Evolution of the WebDSL Runtime

Reliability Engineering of the WebDSL Web Programming Language

Danny M. Groenewegen
Delft University of Technology
The Netherlands
d.m.groenewegen@tudelft.nl

Elmer van Chastelet
Delft University of Technology
The Netherlands
e.vanchastelet@tudelft.nl

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

ABSTRACT

Web applications are ideal for implementing information systems; they can organize and persist the data in a database, do not require installation on client machines, and can be instantly updated everywhere. However, web programming is complex due to its heterogeneous nature, causing web frameworks to suffer from insufficient or leaky abstraction, weak static consistency checking, and security features that are not enforced. We developed the WebDSL web programming language, which supports direct expression of intent, strong static consistency checking, linguistic abstractions for web programming concerns, and automatically enforces security features for web applications. We have used WebDSL for over 10 years to create information systems for academic workflows with thousands of users. Based on our experiences with these applications, we improved the WebDSL compiler and runtime to increase robustness, performance, and security of applications. In this experience report, we reflect on the lessons learned and improvements made to the language runtime.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Compilers**; **Runtime environments**; **Integrated and visual development environments**.

KEYWORDS

domain-specific languages, web programming, web applications, web security, compilers, integrated development environments, experience report

ACM Reference Format:

Danny M. Groenewegen, Elmer van Chastelet, and Eelco Visser. 2020. Evolution of the WebDSL Runtime: Reliability Engineering of the WebDSL Web Programming Language. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3397537.3397553

1 INTRODUCTION

Information systems store data, organize data, and manage business processes. For example, in a university information systems are used to track student progress, store grades, manage individual study program selection, and create overviews for university staff. Web applications are ideal for implementing information systems: they organize and persist all data in a database, protect data by only allowing specific operations, do not require installation on client computers, and can be upgraded without interruption.

Unfortunately, web application development is complex due to its heterogeneous nature. It involves multiple programming languages with their own programming models (e.g. DOM updates with client-side Javascript, server-side Java code to execute operations, SQL database queries), and separate software systems in a network (browser, proxy server, application server, database server) that all need to work together. Additionally, there are non-functional requirements inherent to the web platform such as protecting against request tampering and injection attacks.

Many web frameworks exist to assist programmers in organizing the complexity of web programming by enforcing standard patterns. These frameworks also have their own issues, such as having to write boilerplate code to glue together components, late integration checks between framework components, and weak IDE support for framework concepts [11]. Part of the problem is lack of collaboration between the programming language design and the framework design. For example, where a typical framework provides convenient ways to escape values in queries, query injection attacks can be prevented in a safer way if the programming language is made aware of queries, because the developer can forget about the problem entirely [2].

We have developed WebDSL, a domain-specific programming language that incorporates web framework concepts in the language. WebDSL provides linguistic abstractions for the various aspects of web programming integrated in a single language that produces code for the different tiers in a web application. The language abstractions allow direct expression of intent. Accidental complexity from boilerplate code and non-functional requirements is handled by the code generator and runtime. The language provides static consistency checking in the compiler and IDE which avoids consistency errors in the application definition. The WebDSL compiler generates a full Java web application that can be deployed on an application server.

We have been developing and using WebDSL for over 10 years to create information systems for academic workflows. The initial WebDSL research was focused on DSL compiler design [12, 13], language design for access control and data validation concerns [7, 8], and static consistency checking [11]. The applications we created at that point were prototypes and case studies, with few external users. Because the WebDSL compiler became more reliable over

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
<Programming'20> Companion, March 23–26, 2020, Porto, Portugal
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7507-8/20/03.
https://doi.org/10.1145/3397537.3397553

time, the applications became more ambitious. By now we have developed several applications with thousands of users:
EvaTool: a course evaluation application that supports processes for analyzing student feedback by lecturers and other staff.
WebLab: an online learning management system with a focus on programming eduction (students make programming assignments in the browser), with support for lab work and digital exams, used in multiple courses at TU Delft.
MyStudyPlanning: an application for composition of individual study plans by students and verification of those plans by the exam board, used by multiple faculties at TU Delft.
conf.researchr.org: a domain-specific content management system for creating and hosting integrated websites for conferences with multiple co-located events, used by all ACM SIGPLAN and SIGSOFT conferences.

We learned many lessons while developing these applications, which we used to improve the reliability of the WebDSL language and its runtime. The abstraction layer that the WebDSL language provides between application specification and implementation, entails that the time invested in fine tuning reliability, robustness, performance, scalability, and security of the language and its runtime benefits all applications. Engineering a reliable runtime requires coordination between all the heterogeneous components of a web application, and takes a lot of experimentation to improve. In this paper, we reflect on improvements we made to the WebDSL code generator and runtime and how they related to language design decisions. The contributions of this paper are:

(1) A reflection on design decisions in WebDSL and their impact on the reliability of the resulting applications.
(2) An experience report on robustness, performance, and security problems that came up in real-world web information system scenarios.
(3) Insight into the requirements for the next generation of multitier web programming languages.

2 WEBDSL LANGUAGE

In this section we discuss the design and implementation of the WebDSL language. In subsequent sections we focus on reliability aspects.

2.1 Language Design Principles

Based on the problems we observed in web programming languages, and our experiences in creating applications, we identify 5 design principles for WebDSL which guide the language design decisions.
Linguistic abstractions should enable direct expression of intent. Boilerplate code is generated or hidden in the runtime. Accidental complexity is removed, only essential complexity is expressed. Design language concepts with as much or little flexibility as required for the essential complexity.
Linguistic abstractions should ensure reliability and security. Applications should keep working when deployed in a real setting. This means the runtime should ensure robustness, performance, scalability, and also security, protecting against malicious web technology exploits (e.g. cross-site scripting or remote code evaluation). Exploit countermeasures are enforced in the runtime without adding complexity to application code.

```
entity User {
  username  : String
  username  : Strin
  password  : Secret
  allowEdit : {User} (allowed=from User as u where
                        u.usernam <> ~this.usrname) }
entity Note {
  author : User (inverse=notes)
}
template note( n: Note ){
  navigate editNot( n ){ "Edit note" }
  "~n.authr.username
  output( nn.content
}
principal is User with credentials usernam, password
access control rules
  rule page root { "true" }
  rule page editNote( owner: User ){
    principal == owner || owner in principal.canEdt
  }
```

There is no page with signature editNot(Note)
Press 'F2' for focus

Figure 1: Editor screenshot, example intentionally seeded with faults to show static checks in IDE

Static checking should present errors in terms of the domain. WebDSL is designed from the ground up with static analysis and cross-language consistency checking in mind. The IDE and compiler can analyze the code and immediately report errors. Because of the explicit syntactic constructs for language concepts, semantic errors can be precise and messages in terms of the domain concepts. Figure 1 demonstrates this with a screenshot of the WebDSL editor.
Extensibility should be explicit. Avoid abstractions from becoming leaky, in cases where knowledge of the generated code is required to complete the application. Extension with external components is done through explicit foreign function interfaces in the language, such as for invoking server-side Java or client-side Javascript libraries.
Lessons learned should be consolidated in the language. Language and applications should co-evolve, reflecting experiences from requirements engineering and application development in the language design. General problems found and fixed in applications should become language or library improvements, so that other applications automatically reap the benefits.

2.2 WebDSL Language Concepts

The WebDSL language consists of several concepts or sublanguages that work together in the specification of a complete web application. The three core language concepts in WebDSL, listed at the top of Table 1, are *data model* entities with automatic persistence to the database, *user interface templates* with safe HTML output and data binding in forms, and *functions* to implement operations on data. The rest of the table lists extensions to the core concepts and provides insight into the interactions between concepts. These interactions determine the static consistency checking the compiler and IDE perform and how error messages about failed consistency checks are phrased. For example, user interface templates require well-typed access to the data model; access control rules refer to defined user interface components for weaving in checks; the principal entity declaration needs to refer to entities and properties that

Table 1: WebDSL Language concepts and their interactions

Concept	Functionality
data model	data entity objects with database persistence primitive, reference, and collection types load/save functionality for objects
ui templates	pages connected by navigation links render HTML tags and **data model** values forms with databind update **data model** objects
functions	general-purpose object oriented language actions triggered from **ui templates** update **data model** objects with assignments
queries	query **data model** objects in **functions**
email	email templates, based on **ui templates** render send email trigger in **functions**
data validation	validation phase after databind **ui templates** **data model** invariants, **functions** assertions render messages in **ui templates**
access control	rule-based sublanguage to create security policy declare principal **data model** object rule checks can use expressions from **functions** rule needs to refer to existing **ui templates**
native classes	declare interface of Java code in **data model** create objects and invoke methods in **functions**
services	**ui templates** page request to generate JSON read incoming JSON request data in **functions**
search	search field mapping in **data model** search queries in **functions**
JS CSS embed	embed JS and CSS fragments in **ui templates**
Ajax updates	update subset of **ui templates** inside page

have been declared elsewhere. Access control rules define the accessibility to pages and templates. If a page does not have an associated access control rule, a warning is given to notify the developer that the page is inaccessible. In addition to static consistency checking, the links between language concepts also influence the runtime of the language. For example, email rendering reuses template rendering, but stores rendered content in an email queue instead of returning it in a response to the browser. Note that the table is not exhaustive; WebDSL supports additional concepts.

2.3 Example Application

Figure 2 shows an example WebDSL application to illustrate the main language aspects. Code comments in the example illustrate particular features being used in the line before or above. The example application is a tiny multi-user note taking application, in which a user can give edit access of (all) their notes to other users. The data model consists of two `entity` definitions (User at lines 1-14 and Note at lines 15-19) for data objects with persistence. The `root` page (lines 20-31) is the default landing page of the web application and shows a login form (line 21), navigation to the edit page of the logged in user's notes (lines 23-25), and displays notes from every user (lines 26-30) using the `note` template (defined at lines 32-37). When the logged in user (principal) has received edit access from the author of a note, a link to edit those notes is also visible (line 35, the visibility is controlled by the access control rules at lines 69-71). The `editNotes` page allows editing existing notes (lines 40-51). If the principal is the owner, this page also

```
1  entity User {
2    username  : String (id, name)
3    // id checks uniqueness, runtime uses value for url
4    // name is shown for selection in inputs
5    password  : Secret  // Secret is stored encrypted
6    notes     : {Note}  // set of Note type entities
7    allowEdit : {User}
8        (allowed=from User as u where u <> ~this)
9        // allowed filters input options
10       // safe queries as expressions
11   canEdit : {User} (inverse=allowEdit)
12   // bidirectional relation many-to-many
13   validate( notes.length <= 5, "Only 5 notes allowed")
14 }
15 entity Note {
16   author : User (inverse=notes)
17   // bidirectional relation one-to-many
18   content : WikiText  // output renders Markdown
19 }
20 page root {
21   authentication
22   // generated template based on principal declaration
23   navigate editNotes( principal ){
24     "Edit your notes"
25   } // navigation link to edit page
26   for( n: Note order by n.modified desc ){
27   // iterates over all notes in db
28   // modified timestamp is a built-in entity property
29     div{ note( n ) } // template call
30   }
31 }
32 template note( n: Note ){
33   "~n.author.username: "
34   output( n.content )
35   navigate editNotes( n.author ){ "Edit note" }
36   // access control hides inaccessible links
37 }
38 page editNotes( owner: User ){
39   div{ "Notes of: ~owner.username" }
40   form {
41     placeholder ph {  // mark region as ajax updatable
42       for( note in owner.notes ){
43         label( "Current" ){ output( note.content ) }
44         // input and output in standard library
45         label( "Update" ){ input( note.content ) }
46       } // each iteration unique template identity
47     }
48     submit action{ replace( ph ); }{ "Save updates" }
49     // commits all databind inputs in form
50     // replace triggers ajax update of placeholder ph
51   }
52   if( principal == owner ){
53     // inline access control safe in ui template
54     submit action{
55       Note{ author := owner }.save();
56       // inverse adds it to owner.notes property
57     }{ "Add note" }
58     form{
59       label( "Allow edit" ){
60         input( owner.allowEdit )[ onclick=action{} ]
61         // input shows username values with checkboxes
62         // actions can be attached to onX attributes
63       }}}}
64 principal is User with credentials username, password
65 // principal declaration, enables ac rules,
66 // sets principal, generates authentication
67 access control rules
68   rule page root { true }
69   rule page editNotes( owner: User ){
70     principal == owner || owner in principal.canEdit
71   }
72   // hides inaccessible links
73   // page init check defends against url tamper
```

Figure 2: WebDSL example

allows adding new notes or giving another user access to edit the owner's notes (lines 52-63). An interesting aspect of this edit page is that forms (lines 40 and 58) automatically perform databinding of input values (lines 45 and 60), an action that simply needs to store the inputs does not require code (line 60). Additionally, the form input id attributes are implicit, every `input(note.content)` (line 45) has its own template identity which relates a form input field to a particular `Note` entity being edited (see Section 2.5). Page content can be updated through ajax without refreshing the entire page. This is done through placeholder definitions that mark the area to be refreshed (line 41), and invoking the built-in replace function as part of an action (line 48). Access control is enabled by declaring the entity (User) that is used to represent the principal (line 64). Based on the principal declaration, a default authentication form is automatically generated (called at line 21). Access to pages is denied by default and rules express the condition for allowing access. The `root` page with the login form is accessible to anyone (line 68), while the `editNotes` page can only be accessed if you are the owner, or have gotten access from the owner (lines 69-71). This is an example of a simple discretionary access control model, that allows users to configure access control restrictions to their data [7].

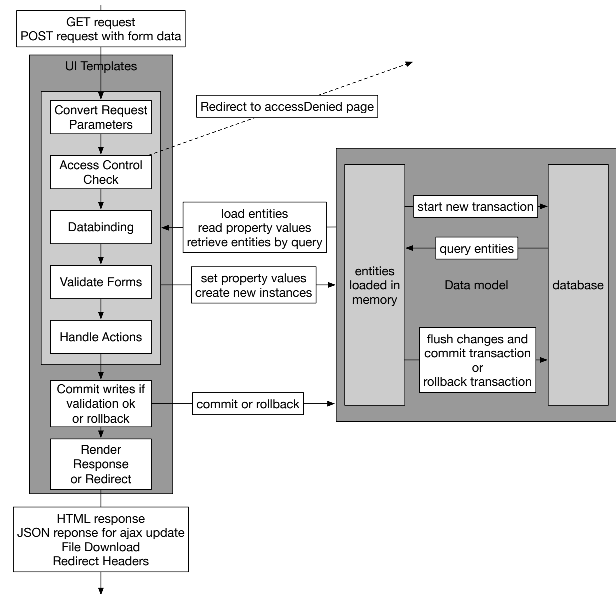

Figure 3: WebDSL request lifecycle

2.4 WebDSL Request Lifecycle

The main runtime behavior of WebDSL applications is handling browser page requests for retrieving a page (GET) and requests for posting form data (POST). The request processing lifecycle is shown in Figure 3. The dispatch handler starts by analyzing request parameters to determine the page that was accessed and load the arguments to the page. Session data is also loaded to determine whether a user is logged in. This is followed by an access control check, which can deny access to the rest of processing and redirect. In case of a GET request that only reads data, the templates are rendered, and no database transaction commit is required. In case of a form submit the templates are evaluated in multiple phases: databinding processes inputs and transforms request data to updates in the loaded entities, validation then checks for failing validation rules before deciding to execute the requested action, finally the requested action is executed. In case of validation failure, the transaction is rolled back, and the page is rendered with errors. When validation succeeds, the transaction is committed and the response is rendered or a redirect is triggered. An ajax update request is a variation of this process, where only part of a page is rendered and returned. Throughout the request phases, the persistent data model is accessed to load entity data by id or through queries. Entity updates are tracked, and flushed back to the database when committing. Database transaction semantics decide how to resolve conflicts in updating persisted data.

2.5 User Interface Template Identity

The user interface templates in WebDSL use a custom framework for dispatching and handling requests. A distinguishing feature in this implementation is a deterministic template identifier generator that uniquely identifies instances of templates. This is the basis for `form` handling, because these identifiers connect input data in a submit request back to the data model value in the input. In many MVC frameworks (e.g. Django, Ruby on rails) the developer is required to specify such id attributes consistently in both the template and action handler. This makes reuse of form templates more difficult, because multiple occurrences need explicit unique id attributes. WebDSL takes a different approach and includes action handling as part of the user interface context. Input tag identities are implicit, and data model values are automatically updated through databinding when a form is submitted. This scales easily to more complex forms with iteration and choice. WebDSL automatically inserts unique identifiers and avoids clashes in the input ids. Submit actions also follow this method and use the template identifier to match a button submit form request to the right action. Input identifiers that are not in the template declaration, or submit ids that are not available, are ignored to avoid security problems caused by form data tampering. The construction of template identity is illustrated in the table below.

Template Element	Template Id Component
`for(e in` *persistent entities*`)`	`e.id`
`for(e in` *transient entities*`)`	*iteration number*
`for(e in` *primitive values*`)`	`e`
`templatecall(args)` *static id*	*assign unique id at compile-time*

The template id is build up dynamically in a stack when processing phases of templates. Part of the construction is static, each templatecall in the AST of the application gets a fresh id assigned at compile-time. This makes multiple calls to the same template unique, which means only iteration constructs could create collisions. For each iteration construct in WebDSL, the runtime decides on an appropriate identity value to add to the stack. When requesting an id for a template, the stack content is hashed to constrain the size of the id attribute.

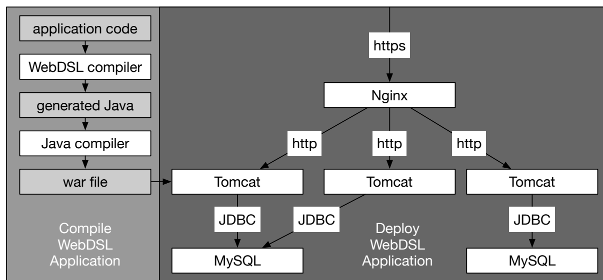

Figure 4: Deployment scenario, each box can be a separate server or the same depending on application requirements

2.6 Application Deployment

The WebDSL compiler creates a war file with a complete Java web application. A typical deployment scenario is illustrated in Figure 4. Here, the war file is copied into the webapps folder of a Tomcat application server. On initialization of the application, MySQL database table schemas are created if they do not exist, and updated if new columns are added. Nginx receives incoming requests first and decides based on the domain which application and application server are requested. Using a reverse proxy server to handle outside requests is more secure than directly exposing an application server to the web. The Apache Httpd and Nginx projects get a lot more scrutiny because these are used everywhere. Additionally, they can be set up to connect to multiple Tomcat instances, and take care of common web deployment configuration, like HTTPS encryption. A Tomcat application server can host one or more web applications, and a MySQL instance can host databases for multiple web applications.

3 ROBUSTNESS ENGINEERING

We define robustness as: applications should not crash and should not show glitches in availability.

Single Application per JVM. In an ideal scenario for deploying Java web applications, it is enough to have one application server instance hosting all the applications, and one database server instance hosting all the databases. What we experienced in practice is that there are many reasons why the JVM can crash:

- Hanging Tomcat JVM due to expensive page request
- Tomcat crashing automatically after 50 days [1]
- Crashed JVM due to bug in JNI code of a library
- Maximum open file handles reached for process

These problems can all be solved, some require changing the OS environment, or JVM parameters, or the bug has been fixed in a newer JVM/Tomcat/library. Even though they can be solved, having all applications in one application server means if one crashes the JVM, all applications are down. For robustness in our application deployment we switched to one application per Tomcat instance. With MySQL we experienced few robustness issues. However, for performance tuning it can be useful to have one application database per MySQL instance to have more control over settings.

Fluent Redeploy. When redeploying an application there is a small delay between the war file being deployed and the first request being accepted. This delay can be reduced by using tomcat war file versioning e.g. by copying a war file with version number appended, e.g. ROOT##42.war. The new war file is deployed next to the old one, and requests are directed to the new application instance as soon as it is finished deploying. A WebDSL application starts with checking the database schema for updates. In the case of adding a new property to an entity with many saved instances, this can be slow. The schema update can also be done in advance to avoid the delay in deployment. Another issue was related to template identity. The identifiers are partially based on a static id assigned to template calls at compile-time. If this id is not stable between recompilations, input and action ids can change. This means that if a user is looking at a loaded page, then a redeploy is performed on the application server, the forms on the loaded page are no longer valid. A page refresh is needed to get back to a working page. We improved this behavior by making the ids more deterministic, using the AST location as unique identifier, which resulted in fewer failed actions after a redeploy.

Transaction Retry. In the majority of requests there is no issue with concurrent edits of the same data. Since we rely on the transaction behavior of the database to handle conflicts, there are specific scenarios where a request fails because another transaction committed changes at the same time, e.g. by code in a background task. This situation was observable as a page sometimes not loading, or an action failing to complete. We added a retry mechanism to handling requests in the specific scenario of a concurrent change. By default, requests are tried upto 3 times before giving up. In most situations this is enough to let the update be processed. The request is handled as if it came after the commit that caused the conflict. Transaction semantics can not be hidden entirely from the WebDSL application developer, as we experienced in a WebLab scenario. A new feature was added to calculate an average grade for all exam assignments, updated every time a change was made by any student. This led to all student transactions being in conflict, because they were trying to update the same row storing the average grade in the database. The problem of describing derived values concisely and deriving a robust evaluation was inspiration for the IceDust language [9, 10].

Submit Failure Feedback. Another improvement was made in the handling of failed actions. When data is updated in the database, it might happen that a form is no longer available, meaning the unique ids of the inputs and actions will be ignored. If the user then submits the form, it is not recognized as a valid action. The initial implementation would cause the button not to get a response, which was confusing for users. We improved this behavior by explicitly notifying the user with a customizable message. For example, in WebLab, the submit button for an assignment becomes unavailable after the deadline passes. The student now gets a message explaining that the deadline has passed, when they press the submit button.

4 PERFORMANCE ENGINEERING

We define performance as: applications should have no noticeable delay in response times, and this should hold also when the amount of data increases (scalability).

[1] https://bz.apache.org/bugzilla/show_bug.cgi?id=56684

In-Memory Page Cache. In many applications there are more users reading data than writing data. In that case, a page cache is very beneficial for performance, e.g. in CMS-style applications such as conf.researchr.org. By building a page cache into the runtime we can automatically handle cache invalidation. After checking access control, the rendered page is retrieved from cache if available. The cache is filled automatically and keeps the most recently used pages in cache. Cache space is allocated for anonymous users, and for logged in users, which helps improve the browsing speed for a user session on the website. All caches are invalidated when an entity change happens. If it is a session entity change, only the page cache for logged in users is invalidated.

Query Prefetching. In the data model of the WebDSL runtime, we use Hibernate ORM to implement objects with persistence. In the runtime, we made the decision to have a default configuration that retrieves reference properties lazily. In many cases this is a good default. If the reference is not used, it does not have to be loaded from the database. The effect is that the queries that get generated are small and fast. However, the number of executed queries is high. When there is an iteration over a large collection, and for each iteration a query is executed, it is often faster to do one query with a join for the extra needed data (eager fetching). This is referred to as the 1+N problem in ORM terminology. We have experimented [6] with deriving automatic prefetching, which showed us that the decision for eager fetching can be partly static, by analyzing access patterns in the application code. However, there is also a dynamic component. The actual speed improvement depends on the table sizes and several database settings. To have more control, we also added prefetching syntax to the language to force eager fetching. This turned out to be very convenient in practical situations where a single page was getting too slow.

5 SECURITY ENGINEERING

We define security as: applications should prevent attacks from malicious sources, where vulnerabilities in the web technology stack are abused.

Improving CSRF Protection. Easy to guess id attributes in form inputs are vulnerable to Cross-Site Request Forgery (CSRF) attacks. A malicious website can create a link or image that is a forged request to execute an action on the targeted application. If the victim is logged in to the targeted application, the browser will perform the request using the victim's credentials. Template id generation in WebDSL depends on the data and is hashed to make them hard to guess. We further improved this protection by including the principal user entity id in all template id attributes. This can be done transparently because the compiler controls id generation, and the entity used as principal is explicitly identified for access control. A common way to do CSRF protection in frameworks like Django is to rely on adding an additional CSRF token to all the forms. The token is a random secret value associated with a user session that needs to be submitted with the request parameters to perform the action. Although it makes protection convenient, it is still something that a developer can forget to include, or cause confusion if it is used incorrectly and blocks a submit unintentionally.

Force HTTPS. A feature that is best solved before requests go to the application server at all, is forcing request to go over HTTPS. This makes sure all sensitive form data and cookies gets sent encrypted. This is simple to configure in Apache Httpd or Nginx and can be configured with HSTS headers so that browsers cache the decision to access the site over HTTPS.

Single Sign-On. The largest security issue we experienced was a bug in the A-Select single sign-on Apache module provided by the university. The module was vulnerable to a directory traversal attack, which would circumvent the filter that blocked access. The lesson learned here is to be very careful with external authentication integration, the application code might have a perfect access control model, but if you cannot trust the signin procedure it is useless.

Deployment Isolation. Since we were running the servers, we also managed a Jenkins instance for our research group. This became a problem when the jenkins user started executing suspicious commands on the server. It turned out that a vulnerability in Jenkins was abused to run arbitrary scripts. The lesson we learned was to not trust that other developers of other web applications get security right, and deploy applications with scripting components in as much virtualization and isolation as possible.

6 RELATED WORK

WebDSL has been designed to integrate web information system concerns into a single language. Comparing with existing languages and frameworks, it functions as a full-stack web programming solution like the Java Spring framework, Django Python framework, and Ruby on Rails. React and Angular are popular client-side rendering frameworks. These are not full-stack solutions, and require a server-side component to handle concerns like persistence. Integrating concerns as language features requires making design decisions up front. This means WebDSL is particularly suited for information systems. However, it might not be a good choice for other styles of web applications. For example, if you need raw performance, or a client-side rendered user interface with mostly web socket communication, then the current language features in WebDSL are not sufficient.

WebML [1] is a modeling language for generating Java web applications. Applications are created in a graphical tool using high-level page components such as login, show all items, search items, item detail. High-level components have the benefit of allowing fast prototyping and easy understanding of applications. The downside is that there is a large gap to get fine-grained control, such as having to write custom components in Java code.

Ur/Web [3] provides verification of rich program properties through type-checking. The Ur/Web code compiles to run on server and client, generating C for the server and Javascript for the client. An example of verification related to input id attributes is that it checks whether the request data ids accessed in an action handler are indeed produced in the HTML form that invokes the action.

The Links [4] web programming language provides a single language from which code for all tiers (client, server, and database) is generated. The database abstraction provides transparent optimized database queries derived from the code. Security aspects are identified as open issues in this work. Current work on Links [5] introduces session types to provide static guarantees that communication between clients and server complies with a specified protocol.

REFERENCES

[1] Marco Brambilla and Piero Fraternali. 2014. Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience. *Science of Computer Programming* 89 (2014), 71–87. https://doi.org/10.1016/j.scico.2013.03.010

[2] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. 2010. Preventing injection attacks with syntax embeddings. *Science of Computer Programming* 75, 7 (2010), 473–495. https://doi.org/10.1016/j.scico.2009.05.004

[3] Adam Chlipala. 2016. Ur/Web: a simple model for programming the web. *Commun. ACM* 59, 8 (2016), 93–100. https://doi.org/10.1145/2958736

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures (Lecture Notes in Computer Science, Vol. 4709)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

[5] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *Proceedings of the ACM on Programming Languages* 3 (2019). https://doi.org/10.1145/3290341

[6] Christoffer M. Gersen. 2013. *ORM Optimization through Automatic Prefetching in WebDSL*. Master's thesis. http://resolver.tudelft.nl/uuid:597b318c-a1af-4fde-865f-4422f548336b

[7] Danny M. Groenewegen and Eelco Visser. 2008. Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns. In *Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14-18 July 2008, Yorktown Heights, New York, USA*, Daniel Schwabe, Francisco Curbera, and Paul Dantzig (Eds.). IEEE, 175–188. https://doi.org/10.1109/ICWE.2008.15

[8] Danny M. Groenewegen and Eelco Visser. 2013. Integration of data validation and user interface concerns in a DSL for web applications. *Software and Systems Modeling* 12, 1 (2013), 35–52. https://doi.org/10.1007/s10270-010-0173-9

[9] Daco Harkes, Danny M. Groenewegen, and Eelco Visser. 2016. IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2016.11

[10] Daco Harkes, Elmer van Chastelet, and Eelco Visser. 2018. Migrating business logic to an incremental computing DSL: a case study. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, David Pearce 0005, Tanja Mayerhofer, and Friedrich Steimann (Eds.). ACM, 83–96. https://doi.org/10.1145/3276604.3276617

[11] Zef Hemel, Danny M. Groenewegen, Lennart C. L. Kats, and Eelco Visser. 2011. Static consistency checking of web applications with WebDSL. *Journal of Symbolic Computation* 46, 2 (2011), 150–182. https://doi.org/10.1016/j.jsc.2010.08.006

[12] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. 2010. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling* 9, 3 (2010), 375–402. https://doi.org/10.1007/s10270-009-0136-1

[13] Eelco Visser. 2007. WebDSL: A Case Study in Domain-Specific Language Engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007 (Lecture Notes in Computer Science, Vol. 5235)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, Braga, Portugal, 291–373. https://doi.org/10.1007/978-3-540-88643-3_7