RI 14009, 15 Dec 2014 Computer Science

IBM Research Report

Simplifying Web Programming

Nishant Sinha
IBM Research, India

Rezwana Karim
Rutgers University, USA

Monika Gupta
IBM Research, India

IBM Research Division
Almaden - Austin - Beijing - Delhi – Bangalore - Haifa - T.J. Watson - Tokyo - Zurich

Simplifying Web Programming

Nishant Sinha
IBM Research
India
nishant.sinha@in.ibm.com

Rezwana Karim
Rutgers University
USA
rkarim@cs.rutgers.edu

Monika Gupta
IBM Research
India
monikgup@in.ibm.com

ABSTRACT

Modern web programming is plagued by a jungle of heterogeneous programming frameworks and lacks adequate abstractions for end-to-end rapid, structured, design and development. We studied the current problems faced by developers using an online survey, and found that integrating client-side interactivity with the back-end is a major source of inefficiency. Based on the reported issues, we developed a new programming environment, called WEBNAT, to reduce the burden of client-server programming. WEBNAT makes it easy to specify bindings of client-side views with server-side data and provides multiple abstractions that enable succinct specifications for interactive web applications. We conducted a user study to understand its usefulness and barriers to adoption. Our participants were able to learn and use WEBNAT in less than 2 hours showing minimal learning curve. We also discovered that although novices embrace the system readily, experience developers are more cautious about adopting a new web programming framework.

1. INTRODUCTION

Programming for the web is quite intimidating. The web developer faces a jungle of low-level web technologies to deal with, for client-side, e.g., HTML, CSS, JavaScript (JS) libraries, server-side, e.g., PHP, J2EE, Django, Ruby on Rails (RoR), and database interaction, e.g., MySQL, MongoDB, Derby. Writing even conceptually simple applications (apps) requires inordinate amount of effort: learning multiple languages and frameworks (most apps combine 5 or more frameworks), diving into nitty-gritty details of each framework and writing glue code, often repeatedly, to make multiple layers inter-operate. Consequently, the entry barrier for web programming is high even for skilled programmers let alone novice end-users.

To discover the key hurdles during development, we conducted an online survey with 40 web developers, having varied experience levels and backgrounds. Most participants were pained by lack of tools for efficient UI design and handling cross-browser compatibility. The next key issue was about structuring the application (separating layers) and writing glue code for synchronizing states across the browser, server-side logic and the database. Finally, the responses also complained about writing secure apps and complex server/database configuration setup needed for development. In summary, besides UI design, writing structured apps and debugging erroneous glue code continues to be a challenge for developers in spite of many sophisticated web frameworks being available.

The model-view-controller (MVC) paradigm [32] is the de-facto structuring principle for web apps. The MVC pattern advocates separation of concerns between three entities: models capturing app data, views describing the visual appearance and the controller which handles transformations between models and views along with navigation control. Without proper separation of concerns, it is easy for programmers to slip into unstructured programming mode where they may intermingle MVC components of the application and *duplicate* functionality unknowingly, making apps inadvertently complex. Besides enabling modular development, a key benefit of model-view separation is *reactivity*: the view is supposed to refresh whenever the *bound* model is updated. Using MVC patterns can dramatically reduce the developer's coding effort, e.g., by using reactive templates provided by the Angular [2], the code size for the Google Feedback application came down to 1.5KLoC from 17KLoC.

Not surprisingly, most modern web programming frameworks spawned over the last decade, e.g., Ruby on Rails, Django, Symphony, ASP.NET, Backbone.js, enforce some form of MVC-based design. However, our survey shows that these frameworks lack suitable programming abstractions which enable MVC-based design *end-to-end*. For example, although RoR contains ActiveRecords which allows the developer to avoid dealing with low-level server-database interactions, RoR does not facilitate data synchronization between models on the client-side and the ActiveRecords in the backend. Thus, programmers are forced to duplicate data or code between client and server. Similarly, popular frameworks like Angular [2], Backbone [3] and React [6] allow convenient model-view *bindings* on the client-side but do not assist with data synchronization between client and server. In absence of strict end-to-end enforcement of MVC, the developer wastes time writing low-level, erroneous boilerplate code for one or more application layers, leading inevitably to complex, bloated and unmaintainable apps.

In this paper, we present a new domain-specific language WEBNAT to enable effective MVC-based design of web apps. Designing a WEBNAT app involves three stages: (a) declaring the data models as *typed* variables (b) defining the views as HTML/CSS *templates* [4] with *placeholders* for dynamic data expressions and (c) defining the controllers containing the presentation as well as business logic over models and views. WEBNAT includes multiple abstractions to enforce MVC separation while avoiding boilerplates:

- *First-class* views: In WEBNAT, view references can be used

ISEC'15, Bangalore, Karnataka, India.

as commonly as variables, e.g., the syntax allows statements of form $\#t = m$, where $\#t$ refers to an HTML `table` element instantiation and *m* is a variable of type `array(int)`.

- *Reactive-by-default.* To enable model-view reactivity, conventional languages rely on special datatypes like `signal` or `stream` [25, 15] and corresponding APIs ($\$E, \$B$ [25]). Statements of form $(a = b + c)$ over *integer* variables a, b and c must then be *transformed* [16] to corresponding expressions over `signal` variables a', b' and c' to enable continuous evaluation. Instead in WEBNAT, all variables are reactive-by-default, i.e., expressions over these variables may be dynamically evaluated by WEBNAT semantics whenever value of any variable changes. This enables intuitive, reactive programs specifications in WEBNAT.

- *Declarative Bindings.* WEBNAT allows *binding* view elements with data variables (or expressions over them) using a native declarative *binding* construct. The developer may use familiar imperative assignments to specify binding, e.g., the statement $\#t = m$ associates the *data context* m with UI element $\#t$ and ensures that whenever m changes, $\#t$ is re-rendered using the new value of m. Again, popular frameworks, e.g., Angular [2], React [6], depend on special idioms (e.g., data-directives [2]) to specify model-view bindings; these idioms have a learning curve and vary between frameworks thus adding to programmer's burden.

- WEBNAT adopts the principle of *one-data*: the specification refers to only a single copy of data, irrespective of whether the data is duplicated across client/server during execution. The WEBNAT runtime duplicates and synchronizes *persistent* data as required: the programmer is thus saved from the burden of specifying explicit synchronization glues, which are often erroneous.

- Finally, an implicit type system allows the developer to write intuitive yet type-correct *coercions* [31] between models and views, e.g., in $\#t = m$ statement, t and m are of types `view(table)` and `array(int)` respectively.

We implemented a research prototype of the WEBNAT language and a runtime which compiles WEBNAT programs to JavaScript on both client and server. We conducted a within-group study to evaluate the benefits and understand the barriers to adoption. Most of the participants respond that the inbuilt abstractions simplify web programming to a large extent and help them avoid writing glues between different tiers. Programmers with lesser experience (<5 years) are eager to adopt the platform for regular use. More experienced programmers also appreciate WEBNAT's' simplified design paradigm but are less interested, primarily because they have multiple years of familiarity with their framework and demand the same low-level control from our framework.

1.1 Background Survey

Quoting from [1]: *Almost every Web designer can attest that much of their work is repetitive. We find ourselves completing the same tasks, even if slightly modified, over and over for every Web project.*

We conducted an online survey to understand the challenges that programmers encounter during web development.

Participants. Participants were recruited through internal mailing lists of a university and an IT organization, and various social networks. The survey was open for two weeks. We received 40 responses (13 females). All the participants were well-educated with

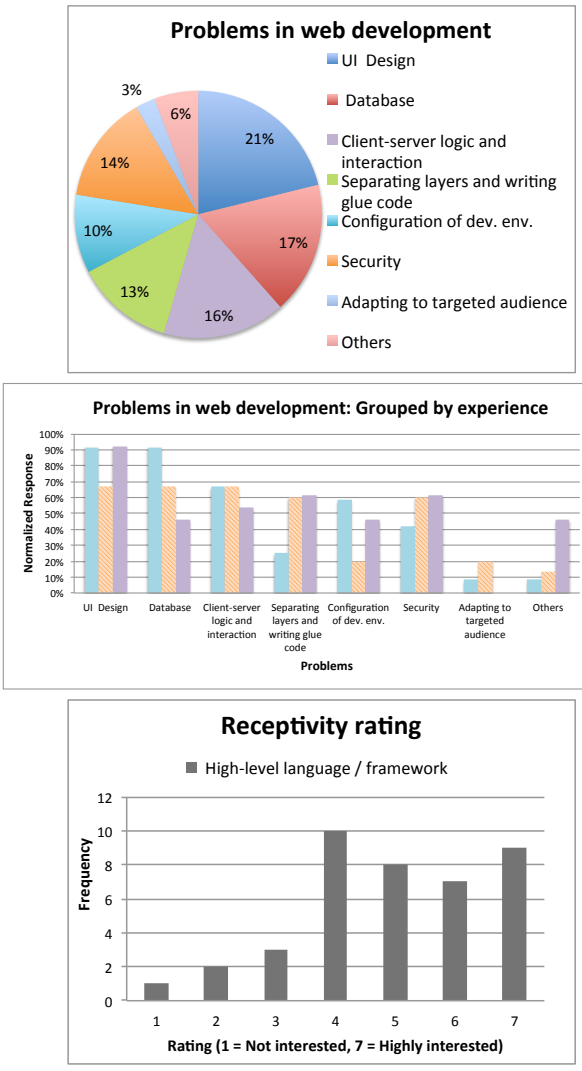


Figure 1: Survey Response for (a) problems in web development (b) user receptivity (c) challenges in web development for different groups.

15 having Bachelor's degree, 9 with Master's degree, and 8 with PhD (rest did not answer this question). In terms of web development experience, 12 participants reported <1 year (*novice*), 15 with 1-5 years (*semi-professionals*) and 13 with >5 years (*professionals*).

We asked the participants to select the challenging web development tasks from a pre-specified list (based on [33]) including UI design, cross-browser compatibility, coding client-server and database interactions, ensuring security, writing server logic and configuration. We also asked them to name the tasks which involved most coding/debugging effort and how helpful their current web programming framework was in reducing their effort for each difficult task.

Fig. 1 (a) shows the responses for different web development challenges and Fig. 1(b) shows the breakdown of the survey results, grouped by the participants' experience. The results show that the main challenge for developers (21%) is in designing HTML/CSS UIs and making them look good across browsers. Interaction with database is the next challenge mentioned by 17% of the partici-

pants. Client-server interaction, which includes coding and debugging client/server-side logic and writing code for serializing/deserializing code, was mentioned by 16% participants. 13% participants reported organizing and separating different layers of the app as a challenge. 14% of the participants reported ensuring security of web apps as a challenge.

The survey also questioned participants about their familiarity with MVC and related web programming concepts. The results show that 83% developers are familiar with asynchronous event handlers, 73% with MVC patterns and 93% with object types and casting in programming languages. In fact, 68% developers think about MVC while designing and developing the app. While 25% developers said that their current framework enforces MVC during app development, 50% reported partial enforcement and 25% said that MVC is not enforced.

We found that semi-professional programmers (67% and 67% respectively) and even half of the professionals (46% and 54% respectively) mentioned concerns for handling database and client-server interaction. This is interesting given that most of the popular frameworks (e.g., RoR, Hibernate) provide database abstraction with an object-relational mapping (ORM). Participants mentioned that they spend majority of debugging effort in debugging SQL queries and synchronizing browser state with the database. Also, 62% of the professional and 60% semi-professional developers reported that they face problems with keeping different application layers decoupled and are forced to write a lot of boilerplate code on a regular basis.

We also asked the participants to rate how open they are to learn a high-level web development environment that abstracts away low-level details. Fig. 1(c) shows that in spite of abundance of web frameworks, participants in our survey were overall receptive towards a high-level environment that would simplify the development process.

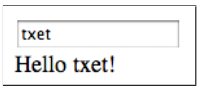
1.2 Model-View-Controller Paradigm

The MVC paradigm [32] is central to designing data-driven user-interfaces in a structured manner. For web apps, the models refer to in-memory or persistent data store, views are specified in HTML/CSS and rendered by a browser and the controller is written in one of the web-centric languages, e.g., JS on client and PHP/Python/Java on server-side. Although the MVC paradigm is well-known, web developers have started using MVC extensively only recently. The earliest web applications were server-centric, where data resided fully on the server and controllers used the data to generate views and send back to the client. These apps had little separation of concerns and all the components were intertwined in the server-logic, e.g., PHP or JSP code. With the advent of Web2.0, apps became client-heavy and highly interactive. To enable real-time behaviors, views are no longer rendered at server; now many apps persist the views at the client and only the models are exchanged across client-server. This leads to a natural MVC-based decomposition of the app. A well-structured app consists of three distinct components: *data model* definitions, view *templates* with *holes* for plugging in data, and *controllers* for (a) rendering views, (b) transforming data between client, server logic and database, and (c) routing URL requests to appropriate controllers.

1.3 MVC by Example

Fig. 2(a) shows the HTML and JS code (using the popular JQuery [7] library) for a `hello` app which renderes the output based on the input text reactively. Whenever the user modifies the text in the input box to say, *txet*, the output *Hello txet!* is shown below the box. In terms of the MVC paradigm, the application contains two visual components (views), the input *inp1* and the *outdiv*, each having a data model and a view template, and a controller which copies the data models from the first view to second. The view *template* for *inp1* consists of the HTML text `input` tag and the model is the text entered. For *outdiv*, the template consists of the HTML `div` element together with a nested text template T = *Hello* ρ*!*, where ρ denotes a placeholder, corresponding to the model of *outdiv*. Note that ρ is never explicitly declared or assigned to in the code in Fig. 2(a). Similarly, the text template T is not explicit in HTML: it is dynamically inserted by the JS code (line 9) after obtaining the model value. Finally, the controller code extracts *inputbox*'s model (line 7) and uses it to create the view for *outdiv* (line 8-10). This example illustrates how easy it is to write code which intertwines MVC components using, e.g., JQuery [7].

```
1  <script type='text/javascript'>
2
3  $(document).ready ( function () {
4    var inputbox = 'input[name=inp1]';
5
6    $(inputbox).change( function () {
7      var v = $(inputbox).val();
8      if (v !== '')
9        $('#outdiv').html ('Hello ' + v + '!');
10     else $('#outdiv').html('');
11   });
12 });
13 </script>
14 <body>
15 <input name='inp1' type='text'> </input>
16 <div id='outdiv'> </div>
17 </body>
```



(a) In HTML/CSS/JavaScript

```
always -> {
  #outdiv =
    (#inp1 != "")? "Hello" + #inp1 + "!" :""
}
```

(b) In WEBNAT

Figure 2: A `hello` app.

Although most modern frameworks enforce MVC separation partially, they do not enforce MVC end-to-end. For example, RoR, Django and Spring, focus on the server-side, while Angular, Backbone and Ember enforce MVC on the client side. In particular, the programmer still has to write glue code for propagating model changes to views and coordinating front-end UI changes (primarily in JavaScript) with back-end models. Data is duplicated between client and server and hence the onus lies with the programmer to synchronize client-side data with server-side data by writing complex, error-prone, JS glues. Moreover, serializing data and converting between client-server formats leads to high debugging effort, eventually overwhelming the web programmer and undermining the advantages of the adopting the MVC development style.

2. THE WEBNAT LANGUAGE

We have developed the WEBNAT specification language to simplify end-to-end web programming and provide web developers

with an intuitive language syntax, oblivious to client-server-database topology. Our goal is to reduce their cognitive load allowing them to focus on the app design and improve productivity.

WEBNAT builds over the JavaScript (JS) based web development ecosystem by adding multiple programming abstractions which allow the developer to convey her design intent succinctly. For example, Fig. 2(b) shows how the controller for `hello` app is specified in WEBNAT, in a single line. It refers to view elements directly, e.g., *#inp1* and uses them to read and write the *bound models* indirectly; model-view binding is thus exploited for succinctness. The specification is *reactive* by default (using the custom event *always*): any change to *#inp1*'s model triggers this controller, which updates *#outdiv*.

To specify the app in WEBNAT, the developer needs to specify (a) views as HTML/CSS templates [4] (b) the data (model) variables with appropriate types, and specify which of them are persistent, (c) bind the models with views, and (d) specify the interaction and transformations between data and views via *handlers*. Finally, the developer runs the WEBNAT compiler to create a deployable, multi-tier web application in JS. We now discuss the main design characteristics of WEBNAT.

2.1 Key Design Decisions

View References. Any UI widget or a hierarchical collection of widgets is said to be a WEBNAT *view*: views are *first-class* entities in WEBNAT. Each view (including sub-views and widgets) is *named* uniquely; the developer can use these view references in WEBNAT syntax directly, e.g., when binding models to views or specifying handlers.

Reactivity, Model-view Bindings. Reactive specifications capture the core data flows of the system succinctly and thus omit the need for polling the data sources at regular intervals to monitor changes. All data variables are assumed to be *reactive* by default; consequently, all expressions and assignments dependent on these variables are also evaluated reactively. Views are defined as templates with *holes* containing data-dependent expressions and may be *bound* with a data variable or a record field. Handlers contain statements over data variables and view references. Therefore, all views and handlers are reactive implicitly, i.e., WEBNAT semantics guarantees that views are re-rendered as soon as the bound data changes and handler expressions re-evaluated as required. Thus, writing glue code for observing changes to data or views is avoided. *Binding* views to data variables is done via ordinary assignments; $\#t = v$ specifies a one-way binding, i.e., $\#t$ is re-rendered with data context of v whenever v changes. Reactive variables omit the need of an explicit `signal` or `stream` datatype [16, 25, 15] for specifying data models.

One Data. Duplication of model data across client, server and database leads to synchronization issues: developer must serialize, de-serialize data across tiers, ensure client data copy is synchronized with server copy at all times. All of this leads to erroneous, potentially redundant, glue code. WEBNAT has a simplified data model: all data is specified using typed variables (of Boolean, Integer, String, Array, Map and Record types) and apps work with a single copy of data. Variables required to be stored in a database are simply marked *persistent* during declaration; otherwise they are assumed to be volatile. The handlers read and write all data variables directly, as in e.g., Java, without handling persistent variables separately. Because most web apps need to query persistent data, WEBNAT includes an inbuilt *data query language* to enable filtering and manipulation of arbitrary data collections (Arrays and Maps), directly inside handlers. The *one-data* principle allows the developer to design apps in a simplified manner, *oblivious* to both database and client-server separation.

Integrate with JavaScript ecosystem. WEBNAT builds over the JS ecosystem: it borrows JS syntax and adds additional constructs to specify first-class views, handlers and queries. Data placeholders inside views are specified using Handlebars [4] syntax (double curly braces, cf. Fig. 3). WEBNAT interoperates with JS libraries both inside browser (Document Object Model) and server-side (*nodejs* [11]). This allows us to benefit from the rapidly growing JS web platform [11], fast event-driven programming model, rich set of server-side libraries [10], share code between client and server, as well as reuse conventional DOM manipulation libraries. WEBNAT also provides a utility library to abstract interaction with external REST [17] APIs via AJAX [21].

Compiler. Translating the high-level WEBNAT designs to executable client-server code is done by the WEBNAT compiler, which performs systematic multi-phase transformations. Currently, the compiler generates JS code for both client and server. Client-side reactivity is implemented using the Backbone library [3] with a Handlebars compiler [4]. The server-side code uses the `nodejs` [11] platform and `MongoDB` [9] database. Models are transformed to JS objects; persistent models are converted to collections in MongoDB. Reads or writes to persistent variables are translated to under-the-hood database calls. WEBNAT handlers are translated to asynchronous JS event-handling callbacks. We chose MongoDB to store persistent data because translating from typed (array or map) WEBNAT variables to JSON-like MongoDB collections with dynamic schemas is simpler than normalizing them into fixed table-like schemas of relational databases. Note that although our prototype compiler targets specific languages or libraries, WEBNAT programs can be compiled to other web frameworks also, e.g., based on Python or PHP. A formal description of various compiler phases and type checking is beyond the scope of this paper. Our goal in this work is to motivate the syntax and the design decisions for WEBNAT and carry out a user study to understand user reactions and adoption barriers.

2.1.1 WEBNAT *by Example*

We illustrate the WEBNAT language using the *todo* list app shown in Fig. 3: the app allows user to dynamically add and remove work items. The app consists of a set of view templates, model declarations, binding definitions and handlers for dynamic behavior. Although rather simple, this app employs many patterns that the majority of modern web applications may have.

View Templates. The *todo* app consists of three main UI elements (view templates): a label $\#lab1$ with text *Enter an item:*, an input $\#textbox$ to enter a new item and a list container $\#Tab$ to display the pending items. In WEBNAT terminology, $\#Tab$ is a *repeater* view [30]; such views are created by multiple instantiations (here, a list) of a particular *prototype* view. Here, each prototype view of $\#Tab$ (Fig. 3) in turn consists of three sub-views: a checkbox $\#checkb$, a text label $\#data$, and an image $\#cross$. Note the placeholder `{{this.value}}` inside the label #data: this syntax denotes that the label will dynamically obtain its content text from the $value$ field of the data bound to the prototype view.

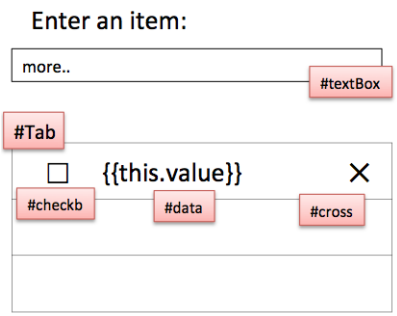
Models. The model for *todo* app is an array *todos* with elements of type *Item* which is *persistent*, as specified below. Each type has a default object instantiation, e.g., {value: '', done: false} for type *Item*.

```
type Item = {value: string, done: boolean};
var todos: array(Item), persistent ;
```

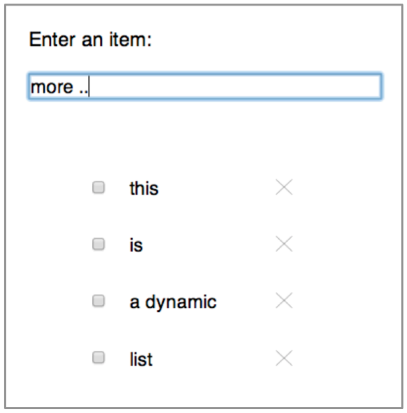
Bindings. The WEBNAT primitive `bind` is used to associate the array *todos* with the view reference #Tab as follows.

```
<head>
  <title>todo</title>
</head>

<body>
  <input id='textBox' type='text'
         placeholder='more..'/>
  <ul id='Tab'>
  {{#prototype}}
    <li>
       <input type='checkbox' id='checkb'/>
       <span id='data'> {{this.value}} </span>
       <span id='cross'> X </span>
    </li>
  {{/prototype}}
  </ul>
</body>
```

(a)



(b)



(c)

Figure 3: Todo list example in WEBNAT. (a) View template in text (Handlebars format [4]) (b) template visualization with labeled view elements, and (c) the generated app.

```
bind ({'#Tab': "todos" });
```

or, as an imperative assignment using the *always* handler:

```
always -> { #Tab = todos }
```

Consequently, each prototype view of $\#Tab$ is bound to an element of *todos* with type `Item` (the $\#data$ label obtains its value from $value$ field of `Item`). Because all data variables are reactive, any changes to *todos*, e.g., addition, deletion of elements, result in reactive updates (add, remove rows) to $\#Tab$. Note that by declaring *todos* as persistent, the developer avoids synchronizing duplicates of *todos* across client and server. Also, by binding the repeater $\#Tab$ with *todos* declaratively, the explicit view refresh on each update is avoided. Using `bind` is convenient when binding multiple view names and variables; *always* instead is preferable when binding with expressions or queries over variables.

Handlers. Handlers capture interactive behaviors and are specified in the *event -> actions* format, similar to event listeners in JS: when the triggering *event* occurs, the *actions* are executed. Event names are either standard DOM events or thin macros over them. The actions are either imperative statements over models and views or may invoke external JS functions. Although handler actions depend on reactive variables, they do not re-evaluate unless the triggering event occurs.

The *todo* app contains two handlers defined below. The first handler executes when the input $\#textBox$ receives *enter* key event: the contents of $\#textBox$ (if non-empty) are added to the *todos* model and $\#textBox$ cleared. The second handler specifies that on clicking the $\#cross$ image for any prototype sub-view, say $\#p$ in $\#Tab$, the model ($item$) bound to the view $\#p$ is removed from *todos* array (thus removing the sub-view $\#p$ reactively). Note that handlers over repeater elements take in the bound data as parameter, e.g., $item$ for $\#cross$ in the second handler, which can be read and modified as required in the handler.

WEBNAT includes two custom events, *init* and *always*: an handler with *init* event is executed only once when the app loads; *always* handlers re-execute whenever the dependent variables change.

```
keyenter #textBox -> {
  if (!empty(#textBox)) {
    todos += #textBox;
    clear(#textBox);
  }
}
```

```
click #cross(item) -> {
  remove(todos,item);
}
```

Succinct yet Expressive. The WEBNAT todo app specification is quite succinct while detailed enough to create the app in Fig. 3(c). A number of characteristics of WEBNAT handlers enable such intuitive specifications. (A) View references, e.g., $\#textBox$, may be used freely in statements, similar to data variables. View references may also be overloaded depending on the context, e.g., in the first handler, $\#textbox$ is used both to refer to the UI widget as well as the model (string entered by user) at different places. (B) We are able to avoid writing additional glue code to refresh $\#Tab$ on *todos* model updates because (i) $\#Tab$ is specified as a *repeater* view and (ii) *todos* is bound to $\#Tab$. (C) WEBNAT also uses *implicit* type coercions [31] extensively, e.g., in the first handler, the view reference $\#textBox$ is first coerced into its model value, say s of type `String`, which is further coerced to a record value (based on default field values) before adding to the array. This is roughly equivalent to the following JQuery-based code.

```
var s = $('#textBox').val(); //JQuery
var v = {value: s, done: false}; todos.push(v);
```

(D) WEBNAT's *one-data* design makes persistent objects, e.g., *todos*, indistinguishable from volatile objects. Both are manipulated via same set of read/write operations; the compiler ensures that persistent objects are handled under-the-hood correctly. Consider the statement *remove(todos,item)* in the second handler. Because *todos* is persistent, the app runtime keeps two data copies, one at the client and other in a MongoDB collection at the server end. The WEBNAT compiler transforms the above high-level statement into client/server operations as follows. The client code issues an Ajax [21] DELETE request to invoke the server-side logic, which in turn, makes a database query to remove the *item* value from the *todos* collection in database. On successful completion of this request, *item* is also removed from the local *todos* collection. This results in an reactive update of the $\#Tab$ view on client-side

and completes the remove operation.

Similarly, the update `todos += #textBox`, is implemented by sending the model string of $\#textBox$ with an Ajax PUT request to the server-side logic, which inserts the corresponding entry in the *todos* collection in the database. Further, WEBNAT enables inbuilt model querying based on MongoDB syntax, e.g., `todos{done: true}` will select all items whose *done* field is *true*.

In summary, WEBNAT exploits the combination of model-view binding, one-data modeling and reactivity to devise powerful syntactic abstractions which simplify specification of web apps. Note that, in absence of these abstractions, the developer must manually write the error-prone glue code to perform data synchronizations across multiple tiers and update views reactively.

Asynchronous Calls, APIs, Navigation. WEBNAT includes a utility library with functions for invoking external services, e.g., *restGet, restPost, restPut, restDelete* allow communication with external REST [17] services. Errors due to remote asynchronous calls are handled in the usual JS-style by passing additional *onerror* handlers to the calls. The library also contains wrappers functions for DOM manipulation and type transformations and custom event emitters. Each web page corresponds to a collection of view templates; navigation between pages is achieved via *routes* similar to Backbone [3]. A module system for encapsulating MVC elements to enable reuse is under development. Few other extensions, e.g., role-based access control of data models are also planned.

Expressiveness of WEBNAT. We have designed a wide variety of complex apps in WEBNAT, including form-based applications and personal information management, e.g., mail clients, authentication and inventory management. We have also developed a *map* application prototype in WEBNAT, similar to Google Maps. Wright [35] aggregates the features of common rich internet apps (RIAs) in a *ticket* handling app. All these features can be specified in WEBNAT faithfully. A specification of a large app, a mail client, in WEBNAT is provided in the Appendix of this paper.

Illustration: A REST-based Music Browser. We describe how a simple music browser app is specified in WEBNAT: the app first gets a list of songs from an external REST service (in JSON format) and then displays the song details in a master-detail format. Fig. 4 shows the views of the app.

```
type Song = {album: string, song: string, lyrics:
     string};
var songs: array(Song);
bind ({'#catalog': "songs" });

init -> {
  #catalog = restGet("http://songdb.com/songs")
}
click #catalog-item (item) -> { #detail = item }
```

The first handler is invoked on page initialization, when *init* (a custom WEBNAT event) is triggered. The call to *restGet* is made asynchronously and the returned value (in JSON format) is bound to the model of the *repeater* view $\#catalog$. The compiler translates this handler into a callback invoked after results from the AJAX request is obtained. In the second handler, $\#catalog-item$ denotes the prototype sub-view of $\#catalog$ which is bound to *item* model. On clicking any sub-view, the model of $\#detail$ view is set to *item*, thus refreshing the view with the current song details. Note that automatic type coercion takes care of transforming the JSON object from *restGet* into the typed *songs* variable bound to $\#catalog$.

Comparison with Web Frameworks. Upcoming web frameworks like Angular [2], React [6], Meteor [8] include primitives for simplifying web programming. Angular enables reactive data-bindings (only on client-side) via a collection of new primitives, e.g., `scope`, `ng-controller`, `ng-bind`, which involves learning multiple concepts for specifying binding and reactive controllers. Instead, React allows declaring new *view components* which embed the templates and the models and can be dynamically instantiated. In contrast to WEBNAT, which ensures a clear *separation* between model and view references via `bind`, both Angular and React lead to quite interwoven HTML/JS code; also, they do not allow native view references (and therefore model-view type coercions) or enable end-to-end reactivity. The Meteor framework [8] is closest to WEBNAT: it provides a unified data model and end-to-end reactivity using a special datatype for reactive variables and APIs for constructing reactive expressions. The key distinguishing aspects of WEBNAT are (i) native view references (ii) statically-typed, natively-reactive data variables, (iii) imperative bindings, (iv) implicit coercions. Also, the static type system ensures easy bindings, e.g., between repeater data and array models, via implicit type coercions. Finally, although all these frameworks are quite popular among developers, we do not know of any systematic user study with them.

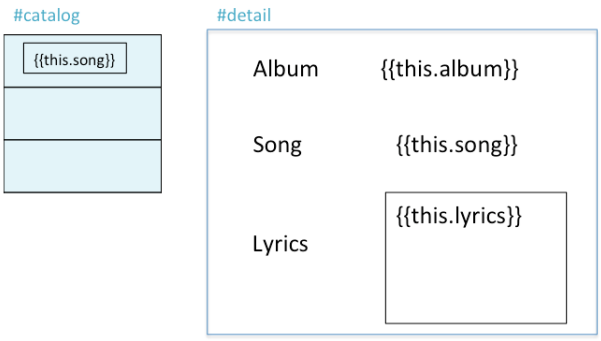


Figure 4: Mockup of the Music Browser.

3. USER STUDY

We conducted a study of the WEBNAT environment to investigate if programmers find it useful as compared to existing web frameworks and to discover the hurdles in adoption.

3.1 Participants

Twelve participants (3 females, mean age=28.5, sd=3.435) were recruited from an IT organization. Participants' experience with web programming varied with 4 novice (less than 1 year experience), 3 semi-professional (1-5 years of experience) and 5 professional (more than 5 years of experience) programmers. The criterion for participation was that they have developed at least one end-to-end, multi-tier web app in past. All the participants were familiar with the MVC design pattern, with eight of them reported using MVC frameworks regularly. We used a combination of e-mailing to organization's internal mailing list, word-of-mouth, and snowball sampling to recruit our participants.

3.2 Method

A within-subject study was conducted. Participants were asked to develop a web app, using their preferred framework and using WEBNAT. The ordering of the condition was counterbalanced across participants. Participants were asked to think-aloud [24] during the whole session to obtain insights into their thought processes, and their web development process. Participants were allowed to use any search engine for their preferred framework, and able to

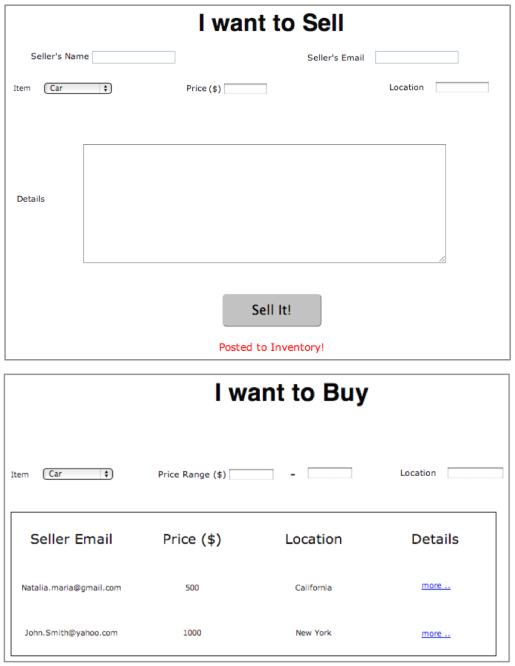


Figure 5: Initial mockups for buy and sell pages in the study task. Final mockups and WEBNAT code are shown at the end of the paper.

ask syntax-specific questions to the facilitator for WEBNAT (e.g., how do I add an object to an array?, what is the syntax for writing a query?). All the sessions were voice recorded and later transcribed.

At the beginning of the study, the facilitator provided the functional specification of the app and answered participants' questions about the app. For the preferred framework condition, the task was to develop the app in their framework of choice in 60 minutes. After that, participants were asked to fill a questionnaire with questions rated on a 5-point Likert scale and free-form questions about the limitations of their current framework and suggestions for improvement. Participants who were not able to finish this task in the specified time were asked then to enumerate all the low-level design steps they have to perform in their framework to implement this app and provide an estimate of the total time required.

For the WEBNAT condition, participants were given a 30-minutes tutorial explaining how to create the *todo* app in WEBNAT, to make them familiar with our framework. The tutorial included step-by-step instructions with screenshots which allowed the participants to get hands-on programming experience WEBNAT's syntax for writing models and handlers and basic Handlebars syntax [4]. After the participants completed the tutorial, they were asked to create the task web app using WEBNAT in 60-minutes. This was followed by the same questionnaire as in the preferred framework condition. The participants were also provided a quick reference sheet, designed using Neilsen's heuristics [29], describing how to specify model-view binding, repeaters and queries and diagnose errors quickly. The prototype WEBNAT tool provided support to automatically check the syntactic correctness of the program.

3.2.1 Tasks

The task involved building an online *buy-sell* app consisting of three pages: a *seller* page where users can post items to sell (Fig. 5(left)), a *buy* page where users can search for a particular item (Fig. 5(right)), and a *main* page for navigating to buyer and seller page (not shown).

As the aim of our study was not to test UI-designing capabilities of the frameworks, we provided the participants with HTML and CSS files for the three pages. Participants were asked to implement the core functionalities, i.e., posting a new item to sell, validating input fields, and displaying all the items satisfying a basic word-based search criterion.

For the first task, the developers were given the specification together with the three HTML/CSS files and asked to implement the core functionalities of the app using their *preferred* framework. Some participants expressed inability to finish this task in 60 minutes. We instead asked them to enumerate all the low-level design steps they must perform in their framework to implement this app and estimate the total time. Also, they were asked to note down the key error-prone steps which require a lot of debugging.

For the second task, asked them to specify the app in WEBNAT using the three HTML/CSS files. We asked the participants to think aloud [24] to obtain insights into their thought processes and the problems they encountered.

Limitations. The validity of our study results is dependent on the set of participants we recruited, all of whom were from an IT organization. None of the participants were familiar with newer frameworks, e.g., Backbone, Angular or Meteor. Our study considered only a single web programming task which involved developing a simple app, given the time constraints. We did not consider aspects of building secure apps in this study. Additional studies may be required to examine the development of richer apps. Prior familiarity with WEBNAT may also lead to different results.

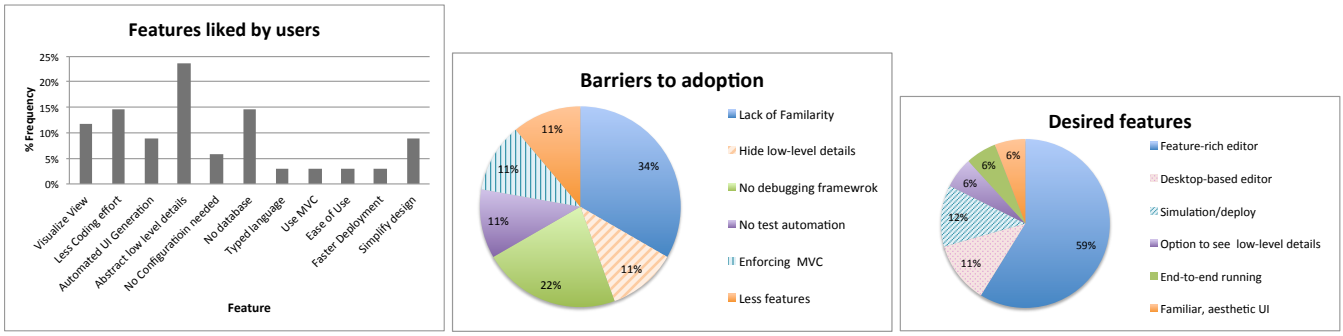
3.3 Study Results

Using their preferred system. The participants used a wide variety of preferred frameworks to complete their task: PHP, J2EE, Groovy on Grails, Ruby on Rails, Perl, nodejs for server-side logic and MySQL, MongoDB, Hibernate for database interaction. On the client side, besides HTML/CSS and form actions, some used template libraries, e.g., JSP, GSP and Ajax calls for client-server interaction. Out of 12 participants, 8 were not able to finish the task in the given period using their preferred system. The main reasons for not finishing included low-level coding required by their framework, coding and debugging client-server communication for Ajax queries, writing correct SQL queries, server and database configuration and setup issues.

Programming with WEBNAT. All 12 participants completed the WEBNAT tasks. To program the app using WEBNAT environment, the participants declared variables (models), the model-view bindings and the handlers for each of the three app pages. For the seller page, most participants identified that models need to be bound with input text-boxes and their value needed to be stored in a database. However, some of them initially overlooked the method to specify a persistent variable in WEBNAT and instead declared a separate variable representing the database. The mistake was corrected when writing the handlers for the *sell* button. Some of the participants also took time to decouple the model and the view in the search results table on the buy page.

At the end of the survey, all participants agreed that the coding overhead in WEBNAT was low. All except one participant said that they would have to write a lot more code in their favorite framework for this app. Many participants enjoyed coding in WEBNAT: participant P2 commented *".. this was a new paradigm of programming"*. Similarly P5 said *" ... I needed to program in a completely new programming style"*. P9 noted *"Seems like a good environment for collaborative development..focus on core functionality, worry less about low-level details"*.

Other frameworks. Participant P5 commented *"although you can*



(a) (b) (c)

Figure 6: (a) Features of our tool liked by participants (b) Barriers for professional participants in adopting our tool (c) Proportions of tool features desired by the study participants.

drag-drop in iOS storyboards also, you cannot write views so easily..that takes 2-3 steps...if the widget is not inbuilt, you need to do lot of Objective-C coding.". P10 said *"Writing standard form apps in my framework doesn't require much coding. However, I spend lot of time debugging Ajax queries to backend...they are hard to get right.".*

Syntax. Participants were able to appreciate WEBNAT despite the new WEBNAT syntax and programming style. P5 commented *" ... it is hard to learn a language in such a short time...this only needs practice, the learning curve is OK."* . Most participants agreed that the usefulness of the inbuilt abstractions outweighed the initial learning curve. P12 commented *" ... I didn't have to worry about so many low-level details. I liked the idea of modeling database as variables. .. it does need me to learn the syntax, but that is just the same as learning any new language."*

Learning Curve and Adoption. Six of the participants said that they will adopt our system for regular use, 5 declined and 1 participant answered *maybe*. Professional developer P4 commented that *"I'm not sure this system can design complex apps. But this seems good for designing simple mobile apps.".* We asked participants the major reasons behind their decisions of not adopting our tool. Fig. 6(b) summarizes their responses: the main hurdle was lack of familiarity with our new environment and the programming language. Many participants also asked for extensive debugging and testing automation support, which is absent in the current prototype. Some participants were quite skeptical - P10 said *".. my framework (Groovy on Grails) has so many features, session cache management, duplicate form submits, authentication, pagination, third-party integration .. not clear if this tool has all these features.".*

We also asked participants about the features they wanted to have in our tool (Fig. 6(c)). The majority of the participants wanted a feature-rich editor with more keyboard shortcuts, better context-sensitive assistance for writing code (reusing and refactoring model/handler definitions, help menus, drag-drop over tree items). Some participants wanted more low-level control over the generated code. P5 commented *"For each handler I write, I would like to see the code for it. Just to make sure the tool is doing right. For complex apps, I want to see the code and debug myself."* Other participants asked for real-time simulations of the model, desktop-based editor instead of web-based, and a more professional UI for the tool.

We asked all participants to rate the tool on a scale of 1 to 5. All the participants rated the tool 3 or more (mean=3.75, sd=0.622) (cf. Fig. 7). The Pearson correlation coefficient value of -0.749 between participants' years of experience and corresponding rating indicates

Years of Experience	Avg. Rating	Std. Dev.	Decision
< 1	4.25	0.5	Yes
$2-3$	4	0	Yes
$4-5$	4	0	No
> 5	3.2	0.447	Maybe, No

Figure 7: Participant's overall rating of our tool (mean and sd) grouped by their years of experience. Decision = whether they would like to switch to our tool.

that novice and semi-professional developers gave a higher rating to our tool compared to professionals. We also asked participants if they would use our tool instead of their current framework. Fig. 7 shows that novice users may be ready to switch to our tool while the rest are either hesitant or not ready to switch to the current prototype of our tool.

3.4 Implications

Abstractions for Web. Our study shows that the web development can be accelerated by incorporating new programming abstractions based on the MVC paradigm. Most popular frameworks do contain some abstractions, e.g., database ORMs, templates, client-side reactivity. However, in each framework, developers continue to struggle with low-level details between one or more layers, e.g., client-server data transformation and switching between syntax of multiple languages. New frameworks for web development should try to simplify *end-to-end* development across all tiers.

Low-level control. Although abstractions help developers reduce their coding effort, professional programmers often want full control over the generated code for debugging, adding optimizations or simply browsing. By exposing the low-level code in a user-friendly format, the development tool can cater to all programmers, from novice to professionals. Enabling modifications to the generated code is feasible but requires more investigation.

Adoption barriers to new Web DSLs. We observed that experienced developers are often quite critical of newer frameworks/DSLs and hesitant to move away from their favorite framework. This is because the established frameworks provide the developer with a wide set of features and default configuration options, enabling them to build complex, state-of-the-art systems. Developers become oblivious to hardships in their framework, e.g., writing and fixing low-level, client-server glues, debugging database queries, as long as it helps them build industrial apps. Therefore, in spite of finding value in reactive programming, one-data and easy bindings in WEBNAT, experienced developers are skeptical unless the new framework also provides them with all the debugging, deployment

and configuration features that they have used in building real apps. We believe that this is the crucial lesson that we learned during the course of evaluating this work.

Consequently, to encourage rapid adoption, it is more pragmatic to incrementally augment existing popular frameworks with new abstractions, e.g., by creating framework-specific plugins/libaries, instead of creating a new ecosystem from scratch. Our choice of building WEBNAT upon the JS ecosystem is a step in that direction – we are investigating ways to embed custom WEBNAT syntax, e.g.,imperative bindings and implicit coercions into conventional JS. This approach may inspire a slow-paced, gradual improvement of development practices even among hesitant professionals.

4. RELATED WORK

The UWE [23] tool enables systematic MDD of web applications by separately modeling the content, navigation, business processes and presentation elements based on UML extensions. Platform-independent models are transformed systematically to obtain platform-specific application. WebML [13] provides a platform-independent conceptual model for data-intensive web applications with extensions for web services, process modeling and context awareness. WebML has evolved and is standardized into IFML [5] which provides a visual notation for declaring models, views, data bindings and flows. Hera [18] also allows specifying abstract presentations which are combined with different models in a run-time page creation environment.

In contrast to fully-visual approach of UWE and WebML, WEBNAT is a conventional language where models and controllers are written textually while views may be written using a template language or drawn, e.g., using a WYSIWYG editor. Textual specifications are attractive to most developers because they are familiar with one or more web programming frameworks; hence they can adapt to WEBNAT quickly. Also, WEBNAT controllers can interoperate with existing JS-based libraries easily. Our main contribution is a set of abstractions to simplify writing web apps: first-class views, inbuilt reactivity and simplified model-view binding.

The MARIA [30] framework allows users to specify UIDLs which are fully platform-independent (abstract) or include partial platform details (concrete). WEBNAT may be viewed as a concrete language which builds upon the conventional JavaScript syntax and event nomenclature and provides new abstractions to overcome common hurdles during app development. The PUC project [28] used model-based interface generation to control common appliances by programming mobile phones and PDA devices. In contrast, WEBNAT makes it easy to specify model-view transformations but does not guess views - the designer must provide them. García et al. [20] provide a recent survey of various UIDLs.

Several researchers have conducted surveys to understand web development processes and needs of end-users, semi-professional and expert developers. Rode et al. [33] investigate the key development issues faced by web developers, where cross-browser compatibility and security turn out to be central, followed by integration problems between different tiers. The multiple phases involved in UI design [26] are studied in [27] by interviewing 11 experts. We are unaware of any user studies with new JS web frameworks, e.g., Backbone, Angular or Meteor.

Reactive functional programming [16] frameworks, e.g., [25, 15], focus on programming with signals/streams to simplify interactive programming but do not consider implicit model-view bindings or enabling MVC-based separation. The seminal Links [14] framework proposed web programming based on a monolithic, functional, statically typed language which is compiled to different languages for each web tier. However, the principles of client-side templates and implicit, reactive data binding are not discussed; the programmer must code explicit view refreshes and track model changes. A number of libraries and DSLs for easing web programming have been proposed recently [22, 34, 19, 15, 12]. Although many of these DSLs, e.g., Mobl [22], Dog [12], endorse one or more helpful abstractions, e.g., unified client-server state, inbuilt data queries, they fail to enforce end-to-end MVC design: the programmer has to be careful about separating models, views and controllers and write additional glue code between views and models. Other DSLs, e.g., TouchDevelop [34], Pulpscription [19] and Elm [15] contain primitives for animation and game development which WEBNAT lacks currently. Because WEBNAT has MVC as its foundation, we believe that it is possible to augment the language by adopting useful primitives from other DSLs and providing library support.

5. CONCLUSION

We presented a new programming language WEBNAT based on the model-view-controller (MVC) paradigm [32] to enable simplified end-to-end web programming, agnostic of the client-server-database tiers. The language consists of primitives to specify typed, reactive models, native view references and declarative model-view binding across tiers. A within-subject study of a prototype WEBNAT implementation shows that web programmers like the simplifications and reduced coding effort provided by this environment. In contrast to novice web developers, professionals hesitate to adopt WEBNAT because they are used to low-level control over code and feature-rich, industrial web frameworks. In future, we plan to provide additional context-sensitive help, debugging tools and convenience features to improve the programming experience. We also plan to improve the compiler to match the quality of hand-coded apps and support other non-JS web frameworks.

6. REFERENCES

[1] http://www.smashingmagazine.com/2011/06/22/following-a-web-design-process/.

[2] Angularjs - superheroic javascript mvw framework. http://www.angularjs.com/.

[3] Backbone.js. http://backbonejs.org/.

[4] Handlebars.js: Minimal templating on steroids. http://handlebarsjs.com/.

[5] Ifml: The interaction flow modeling language. http://www.ifml.org/.

[6] A javascript library for building user interfaces. http://facebook.github.io/react/.

[7] jquery: The write less, do more, javascript library. http://jquery.com.

[8] Meteor. http://www.meteor.com/.

[9] Mongodb. http://mongodb.org/.

[10] Node packaged modules. https://www.npmjs.org/.

[11] node.js. http://nodejs.org/.

[12] S. Ahmad and S. Kamvar. The dog programming language. UIST '13, pages 463–472, 2013.

[13] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers Inc., 2003.

[14] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. FMCO'06.

[15] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. *SIGPLAN Not.*, 48(6):411–422, June 2013.

[16] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, 1997.
[17] R. T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, Univ. of California, Irvine, 2000.
[18] F. Frăsincar, G.-J. Houben, and P. Barna. Hypermedia presentation generation in hera. *IS*, 35(1), 2010.
[19] M. Funk and M. Rauterberg. Pulp scription: a dsl for mobile html5 game applications. ICEC'12, 2012.
[20] J. G. García, J. M. González-Calleros, J. Vanderdonckt, and J. M. Arteaga. A theoretical survey of user interface description languages: Preliminary results. In *Proc. LA-WEB/CLIHC*, 2009.
[21] J. J. Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/`.
[22] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. *SIGPLAN Not.*, 46(10):695–712, Oct. 2011.
[23] N. Koch, A. Knapp, G. Zhang, and H. Baumeister. UML-Based Web Engineering - An Approach Based on Standards. In *Web Engineering*. 2008.
[24] C. H. Lewis. Using the "thinking aloud" method in cognitive interface design. rc 9265, ibm, 1982.
[25] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. OOPSLA '09.
[26] B. A. Myers, S. E. Hudson, and R. F. Pausch. Past, present, and future of user interface software tools. *TOCHI*, 7(1), 2000.
[27] M. W. Newman and J. A. Landay. Sitemaps, storyboards, and specifications: a sketch of web site design practice. DIS, pages 263–274, 2000.
[28] J. Nichols and B. A. Myers. Creating a lightweight user interface description language: An overview and analysis of the personal universal controller project. *TOCHI*, 16(4), 2009.
[29] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. CHI '90, pages 249–256, 1990.
[30] F. Paternò, C. Santoro, and L. Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *TOCHI*, 16(4), 2009.
[31] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
[32] T. Reenskaug. Thing-model-view-editor. Tech. report, Institutt for informatikk, University of Oslo, 1979.
[33] J. Rode, M. B. Rosson, and M. A. P. Qui. End user development of web applications. In *End User Development*, pages 161–182. Springer, 2006.
[34] N. Tillmann, M. Moskal, J. de Halleux, M. Fähndrich, and S. Burckhardt. Touchdevelop: app development on mobile devices. In *SIGSOFT FSE*, 2012.
[35] J. Wright. *A Modelling Language for Interactive Web Applications*. PhD thesis, Massey Univ., NZ, 2011.

Buy-Sell specification in WEBNAT

```
type Item = {name: string, email: string,
   item: string, price: int, location: string,
   details: string};
```

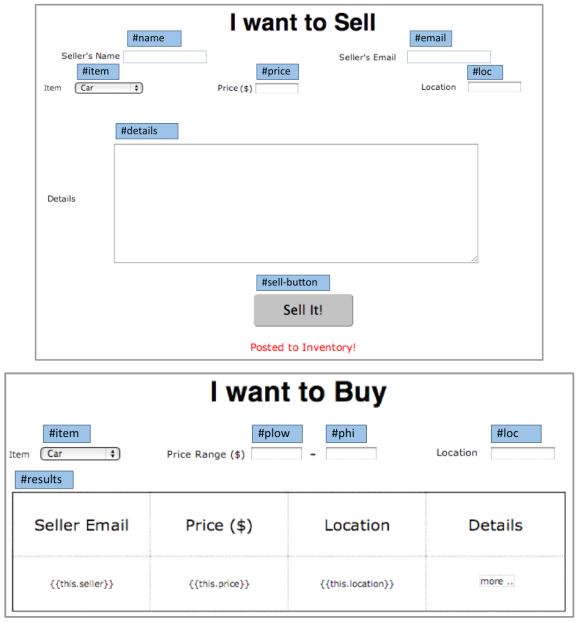


Figure 8: Visual mockups (final) for buy and sell pages in the study task.

```
//WebNat helper for checking validity of inputs
//Validity is specified on type fields
validCheck ({
 'Item.email': function (email) {..},
  ...
 });

var item: Item;
var inventory: Array(Item), persistent;
```

For the **Sell** page:

```
bind ({
   '#name': 'item.name',
   '#email': 'item.email',
   ... //all other view elements
})
click #sell-button -> {
  inventory += item
}
```

The above handler reads *item* variable and adds to the inventory on clicking the Sell button. Note that reading *item* corresponds to reading from all the view elements bound to its fields. The validator for each field of *item* is fired each time *item* is read.

For the **Buy** page:

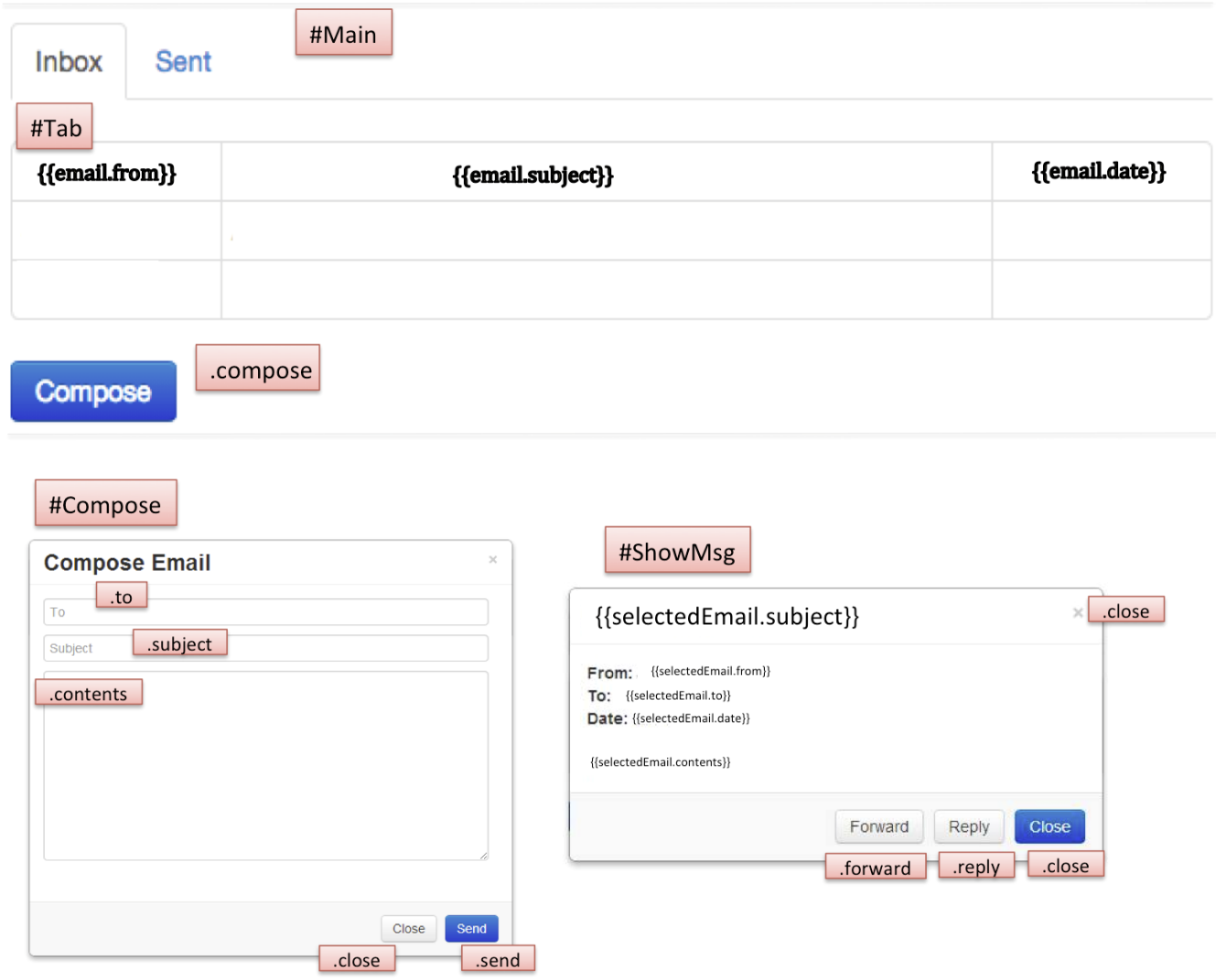
```
always -> {
  #results =
    inventory{item: #item, location: #loc,
      price: {$lte: #phi, $gte: #plow} }
}
```

This handler binds the result of query on *inventory* to the $\#results$ repeater view. Note how view references are used to refer to corresponding models indirectly. Also, the *always* event ensures that $\#results$ is updated on-the-fly as the user types in a valid input for one or more query fields.

Appendix: A Mail client in WebNat

The following is a textual description of a simple mail client in WebNat, followed by the corresponding views, labeled by view references.

```
type Email = {from: 'string', to: 'string',
    subject: 'string', date: 'string', contents:
    'string', box: 'string'};

var allEmails : array(Email), persistent;
var selectedEmail, composeEmail: Email;
var hideShowMsg, hideCompose: bool;
var currTab: string;

//binding views to their default models
bind ({
   '#Compose': {
     ".to": 'composeEmail.to',
     ".subject" : 'composeEmail.subject',
     ".contents" : 'composeEmail.contents'
   }
 });

handlers #Main {
  init -> {
    currTab = 'inbox';
    allEmails = [
       {
          from: 'WebNat',
          to: 'me',
          subject: 'Welcome to WebNat!',
          date: 'Jan 1',
          contents: 'hello world!'
          box: 'inbox'
       }
    ]
  }

  click tr (item) -> {
      selectedEmail = item;
      hideShowMsg = '';
  }

  click .compose -> {
    hideCompose = '';
  }

  click li .inbox -> {
    currTab = 'inbox';
  }

  click li .sent -> {
    currTab = 'sent';
  }

  always -> {
    activeInbox = (currTab == 'inbox')? 'active':
        '';
    activeSent = (currTab ==='sent')? 'active': ''
        ;
    //filter out inbox emails or sent emails from
        allEmails based on the currTab
    #Tab = (currTab == 'inbox')? allEmails{box : '
        inbox'} : ( (currTab == 'sent') ?
        allEmails{box: 'sent'} : [] );
  }

}


handlers #ShowMsg {
   init -> {
     hideShowMsg = 'hide';
   }

   click .close -> {
     hideShowMsg = 'hide';
   }

   click .reply -> {

     composeEmail.to = selectedEmail.from;

     composeEmail.subject = "Re: " + selectedEmail
         .subject;
     composeEmail.contents = "\n\n=======\n" +
         selectedEmail.contents;
     hideShowMsg = 'hide';
     hideCompose = '';
   }
}

handlers #Compose {
   init -> {
     hideCompose = 'hide';
   }

   click .close -> {
     hideCompose = 'hide';
   }

   click .send -> {
     composeEmail.from = 'me';
     composeEmail.date = new Date();
     composeEmail.box = 'sent';
     //replace this by javascript smtp client
     console.log ('send mail to ' + composeEmail.
         to + ', subject: '+ composeEmail.subject)
         ;
     allEmails += composeEmail;
     hideCompose = 'hide';
   }
}
```


#Main
Inbox
Sent
#Tab
{{email.from}}
{{email.subject}}
{{email.date}}
Compose
.compose
#Compose
Compose Email
.to
To
Subject
.subject
.contents
Close
Send
.close
.send
#ShowMsg
{{selectedEmail.subject}}
.close
From: {{selectedEmail.from}}
To: {{selectedEmail.to}}
Date: {{selectedEmail.date}}
{{selectedEmail.contents}}
Forward
Reply
Close
.forward
.reply
.close