assign papers to reviewers according to their bids.

We argue for the effectiveness of the approach by providing a demo that shows an order of magnitude lines-of-code reduction. Furthermore, Section 5 describes ongoing and future work that will further push the advantages of declarative computation by automating optimizations.

The FORWARD project was recently licensed by app2you Inc, which is a provider of a Do-It-Yourself (DIY) platform [11]. App2you's DIY platform is being refactored so that it produces FORWARD code as the user builds pages with the DIY tool. In the meantime, app2you Inc has recently deployed earlier versions of FORWARD in three human-centric business process management applications and is currently in the process of deploying the presently-described FORWARD version in (a) an analytics-oriented large-scale business intelligence application in the pharmaceutical area, and (b) a business process management application involving hundreds of pages collecting data via smart forms.

## 2. RUNNING EXAMPLE

The running example, which is also the accompanying CIDR demo, is a proposal reviewing Ajax application. The paper focuses on its `review` page (see Figure 1), where the reviewers submit reviews that consist of a comment collected using a web-based text editor, and depth, impact and overall grades collected using slider Javascript components. The editor appears either in edit mode (e.g. first proposal of Figure 1) or in display mode (e.g. second proposal). The page reports the titles of the proposals as hyperlinks that lead to the proposals' pdf, the full set of reviews using a Javascript component with the familar "More" and "Less" buttons, and a bar chart with the average grades.

The full version of the reviewing application is available online at `demo.forward.ucsd.edu/reviewing`, where instructions are provided on how to use it in any of the three roles it supports (namely, "applicant", "reviewer", "chair"). The application's FORWARD specification is available at `demo.forward.ucsd.edu/reviewing/code`. Online instructions on `demo.forward.ucsd.edu` also teach how to create your own FORWARD application.

The implementation of the discussed review page required 102 lines of FORWARD page specification and the implementation of review submission required 42 lines of FORWARD action specification. Building the same required 1,642 lines of HTML, Java, Javascript and SQL code in a "quick-and-dirty" Ajax application where the code is compact, often against the principles of good MVC coding, which requires separation of the control flow part of the code from the visual/interactive part of the code. In a disciplined MVC-based implementation it would take even more.[3]

## 3. CREATING AN APPLICATION

A developer creates an application by providing to the FORWARD interpreter source configurations, schema definitions, page configurations and program configurations.

### 3.1 Sources, Objects and Schemas

The source configurations dictate the type of the sources (e.g., relational database, LDAP server, spreadsheet) and how to connect to them. As a convenience for the development of cloud-based applications and databases, FORWARD implicitly provides to each application an SQL++ database source named `db`. In the running example, `db` provides the full persistent data storage of the application.

A FORWARD application's programs and pages also have access to the `request`, `window` and `session` main memory-based sources, which in the spirit of the session-scoped attributes provided by application servers, have limited lifetimes and there may be more than one instances of them at any time. For example, the `session` source lives for the duration of a session. All the pages of a browser session and all the programs initiated by http requests of the browser session have access to the same instance of the `session` source, while pages and actions of other browser sessions have their own `session` instances. Similarly the `request` source lives for the duration of processing an http request by a program (as discussed in Section 3.2) and each in-progress program has its own `request` instance. A `window` source lives for the

[3] Note that current MVC frameworks interact very poorly with Ajax due to their literal adherence to the program-page cycle. In light of the MVC frameworks' mismatch to Ajax functionalities we did not attempt measuring how many lines of Struts or Spring would be required to build the running example.

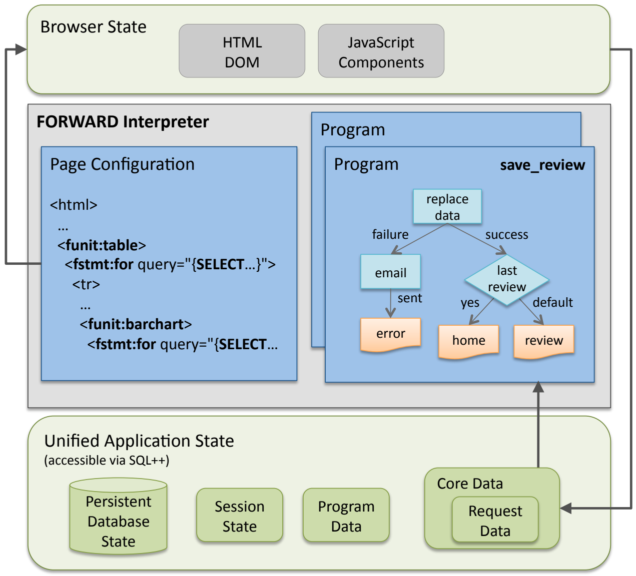


Figure 2: The FORWARD interpreter hosts applications that consist of pages and programs

duration of a `browser` window and becomes available to all pages and programs of such window.

Each source stores one or more objects. Each object has a *name*, *schema* and *data*. The schema may be explicitly created with the Data Definition Language (DDL) of SQL++ or may be imported from the source. For example, the schema of a relational database source is imported from its catalog tables. The DDL of SQL++ is a minimal extension of SQL's DDL.

## 3.2 Operation: The Program/Page Cycle

In the spirit of MVC-based frameworks such as Struts and Spring, a FORWARD application's operation is explained by program-page cycles. (In that sense FORWARD's programs correspond to Struts' actions.) An http request triggers the interpreter to run the program configuration that is associated with the request's URL. The program reads and writes the application's unified state (i.e., database, request data, session data) and possibly invokes services that have side effects beyond the application's state (e.g., sends an email). The program's run typically ends with identifying which page $p$ will be displayed next. Conceptually, FORWARD's intepreter creates a new page according to $p$'s page configuration, which may be thought of as a rendered SQL++ view definition, and displays it on the browser. A displayed page typically catches browser events (such as the user clicking on a button, mousing over an area, etc; or a browser-side timer leading to polling) that lead to action invocations (via http runs) therefore continuing the program-page cycle.

Notice that FORWARD enforces the full separation of the Controller functionality from the View functionality, which current MVC frameworks only encourage but do not enforce: The page configurations are literally views, unable to side effect the application state.

## 3.3 The Page Configuration

A page configuration is an XHTML file with added:

1. FORWARD *units*, which are specified as XML elements in the configuration and are rendered as maps, calendars, tabbed windows and other Javascript-based components. Internally FORWARD units use components from Yahoo UI, Google Visualization and other libraries and wrap them so that they can participate in FORWARD's pages without any Javascript code required from the developer.
2. SQL-based inlined *expressions* and `for` and `switch` *statements*, which are responsible for dynamic data generation.

Figure 3 provides the page configuration of the `review` page. Figure 3 excludes a few parts, which are marked by `<!-- in demo -->` and can be found on the online demo. The complete page configuration's size is 102 lines.

Lines 2-6 of the page configuration list the HTML that generates the top of the page and contains the FORWARD unit `funit:table`.[4] Notice that a unit may contain other units; e.g., the `funit:table` (line 6) contains the `funit:bar_chart` (line 19), the `funit:editor` (line 33) and the `funit:slider` (line 55). A unit may also contain XHTML, which may, in turn, contain other units.

A `fstmt:for` statement evaluates its query and for each tuple in the result (conceptually) outputs an instance of its body configuration, i.e., of the XHTML within the opening and closing `fstmt:for` tags. For example, the `fstmt:for` on line 11 outputs for each proposal one instance of the `tr` element on line 15 and its content.

The syntax and semantics of the `fstmt:for` and `fstmt:switch` statements are deliberately similar to the `forEach` and `choose` core tags of the popular JSP Standard Tag Library (JSTL) [10]. The same applies for FORWARD expressions and JSTL expressions. However, FORWARD's `fstmt:for` iterates directly over a query, whereas JSTL's `forEach` iterates over vectors generated by the Java layer, which are in turn produced by iterating over query results. Besides the obvious code reduction resulting from removing the Java middleware, we will see many more important benefits that are delivered because FORWARD analyzes the queries behind the dynamic data of the page.

FORWARD's page configurations enable nested, correlated structures on the pages. In particular, the query of a `fstmt:for` statement found within the body configuration of an enclosing `fstmt:for` may use attributes from the output of the query of the enclosing `fstmt:for`. For example, the table `reviews` has a foreign key `proposal`. For each "proposal" instance the correlated query on line 20 produces a tuple with the average grades of its reviews by using the `proposal_id` attribute of its enclosing query. Furthermore, the page configurations allow variability (e.g., the current user's review appears either in edit mode or in display mode) utilizing the `fstmt:switch` statement (line 34).

Expressions, which are enclosed in { }, can reference attributes of the query's output. Furthermore, an expression may be itself a query. In the interest of flexibility, which has been the norm in tools and languages for web page creation,[5] FORWARD coerces the types produced by the expressions to the types required by the FORWARD units or XHTML,

[4]Its syntax is identical to the standard HTML table but, unlike the HTML table, it aligns the columns of its nested tables.

[5]For example, JSP pages convert JSP expressions into strings whenever possible.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <html xmlns:funit="http://forward.ucsd.edu/units"
3         xmlns:fstmt="http://forward.ucsd.edu/statements">
4     <!-- in demo: html for css, images, headers, footers-->
5     Hello {session.user}!
6     <funit:table>
7       <th> <!-- table headers -->
8         <td>Title</td>
9         <!-- in demo: rest of the headers -->
10      </th>
11      <fstmt:for name="proposals"
12                query="{SELECT id AS proposal_id, title, pdf_link
13                        FROM db.proposals
14                        ORDER BY proposal_id}">
15        <tr>
16          <td><a href="{pdf_link}">{title}</a></td>
17          <!-- in demo: comments and reviewers -->
18          <td> <!-- bar chart showing average grades -->
19            <funit:bar_chart>
20              <fstmt:for query="{SELECT AVG(depth) AS avg_depth,
21                                        AVG(impact) AS avg_impact,
22                                        AVG(overall) AS avg_overall
23                                 FROM db.reviews
24                                 WHERE proposal = proposal_id}”>
25
26                <bar bar_value="{avg_depth}" color="red"/>
27                <bar bar_value="{avg_impact}" color="blue"/>
28                <bar bar_value="{avg_overall} " color="green"/>
29              </fstmt:for>
30            </funit:bar_chart>
31          </td>
32          <td>
33            <funit:editor name="my_review">
34              <fstmt:switch>
35                <fstmt:case condition="{EXISTS (
36                    SELECT * FROM db.reviews
37                    WHERE reviewer = session.user
38                    AND proposal = proposal_id)}”>
39                  <text>{SELECT comment FROM db.reviews
40                         WHERE reviewer = session.user
41                         AND proposal = proposal_id}</text>
42                  <state>display</state>
43                </fstmt:case>
44                <fstmt:else>
45                  <text>Type your comment here</text>
46                  <state>edit</state>
47                </fstmt:else>
48              </fstmt:switch>
49            </funit:editor>
50          </td>
51          <td> <!-- sliders -->
52            <funit:table>
53              <tr>
54                <td>
55                  <funit:slider min="1" max="5" name="depth"
56                         value="{SELECT depth FROM db.reviews
57                                 WHERE reviewer = session.user
58                                 AND proposal = proposal_id}"/>
59                </td>
60              </tr>
61              <!-- in demo: sliders for impact and overall grades →
62              <funit:button onclick="save_review">Submit</funit:button>
63  <!-- in demo: several closing tags -->
```

Figure 3: The page configuration of `review`

depending on where the expression appears. Therefore, the developer need not worry about fine discrepancies between the types used in the database (which are dictated by the business logic and are often constrained) and the types used for rendering, which are often as general as they can be. For example, the expression that feeds the `value` attribute of the `funit:slider` on line 56 is a tuple that has a single integer attribute. However, it is coerced into a float, which is the type of argument that the `funit:slider` expects.

**The page as an automatically updated rendered view** Conceptually, the page configuration is evaluated after every program execution. While such an explanation is simple to understand, it is only conceptual. If the page that is displayed on the browser window before and after a program's execution is the same, then FORWARD will incrementally update only the parts of it that changed, therefore achieving the user-friendliness and efficient performance of Ajax pages, where the page does not leave the user's screen (and therefore avoids the annoying blanking out while waiting for the page to reload) but rather incrementally re-renders in response to changes. [9] explains how FORWARD utilizes incremental view maintenance in order to efficiently and automatically achieve pages as incrementally rendered views.

**To XQuery or not to XQuery** The page configuration syntax and semantics of FORWARD closely resemble those of XQuery, as they both enable nesting, order, variability and coercions. Indeed, FORWARD allows similarity to XQuery to become even more obvious by allowing omission of the SELECT clause of the queries, which is tantamount to an automatic `SELECT *`. We have chosen SQL for three reasons:

1. The typical query input is the application's database, which is almost always relational and therefore we do not need to complicate the query language semantics with conventions on how the relational database is wrapped into XML. In that sense, the page configuration language is close to the SQL/XML relational input and XML output approach.

2. A main audience of our approach are SQL developers who feel frustrated by how hard it is to create a rendered view. There is no significant number of XQuery Web developers yet.

3. Numerous processing, optimization and consistency checking algorithms around the page configuration are amenable to cleaner reductions to respective SQL problems, without subtle complications that XQuery's semantics introduce.

Nevertheless, a limited XQuery implementation is also in the plans, as XQuery provides high power in certain cases, such as XQuery-Text.

**Summary** The page configuration is essentially an SQL view embedded into a visual template, consisting of HTML and funit tags. Therefore the page configuration resolves Challenge 1, since it enables the production of pages without requiring Java and javascript code in addition to SQL. Furthermore, it is implemented as an Ajax page that is automatically updated to reflect the database state, therefore resolving Challenge 2 of Ajax application programming.

## 3.4 Unified Application State

The FORWARD unified application state, in addition to the conventional persistent data that an application has, also includes transient application data, such as an automatically-created logical-level representation of page data, which are heavily used in web application programming.

Structurally, the unified application state consists of the set of all sources, such as the `session`, `db` and `core`, which are described by an SQL++ schema. In order to accommodate the needs of pages, of session data and of other data typically occurring in web application programming, SQL++ is a minimal extension of SQL, whereas each schema is a tuple. An attribute of the tuple may be either a scalar type or a table, whose tuples have attributes that may recursively contain nested tables. Notice that standard SQL corresponds to the case where a schema is simply a tuple of tables (think of SQL's table names as being the attribute names of the top level tuple) and each table's tuples may only have scalars (as opposed to nested tables). In order to allow variability in the spirit of XQuery and OQL, SQL++ also supports an OQL-like union construct.

An example of a schema that uses the extra features of SQL++ is the session, which in the running example is simply a tuple containing only the standard scalar attributes `session_id`, `user` and `role` that are set by FORWARD's session management and authentication/authorization utilities (not shown). The expression {`session.user`} on line 5 is a SQL++ query accessing the attribute `user` of the tuple of the `session` schema. A key integration contribution of FORWARD, which attacks Challenges 1 and 3, is the ability of SQL++ to combine persistent data with transient data using just a single SQL++ query. For example, consider the queries on lines 36 and 39 that combine the `session.user` with the table `reviews` of the persistent schema `db` to produce the reviews of the currently logged-in user in just three lines of SQL. More integration contributions are made by the core schema and data, discussed next.

The *core* schema captures in an SQL++ schema the subset of page data that have been named by the developer, using the special tag `name`, in the page configuration. Therefore the core enables the developer to resolve part of Challenge 3, by enabling him to create a data structure encompassing the data of interest to the programs as opposed to, say, visual details. FORWARD infers the core schema by inspection of the page configuration. In the running example, the core schema, which happens to fall within standard SQL, is a table of the proposals that appear on the screen along with their reviews and grades. In particular, it is a table named `proposals` (due to the `fstmt:for` on line 11) with a string attribute named `my_review` (due to line 33) and float attributes `depth`, `impact` and `overall` due to the sliders (the last two not shown in Figure 3). The table `proposals` also has the key attribute `proposal_id` so that one can associate the data collected by the multiple instances of the editor and the sliders with proposals. Mechanically, FORWARD infers this attribute to be the key of the query that feeds `proposals` using a straightforward key inference algorithm. Notice that such inference relies on the underlying `db.proposals` table having a known key, which is an unavoidable assumption of the running example, no matter what technologies one uses to implement it.

Notice that the algorithm that infers the core schema utilizes statements and queries of the page configuration, i.e. the page's logical aspects, while it mostly ignores the unit structure and XHTML (the visual aspects). The only unit aspect that matters is the types of data collected by the user, i.e., string from the `funit:editor` and floats from the `funit:slider`s. This is a key advantage over page mirror-based frameworks, such as Microsoft's ASP.NET, that offer to the developer a server-side mirror of the page data, so that the developer does not have to code in Javascript. Unfortunately, the structure of the mirror follows the (typically very busy) visual structure of the page, as opposed to the data structure that best fits the database (a typical "Challenge 3" problem). In the running example, had we followed ASP.NET's approach, instead of having three attributes names named `depth`, `impact` and `overall`, corresponding to the three sliders, we would have a hard-to-use nested table whose first tuple would be implicitly assumed to contain the `depth`, its second tuple to contain the `impact`, etc. The fact that FORWARD's page configuration enables extracting the core data is testimony of the power of a declarative approach fueled by logical statements and queries.

An important piece of data in the core is information about which program was invoked. In the running example, the page invokes the program `save_review` (line 62), and therefore it is important to know upon invocation which one of the many instances of the `save_review` was invoked. There are as many instances as proposals on the page. The core identifies the `proposal_id` for the invoked `save_review`.

FORWARD guarantees that the core data is automatically up-to-date when a program starts its execution. This is a key contribution towards resolving Challenge 3. In a conventional Ajax application, Javascript and Java code has to be written to establish a copy of relevant page data on the server, in a way that they can be subsequently combined with the database.

The `name` attribute convention is reminiscent of the HTML standard's convention to allow a `name` to be associated with each form element and consequently generate request parameters with the provided names. Drawing further the similarities to HTML's request parameters, the *request* objects keep only the tuple of the core that correspond to the invoked program.

Mappings (Section 3.5) raise the level of programming even higher by allowing the developer to select, project and combine data utilizing a mapping interface.

## 3.5 The Program Configuration

A program configuration is a composition of synchronous *services* in an acyclic structure with a single starting service, which is where the program's execution starts. For example, Figure 2 shows the graphical representation of the `save_review` program configuration, which is composing services `replace_data` and `email`, among others.

Services input data from the unified application state. For example, the starting service `replace_data` takes as input data indicated by the mapping of Figure 4. The invocation of a service has one or more possible *outcomes*, and each outcome (1) generates a corresponding service output, which becomes part of the unified application state, and (2) leads to the invocation of a consequent service or the end of the program. The `replace_data` service has a `success` outcome and a `failure` one. The former adds the `replaced_tuple` to the unified application state and the latter adds a failure `message` in order to facilitate the invocation of the consequent `email` service, which will email the failure message to the application administrator.

FORWARD offers a special service called `page` (depicted by the page icon in Figure 2) that programs use to instruct the FORWARD interpreter of which page to display next.

The transactional semantics of services differentiate the ones that have *side effects* from the ones that do not. A

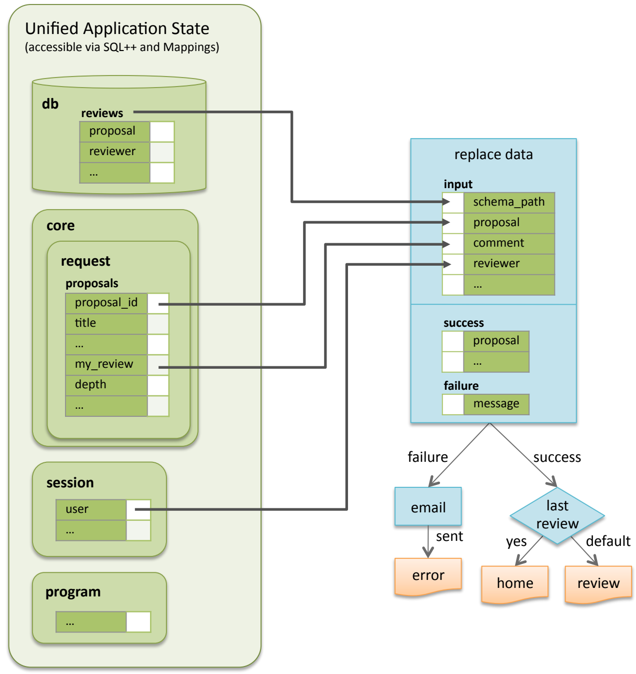


Figure 4: Building the `save_review` program configuration

service has side effects if it makes changes observable by the outside world, be it by updating the unified application state or by invoking an external service, such as sending an email or charging a credit card. The `replace_data` and `email` are examples of services having side effects and are graphically depicted by a rectangle in Figure 2. A service with no side effects is depicted by a rhombus and merely chooses what service will be invoked next. For example, if the outcome of `replace_data` is `success`, then the program will invoke the `last_review` service, which simply checks if the review just saved was the last one to be submitted, and hence does not have any side effects. If so, another `page` service will set the next page to be the `home` page of the application. Otherwise, the next page will be the same as the current `review` page.

Building a program configuration is greatly simplified by SQL++ access to the unified application state. It is further simplified bythe FORWARD *mapping language*. In principle, the service input data can be generated by a SQL++ query. In practice though, the developer rarely does so since the input data of services can be specified much easier using *mappings*. Figure 4 demonstrates how mappings are visually designed inside FORWARD's development environment between the unified application state and the input schema of the `replace_data` service. Notice that the developer seamlessly draws mappings from the persistent database state `db`, the `request` data and the `session` data to the input schema of the service. For consequent services, the developer is also able to draw mappings from the `program` data generated by previous services invocations.

## 3.6 The Scope of FORWARD Applications

The SQL++ based program and page configurations have limitations, when compared against programs and pages written using combinations of Java, Javascript and SQL. We argue that these limitations are within the spirit of "make easy things easy and difficult things possible": Still every application is doable, while common applications are much easier to write.

In the case of programs, notice that a program is an acyclic graph.Therefore programs comprise (1) sequencing of services, most of which are plain SQL statements, and (2) if-then-else control. However, programs lack explicit loop structures. The restriction is much less severe than it initially appears to be because there are implicit loops in the service implementations. For example, the email service of the example sends an email to each of the many recipients. Nevertheless, the limitation raises three key questions:

First, what percentage of applications do not require explicit loops? Database theorists had introduced the relational transducer [1], which described the program executed after a user interaction by a plain sequence of SQL commands, and essentially conjectured that the business process of most web applications is describable within the expressive power of SQL (notably without recursion). Later, the WAVE project showed that this conjecture applies to well-known web applications, such as `dell.com` [5]. The authors have collected an interesting piece of evidence pointing in the same direction: Two of the authors have given a web application development class, at SUNY Buffalo and UCSD respectively, asking students to specify and build a web application of their choice as a class project. The vast majority of student applications avoid the limitation. For example, in the Winter 2010 UCSD offering of the class, all student projects avoided the limitation.

The next question is how can the limitation be overcome when explicit loops are truly needed in a program? In such cases the developer may write a service in Java and enjoy the full power of Java. Indeed, the developer may write the whole program in Java.

Finally, what are the benefits of the no-loops limitation? The lack of explicit loops simplifies the semantics of service compositions to the point that a program is simply plugging together services via mapping a service's output to other services' input. The practical failure of prior visual programming tools and business process languages that attempted to capture the full extent of programming makes us believe that visual programming should be limited to the simple cases, while Java provides the "bail-out".

# 4. INTERNAL ARCHITECTURE

The internal architecture of the FORWARD interpreter is illustrated in Figure 5, which displays internal modules (labelled with red numbers) in association with developer-visible concepts of Figure 2. For efficiency of storage and communications, FORWARD maintains a *visual page state*, which provides an abstraction over the browser state by including the externally visible state of visual units, but excluding their implementation details. Two copies of the visual page state lazily mirrored between the browser and the server.

When the user performs interactions such as typing in a text box, the HTML DOM and the JavaScript variables of visual components change in the browser state. The respective *state collectors* of each FORWARD unit synchronize the appropriate part of the browser state with the corresponding part of the *browser-side visual page state*. When the user eventually triggers an event that leads to invoking a

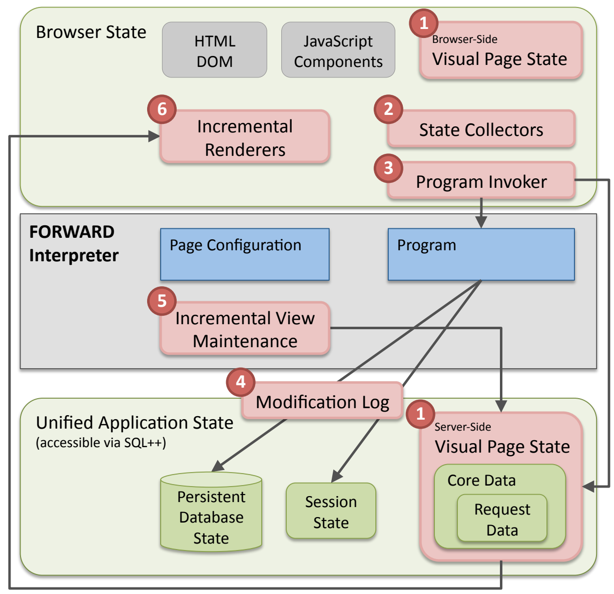


Figure 5: Internal FORWARD Architecture

program, such as clicking the submit button of Figure 1, the *program invoker* guarantees that the browser-side visual page state has been fully mirrored onto the server-side before the program executes. This guarantee is efficiently implemented via incremental writes to the prior visual page state.

Using the program invocation context and page configuration, the interpreter calculates (1) the core data, by projecting only the named attributes of the visual page state, and (2) the request data. As services within the program read from and write to the unified application state, the system also uses a *modification log* to intercept all changes to the unified application state. By using the modification log in combination with the unified application state, the interpreter employs *incremental view maintenance* optimizations to incrementally maintain the current visual page state to the next visual page state [9]. The current implementation uses an off-the-shelf relational database without modification to the database engine. It intercepts changes by instrumenting database-related services. By storing the modification log as well as transient parts of the unified application state in memory-resident tables of the RDBMS, the current implementation issues incremental queries that are more efficient than the original queries in the page configuration, since they retrieve data primarily from the memory-resident tables. As illustrated in [9], incremental view maintenance can speed up the evaluation of page queries by more than an order of magnitude.

Finally, the interpreter uses data diffs to efficiently reflect changes back to the browser-side visual page state. The same data diffs are also provided to the respective *incremental renderers* of each visual unit, which, in turn, programmatically translates the data diff of the visual page state into updates of the underlying DOM elements or method calls of the underlying JavaScript components. Essentially, the incremental renderers modularize and encapsulate the partial update logic necessary to utilize Javascript components, so that developers do not have to provide such custom logic for each page. Also illustrated in [9], in addition to performance gains due to less DOM elements / JavaScript components being initialized, incremental rendering also delivers a better user experience by reducing flicker and preserving unsaved browser state such as focus and scroll positions.

To convert between the different schemas of the core data and the visual page state, the compilation of the page configuration automatically produces a mapping between these two schemas. For example, as discussed in Section 3.2, the core data comprises three (flat) attributes `depth`, `impact` and `overall`, whereas the visual page state represents the corresponding sliders nested within a table unit, thus the mapping language mitigates such structural differences including extraneous attributes (by projection) and nested tuples (by flattening). Mappings are also used to produce type coercions, such as automatically converting an integer attribute in the core data to a float attribute in the visual page state, and vice versa.

# 5. FUTURE WORK ON OPTIMIZATIONS

As the history of SQL-based systems has shown, a key benefit of declarative approaches is the enablement of (i) automatic optimizations and (ii) static analysis that can improve the operation of the system or detect possible erroneous behaviors. As discussed in Section 4, the efficient incremental maintenance of the Ajax pages is a derivative of the declarative SQL-based approach. FORWARD ongoing and future work will extend the benefits of the declarative approach towards the following optimizations and functionalities, which would otherwise require programmer efforts to be accomplished. Notice that the declarative approach reduces each of the following problems into an extension or specialization of respective problems where the data management community has delivered automatic optimization techniques.

**Page query optimization** The obvious (according to the semantics) way to collect the data needed by a page configuration is not necessarily the most efficient. For example, the data needed for the page of the running example can be collected by running the outer query of Lines 12-14, which returns proposals, and then, for every proposal tuple, instantiate the `proposal_id` and run the query of Lines 20 to 24, which returns their grades. A more efficient way to collect proposals and grades is by sending a single query. Research in OQL and XQuery has ran into similar issues and has provided a list of rewritings that can be employed for performance gains in such situations.

**Pub-sub optimizations for updating myriads of open pages** Consider a popular application where there are myriads of pages open at any point in time. When the database data change, these pages have to be correspondingly updated. For example, when a Facebook user changes his status, the currently open pages of his friends must be updated. FORWARD's reduction of the data reported by a page into essentially a rendered view, opens the gate to leveraging database work (e.g., [4, 6]) for the rapid update of only the relevant open pages.

**Location transparency, browser side operation and mobility** The SQL++ queries over the unified application state are location transparent in the sense that they do not dictate whether they are executed fully at the server or whether they are executed partially at the server and partially at the browser. While the current version of FORWARD mirrors the browser data on the server and runs SQL++ fully on the server, alternate methods are also possible. A possibility that is beneficial to mobile applications is to create caches of the parts of the application state on the browser. Then certain queries and programs will be able to operate on a disconnected browser.

## 6. ACKNOWLEDGEMENTS

We thank Kevin Zhao for his worthy-of-authorship contributions. Kevin does not appear in the author list since he is the author of another CIDR 2011 paper. We thank Fernando Gutierrez, Ryan Mourey, Sam Wood and Erick Zamora for the contribution of multiple FORWARD units. We thank Alin Deutsch for many discussions on FORWARD.

## 7. REFERENCES

[1] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 179–187, New York, NY, USA, 1998. ACM.

[2] Asp.net, 2009. `http://www.asp.net/`.

[3] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[4] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[5] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD Conference*, pages 539–550, 2005.

[6] Y. Diao and M. J. Franklin. Query processing for high-volume xml message brokering. In *VLDB*, pages 261–272, 2003.

[7] Echo web framework, 2009. `http://echo.nextapp.com/site/`.

[8] 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.

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

[10] Javaserver pages standard tag library, 2010. `http://java.sun.com/products/jsp/jstl/`.

[11] K. Kowalczykowski, K. W. Ong, K. K. Zhao, A. Deutsch, Y. Papakonstantinou, and M. Petropoulos. Do-it-yourself custom forms-driven workflow applications. In *CIDR*, 2009.

[12] S. Miro. Who moved my state? *Dr. Dobb's*, 2003.

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