Tiago Delfim Parada Gonçalves Queijo Lopes

Bachelor of Computer Science and Engineering

Language-Based Data Sharing in Web Applications

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Ricardo Viegas da Costa Seco,
Assistant Professor, NOVA University of Lisbon

Examination Committee

Chairperson: Sérgio Duarte, NOVA University of Lisbon
Raporteur: Francisco Martins, University of Lisbon
Member: João Costa Seco, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

August, 2017

Language-Based Data Sharing in Web Applications

This document was created using the (pdf)LaTeX processor, based in the "unlthesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

I would like to dedicate my dissertation work to my family, friends, and professors that helped me throughout the last five years. A special gratitude to my parents Leopoldo and Lurdes Lopes for their unfailing support and continuous encouragement, and to my best friend João Augusto for his friendship over the years.

Acknowledgements

First, I would like to thank my adviser, Prof. João Costa Seco, for his guidance and support during the elaboration of this thesis. I would also like to acknowledge Nuno Martins, Bernardo Albergaria, and Guilherme Rito for their contributions to this thesis.

I would like to thank my colleagues, friends, and professors that helped me throughout the course, and a special thanks to my best friend João Augusto for his continuous support and friendship.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Abstract

Cloud development and virtualization of applications is crucially becoming the common practice in the software engineering industry. Many systems and database tools are available to support applications with many instances and views, but all the orchestration of data and functionality in the so-called multi-tenant applications comes with a high development and maintenance cost. Due to the high costs of developing and maintaining such applications, there is an increasing need for languages and tools that support the gradual development of software for a highly shared environment, at the developer and user level.

We extend a typed, reactive and incremental programming environment and language with parameterized modules that increase application modularity, with lenses that provide a (filtered) data sharing mechanism between modules, and the (dynamic) verification of module access conditions to implement data privacy. The combination of these mechanisms is a safe and powerful mechanism to design and evolve cloud and web applications.

We present a pragmatic programming language supported by a deployed prototype where several examples of applications illustrate this new programming paradigm. We also provide a larger web application example as a means of showing how the combination of the introduced mechanisms allows for the development of multi-tenant applications, and to compare it against implementations in modern frameworks.

Keywords: Multi-tenant Applications, Data Isolation, Data Privacy, Data Sharing, Programming Lenses

Resumo

Desenvolvimento na *cloud* e virtualização de aplicações está, crucialmente, a tornar-se a prática comum na indústria de engenharia de software. Existem muitos sistemas e ferramentas de bases de dados para suportar aplicações com muitas instâncias e vistas, mas toda a orquestração de funcionalidade e dados nas chamadas aplicações multi-tenant tem um preço alto no desenvolvimento das aplicações e nos custos de manutenção, existindo assim uma crescente necessidade de linguagens e ferramentas que suportem o desenvolvimento gradual de software para um ambiente altamente partilhado ao nível do programador e do utilizador.

Estendemos uma linguagem tipificada, reativa, incremental e respetivo ambiente de programação com módulos parametrizados que aumentam a modularidade das aplicações, com *lenses* que fornecem um mecanismo (filtrado) de partilha de dados entre módulos, e com verificação dinâmica de condições de acesso a módulos para implementar privacidade de dados. A combinação dos mecanismos enunciados é um mecanismo forte e seguro de desenhar e evoluir aplicações para a *cloud* e *web*.

Apresentamos uma linguagem de programação pragmática, suportada por um protótipo *deployed* com diversos exemplos de pequenas aplicações para ilustrar este novo paradigma de programação. Proporcionamos também uma aplicação *web* de maior dimensão com o intuito de mostrar como a combinação dos mecanismos enunciados permitem desenvolver aplicações *multi-tenant* e comparar com implementações em *frameworks* modernas.

Palavras-chave: Aplicações Multi-Tenant, Isolamento de dados, Privacidade de dados, Partilha de dados, Programação com lentes

Contents

Acronyms

API Application Programming Interface.

CSS Cascading Style Sheets.

DDP Distributed Data Protocol.

GHC Glasgow Haskell Compiler.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

JS JavaScript.

MTA Multi-Tenant Application.

QQ QuasiQuotes.

QRcode Quick Response Code.

RAM Random Access Memory.

REPL Read–Eval–Print Loop.

REST Representational State Transfer.

SQL Structured Query Language.

SSL Secure Sockets Layer.

TCP Transmission Control Protocol.

TH Template Haskell.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

USID Unique Session Identifier.

Chapter 1

Introduction

Cloud development and virtualization of applications is becoming a crucial common practice in the software engineering industry [Vel+10; You+11]. Software-as-a-Service applications have become increasingly popular within the cloud [SR11; Tsa+14]. The development of these customizable Mutli-Tenant Applications (MTAs), increases the reuse of code, but demands developer skills and attention to ensure data separation and privacy between users and tenants [Bez+10]. There are many tools and frameworks that work at the system and database levels to help support many instances and views of an application, as well as share and replicate databases [MK11; Rod+12]. The use of virtualization infrastructures like the Amazon AWS containers [Ama] is widely accepted, and heavily based on smart system management and virtual configuration tools. However, application instances are many times separate machines with explicit sharing and coordination code. The amount of complex hand-written code that keeps data and functionality separated comes with a high development, testing, validation and maintenance cost. This long development process is prone to errors, and as the application scales so does the chance for errors.

It is important to use sophisticated language based approaches in order to validate data separation between users and tenants. But, more importantly, there is an increasing need for languages and tools that support the gradual development of software for a highly shared environment, both at the developer and user level. Additionally, there should be language based mechanisms for defining (and validating) single user applications and then transform them, by adding only user management code, to a context of a multi-user application.

In this thesis we build on top of a typed, reactive and gradual programming language and the corresponding programming environment. The core language allows the definition of a set of data variables and active expressions associated to public names. It allows

for a type safe redefinition of a data variable or method, as well as the redefinition of the type of an expression. This is obtained by keeping all data dependencies between names and preventing changes to propagate erroneously through the dependency graph.

We extend the language with a module abstraction to isolate data and functionality and module mechanisms that map to core language constructs. We introduce module mechanisms such as: parameterization to index the isolated data and functionality; nesting to create module hierarchies with role based development in mind; guard condition mechanism to dynamically verify module access conditions and implement data privacy; and inheritance to give modules full access to inherited modules. Data sharing is supported by lens [Boh+06; Hof+15; Ste15] based filters and sliced data sharing mechanisms between modules. With a language-based approach, the incremental and reactive properties of the core language are still present in the introduced abstractions. The combination of the mechanisms that are presented in this thesis is a safe and powerful paradigm to design and evolve cloud and web applications.

We present a pragmatic programming language supported by a deployed prototype where several examples of applications illustrate this new programming paradigm. We will show how authentication is easy to achieve, allowing the developer to build authentication mechanisms from scratch and use them throughout the rest of the development to validate data access dynamically. With this new paradigm we will build a large To-do list application for groups of users to share tasks, with each user possibly having an administrator role in a group. Each user will also be able to filter the tasks to customize his own view. Traits like role based development, user tailored views, and group shared information are multi-tenant traits which we want to address with the combination of the introduced mechanisms. With this application we will show how the introduced language abstractions speed up development, and simplify code. We will also compare with modern frameworks with smaller examples to benchmark our solution.

1.1 Reactive and Incremental Language

We start with a reactive and evolvable framework [Mat15] for web and cloud applications. It provides operations for dealing with data and how it can be displayed, as well as how the application behaves. Due to the reactive nature of the language [DS15], every change made to the state of the application is propagated through a graph of dependencies, giving the developer immediate feedback and keeping the state synchronized. Each change made to an application is statically verified to guarantee that the application evolves safely. These changes are automatically integrated with the running application without disrupting the availability of the application. The combination of the reactive and incremental aspects allow the developer to build applications in a live environment, which greatly increases productivity in the development process.

There are three core operations in the language: **var**, **def**, and **do**. The purpose of the **var** operation is to declare names that store the application's persistent state. The **def**

operation declares named pure data transformations. Finally, the **do** operation performs actions that change the state, and it is through this operation that the user interacts with an application. Actions are introduced with the keyword **action**.

The reactive nature begins with the **def** operation, which builds a graph of dependencies to propagate changes made to the state, keeping it always up-to-date. Pure data transformations (**def**) are only modified through the propagation of changes, and variables (**var**) through actions.

```
1  var x = 2
2  def square = x * x
3  def setX y = action { x := y }
4  def calc =
5    <div>
6      <p>(x ++ "^2 = " ++ square)</p>
7      <input type="number" id="newValue" value=x />
8      <button doaction=(setX #newValue)>
9        "Calculate square"
10     </button>
11   </div>
```

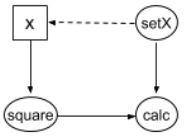

Figure 1.1: Square calculator example and the associated data dependencies graph

To help further understand how to use the language, fig. 1.1 illustrates how a simple square number calculator can be implemented. This example defines four names: `x`, `setX`, `square`, and `calc`. `x` is a state variable initialized with the number `2`, and it will store the number that we wish to square. `square` is a pure data transformation storing the result of the square of `x`. `setX` creates an action for modifying the state stored by `x`. Since `square` depends on `x`, changing `x` updates the value of `square`, thus reflecting the desired calculation of the square of a given number. `calc` defines an Hypertext Markup Language (HTML) document in which we display the current value stored in `x`, and the current value of the square of `x` stored in `square`. To provide a new number we create an input in the document with a specific `id`, `newValue`, which we can then use to execute the `setX` action with a button. The action is given the argument `#newValue` which refers to the input with the corresponding *id* attribute. Because the document is also a pure data transformation, it will be updated every time that the state changes in one of the dependencies.

1.1.1 Runtime Support System

The framework is supported by a runtime system [Mat15] with three main components:

- **Interpreter** – parses, verifies, evaluates, and executes code;
- **Database** – stores application data;
- **Web Server** – provides a Representational State Transfer (REST) Application Programming Interface (API) for the interpreter, and pushes updates via WebSockets [Webb]. The three most important routes of the REST API are described in

Table 1.1: REST API

Description	HTTP Request
Get the value of a name	GET /:**name**/:**args**

Execute code	POST /
	data: {
	exp: **code**
	}

Do actions	PUT /
	data: {
	action: **action**,
	args: {
	arg: {
	type: **type**,
	value: **value**
	}
	},
	env: {
	name: **value**
	}
	}

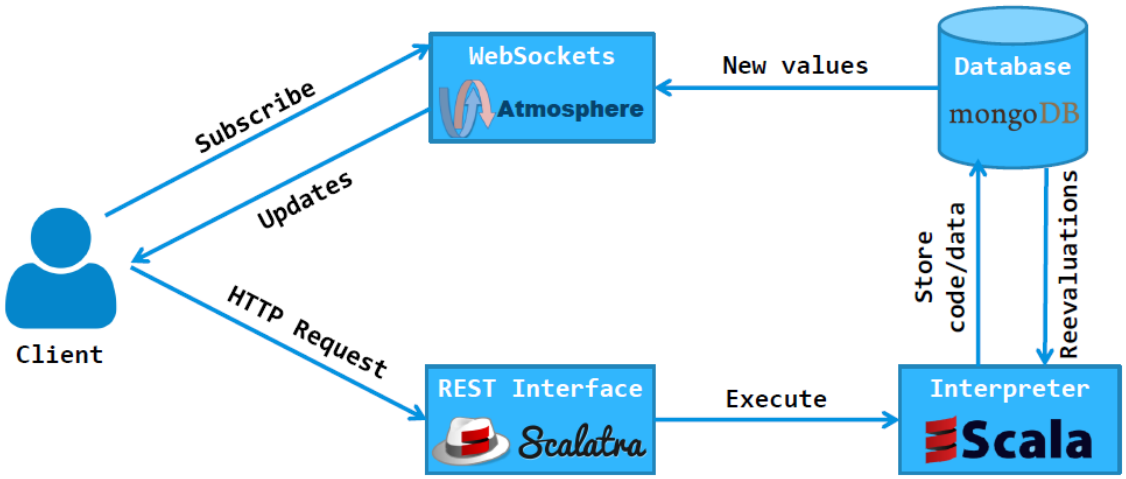

Figure 1.2: Runtime Support System architecture [Mat15]

table 1.1.

Figure 1.2 illustrates the interactions between each component. A typical cycle of interactions with between a client and the system starts with the client subscribing to a workspace through WebSockets. When a client subscribes, the existing workspace data is pushed to the client. The client then performs requests to the server REST interface to execute code in the interpreter, storing any new or updated data in the database. All changes

in the database are pushed back to all subscribed clients through the WebSockets.

1.2 Approach

To tackle the problems that were previously discussed, this thesis seeks to introduce language abstractions that help design and evolve cloud and web applications that require data to be shared in, and between, different isolated groups of users.

The development of authentication mechanisms require client identification. The most common approach to client identification in modern frameworks is a server generated unique identifier stored on the client side. We adopt the same approach and introduce a server generated Unique Session Identifier (USID) to enable the runtime system to distinguish clients. In the context of the programming language, this identifier is exposed through the reserved, string typed, keyword **usid**. The introduced USID allows the developer to build authentication

Many languages like C [Cmo], and OCaml [Oca] employ a modular approach as a means of isolating code to manage the complexity of programs, or to hide information. Keeping in mind the modular approach of these languages, we introduce our **module** abstraction as isolated environments where data is shared with all the users. To control user access to modules we introduce guard conditions that can be defined with modules.

Modules are isolated environments, thus the sate outside of a module is not accessible from inside the module. Most languages provide an import operation for their modular approach, typically granting access to an exposed interface of *getters* and *setters* to manage encapsulated data. Thus, we introduce an **import** operation with which the developer can create views in a module over states defined in an outside environment. Import operations are supported by **lenses** [Boh+06; Hof+15; Ste15], which provide bidirectional transformations over a concrete state: a *get* transformation that given a concrete state produces a view, and a *put* transformation updates a concrete state with an updated view. We introduce different kinds of lenses to cover a wide range of filters for an imported module state.

MTAs need to provide a different state of the same application to each tenant, so that a tenant can manipulate its own state and not another tenant's state. With this in mind, we expand the introduced module abstraction with parameterization, allowing modules to be defined with a set of parameters. The parameters act as an index for each defined module state, thus transforming the module into an indexed module. Defined parameters can be explicitly set as non-indexing parameters. An indexed module provides views of each module state, with each view being accessed through the module indexing parameters (index). To provide different views of the same state we apply the lens concept over a map structure where each entry is a concrete state, and the lens *moves* to an entry with a given index to apply the lens transformations. The composition of module parameterization and module access conditions allow for the definition of strong access control by defining conditions that depend on indexing parameters.

So far, the introduced abstractions provide tools to create authentication mechanisms and applications that can quickly be transformed to provide an array of different application views and states. However, role based development is not yet covered, which is a very important trait of the MTAs. User roles follow a strict hierarchy of access, and are commonly implemented in modern frameworks by explicitly defining roles and creating rules associated with each role. In our approach, we allow modules to be defined inside other modules as nested modules. Nested modules create a strict hierarchy of access by inheriting the ancestors conditions and isolating the nested module state from them. Typically, after the definition of user roles in modern frameworks, the framework provides a set of operations that helps determine at runtime if a user has a certain role or not. To this end, we introduce a similar runtime conditional access check operation, that closely resembles an *If-Then-Else* statement, called *In-Then-Else*. This operation checks if a given user has access to a given module at runtime and depending on the result, either the branch *then* or the branch *else* is evaluated. Combining nested modules, access conditions, *In-Then-Else*, and import operations provides a powerful tool in the development of role based applications, without explicit long hard-written queries for role definition and access control.

Often, in a modular approach there is a need to access an already defined module to avoid writing the same code multiple times. Commonly languages with a modular approach provide some kind of inheritance or extension mechanism to deal with such needs. In our approach, we introduce module inheritance to also tackle this problem. Considering that our modular approach introduces module parameterization, to inherit a module we have to consider that the module can be a parameterized module. With that in mind, to inherit a module the heir must provide arguments to the inherited module parameters. When a module inherits another module, the heir gains full access to the inherited module state. To guarantee that access to the inherited module state is checked when accessing an inherited state, the module access condition is also inherited.

Our approach is language based, with most of the introduced abstractions mapping directly to core language constructs, thus keeping the incremental and reactive properties of the core language. Additionally, our approach needs only a few modifications to the type system keeping the guarantee of a sound development process, correct code, and that no errors occur when the application is deployed.

The combination of the introduced abstractions gives the developer the ability to develop complex reactive cloud and web applications in a live environment, where each modification made to the application can be used right away. This improves the quality of the development process with faster feedback and faster development, which would otherwise be slower and harder in current web frameworks. To help the development process, we provide a web-based live programming environment to give immediate and continuous feedback to the developer and to simplify development with the module abstraction by providing module navigation and inspection tools.

1.3 Contributions

This thesis has three main contributions:

- A set of language abstractions to simplify and speed up the safe development of complex cloud and web applications that require shared and isolated data between groups of users:
  - Unique session identifiers to distinguish application users and enable authentication;
  - A module abstraction to express isolated environments of shared data;
  - Module parameterization to allow data to be given in function of a set of parameters;
  - Module guard conditions to restrict user access to data in an environment;
  - Module inheritance allowing modules to fully access other modules available in their environment;
  - Module nesting to express hierarchies with modules;
  - A set of programming lenses to support an import mechanism that allows data to be shared across isolated environments with reactive nature;
- A runtime support system capable of handling the new language abstractions that provides a REST API and WebSockets that push updated data to clients;
- A web-based live programming environment that gives immediate and continuous feedback, with tools to help development with the introduced language abstractions;

1.4 Structure of the Document

In this section we present a description of what each chapter in this document discusses.

- chapter 2 presents the detailed solution with the help of a growing example web application;
- chapter 3 discusses the challenges offered by the implementation of the proposed model in the extended language;
- chapter 4 studies existing solutions of current frameworks to understand how the development of complex applications is achieved in those web frameworks;
- chapter 5 provides a benchmark of our implemented solution against an existing framework, comparing performance, code succinctness, and effort in developing MTAs;

- chapter 6 presents some final remarks about the proposed model and future work that we believe could improve the work done in the context of this thesis.

Chapter 2

Language-Based Model

In this chapter we present our programming language, extending [Mat15], and illustrated in chapter 1. The fundamental language mechanisms introduced in our development are session identity, parametric and extensible module introduction, a variety of data sharing mechanisms, and access conditions to modules. We show how it is possible to use and combine such mechanisms to create basic authentication features, and to build different kinds of security layers, including shared, filtered, and isolated environments in the context of a multi-user web application. In the following sections, we introduce the syntax, semantics and some implementation details of the language using a running example. Our example extends the *de-facto* standard benchmark for web technologies, the TodoMVC project [Tod]. We add user authentication to the plain to-do list application, and provide an implementation for sharing and managing task lists among groups of users.

2.1 Base Language

In this section, we briefly overview the new language mechanisms. In the subsequent sections, we give a more detailed explanation of each one of the introduced mechanisms and how their composition can be used to provide more sophisticated patterns.

Session identity The ability to identify users is one of the basic requirements in most web frameworks, and it is also the foundation on which authentication mechanisms are built upon. We explicitly introduce this basic mechanism in the language through a unique identifier for each connecting user (USID). The USID is denoted by the reserved identifier `usid`.

```
module Public {
  var welcome = "Welcome"
}

module User<string name, number age> with Public {
  var msg = welcome@Public ++ " " ++ name
  var myData = 0

  module Adult when(age >= 18) {
    import myData as data
    def inc = action { data := data + 1 }
  }
}
```

Listing 2.1: Module mechanisms

Module introduction Modules define hierarchical and isolated environments, allowing for the explicit sharing of data between modules. Modules are defined through the `module` definition constructs. For instance, listing 2.1 shows how modules can be defined. In the example, we define the `Adult` module nested in the `User` module to create a hierarchy.

Data Sharing Expressions in a module are defined with relation to an environment, which includes all definitions of that particular module. Hence, by default, the names in a module's outer context are inaccessible. In order to use names defined outside a module, we can create a surrogate name, via an `import` operation. In listing 2.1, the `Adult` module imports the `myData` name and uses the local alias `data` to define the action named `inc`. An `import` operation defines a bidirectional view based on the well established concept of lenses [Ste15]. The created view is bound to the `data` name, and is set over the state variable `myData` outside the `Adult` module. Any changes on either end, are propagated to the opposite end. For instance, the defined action `inc` increments the value of the `data` name, thus any increment on the local alias `data` is propagated back to the original `myData` name.

Module parameterization and access control We introduce module parameters as a means of indexing a module state with relation to a set of parameters. In listing 2.1, we define the `User` module with a set of parameters: `name`, and `age`. Thus, for each combination of the parameters, there is a different associated `msg`, and `myData` state. Accessing a module name is done through the REST API described in sections 1.1.1 and 3.2. For example, accessing the `welcome` name in the example is done with a *GET* request to */Public/msg*. To restrict access to a module we also introduce guard conditions to the `module` definition constructs. In the example we define such a condition for the `Adult` module, where access to the module is only granted for values of `age` greater than `18`.

Module inheritance Module inheritance is introduced to allow modules to fully access another module. Using names from the inherited module is done through module identifiers, which are names appended to the module name with a `@` symbol. The example in listing 2.1 defines an inheritance of the module `Public` in the module `User`, allowing the expression of `msg` to use the name `welcome` from the `Public` module.

2.2 Sessions and Authentication

Identification refers to the act of being able to state a person or thing's identity, *authentication* is the process of confirming that identity [CC12]. Nowadays most web applications need to identify devices in order personalize user experience, which they do by creating sessions between the server and the client. To establish a session, the devices need to first have an identity. To accomplish this, we introduced the Unique Session ID [Usi]. The USID is a server generated string token given to each unidentified device. Each device is then responsible for storing the USID and sending it attached in future communications, thus establishing a session. An Internet browser, for example, uses Hypertext Transfer Protocol (HTTP) Cookies [Kri01] to store the given USID. In the context of the programming language, we expose the USID through the reserved keyword `usid`.

The introduction of the USID enables the server to identify users, thus allowing the definition of authentication mechanisms in the language.

We want to build a simple username/password authentication mechanism where each user has a password associated to him. The authentication process will require the user to provide a name and a password. If the given login information matches with the stored information the user is authenticated with the given name (assuming the username/password combination is only known to the user). To build an authentication mechanism like this, we need a collection of users and a store to register authenticated users. The listing 2.2 shows the table `users` where the username/password combinations are stored, and the table `authenticatedUsers` to track authenticated users by associating the token of a user to a name. We populate the user storage with the data found in appendix A.1.

In listing 2.3 we define our authentication function with two parameters, name and password, which constitutes a login function. It first performs a selection action over

```
1 table users {
2   name: string,
3   password: string
4 }
5 table authenticatedUsers {
6   name: string,
7   token: string
8 }
```

Listing 2.2: Data stores for an authentication process

```
def authenticate name password =
  match
    get user in users
    where user.name == name and user.password == password
  with
    user::rest =>
      action {
        insert {
          name: name,
          token: usid
        } into authenticatedUsers
      }
    | [] => action {}
```

Listing 2.3: Authentication process

```
def logout =
  action {
    delete user in authenticatedUsers
    where user.token == usid
  }

def authenticated id =
  match
    get user in authenticatedUsers
    where user.token == id
  with
    user::rest => true
    | [] => false

def userFromId id =
  match
    get user in authenticatedUsers
    where user.token == id
  with
    user::rest => user.name
  | [] => ""
```

Listing 2.4: Authentication page helpers

the `users` collection to retrieve a user with the given name/password combination, and matches the result with either a non-empty collection, or an empty one. A non-empty collection means that a user was found, and so an action associating the user to a token is returned. If the matched collection is empty then no user was found and an empty action is returned. The returned action can later be attached to a login button to execute it.

With the authentication mechanism created, we now want to create a reactive page that shows a login form if the user is not authenticated, otherwise it shows the user name and allows him to log out. To define this page we require some helper functions as seen in the listing 2.4 for additional functionality. Respectively, the first provides an action to create a logout button, the second is a function that checks if a given user is authenticated and the last one retrieves the name associated with an authenticated user.

With the helper functions, and actions, in place we can create the page shown in

```
def page =
  <div>
   (if not authenticated usid then
     <div>
       <div>
         <input type="text" placeholder="username" id="name"/>
         <input type="password" placeholder="password" id="password"/>
       </div>
       <button doaction=(authenticate #name #password)>"Log In"</button>
     </div>
   else
      <div>
        <h1>("Welcome " ++ userFromId usid)</h1>
        <button doaction=(logout)>"Logout"</button>
      </div>)
  </div>
```

Listing 2.5: Authentication page

Figure 2.1: User authentication page

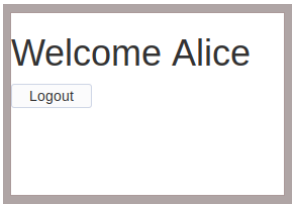

Figure 2.2: Authenticated user page

listing 2.5. Figures 2.1 and 2.2 show a flow of user authentication in the created application [1], where a user is authenticated with a password and the page reacts to the successful operation by showing a welcome message with the user name.

Even thou the authentication process is successful, the application can only sustain one authenticated user due to the shared global state in which the application is defined and the nature of non-delayed expressions. This example is important to show how non-delayed expressions, like the `page`, with a stored global value that is only re-evaluated when a dependency is changed behave in a shared environment. In the example, the `page` name depends on the `authenticatedUsers` collection through the `authenticated` function which causes the page dependency on the state of the `authenticatedUsers` store. The `page` evaluates either to a login page (fig. 2.1) or a simple user page (fig. 2.1) depending on whether the user is authenticated or not when the expression is evaluated. If a user authenticates with the correct credentials the user name is associated with the given user USID in the `authenticatedUsers` store, which in turn causes the `page` name to be re-evaluated for the user that requested the `authenticate` action. When, for example, Alice authenticates with her credentials, the resulting updated state of the `page` name will show the view in fig. 2.2 and any other subsequent request for the `page` name will return the same stored value of fig. 2.2 for all users. At this point, if the

[1] The full example can be found at http://live-programming.herokuapp.com/dev/pP4ta, with the application main page at http://live-programming.herokuapp.com/app/pP4ta/page.

```
def page token =
  <div>
   (if not authenticated token then
     <div>
       <div>
         <input type="text" placeholder="username" id="name"/>
         <input type="password" placeholder="password" id="password"/>
       </div>
       <button doaction=(authenticate #name #password)>"Log In"</button>
     </div>
   else
      <div>
          <h1>("Welcome " ++ userFromId token)</h1>
        <button doaction=(logout)>"Logout"</button>
      </div>)
  </div>
```

Listing 2.6: Authentication page through a function

`logout` action is requested by Alice, then the page will be re-evaluated back to show fig. 2.1 and the `authenticatedUsers` store will be empty again. However, if instead of Alice, a non authenticated user requests the `logout` action, it will not remove Alice from the `authenticatedUsers` store due to the requesting USID not matching Alice's USID. Then, because the `page` still depends on the `authenticatedUsers` store, the `page` is re-evaluated with the non authenticated user USID that requested the `logout` action, thus resulting in the login page in fig. 2.1 while Alice remains authenticated according to the `authenticatedUsers` store. Alternatively, when the `page` state is fig. 2.2, suppose the non authenticated user could authenticate directly through the framework console. This action would produce the same reaction, updating the `authenticatedUsers` store by associating the new authenticated user with his USID and in turn updating the dependent `page` name, but in this case the resulting stored view would be similar to fig. 2.2 but with the new authenticated user name while Alice is still authenticated.

With the behavior demonstrated in the previous example in mind, and since functions are delayed expressions that store no value after being called, it is possible to define a function that takes a USID argument and returns an HTML page. The parameter `token` will then replace all USID occurrences in listing 2.5 as shown in listing 2.6. With the page wrapped in a function with the USID as a parameter, when Alice requests the page with her USID as the argument the page will be evaluated with her USID. If she successfully authenticates, she will have a unique HTML page value refreshed only with the given USID to reflect changes on the `authenticatedUsers` store. As a result of this approach, only Alice will see the page in fig. 2.2 as expected, other non authenticated users can call the `page` function with their own USID to get a unique `page` value with the view in fig. 2.1 to be able to authenticate. This behavior duality between delayed expressions and non-delayed expressions in a shared environment is a recurring, and important, issue throughout this chapter.

This approach [2] can then be applied to the rest of the application as we build it, that is, every state being in function of the USID and/or other data. However, having users provide the USID manually to execute a function that returns data is not safe and user friendly since the user needs to find the Cookie storing the USID, and provide it explicitly. In order to provide a better way of isolating a state of users and groups of users while keeping the reactive properties of the language, illustrated in this example, we introduce the explicit declaration of modules and associated mechanisms in the next sections.

2.3 Modules

All the declarations and data stored in each workspace in the Live Programming framework are accessible to any user. To build web applications where data is shared but also isolated within groups of users with different access conditions, we first need to express isolated environments to contain data. We introduce modules as named environments where the data declared in the module is shared, and all the names outside of a module are inaccessible inside it. `module Public { var x = 1 }` defines a module where state variables and pure data transformations can be declared. After a module is defined, we can add or redefine names of a module using a module block operation. The module block starts with the symbol `@` followed by the module name. For example, `@Public { var x = 2 }` allows names to be added or redefined for the `Public` module. Module definitions map directly to `var` or `def` operations in the global environment. Each name defined in a module is transformed by appending the module name with the symbol `@`. For instance, the name `x` in the previous example, would be internally transformed to `x@Public` and mapped to a `var` operation in the global environment.

Sensitive data such as the one used in our authentication example, should not be accessible to every user of the application. Since modules isolate data, it would be safer to keep the data stores, functions, and actions in the root environment of the work space, while the application page stays isolated inside a module. But isolation means nothing outside can be accessed. In order to access outside data, and possibly filter it, we introduced a familiar mechanism of importing names which is described in detail in the following sections.

2.3.1 Lenses and Imports

The `import` operation is used to create views over given named states that are defined in a parent environment, allowing a module to access its data or filter it. By default the views are named after the imported name.

Going back to our running authentication example, we can now achieve what was established in section 2.3, keeping the data stores and functions outside a module while

[2] Working example of this approach can be found at http://live-programming.herokuapp.com/dev/Ny7si.

```
1  // Data stores of listing 2.2 with seeds from appendix A.1
2  // Authentication function of listing 2.3
3  // Helper functions and actions of listing 2.4
4
5  module Public {
6    import authenticate
7    import authenticated
8    import userFromId
9    import logout
10
11   def page =
12     <div>
13      (if not authenticated usid then
14        <div>
15          <div>
16            <input type="text" placeholder="username" id="name"/>
17            <input type="password" placeholder="password" id="password"/>
18          </div>
19          <button doaction=(authenticate #name #password)>"Log In"</button>
20        </div>
21      else
22         <div>
23             <h1>("Welcome " ++ userFromId usid)</h1>
24           <button doaction=(logout)>"Logout"</button>
25         </div>)
26     </div>
27 }
```

Listing 2.7: Importing outside names

the application page is defined inside a module. Assuming the already defined data stores, functions, and actions, the listing 2.7 shows the intended result by importing the four defined functions and actions that manipulate the data stores without exposing the sensitive information to the module state.

We use an internal mechanism of lenses [Boh+06; Hof+15; Ste15] to support the import operations. Imports map directly to `def` operations with the corresponding lens value. A name defined with a `def` operation has reactive properties, which means that the stored import lens retains the reactive properties. Lenses are views over a state that allow bidirectional transformations between a set of inputs (concrete states), and a set of outputs (abstract states). A lens is comprised of two main operations as shown in fig. 2.3:

- `get`: a forward transformation, from the concrete state to the abstract state;
- `put`: a backwards transformation that takes an old concrete state and updates it with an updated abstract state;

It is also useful to have an operation that creates a concrete state from a given abstract state without an original concrete state. This operation is called `create`. The `create` operation can be achieved with the `put` operation by providing a default concrete state instead of an old concrete state, and update it with a given abstract state.

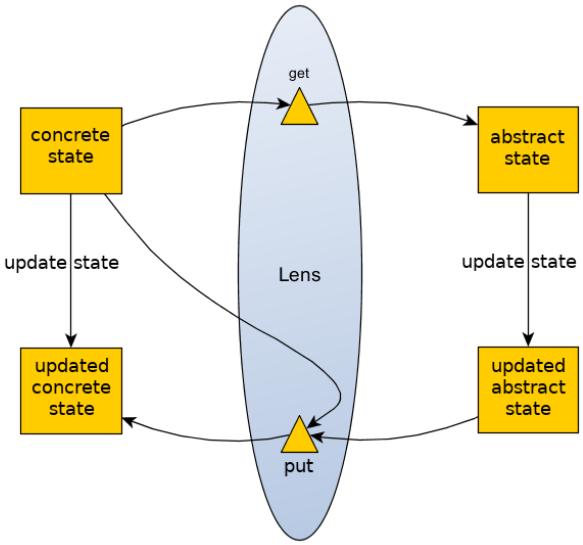

Figure 2.3: Lens *get* and *put* transformations [Dle]

We introduced a number of different kinds of lenses that support the most common situations of importing and filtering data from a module to another, as well as use the composition of lenses to build more complex data filters.

In the following sections, we will describe in detail the kind of lenses and lens compositions operations we adopted and implemented. For each kind of lens we show a small example of the supported import operations.

2.3.1.1 Simple Lens

We introduce simple lenses to create direct abstract states for given named concrete states. The original name is stored in the lens, allowing each operation to execute over it. The available transformations of a simple lens are as follows:

- `get`: retrieves the current value of the original name;
- `put`: changes the current value in the original name to a new given value.

The simple lens supports the most basic `import` operation to view and modify a state directly. Listing 2.8 shows an example of importing a name and defining an action to change its state. In the example, the collection `data` is imported into the module `Public`, and given a new name, `pubData`. If a new name is not given with the keyword `as`, the name of the import defaults to the original name. We then define an action to insert a new value into the imported name. The `addData` action concatenates the current value in the `data` name (retrieved with the `get` operation) with the given `n` value. The result is then stored in the `data` name with the `put` operation of the lens. The type system only accepts lenses over names defined with `var`/`def` operations and imported names in the closest outside environment of the module.

```
var data = [1,2,3,4,5]

module Public {
  import data as pubData

  var addData n = action { insert n into pubData }
}
```

Listing 2.8: Simple name import

```
var data = [1,2,3,4,5]

module Public {
  from n in data where n > 3
    import n as pubData

  var addData n = action { insert n into pubData }
}
```

Listing 2.9: Filtered import of a collection

2.3.1.2 Filtered Lens

The filtered lens is introduced to create a filtered abstract state of a given concrete state. In this particular case, both the concrete state and the resulting abstract state are collections. The abstract state is a collection consisting of all the elements in the concrete state that satisfy the given predicate. The available transformations of a filtered lens are as follows:

- `get`: iterates over the concrete state with an accumulator collection which starts empty, and each element that satisfies the given predicate is added to the accumulator, which is then returned as the abstract state;
- `put`: changes the current value in the original name to a new given value.

Listing 2.9 shows an example of a filtering `import`. First, we define `from n in data` to bind the name `n` to each element when iterating the `data` collection. The predicate, `where n > 3`, matches elements in the `data` collection greater than 3. `import n` declares that we want to import the each element as is.

In listing 2.9, the `get` operation returns the collection `[4,5]`. The `addData` action produces the result the same way as the action in listing 2.8, which is also stored in the name `data` through the `put` operation. Delete and update actions combine their filter conditions with the `import` operation predicate in order to only delete elements pertinent to the abstract state.

2.3.1.3 Filtered First Lens

The *filtered first* lens is introduced as an extension of the filtered lens in section 2.3.1.2, except the resulting abstract state is an element of the original collection instead of a

```
var data = [1,2,3,4,5]

module Public {
  from n in data where n > 3
    import first n as pubData
    default 0

  var changeData i = action { pubData := i }
}
```

Listing 2.10: Import first element of a filtered collection

collection of elements. The abstract state is the first element of the original collection that satisfies the given predicate. Because of the possibility of no matching element, the lens requires a given default value of the same type as the original collection elements. The available transformations of a filtered first lens are as follows:

- `get`: the same selection as the filtered lens is executed, then if the resulting collection has elements, the first element is returned. Otherwise, if there is an already stored value in the lens, the stored value is returned, alternatively the defined default value is returned;

- `put`: if the abstract collection is not empty, all the elements that satisfy the predicate are updated with the new given value. Otherwise, the new value is stored in the lens.

Listing 2.10 shows how we can import the first element matched in a filtered collection. As the example shows, the `import` operation follows the almost the same structure as the filtered lens in listing 2.9. With the exception being that, instead of importing all elements matched, we import only the first match with the keyword `first`.

In listing 2.10, the `get` operation of the lens would return the value `4` in the example. The action `changeData`, when executed, calls the `put` transformation described above with the given result value of `i`. For example, if we execute the action with `changeData 6`, the `put` operation would update all elements greater than `3` to `6`. This behavior is due to the fact that collections are iterated in lens transformations, that is, elements are not accessed by position in the language, and so we adopted this solution in order to not produce erroneous behavior. The `data` collection would be equal to `[1,2,3,6,6]`, and the `get` transformation would return the value `6`. If, however, the filter was `n > 5`, the `get` operation would return the default value `0` due to no matching elements. The given default value does not satisfy the predicate because the condition might contain names which can only be evaluated at run-time (like the USID), thus the lens cannot know if the default value satisfies it. The adopted solution is safe, but doesn't keep the user from defining inconsistent imports with this lens. In this case, the action `changeData 6` would store the value `6` in the lens instead, meaning that subsequent `get` calls return the stored

```
var data = { name: "App", settings:[] }
var users = [{name: "Alice", points: 18}, {name: "Bob", points: 33}]

module Public {
  // Simple lens composition
  import data.name as app

  var change n = action { app := n }
}

module Game {
  // Filtered lens composition
  from user in users where user.points > 20
    import user.name as topPlayers
    default {name: "Dummy", points: 21}

  var addPlayer name = action { insert name into topPlayers }
}
```

Listing 2.11: Record field filtering with composed lenses

value `6`. The value `6` is consistent with the predicate, but if we executed the action with the argument `2`, the stored value would be `2` even thou it is inconsistent with the condition.

2.3.1.4 Record Field Lens

In this section we introduce a type of lens to compose with simple and filtered lenses. This lens makes clear that an import declaration is defined by a lens or composition of lenses. Depending on the composition, the concrete state can be a record if composed with a simple lens, or collection of records if composed with a filtered lens. The available transformations of a record lens are as follows:

- `get`: first the composed lens `get` operation provides a result, and then, the record lens extracts the given field. The extraction process depends on the result of the composed lens. If the result is a record, the extracted field is returned as the abstract state. If the result is a collection of records, the returned abstract state is a collection of extracted fields.

- `put`: if the concrete state given by the composed lens is an object, then the field is replaced with the new given value and the `put` operation of the composed lens is fed the new object result. Otherwise, if the composed lens produces a collection of records, then the field of each record is replaced with the given value, and the resulting updated collection is put back with the `put` operation of the composed lens.

Listing 2.11 shows an example of how these compositions can be defined with the `import` operation. We defined two compositions with the record lens: a simple lens, and a filtered lens.

Simple lens composition The record lens composition with a simple lens is defined just as a simple lens `import` in listing 2.8, but instead of importing the object as is we import the given `name` field of the object, as seen in line 6 of listing 2.11. The record lens concrete state is given by the simple lens abstract state of the record `data` name. In the example, the `get` operation returns the `"App"` value. The action `change` calls the `put` transformation of the record lens with the given `n` value, which creates the updated object that is then given to the `put` operation of the simple lens.

Filtered lens composition The record lens composition with a filtered lens is defined just as a filtered lens `import` in listing 2.9, but instead of importing each matching element as is, we import the given `name` field, as seen in line 13 of listing 2.11. Additionally, the default value `{name: "Dummy", points: 21}` is defined with the `import` operation. The default value is needed when inserting new elements into the lens abstract state in order to populate the rest of the fields in the new object. The record lens concrete state is given by the filtered lens abstract state over the `users` collection, and each object field of the filtered collection is extracted to build the abstract state. In the example, the imported `topPlayers` name returns the collection `["Bob"]`. The action `addPlayer` creates a new object with the field `name` assigned the value of the `name` argument, and populates the field `points` with the provided default value `21`. The new object is then passed onto the `put` transformation of the filtered lens, which inserts the object in the original collection.

2.3.1.5 Indexed Lens

Indexed lenses support the state in module parameterization, a mechanism that is explained in section 2.3.2. Indexed lenses are set over a map structure where each entry is a concrete state, and the lens *moves* to an entry with a given index to apply the lens transformations. We introduce indexed lenses to index concrete states into as many abstract states as the indexing key allows. Unlike the previously introduced lenses, indexed lenses store the produced abstract states as concrete states for each given index, because producing an undetermined number of abstract states for every `get` operation is potentially costly. Each indexed lens is defined with a set of parameters which are used to produce an index key for an abstract state. An index key is created by concatenating each given argument into a `string` to access a unique abstract state, just as a map structure as seen in fig. 2.4. A concrete state stored in an indexed lens is given by a default expression when defining the indexed lens. The available transformations of indexed lenses are as follows:

- `get`: given a set of arguments, the lens combines them into a key, and the associated abstract state is returned. If the key lookup doesn't find an abstract state, the `create` operation is used to store a new abstract state and return it;

- `put`: given an abstract state and a set of arguments, the lens combines the arguments into a key and updates the abstract state in the associated entry;

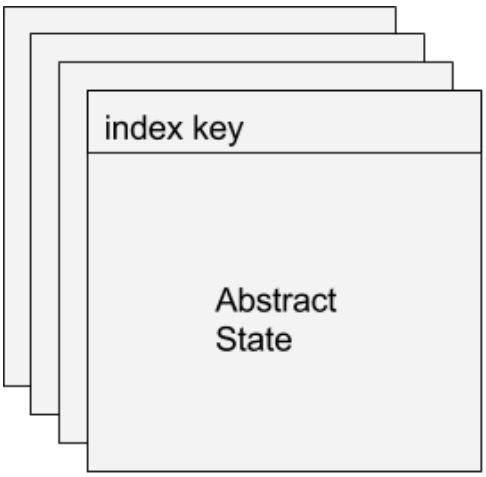

Figure 2.4: Indexed Lens structure

- `create`: the default expression, stored in the lens as the concrete state, is evaluated for a given set of arguments and stored with the composed key in the associated entry.

2.3.2 Module Parameterization

So far, modules isolate data in a shared environment giving a single state equal to all users. In this section we introduce a parameterization mechanism to index modules. In an indexed module, each state is indexed with a set of parameters defined with the module. To index a state, we use the previously introduced indexed lenses in section 2.3.1.5. Each name defined in a module, is mapped directly to a `var`, or `def`, operation as described in section 2.3, but in an indexed module the value stored for each name is instead an indexed lens. The concrete state (default expression) of an indexed lens is given by the declared expression of a `var`, or `def`, commands and the lens index is given by the indexing parameters of the module. With this, each declared expression can have different states for different combinations of arguments. Listing 2.12 shows how module parameters can be declared. Every parameter requires a type annotation, and only basic types (string, number and boolean) are valid. A parameter can also be defined as a non-indexing parameter with the `*` character prepended to the parameter name. Non-indexing parameters are not used to index a module state, thus they are also not used in indexing lenses of the module. For example, in the module `Room` in listing 2.12, two users can access the same room state while providing different names through the parameter `name`. Due to their

```
1 module Room<number room, string *name> {
2   var message = "Welcome to room " ++ str room
3 }
```

Listing 2.12: Module defined with parameters

non-indexing nature, the scope of non-indexing parameters is limited to mechanisms that will be introduced later in sections 2.3.3 and 2.3.4. In contrast, the scope of indexing parameters is the whole module.

In listing 2.12, we have a state variable `message` that depends on the indexing parameter `room`. The expression of the `message` is stored as the default value of an indexed lens. When an index is accessed, the expression is evaluated with the provided arguments from the request to the module state. The value is stored in the given index (formed with the `room` argument), resulting in every room having a different `message` state.

When a dependency of an indexed lens changes, the reactive nature of the language updates all stored values by re-evaluating the lens default expression for each index. Because the index is constructed from the module arguments, each index is deconstructed to populate the module parameters in the evaluation environment.

The USID, being itself an indexation of connecting users, is a prime candidate for a parameter in modules to index module states for each user. We provide a language syntactic sugar when defining a USID parameter to omit the type annotation, for example, `module User<string name, usid> { }`. In the example we define the `module User` with the USID as a parameter, which is internally transformed into a `string usid` parameter. This creates an isolated environment for each user in the `module User` state. For safety, when accessing a USID indexed module state, the server automatically populates the module arguments with the user's USID.

Recalling our running example application last expanded upon in listing 2.7, a problem was keeping the application from growing. The application wouldn't allow for multiple users to authenticate at the same time due to the fact that there was only one state for the `page`. Indexing the `Public module` with the USID parameter provides each user with an authentication page state with the USID as its index. This produces the desired authentication application as seen in listing 2.6 where the page function was made in function of a given token. This new approach keeps the USID safe, and the page value does not need to be re-evaluated for each request by the same user due to the nature of the indexed lens. Listing 2.13 redefines the `module Public`, created in listing 2.7, using the USID as a parameter. The parameterization of the module, allows each user to see the `page` evaluated with his own USID even when another user is authenticated. As listing 2.13 shows, we use the lenses introduced in section 2.3.1.4 to replace the previous `userFromId` imported function. Instead of using a function, now we import the authenticated user name as `currentUser`. The new `currentUser import` filters the `authenticatedUsers` collection with the user's USID, and takes the `name` field of the first record matched. Since the second branch of the `page` is only evaluated for authenticated users, we are guaranteed to never show the `default` value of the `currentUser`.

With an authentication mechanism implemented [3] in our language we can start the

[3] A working example can be found in the workspace at http://live-programming.herokuapp.com/dev/LsT5t with the authentication page at http://live-programming.herokuapp.com/app/LsT5t/Public/page

```
// Data stores
// Authentication function
// Helper functions and actions

module Public<usid> {
  import authenticate
  import authenticated
  import logout
  from user in authenticatedUsers where user.token == usid
    import first user.name as currentUser
    default {name:"",token:""}

  def page =
    <div>
     (if not authenticated usid then
       <div>
         <div>
           <input type="text" placeholder="username" id="name"/>
           <input type="password" placeholder="password" id="password"/>
         </div>
         <button doaction=(authenticate #name #password)>"Log In"</button>
       </div>
     else
        <div>
          <h1>("Welcome " ++ currentUser)</h1>
          <button doaction=(logout)>"Logout"</button>
        </div>)
    </div>
}
```

Listing 2.13: Importing outside names

next step in our To-do list example application mentioned at the start of chapter 2.

2.3.3 Access conditions

By default, a module state is accessible by any user. However, in a web application, it is almost always required to have controlled access to a given state. For example, a user profile is only accessible to an authenticated user, or a group profile to an authenticated user that has access to the group. To tackle this need, we introduce a mechanism for modules to test a given access condition and determine whether a user can access a module state or not. We map this module condition directly to a wrapper delayed expression through a function which takes a *dummy* parameter. This allows us to create a delayed expression which can be evaluated for each request without storing state of the result, just by using core language constructs.

Our growing To-do application requires controlled access to each group, thus we will expand upon it as we further explain the access conditions mechanism. In section 2.3.2 we finished a page where the stored users from appendix A.1 can authenticate themselves. Next, we will expand the application with a user page where the groups he can access are listed using the group seed data from appendix A.2.

To set a module condition we introduced the `when` operator to modules, which takes an expression that must have a `boolean` type. By default, a module defined without a condition is given one with the `true` literal. The condition expression environment is composed of the module's parent environment and module parameters. Each request to a module state must first evaluate its access condition and test if the result is true, or false. As explained in section 2.3.2, accessing a module state requires the provision of arguments for the module parameters, which populate the environment in which the condition is tested. This guarantees that the evaluation of the condition has all the module parameters in the environment.

Listing 2.14 shows the definition of an indexed module with an access condition, using the previously declared `authenticated` function in section 2.2. The defined `module User`

```
module User<usid> when(authenticated usid) {
  import logout
  from user in authenticatedUsers
    where user.token == usid
    import first user.name as username
    default {name:"",token:""}

  def page =
    <div>
      <h1>("User: " ++ username)</h1>
      <button
        doaction=(logout)
        data-redirect=(workspace_path ++ "Public/page")
      >"Log out"</button>
    </div>
}
```

Listing 2.14: User module definition

```
1  @Public {
2    def page =
3      <div>
4        (if not authenticated usid then
5          <div>
6            <div>
7              <input type="text" placeholder="username" id="name"/>
8              <input type="password" placeholder="password" id="password"/>
9            </div>
10           <button doaction=(authenticate #name #password)>"Log In"</button>
11         </div>
12       else
13         <div>
14           <h1>("Welcome " ++ currentUser)</h1>
15           <a href=(workspace_path ++ "User/page")>"Go to your page!"</a>
16           <button doaction=(logout)>"Logout"</button>
17         </div>)
18     </div>
19 }
```

Listing 2.15: Public module redefinition

is indexed with the USID parameter in order to create an isolated environment for each user, similar to listing 2.14 in section 2.3.2. Each access to the `module User` will test if the `authenticated` function for a given USID, evaluates to `true`. With this module, each authenticated user has a different state. In the new `User` module we define the `username import`, similar to the way we do in listing 2.13, to view the name associated with an authenticated user. Finally we define an HTML page that displays the `username`, and allows the user to log out and be redirected back to the log in page.

In listing 2.15, we redefine the `page` in the `Public` module, previously defined in listing 2.13, and add a link to the user's page (line 15).

With these modifications, we have a boilerplate for user authentication [4] on which we will build the rest of our running To-do list application.

2.3.4 Inheritance

In this section, we introduce an inheritance mechanism to the language as a means of giving a module access to another module. The next step in the development of our running To-do example application is to add a group environment for authenticated users to share information, which we will build with the help of module inheritance.

First, we define the state variable `groups` in the global environment, populated with the data found in appendix A.2. With the new information stored, we want each user to have a list of the groups he can access. Listing 2.16 redefines the `User` module of our running example application, with the help of a function that determines if a `string` is contained in a list of records that have a field `name`. The function `listContains` is

[4] A working example can be found in the work space at http://live-programming.herokuapp.com/dev/H36KP, with the authentication page at http://live-programming.herokuapp.com/app/H36KP/Public/page

```
def listContains list name =
  match
    get item in list
    where item.name == name
  with
    u::us => true
  | [] => false

@User {
  import listContains
  from group in groups
    where (listContains group.users username)
    import group as groups
}
```

Listing 2.16: Group listing in the user page

```
module Group<string groupName, *usid> with User(usid)
  when(listContains groups@User groupName) {}
```

Listing 2.17: Group module definition

imported to help create a view that filters the newly defined `groups` collection, and returns the list of all the groups a user can access.

Now consider the module definition `Group` in listing 2.17 which inherits the module `User`. As seen in the example, a module inherits another module through the keyword **`with`**, followed by the inherited module name. If the inherited module has a non empty set of parameters, the host module (module `Group`) must provide arguments to the inherited module parameters when the inheritance is defined. We provide the `Group` module USID parameter as the argument expression for the inherited module `User` USID parameter. This means that, when a user accesses the `Group` module, the `User` module state that is accessible to the user depends on the given USID parameter. Additionally, because the `User` module has an access condition, any access to the `Group` module must first test the inherited module condition. Accessing an inherited name inside the host module is done through the appendage of the inherited module name with the `@` symbol to the inherited name. We also define an access condition for the `Group` module using the inherited view name `groups@Users`, and test if a given `groupName` parameter is contained in it..

Next, we want each group to keep track of a to-do list, as well as a page to execute actions over each task. The listing 2.18 defines a simplified, non-styled, example [5] of the TodoMVC application built on top of our running example application. In the example we define three actions to manipulate the `todos` store:

- `AddTodo` adds a new to-do to the list with a given text
- `deleteTodo` deletes a to-do, identified with a given **`id`**

[5] A working example can be found in the work space at http://live-programming.herokuapp.com/dev/gCkg7 with the authentication page at http://live-programming.herokuapp.com/app/gCkg7/Public/page

```
@Group {
  var todos = [{
    id: 0,
    text: "Welcome to group " ++ groupName,
    done: false
  }]

  def size = foreach(todo in todos with y = 0) y+1

  def addTodo text = action {
      insert { id: size, done: false, text: text }
      into todos
    }

  def deleteTodo id = action {
      delete todo in todos
      where todo.id == id
    }

  def toggleComplete id = action {
      update todo in todos
      with { id: todo.id, done: not todo.done, text: todo.text }
      where todo.id == id
    }

  def todoItem todo =
    <li>
      <checkbox type="checkbox" value=(todo.done)
        docheck=(toggleComplete todo.id)
        douncheck=(toggleComplete todo.id)
      />
      <label>(todo.text)</label>
      <button doaction=(deleteTodo todo.id)>"Delete"</button>
    </li>

  def page =
    <div>
      <header>
        <h1>groupName</h1>
        <input placeholder="What needs to be done?" onenter=(addTodo) />
      </header>
      <section>
        <ul>(map (todo in todos) todoItem todo)</ul>
      </section>
    </div>
}
```

Listing 2.18: Simplified, non-styled, TodoMVC application for groups of users

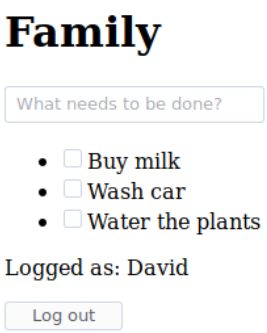

Figure 2.5: Alice's view

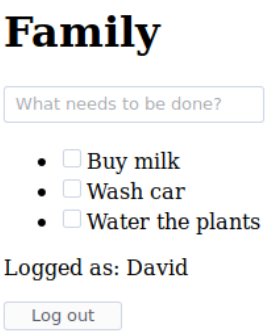

Figure 2.6: David's view

Figure 2.7: Group page with footer duality issue

- `toggleComplete` toggle a to-do completed state

To keep track of how many tasks are in a to-do list, we define `size` that counts the elements in the `todos` collection. We also defined the `todoItem` function, that, given a to-do item returns an HTML value displaying the to-do text, a checkbox to change the completed state of the task, and a button to delete the item. The defined `page` allows the user to input a to-do by pressing enter through the attribute `onenter`. This attribute takes a function with a `string` parameter, and when the *enter* is pressed, the input value is used as the argument of the function (`addTodo`). Finally, in the `page`, we iterate over the `todos` storage, and for each to-do we call the `todoItem` function in order to display each to-do with an HTML value.

Notice how the example does not make use of any of the inherited names except in the definition of the module access condition (because it is a delayed expression), and recall the problematic behavior of non-delayed expressions in a shared environment previously described in section 2.2, page 13. With the inheritance mechanism introduced this behavior is now experienced when using inherited names on non-delayed expressions defined in the host module. To demonstrate this behavior, consider the addition of a footer in the group `page` in which we display the authenticated user name associated with the given USID. Figure 2.7 shows the example with the added footer view with two authenticated users, Alice and David, where David was the first to authenticate and access the group page. The group `page` is a non-delayed expression with a stored value for each group, and using a name such as the `username@User`, which depends on the USID of the requesting authenticated user, causes the group `page` to evaluate with the corresponding authenticated user name. This is due to the fact that the `Group` name is only indexed by the `groupName` parameter, while the inherited `User` module is indexed by the USID. Furthermore, changing the `Group` module USID parameter to an indexing parameter, means that each user has different to-do list, even thou the added footer would correctly display the value of `username` for each user. Our goal however, is to have a to-do list for each group while allowing the free usage of inherited names, and so the next

section introduces a new mechanism that allows us to deal with the recurring non-delayed expressions behavior in shared environments.

2.3.5 Module Nesting

In this section, we introduce module nesting. Nested modules create a hierarchy, thus, access to an inner module must first test the access conditions of all outer modules. This means that, a nested module requires at least the same set of parameters as the outer modules in order to evaluate any outer conditions that might use parameters. Thus, by default, a nested module automatically inherits the set of parameters of the closest outer module, which we will call *nesting* parameters. However, the *nesting* parameters can be overridden for the nested module, meaning we can change a parameter from indexing to non-indexing and vice-versa without changing the actual outer module indexing parameters. New parameters can also be added to the nested module.

Consider the module `Group` redefinition in listing 2.19, which follows the development of our running To-do application. We want to limit the `deleteTodo` action to administrators of each group using module nesting. First, we defined a filtered view of the `groups` store to extract the members list of a given group containing all the information about each member of the group. Then, we define the `isAdmin` function to test if a given member is an administrator in the group. To create the administrator role, we define a nested module in the `Group` module, and override the *nesting* parameter `groupName` to a

```
@Group {
  from group in groups
    where group.name == groupName
    import first group.users as members
    default {name:groupName, users:[]}

  def isAdmin name =
    match
      get member in members
      where member.name == name
    with
      m::ms => m.admin
    | [] => false

  module Admin<string *groupName> when(isAdmin username@User) {
    import todos

    def deleteTodo id = action {
      delete todo in todos
      where todo.id == id
    }
  }
}
```

Listing 2.19: Module nesting

```
@Group {
  def todoItem todo =
    <li>
      <checkbox type="checkbox" value=(todo.done)
        docheck=(toggleComplete todo.id)
        douncheck=(toggleComplete todo.id)
      />
      <label>(todo.text)</label>
      (in Admin(groupName, usid) then
        <button doaction=(deleteTodo todo.id)>"Delete"</button>
       else <span></span>)
    </li>
}
```

Listing 2.20: Checked module access with In-Then-Else

non-indexing parameter. We keep the USID *nesting* parameter as a non-indexed parameter since we don't have any state that needs to be indexed in the `Admin` module. The `Admin` module is also defined with an access condition, with the help of the previously defined function `isAdmin`, to only allow access to administrators of the group. Finally, we import the `todos` store to redefine the `delete` action from the `Group` module inside the `Admin` module. The composition of mechanisms illustrated in the example creates a a non-indexed, isolated, and conditioned environment (the module `Admin`), inside an already existing isolated, and indexed environment (the module `Group`).

Next, we would like to display the administrator actions in the group page defined in module `Group`, but only when the user is an administrator. To allow this kind of checked access to a module defined in another module's environment, we introduce the `In-Then-Else` operation. This new operation is similar to the `If-Then-Else` statement, except instead of providing a condition to test, we provide a module name and arguments for each module parameter in order to test the given module access condition. Listing 2.20 redefines the `todoItem` function, previously defined in listing 2.18. We use a `In-Then-Else` statement with the `Admin` module as the target, and provide an argument for each one of the `Admin` module parameters. In the first branch, where the `Admin` module condition test was positive, we display the delete button for the task. The second branch simply shows an empty `span` HTML element.

The composition in the example application[6] still does not allow the inherited user state in any non-delayed expression of the `Group` module to have the desired result just as is previously described in section 2.3.4, page 29, and in section 2.2, page 13. Additionally, because the defined `page` in the module `Group` is only indexed by the `groupName` parameter each group only has a `page` value. Thus, for each given `groupName`, if an administrator is the first authenticated user and subsequently the first to request the `page` name, the administrator actions will be available to all users due to his USID being used in the evaluation of the non-delayed expressions to be stored. However, with module

[6]Working example in the work space at `http://live-programming.herokuapp.com/dev/5lYB5`, with the authentication page at `http://live-programming.herokuapp.com/app/5lYB5/Public/page`

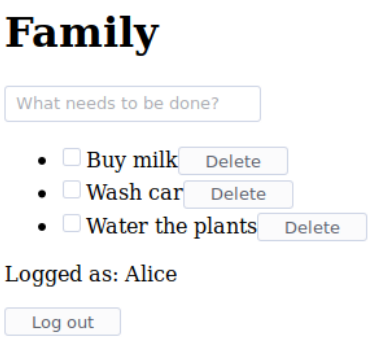

Figure 2.8: Alice's view

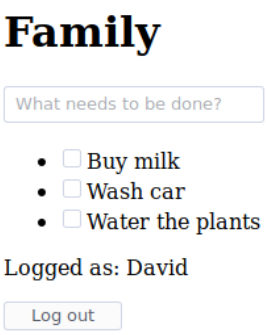

Figure 2.9: David's view

Figure 2.10: Group page with different indexed states

nesting we can create a different composition in order to express the previous example without having the undesired behavior of non-delayed expressions. Listing 2.21 shows this compositions. We define the module `Member` nested inside the `Group` module, and nest the previously defined `Admin` module inside the module `Member`. We override the `Member` *nesting* USID parameter so that both the `groupName` and USID parameters index the `Member` module. Since the `Admin` module now inherits the `Member` module parameters, we override both the `groupName` and the USID parameters as non-indexing parameters. Then, we redefine all the previously defined module `Group` names inside the `Member` module, with the exception of the `todos` store. The `todos` collection is instead imported from the `Group` module.

The composition created in listing 2.21, provides the state of `page` as an indexed state by both the `groupName` and USID parameters of the `Member` module. The `todos` is imported to the `Member` module, which keeps the original collection indexed only by the `groupName` parameter. It is also now possible to use the inherited names from the `User` module without the duality issue affecting non-delayed expressions, because the `Member` module is indexed with at least the same parameters as the `User` module (the USID parameter). For instance, in the redefined `page` name, we can add a footer to the HTML page where we show the `username` value for each user, and a logout button. Additionally, the definition of the inner module `Admin`, and use of the `In-Then-Else` statement, grants extra actions to some users.

Figure 2.10 illustrates two users, Alice and David, authenticated in different devices and viewing the to-do list for the family group. Since Alice is an administrator of the family to-do list group, the delete button is displayed as fig. 2.8 shows. In fig. 2.9 we see David's view, and because he is not an administrator the delete button is not displayed. Figure 2.10 also shows that both views have their respective authenticated user names at the bottom.

To finish our running application [7], we redefine the user page in the `User` module to

[7] Working example in the work space at `http://live-programming.herokuapp.com/dev/BluQ1`, with

```
@Group {
  module Member<usid> with User(usid) {
    import todos
    import isAdmin

    module Admin<string *groupName, *usid> when(isAdmin username@User) {
      import todos

      def deleteTodo id = action {
        delete todo in todos
        where todo.id == id
      }
    }

    def size = foreach(todo in todos with y = 0) y+1

    def addTodo text = action {
        insert { id: size, done: false, text: text }
        into todos
      }

    def toggleComplete id = action {
        update todo in todos
        with { id: todo.id, done: not todo.done, text: todo.text }
        where todo.id == id
      }

    def todoItem todo =
      <li>
        <checkbox type="checkbox" value=(todo.done)
          docheck=(toggleComplete todo.id)
          douncheck=(toggleComplete todo.id)
        />
        <label>(todo.text)</label>
        (in Admin(groupName, usid) then
          <button doaction=(deleteTodo todo.id)>"Delete"</button>
         else <span></span>
        )
      </li>

    def page =
      <div>
        <header>
          <h1>groupName</h1>
          <input placeholder="What needs to be done?" onenter=(addTodo) />
        </header>
        <section>
          <ul>(map (todo in todos) todoItem todo)</ul>
        </section>
        <footer>
          <p>("Logged as: " ++ username@User)</p>
          <button doaction=(logout@User)
                  data-redirect=(workspace_path ++ "Public/page")>
            "Log out"
          </button>
        </footer>
      </div>
  }
}
```

Listing 2.21: Member module

```
@User {
  def page =
    <div>
      <h1>("User: " ++ username)</h1>
      <button doaction=(logout) data-redirect=(workspace_path ++ "Public/page")>
        "Log out"
      </button>
      <h2>"Groups"</h2>
      <ul>
        (map (group in groups)
          <li>
            <a href=(workspace_path ++ "Group/Member/page/" ++ group.name)>
              (group.name)
            </a>
          </li>
        )
      </ul>
    </div>
}
```

Listing 2.22: User page updated links

update the groups list with new links because the `page` name in `Group` module as been moved to the inner module `Member`. listing 2.22 redefines the user page.

2.4 Syntax

In this section we provide the language grammar and explain the direct mapping of the introduced mechanisms to the core language constructs. Most of the grammar remains unchanged from the language we extend (see section 2.2 of language thesis [Mat15]). Figures 2.11 and 2.12 shows, respectively, the top-level operations, and expressions of the language.

The module abstraction given by the **`module`** operation maps to a module typed value that stores all the necessary information about a module: name, set of parameters, access condition, inherited module, and inherited module arguments. As described in section 2.3, each definition is directly mapped to a **`var`**, or **`def`**, operation with a transformed internal name. If the module is indexed by a set of parameters, the value stored for each definition is an indexed lens defined with the module set of parameters.

`Import` operations are mapped into **`def`** operations with the same internal transformed name as simple **`var`**/**`def`** definition in the module, and with the respective lens value. Because lens store the original name, verifying actions over lenses correctly by following the reference in a lens to find out if the original name is a **`var`**, or a **`def`** which is not allowed in actions by the core language.

The `@a` module block operation allows for the (re)definition of names in an existing module. This operation maps directly to the previously, described **`module`** operations,

the authentication page at http://live-programming.herokuapp.com/app/BluQ1/Public/page

o	::=	*r*	Construction Operations
	|	`do` *e*	Interaction Operation
r	::=	`var` a = *e*	State Variable
	|	`def` a = *e*	Pure Data Transformation
	|	`atomic` {*r**}	Composition of Operations
	|	`module` a<*param**> `when`(*e*) `with` a(*e**) {*i** *r**}	Module
	|	`@`a {*i** *r**}	Module Block Redefinition
	|	`table` a {(x: *t*)+}	Database Table
	|	`delete` a	Remove Name
	|	`deleteall`	Remove All Names
i	::=	`import` a `as` a	Import Declaration
	|	`from` a `in` a `where` *e* `import first?` *e* `as` a (`default` *e*)?	Filtered Import
p	::=	`usid` | *t* `*?` a	Module Parameters
t	::=	`number` | `string` | `boolean`	Base Types

Figure 2.11: Operations syntax.

which maps to `var`, or `def`, operations, keeping in the extended language the incremental property of the core language.

The only added expressions to the syntax are: `In-Then-Else` operation, *Access Module Identifier*, and `workspace_path`. The `In-Then-Else` is explained in section 2.3.5, and provides a dynamic runtime module condition check to determine which branch to evaluate. The *Access Module Identifier* is how the heir of an inherited module accesses the names defined in the inherited module, as seen in listing 2.17. Finally, the `workspace_path` is an expression that returns the workspace Uniform Resource Locator (URL) when evaluated.

Besides the added module type, the type system remains unchanged, meaning that with the direct mapping of the introduced mechanisms to core language constructs we allow the extended language to keep the reactive and incremental nature of the core language.

e	::=	a	Names
	|	b	Base Values
	|	x	Variables
	|	#a	Input Names
	|	a@a	Access Module Identifier
	|	`usid`	Unique Session Identifier
	|	`action` {*assign**}	Action
	|	<tag *attr**>*e**</tag>	HTML Element
	|	*e op e*	Binary Operations
	|	`not` *e*	Negation
	|	`let` x = *e* `in` *e*	Scope
	|	*e* `?` *e* `:` *e*	Ternary Operator
	|	`if` *e* `then` *e* `else` *e*	If-Then-Else
	|	`in` a `then` *e* `else` *e*	Module If-Then-Else Access
	|	(x+) => *e*	Function
	|	*e e*	Function Call
	|	[*e**]	Homogeneous Collection
	|	`iter` *e e e*	Iterate Collection
	|	`foreach`(x `in` a `with` y = *e*) *e*	Iterate with Accumulator
	|	`map`(x `in` a) *e*	Map Collection
	|	`get` x `in` a `where` *e*	Query Collection
	|	`match` *e* `with` x::xs => *e* | [] => *e*	Match Query
	|	{(x‘: *e*)+}	Record Literal
	|	*e*.x	Access Record Field
	|	`linkto` a *e*+	Build URL
	|	`str` *e*	Convert to String
	|	`workspace_path`	Work space base URL
attr	::=	attrName = *e*	HTML Attribute
assign	::=	a := *e*	Simple Assign
	|	`insert` *e* `into` a	Insert Element
	|	`update` x `in` a `with` *e* `where` *e*	Update Elements
	|	`delete` x `in` a `where` *e*	Delete Elements
op	::=	+ | - | * | / | %	Arithmetic Operators
	|	== | != | > | < | >= | <=	Comparison Operators
	|	`and` | `or`	Logic Operators
	|	::	Append Element
	|	++	String/Array Concatenation

Figure 2.12: Expressions syntax.

Chapter 3

Implementation Challenges

In this chapter we list and explain some of the challenges found when implementing the introduced mechanisms on top of the existing framework [Mat15]. The biggest challenges of the work done in the context of this thesis were found in the implementation of the language, architecture, and Integrated Development Environment (IDE). The following sections describe the challenges of each component and how we tackled those challenges.

3.1 Language

The biggest challenge in introducing a modular approach to the language was designing module mechanisms that when composed allow for the definition of applications with multi-tenant traits. Applications such as the one we developed with the TodoMVC [Tod] project adapted to groups of users in chapter 2. Additionally, the introduced mechanisms had to support, and keep, the reactive and incremental nature of the core language [DS15]. As language-based abstractions, the mechanisms were designed to map directly to core language operations in order to keep the described reactive and incrmemental properties.

The duality of delayed vs non-delayed expressions discussed in section 2.3.4, page 29, and in section 2.2, page 13, was the biggest design challenge. This duality in a global environment, where all users are seen as one, is explicit and easy to understand when writing code. However, when user identification is introduced, non-delayed expressions with user data produce the same stored value for all users. Examples like listing 2.5 expose the challenges of introducing user identification in a global environment, where a login page has a global state instead of a state for each user.

Designing the module mechanisms with the duality issue in mind was the hardest challenge in extending the language. We designed each mechanism as base constructs, each capable of expressing different aspects of an application. But the composition of

these mechanisms is what allows users to fully express a multi-tenant applications, without any duality issues, as shown in listing 2.21.

The introduced lens mechanisms also offered challenges when designing and composing different lenses to provide more complex filters. As an internal mechanism that supports different `import` operations, it was important that lenses had a simple and common interface (`get`, `put` operations described in section 2.3.1) to allow for the future addition of more `import` operations without making changes to the interpreter core code. This offered a challenge in the transformation of actions into correct assignments, because the core code of the interpreter should only deal with a common interface without knowing what type of lens it is. This challenge was made harder when we introduced lens compositions for the same reasons. Each composition deals with a common interface and one lens can be composed with multiple lenses.

3.2 Architecture

Although the system architecture remained the same, the introduction of modules to the language required some modifications to how the REST interface handles requests in different paths, and how the WebSockets handle updates to page names inside modules.

We decided that the REST API described previously in table 1.1 should keep the same routes. However, in order to access a module name, we had to modify how the routes handle other paths besides the root path (/). So, for example, to access the value of a name `x` in the global environment the request is *GET /x*. If the name `x` is inside a module named `Public`, the path to this module is */Public*, therefore the request to access the value of `x` inside the module is *GET /Public/x*. Nested modules translate directly into a path, for example, a module named `Member` inside a module named `Group` translates into the path */Group/Member*. When dealing with indexed modules, requesting a name is similar to requesting a function with arguments. For example, consider a module named `User` indexed by a `string` typed parameter named `name` with a name `f` defined inside. To access the value of `f` with the index `"Alice"` the request is *GET /User/f/Alice*. If `f` is a function, for example `var f i = i + 1`, the arguments of the function come after the arguments of the module, because first we access the value (a function value), and only then we call the function. So to call `f` with the argument `1`, the request is *GET /User/f/Alice/1*.

When a user requests a name with an HTML value, a page subscription is created for the client in order to keep track of where to push updates for a given HTML page. In indexed modules, each requested name with an HTML value is associated with a set of arguments for the module parameters. This means that, when such a name is updated we need to re-evaluate the page with the set of module arguments with which the client requested. Even thou the previous implementation of page subscriptions handled the list of arguments for functions that return HTML values, adding the module arguments to the same list is not enough, because there is no information in the page subscription about the module, for example: how many parameters, and what is the type of each parameter.

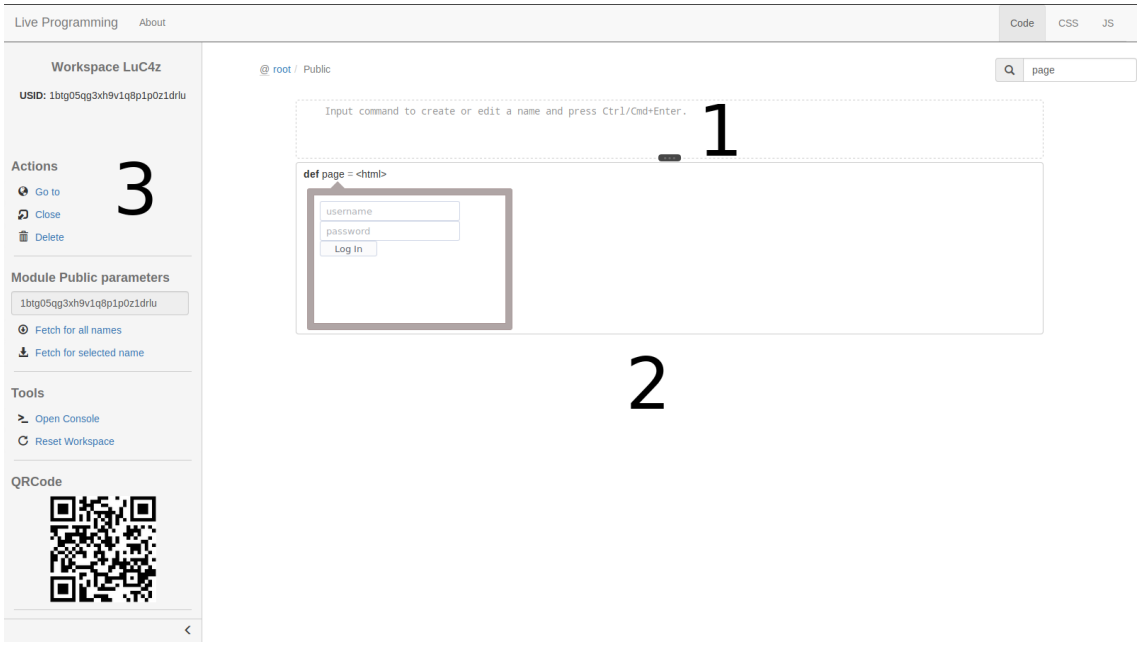

Figure 3.1: IDE

In order to keep track of the module arguments a page was requested with, we extended the WebSockets page subscriptions with information about the module a page is defined in, as well as the module arguments used to evaluate the requested page.

3.3 IDE

In this section we introduce the browser-based IDE[1] for our language. We reworked an already existing IDE for the extended language [Mat15] in order to support the concept of modules and paths. The IDE allows a user to develop applications incrementally with immediate feedback, and use the created applications in the same environment. The new IDE provides new tools to better navigate the development of an application using modules and session identifiers.

The IDE is divided in three main areas as indicated in fig. 3.1. Each numbered section is described as follows:

1. **Live Editor** – In this re-sizable panel the user can write code and send it to the interpreter, through the server. Errors are shown with a pop up above this panel. The module path in which the written code runs is indicated above the editor followed by the symbol @. This path navigator is composed of clickable breadcrumbs which provides an easier backwards navigation. In fig. 3.1 the current path is */Public*.

2. **Names Panel** – This panel lists all the names in the current path. Each name can be selected, and always displays his respective expression and/or value. The list of

[1] IDE available at `http://live-programming.herokuapp.com`

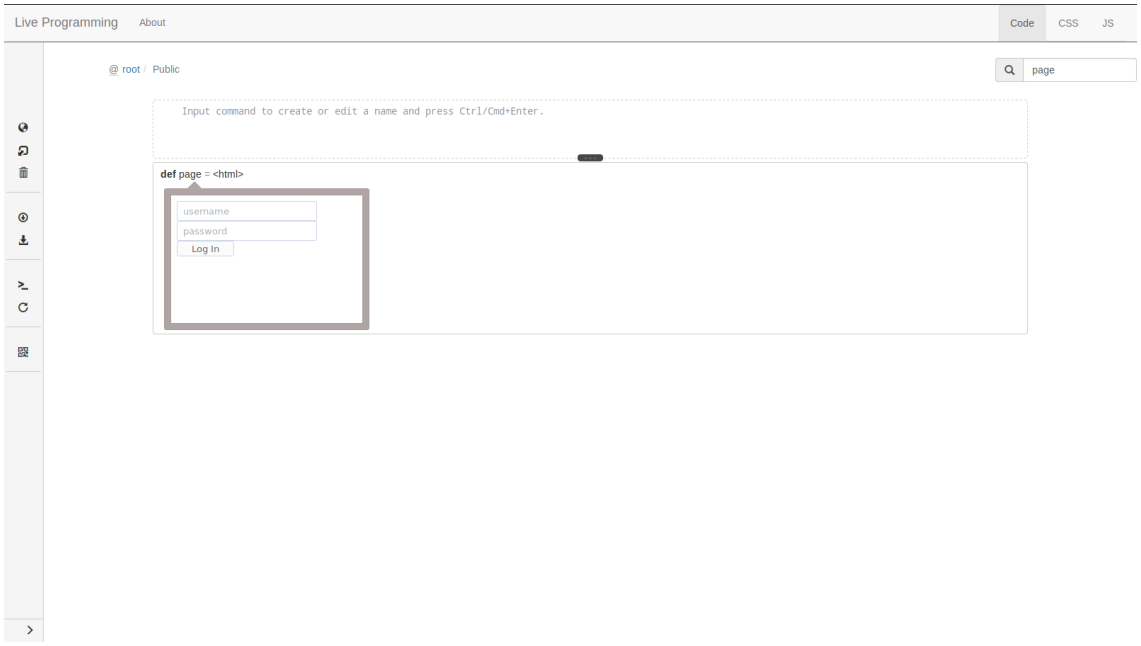

Figure 3.2: IDE with collapsed side bar

names navigated names using the arrow keys, and when a `module` is selected the user can open it by pressing *enter*. *Backspace* will navigate the user backwards by one module.

3. **Side bar** – This panel provides the workspace name, the USID of the user, as well as a four main sub bars: action bar, module parameters bar, tool bar, and a Quick Response Code (QRcode) bar. When a name is selected, the side bar is updated to show all the available options. The action bar gives all names the `delete` action. Names with HTML values provide two unique actions: a *go to* page action that sends the user to the page in a new window, and an action to open the page directly in the IDE with a small frame. Names with action values can be executed in the action bar. Names with function values provide an action to run the function accompanied by a text box in which the user can write the arguments. When inside a module, each module parameter has an *input* available with the respective parameter type, and ech action from the side bar uses a set of arguments managed by the user. The tool bar also provides a way to reset the workspace, which restores the workspace to an empty one. A console can be opened from the tool bar to provide the user with a Read–Eval–Print Loop (REPL). All executed actions in the side bar are also printed in the console display. All names generate a QRcode that links to the name directly.

There are other relevant elements in the IDE, such as the search bar located opposite to the path breadcrumb. The search bar filters names in panel *1* and is a retractable element, so it can be closed and opened as needed by clicking the icon. The tabs on the top right of the IDE allow the user to navigate between the JavaScript (JS), Cascading Style Sheets (CSS), and language editors. The CSS and JS editors are based on the Ace [Ace]

editor, which provides highlighting support and live syntax checking. It is also possible to collapse the tool bar (panel *3*) as shown in fig. 3.2. Most actions and tools are kept in the collapsed bar, with the respective icon identifying each one. The *About* page can be accessed in the top bar, and contains information about the IDE as well as a list of small example applications that can be copied to an empty workspace.

Chapter 4

Related Work

In this chapter we study existing solutions for the problems we tackle in this thesis and compare them with our introduced abstractions. We first explore existing basic session mechanisms used to tackle user identification and authentication. Then, explore modern web frameworks too study the most common approaches to authentication, access control, role based development and synchronization in each one. Finally, we study a programming language for writing lenses in order to better understand lens programming.

4.1 Basic Session Mechanisms

In web applications, a session is a semi-permanent interactive data interchange, between two or more communicating devices, or between a computer and a user [Ses]. A session is established at a certain point in time, and has an expiration date. An established session may involve more than one message in each direction. Usually, a session has an associated state, meaning that at least one of the communicating parts needs to store session data in order to communicate. Establishing a session is one of the basic steps to performing a connection-oriented communication. In the next section we will study some of the session mechanisms used in providing state to communications in stateless web and Internet protocols.

4.1.1 Cookies and tokens

An HTTP Cookie, or just Cookie, is the most primitive mechanism, embedded into the HTTP protocol, to store small amounts of information on the user's browser. Cookies are designed as a reliable mechanism for websites to persist user's information on the browser and recognize the user in a later interaction. The browser sends the Cookies back to the respective sites every time the user accesses them. Cookies are set in HTTP

requests (response by a server to a request) through a Set-Cookie header which instructs browsers to store the data in a Cookie. Afterwards, the cookie data is sent along in every request made to the same server in the form of a Cookie HTTP header. Additionally, an expiration date, and restrictions to a specific domain can be specified.

Perhaps the most important function a Cookie performs in the modern web is supporting authentication between requests, a form of stateful communication between client and server. Websites send back to the client some unique user information, usually a server-generated token, allowing the website to check in future requests if a user is authenticated or not and respond accordingly to the identified user.

4.1.2 Authentication

Authentication is the act of verifying an identity, more precisely in web and cloud applications, the identity of a user [Aut; CC12]. There are three existing authentication factors in the process of identifying a user: knowledge factor, ownership factor, and inherent factor. Knowledge factors include elements that a user knows (e.g. password, security question). Ownership factors include elements that a user has (e.g. tokens). Inherent factors include elements that a user is or does (e.g. fingerprint, voice, bio-metrics, signature). For a positive authentication at least elements from two factors are required. Authentication in web and cloud applications is commonly performed using a two-factor authentication with something the user has (e.g. username, id) and something the user knows (e.g password).

HTTP supports basic schemes for authentication like basic access authentication (providing a username and a password) and digest access authentication (applies an hash function before sending the credentials over the network). These mechanisms operate via challenge-response mechanisms in which servers identify and issue challenges before answering to requests. HTTP also allows the definition of separate authentication scopes under one root Uniform Resource Identifier (URI), called authentication realms.

Another approach to authentication in the modern web is authentication using third-party services, eliminating the need for application developers to build their own ad-hoc login systems, allowing users to login to multiple unrelated services with possibly the same identity and credentials. OpenID [Ope] is an open standard for this decentralized authentication protocol which several large organizations either issue or accept (e.g. Microsoft [1], Google [2]). Users just need to create an account in OpenID by selecting an identity provider and then sign onto any website that accepts it.

4.1.3 Session Mechanisms

HTTP is the foundation of data communication in the World Wide Web, and is a stateless protocol [Rfc], which means that, every time a communication between a server and

[1] https://www.microsoft.com
[2] https://www.google.com

a client ends, the information about the session is lost from the communication stack. Sometimes it is convenient to maintain session related data, for example, to avoid asking for a password every time a client makes a request to the server, to keep track of a shopping cart over multiple requests to the server. In order to maintain session data there are two obvious possibilities: Server Side web sessions and Client Side web sessions.

In a Server Side web session implementation, session information is stored on the server and a token is used to uniquely identify the session. The token is either explicitly stored in an HTTP Cookie on the client browser or, explicitly sent as a parameter in the request also known as URL Rewriting. Each request carries the Cookie to the server so it can match the request with the right session information in the server.

On the other hand, in a Client Side web session implementation, HTTP Cookies are used to directly store the session information on the client. Cookies are automatically carried over on each request to give the server all the information needed about the session.

To show how these two styles are commonly implemented in frameworks we consider the case of Java Servlets [3], which mainly follows a Server Side web session style through the java interface HttpSession [Httb]. When a client makes a request to a servlet, the request object (HttpServletRequest java object [Htta]) provides a method (*getSession*) that retrieves, or creates, an HttpSession object to manipulate session information on the server. By default the server maintains the HttpSession objects in a map, meaning that if the server goes down, all the session information is lost. However, servers can be configured to persist session data to disk. To uniquely identify each HttpSession object in the map, an HTTP Cookie (named JSESSIONID) containing a unique identifier (generated by the servlet), is kept on the client. In this way, when a client makes a request with an HTTP Cookie named JSESSIONID, the HttpSession object that corresponds to that particular session is returned by the *getSession* method, and with it the server can manipulate the session information for that particular client. Additionally, Java Servlets also implements Client Side web sessions by allowing the creation and manipulation of HTTP Cookies kept on the client to store any kind of session information.

Aside from Server Side (with or without URL Rewriting) and Client Side web sessions, an alternative, and well known approach for storing session information is also commonly used. In this approach, hidden fields are inserted in web pages to store session information related to the client accessing the web page. This technique obviously raises security concerns, since, even thou they are hidden in the page, the source of the web page can be inspected to uncover its content. The same concerns apply to HTTP Cookies, which are stored in the browser but can be viewed manually, or even stolen if given the required access to the machine. In both HTTP Cookies and hidden field storage of session data, cryptography can be employed to mitigate these security concerns.

[3] http://docs.oracle.com/javaee/6/tutorial/doc/bnafd.html

4.1.4 Analysing Session Mechanisms

While Server Side sessions are usually efficient and secure, in high-availability systems with no mass storage it becomes difficult to maintain efficiency. Although limiting the number of clients accessing the server severely weakens the availability of a server, it makes it possible to reserve a portion of the Random Access Memory (RAM) for storage of session data to tackle the absence of mass storage. In contrast, Client Side sessions can mitigate the weight of storing and loading all session information on the server in high-availability systems, however, as previously mentioned data stored in the client is vulnerable to tampering. In order to increase security of session data stored on the client, the server must be the only location to initiate a valid session, as well as the only system able to interpret and manipulate the data. To guarantee confidentiality and integrity requirements a variety of cryptography methods can be used. A clear problem of storing session information on the client is the size of the data being transmitted between the client and the server with every request, which is aggravated by the limitations that some browsers impose on the size and number of Cookies each site can store on the client. To improve efficiency in carrying large amounts of session data and meet the limits of browsers, servers may compress data before creating a Cookie and decompressing it when the Cookie is received in a future request.

Many frameworks already provide at least either Server Side sessions or Client Side sessions in the most basic form without giving too much increased security. While it is possible to code all the increased security discussed previously, it is an extensive process, prone to mistakes. To make it simpler and safer to develop applications with sessions, a framework could infer what information needs to be sent to the client and, be able to tell when certain security invariants are being broken by knowing what information should not be sent to the client. The only choices the developer should have to think about are the configuration choices of the employed mechanisms and perhaps where information is allowed to flow by explicitly marking data with annotations for security purposes. However, the underlying mechanisms used by the framework should guarantee all the confidentiality and integrity requirements without compromising efficiency.

4.2 Related Frameworks

In this section we study some of the existing web frameworks, focusing on the features relevant to our topic. For each framework, we make a brief introduction highlighting key aspects of the framework, and give an example on how a simple web application is built. We explore how several frameworks provide session management, access control, and synchronization. Finally, we examine how ensuring data privacy in the studied frameworks is addressed.

```
application example

entity Message {
  author :: String
  text :: Text
}

entity Wall {
  posts -> List<Message>
}

var wall := Wall{}

define page root() {
  title { "Wall" }
  navigate(post()){ "Post" } " "
  for(m: Message in wall.posts) {
    "Author: " output(m.author)
    "Message: " output(m.text)
  }
}

define page post() {
  title { "Post a new message" }
  var m := Message{}
  form {
    label("Input Message:"){input(m.text)}
    label("Author:"){input(m.author)}
    submit("Post", action {
      m.save();
      wall.posts.add(m);
      message("New post created.");
      return root();
    })
  }
}
```

Listing 4.1: WebDSL example application

4.2.1 WebDSL

WebDSL [Gro+08] is a domain-specific language for developing dynamic web applications that translates to Java web applications, and has some key features that are relevant to our topic, such as a rich data model (entities), access control and generation of a synchronization framework. WebDSL applications are organized in _*.*app*_ files. Modules in WebDSL can be written in .app files and imported to a main .app file where the application header is declared. To show how an application is built in WebDSL, consider listing 4.1, a very simple example application in which a client can post messages and view a wall of all of the posted messages. Following the application header, two entities are declared, an entity representing a message, and an entity representing a wall to store all the messages in a collection. The `Wall` variable is instantiated followed by two pages, one iterates the collection of messages to show a wall of messages, and the other provides a form in which a client can input a text message to post.

```
session wall {
  messages -> List<Message>
}
```

Listing 4.2: WebDSL session entity

```
extend session wall {
  friends -> List<User>
}
```

Listing 4.3: WebDSL session entity extension

```
entity User {
  name :: String
  password :: Secret
}
principal is User with credentials name, password
```

Listing 4.4: WebDSL principal definition

4.2.1.1 Sessions

WebDSL follows a Server Side web session style where session data is stored on the server with globally visible variables in the application, also known as session entities [Weba]. Listing 4.2 shows an example of how to define a session entity for a wall of messages. A session entity object is automatically instantiated when a browser makes a request to a server, which responds with a cookie named *WEBDSLSESSIONID* with the identifier for that client session. Session entities, just like regular entities, can be extended in order to add new properties to an already existing session entity. In listing 4.3 we add a list of friends to our wall session entity.

4.2.1.2 Access Control

Access control, in WebDSL, is defined using a sub-language used to define rules over resources. To allow the creation of rules over an authenticated principal, the sub-language supports the declaration of a principal using a user defined entity and a set of credentials, which generates a session entity to hold the currently signed in user. Listing 4.4 shows a configuration of a principal using the `User` entity. The generated session entity is show in listing 4.5. In this declaration two things are introduced, the principal representing the currently signed in user, and a function to verify whether the principal is signed in

```
session securityContext {
  principal -> User
  loggedIn :: Bool := this.principal != null
}
```

Listing 4.5: WebDSL security context

```
access control rules

rule page editMessage(m:Message) {
  m.author == principal
}
```

Listing 4.6: WebDSL access control rules

```
// Compiles
rule page editMessage(m:Message) {
 m.author == principal

 rule action save() {
   m.author == principal
 }

 rule action cancel() {
   m.author == principal
 }
}

// Does not compile
rule action save() {
 m.author == principal

 rule page editMessage(u:Message) {
   m.author == principal
 }
}
```

Listing 4.7: WebDSL access control constraints

or not. In the current implementation of WebDSL the authentication credentials are not used, but in the future, the WebDSL developers pretend to derive a default login template from the given authentication credentials. Additionally, login and logout templates are generated along side an authenticate function that checks if the given credentials are correct, and setting the principal property if they are. The auto-generated templates and function, can be overridden to allow further control of the authentication step. Creating access control rules becomes very simple after the `securityContext` is configured, for example, in listing 4.6 we define a rule stating that only the author of a message can edit it. The first line states that the following declarations will be access control rules. Rules can be applied to other resources like templates, page actions, functions or pointcuts, which will be explained later. In this example, our resource is a page, named `editMessage`, with a message as an argument. The `securityContext` session is available inside the rules, and by using the principal declaration we can check if the author of the message is the currently logged user.

Nested rules are allowed for a finer-grained control, but with some constraints as show in listing 4.7. Pages are parent resources of actions, this implies that a nested rule is only valid for usage of that resource inside the parent resource. Often, a page resource

```
rule page editMessage(m:Message) {
  m.author == principal

  rule action *(*) {
    m.author == principal
  }
}
```

Listing 4.8: WebDSL unfolded rules

```
pointcut userSection(u:User) {
 page editUser(u),
 page post(u)
}

rule pointcut userSection(u:User){
 u == principal
}
```

Listing 4.9: WebDSL pointcuts

```
entity Message {
  id :: String

  synchronization configuration {
    toplevel name property : id
  }
}
```

Listing 4.10: WebDSL top level synchroinzation

allows all its actions with the same rule for accessing the page, therefore describing a page rule automatically makes all its actions follow the same rule. For example, listing 4.8 shows the unfolded `editMessage` page rule. The symbol `*` states that any action with any number of arguments will be matched.

Resources can also be grouped into `pointcuts` in order to apply the same rule to a group of resources. In listing 4.9 we create rules to control user actions with `pointcuts`.

4.2.1.3 Synchronization Framework

WebDSL provides generation of code through the WebDSL IDE for a synchronization framework, which exposes web-services for external applications to use. To generate the synchronization framework it is required that a WebDSL application, with at least a complete model, is already in place. Setting up the framework requires a declaration of a `String` property that represents the object and enables data partitioning, which is primarily used to reduce data sent to mobile applications that access the synchronization webservices. Listing 4.10 defines a synchronization configuration for the `Message` entity. The configuration allows for the definition of access control rules over data to control

```
entity Message {
  id :: String

  synchronization configuration{
    toplevel name property : id
    access read: true
    access write: Logedin()
    access create: principal.isAdmin()
  }
}
```

Listing 4.11: WebDSL synchronization access control

```
entity User {
  name :: String
  fullName :: String

  synchronization configuration{
    restricted properties : fullName
  }
}
```

Listing 4.12: WebDSL synchronization property restriction

which external sources accessing the data synchronization framework can read, write entities or create instances of those entities. These access control rules can only be written if a principal is previously defined, which in this case should be for authenticating devices accessing the framework. In listing 4.11 we define access control rules for the synchronization of the `Message` entity. Lastly, it allows the configuration of restricted properties, meaning that the declared properties will not be shared in the synchronization. Listing 4.12 defines the property `fullName` restriction for the `User` entity synchronization.

After the synchronization framework files are generated, they are imported to the application to make the web-services available to the applications. The available web-services are called with *POST* requests to access the core synchronization functions. The URL is structured as follows:

http://<websiteurl>/webservice/<webservicename>

The services provided at the URL are:

- getTopLevelEntities
- getTimeStamp
- syncNewObjects
- syncDirtyObjects
- sync

```
<body>
  <h1>Post</h1>
  <form class="new-message">
    <input type="text" name="message" placeholder="Message..." />
    <input type="text" name="author" placeholder="Author" />
  </form>
  <h1>Wall</h1>
  <ul>
    {{#each getMessages}}
      <div>Message:{{message}}</div>
      <div>Author:{{author}}</div>
    {{/each}}
  </ul>
</body>
```

Listing 4.13: Meteor application - HTML

The framework also adds three functions to entities for serializing data, because the values in the database are not in the format required for transmission through the web-services. The framework provides a page for each entity where stored data can be browsed.

4.2.2 Meteor

Meteor [Meta] is a JS Web Framework written in Node.js [Nod] which allows for rapid prototyping and cross-platform code production. One of the key features of Meteor is its ability to automatically propagate data changes to clients without any additional code from the developer. The Synchronization process is achieved with the Distributed Data Protocol (DDP) [Metb] and a publish-subscribe pattern [Xep]. By default, the publish-subscribe pattern automatically publishes everything (*auto-publish* package, which should only be used for fast prototyping), meaning that the entire database is present on the client without any filtering.

Another very important feature of Meteor is the use of collections. In Meteor, data is stored in synchronized collections that are available both on the server and on the client. Although there are several projects to support more database systems, MongoDB [Mon] is currently the most stable and maintained. On the server, the collection is stored on a MongoDB database by default, and on the client the collections are managed through a JS implementation of MongoDB in memory [Metc]. This gives the client access to a collection without talking to the server. To keep certain fields of the collection values from being transferred to the client, the publish-subscribe pattern can be customized for each collection accordingly. Finally, when creating a Meteor collection, a connection can be fed to the constructor which specifies a server to manage that collection, making Meteor collections very powerful in multi-server applications.

To show how an application is built in Meteor, consider the simple example application in listings 4.13 and 4.14, which a client can post messages and view a wall of all of the posted messages, live, in the same page. First, we look at the JS code, then the HTML. In Meteor we can write client and server code in the same file if we want, and

```
Wall = new Mongo.Collection("wall");

if (Meteor.isClient) {
  Template.body.helpers({
    getMessages: function () {
      return Wall.find({});
    }
  });

  Template.body.events({
    "keypress .new-message": function (event) {
      if (event.which != 13) return;
      var message = event.target.parentElement.message; // get message
      var author = event.target.parentElement.author; // get author
      Wall.insert({
        message: message.value,
        author: author.value
      });
      message.value = ""; // clear message
      author.value = ""; // clear author
    }
  });
}

if (Meteor.isServer) {
  Meteor.startup(function() {});
}
```

Listing 4.14: Meteor application - JS

to distinguish client code from server code, two boolean values are available through the Meteor object: `Meteor.isClient` and `Meteor.isServer`. Code written outside these contexts is ran on both sides. The first thing we declare is a MongoDB collection to store the messages, which will be available both on the client and on the server as explain before. On the client, we introduce two new things: template helpers and template events. Template helpers provide the views on the client with functions or values, in our example we implement a function that returns every record in the `Wall` collection. Template events are useful to map functions to events that might occur in the template. We map a `keypress` event on an element with a class named `new-message` (in our example the element is a form) to a function that, in case the pressed key is an *Enter*, retrieves the values (message and author) from the inputs and adds a new record to the collection, that is, a new message. On the server side, it simply starts the server with one function and a callback. Finally, in the view, we have a simple form with an input field for the new message, and another for the author name. Bellow the form, we list the collection by using the template language of Meteor (Spacebars), to iterate over all the messages that the template helper `getMessages` returns.

```
<template name="main">
  <p>Session var has: {{showX}}</p>
</template>
```

Listing 4.15: Meteor reactive session - HTML

```
Template.main.helpers({
  showX: function () {
    return Session.get("x");
  }
});

Session.set("x", "1");
// Page will say "Session var says: 1"

Session.set("x", "2");
// Page will say "Session var says: 2"
```

Listing 4.16: Meteor reactive session - JS

4.2.2.1 Sessions

In Meteor, sessions are accessed through a global object on the client in which the user can store an arbitrary set of key-value pairs. Sessions are also reactive, when a value for a session key is set, the change is propagated to the client and the user can see the page automatically change with the new value. Listings 4.15 and 4.16 defines an example of a reactive session. The `Session` keyword is the global object that accesses the session key-value pairs, and it is where we store the variable `x`. In the template `showX` function we fetch and return the value of `x` stored in the session. This way, when the value is changed, the page with the template will update automatically. By default, sessions in Meteor are not permanent, if a page is refreshed the session object will be a new one. There are, however, available packages which provide implementations for persistent sessions, some with Server Side Session others Client Side Session approach.

Meteor provides a package manager [Atma] for developers to publish and distribute packaged code. To implement user authentication, Meteor provides a set of packages to manage accounts. The accounts base package uses a Meteor collection to store the users and keeps the session token in the local storage of the browser to make the session permanent. Additionally, a package can be added that exposes templates to add login forms to an application that work out of the box with the base package.

4.2.2.2 Access Control

Meteor provides *allow* and *deny* methods for collections which give the developer the freedom of controlling every action done on a collection, be it an insert, update or remove operation. For example, listing 4.17 shows how to control who can insert in a collection of messages. Whenever a client executes such operations, the defined methods are called to check whether the operation is allowed to continue or not. Although this is a very

```
Wall = new Mongo.Collection("wall");

Wall.allow({
  insert: function (user, message) {
    return (user && message.author === user);
  }
});
```

Listing 4.17: Meteor access control

```
Roles.addUsersToRoles(AliceId, ['moderator'], 'wall')
Roles.userIsInRole(AliceId, 'moderator', 'wall')  // => true
Roles.userIsInRole(AliceId, 'admin', 'wall')  // => false
```

Listing 4.18: Meteor roles package

flexible way of creating access control, it requires the developer to create code for every operation of every collection if he wishes to specify what is allowed.

To simplify the steps in defining access control in Meteor, a package named *meteor-roles* offers the ability to attach permissions to the already existing users collection. With it, the developer can define the permissions for a user inside a domain. Then, those permissions can be verified. For example, in listing 4.18 we assign the role of *wall moderator* to the user Alice, to check if Alice does indeed have the `moderator role`, and that she does not have the `admin` role. Even thou the developer still needs to address each operation with this package, it simplifies the aggregation of permissions in a large set of users.

4.2.2.3 Synchronization

Meteor synchronization is achieved with the DDP and publish-subscribe pattern. DDP is a very simple protocol for fetching structured data from a server, it works like a REST service but through web sockets, giving live updates when data changes. In Meteor, a client communicates with the server through this protocol. Note that the protocol works as a REST interface, meaning that a connection can be made outside a Meteor application, both server or client side, to receive updates following the DDP. To control what goes where Meteor uses the publish-subscribe pattern, where the developer can specify what is published and who subscribes through a couple of methods (publish and subscribe).

4.2.3 Opa Language

The Opa Language is a full-stack web framework language for the development of web applications [Opaa]. It can be used for both client-side and server-side scripting with the Opa language, which is then compiled to Node.js on the server and JS on the client. The database programming is in MongoDB. The Opa language implements strong, static typing which helps avoid many security issues, for example, Structured Query Language

```
type Session.instruction(state)=
  {set: state} /** Carry on with a new value of the state value.*/
  {unchanged}   /** Carry on with the same state.*/
  {stop}        /** Stop this session. Any further message will be ignored */
```

Listing 4.19: Opa Lang session instructions

(SQL) injections or cross-site scripting attacks. One of the biggest features of Opa Language is the *Power Rows*, which are extended JS objects, they have the flexibility of dynamic languages but with a type checker that keeps the language safe. Another feature, and one that we explore, is the slicer that Opa lang uses to determine where each top-level declaration runs (client side, server side, or both).

4.2.3.1 Sessions

Opa language provides three primitives for communication between clients and servers: Session, Cell and Network. Network is used to broadcast messages for multiple clients, and Cell is a two-way, synchronous communication. Session is a one-way asynchronous communication, and it is the one we will be focusing on.

The Opa standard library description of session says:

"A session is a unit of state and concurrency. A session can be created on a server or on a client, and can be shared between several servers."

Sessions in Opa, are supported by encapsulating an imperative state and using message passing to communicate with message handlers [Kop11]. Upon receipt of a message by the handler, the session can be changed, or even terminated, with a set of available instructions illustrated in listing 4.19.

Creating a session requires an initial state of the session and a message handler, and returns a channel to which messages can be sent to be processed by the message handler. Messages can be sent from/to different servers/clients. Listing 4.20 shows an example of a session that keeps track of a friends list. Here, we define a message handler to change the state when a message matches the first case by returning the `set` instruction, and by default keeping the state intact with the `unchanged` instruction. The session is then created with an initial state, an empty list, and the handler previously declared. The returned value is a channel which we then use to define a function that adds friends in a session.

4.2.3.2 Slicer

The Opa language can be executed on both the server and the client, and so, it must be decided, at compile time, where the code actually runs. This decisions are made by the Opa language slicer [Opab]. With slicing annotations the slicer can be told where each top-level declaration should run. There are three possible slicing annotations that can be written before the `function` keyword: `client`, `server`, and `both`. Each one tells the slicer

```
function handleFriends(oldState, m) {
  match(m) {
    case {add : user} : {
      set : List.add(user, oldstate)
    }
    default : {
      unchanged
    }
  }
}

friends = Session.make([], handleFriends)

function addFriend(user) {
  Session.send(friends, {add : user})
}
```

Listing 4.20: Opa Lang session example

```
number = @sliced_expr({server: 2, client: 1})
do println(side)
```

Listing 4.21: Opa Lang explicit sliced states

where to run the code, but it does not mean that it is invisible to the other side. When running on both sides, it either executes the side effects on both sides or it only executes on the server and shares the results.

When slicing annotations are omitted, the slicer decides to place declarations on the server if possible, otherwise it places them on the only possible side. Writing a slice annotation on a module defaults all the declarations inside it to the same annotations, but they can be overridden with other slice annotations. There are however some rules due to the simple fact that everything can not be placed on both sides. Primitives declared on one side can only be placed on that side. If a primitive is sliced server only, it also means that it is *server private*. Primitives tagged as *server private* cannot be called by the client, and all declarations using it will become *server private* themselves. There is however a directive (*publish*) to stop this propagation of the tag, essentially telling the client can now see the declaration (e.g release data after an authentication mechanism succeeds).

Sometimes the developer wants to have different behaviors for a declaration depending on the side it is running. To this end Opa lang provides a way of this as shown in listing 4.21.

4.2.4 Yesod

Yesod is a web framework based on the Haskell language [Has] for developing type-safe REST model based web applications [Yes]. Type safety is the key feature of Yesod, by giving high-level declarative techniques, the developer can define the expected input types,

```
-- TH function
$(hamletFile "template.hamlet")

-- QQ function
[hamlet|<p>This is quasi-quoted Hamlet.|]
```

Listing 4.22: Yesod TH and QQ functions

as well as having the guarantee that the output is also well formed through the process of type-safe URLs. Yesod provides entity definition in a higher level, with all the necessary process of persisting and loading data being performed inside so that the developer can remain ignorant to the details. The Yesod framework also shines performance wise. Using the Haskell's Glasgow Haskell Compiler (GHC) as well as allowing HTML, CSS and JS to be analyzed at compile time, Yesod provides great performance by avoiding disk I/O at runtime.

Yesod makes good use of Haskell's features to save time in developing with code generation. Code generation comes in two forms, scaffolding for starting projects faster, and libraries. Using the libraries means that the generated code will always be up to date and everything is taken care of at compile time. All the code can be written manually, without using library specific code generation, if more control is required.

One such library is the Template Haskell (TH), which essentially generates an Haskell abstract syntax tree, reducing a boilerplate code in many occasions. QuasiQuotes (QQ) is a minor extension to the TH library, and an important library. It allows arbitrary content to be embedded within Haskell source files. Consider listing 4.22, while a TH function like `hamletFile` can read the template contents from a file, QQ provides one named `hamlet` that reads the contents inline.

In listing 4.23, we defined an example wall application to demonstrate how a small application is built in Yesod.

```
{-# LANGUAGE EmptyDataDecls             #-}
{-# LANGUAGE FlexibleContexts           #-}
{-# LANGUAGE GADTs                      #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE MultiParamTypeClasses      #-}
{-# LANGUAGE OverloadedStrings          #-}
{-# LANGUAGE QuasiQuotes                #-}
{-# LANGUAGE TemplateHaskell            #-}
{-# LANGUAGE TypeFamilies               #-}
{-# LANGUAGE ViewPatterns               #-}

import Control.Applicative ((<$>), (<*>))
import Data.Text (Text)
import Yesod
import Yesod.Form.Jquery
import Database.Persist.Sqlite
import Control.Monad.Trans.Resource (runResourceT)
```

```
18 import Control.Monad.Logger (runStderrLoggingT)
19
20 share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
21 Message
22     author Text
23     text Text
24     deriving Show
25 |]
26
27 data Wall = Wall ConnectionPool
28
29 mkYesod "Wall" [parseRoutes|
30 / WallR GET
31 /post PostR POST
32 |]
33
34 instance Yesod Wall
35
36 instance RenderMessage Wall FormMessage where
37     renderMessage _ _ = defaultFormMessage
38
39 instance YesodPersist Wall where
40     type YesodPersistBackend Wall = SqlBackend
41
42     runDB action = do
43         Wall pool <- getYesod
44         runSqlPool action pool
45
46 messageAForm :: AForm Handler Message
47 messageAForm = Message
48     <$> areq textField "Author" Nothing
49     <*> areq textField "Message" Nothing
50
51 messageForm :: Html -> MForm Handler (FormResult Message, Widget)
52 messageForm = renderTable messageAForm
53
54 getWallR :: Handler Html
55 getWallR = do
56   wall <- runDB $ selectList [] []
57   (widget, enctype) <- generateFormPost messageForm
58   defaultLayout [whamlet|
59     <p> Post a new message!
60     <form method=post action=@{PostR} enctype=#{enctype}>
61       ^{widget}
62       <button>Post
63     <p> Messages!
64       $forall Entity messageid message <- wall
65         <p>Message: #{messageText message}
66         <p>Author: #{messageAuthor message}
67   |]
```

```
68
69 postPostR :: Handler Html
70 postPostR = do
71   ((result, _), _) <- runFormPost messageForm
72   case result of
73     FormSuccess message -> do
74       runDB $ insert message
75       redirect $ WallR
76     _ -> redirect $ WallR
77
78 openConnectionCount :: Int
79 openConnectionCount = 10
80
81 main :: IO ()
82 main = runStderrLoggingT $ withSqlitePool "demo.db3" openConnectionCount $
83   pool -> liftIO $ do
84     runResourceT $ flip runSqlPool pool $ do
85         runMigration migrateAll
86     warp 3000 $ Wall pool
```

Listing 4.23: Yesod Wall application example

At the top, we declared the extensions being used in the application, also known as language *pragmas*, followed by the imported classes required for what we need to build. The `share` function generates the code for building the entity representing the messages as well as the code required for persisting that entity (in our example we use the SQLite [Sql] library), while the quasi-quote function `persistLowerCase` converts a whitespace-sensitive syntax into a list of entity definitions. The data type declared as `Wall` represents the foundation data type of the application, and it must be an instance of the `Yesod` type class as declared afterwards. The foundation data type of an application can store a variety of things, and in this case it stores the connection pool for persisting data. Next, we have the route definitions. One route for the main page, where we will show all the message and a form for posting a new message, and a route for the form action that inserts new messages. The following defined instances included in our application are, respectively, used for automatic generation of our `Message` entity form, and for persisting data in the application with a function that runs an action in the connection pool. The following two functions, `messageAForm` and `messageForm`, are used to ultimately build a table with a form for the `Message` entity. The next two functions, `getWallR` and `getPostR`, respectively handle the requests for the `WallR` and `PostR` routes. In `getWallR` we first fetch all the messages, then, we generate the widget for the `Message` entity for building our form, and finally, we create the page to be returned. In the page we have the form for posting new messages as well as a list of all the messages, which are iterated with a `forall` function given the already fetched list of messages from the database. For handling the POST requests, the `getPostR` checks the result of the form, if it is a success it inserts the new message in the database and redirects to the main page.

For simplicity, any other result redirects to the main page. Finally, we declare the number of possible open connections in the pool, and we define our main function. In it we start a logger, the connection pool for SQLite, and the Warp Webserver with our `Wall` foundation and connection pool.

4.2.4.1 Sessions

Unlike the previously studied frameworks, Yesod by default implements sessions in a Client Side web session style with a package named `clientsession`. Data is stored in an HTTP Cookie using encryption and signatures, which overcomes the security concerns that rise from using Client Side web sessions. Encryption ensures that the user can't inspect the Cookie and understand its contents, and signatures protect the Cookie from being tampered with. To change the file path for the encryption key for client session or the session timeout, one can override the `makeSessionBackend` method in the `Yesod` type class. Also, if we want to turn off session handling this method can be overridden to return `Nothing`, although Cross-Site Request Forgery protection is also disabled along side session handling. There are, however, other functions for finer grain control of session configurations.

The only remaining security concern is that using the client sessions over HTTP brings the inherent vulnerability of an attacker being able to read the traffic and impersonating the user by obtaining his Cookie. Secure Sockets Layer (SSL) is the only solution to this vulnerability, and preventing browsers from accessing the site with HTTP. To run the entire site over SSL, Haskell has a solution called *warp-tls*, and to prevent the site from sending Cookies over insecure connections we can apply transformations to the `makeSessionBackend` method. This transformation turns on the Secure *bit* of Cookies in order for the browsers not to transmit over HTTP to the domain.

The API for the base session is available through four functions: `lookupSession` to retrieve a value (if available) with a given a key, `getSession` to retrieve all of the key/value pairs, `setSession` to set a value for a given key, and `deleteSession` to clear a value for a given key. Recalling listing 4.23, in listing 4.24 we set a session value with the most recent post when it is successfully inserted in the `PostR` route, and retrieve it when handling the `WallR` route.

Yesod also provides a pair of functions to enable the storage of messages in the session for sending success and failure messages to a redirected page. A `setMessage` to store a message in the session, and `getMessage` to read and clear the previously stored message. In listing 4.25 we define a success message and error messages to be stored when posting on the wall, which are then shown in the wall page.

4.2.4.2 Access Control

Authentication in Yesod is supported through third-party authentication systems, like OpenID [Ope], BrowserID [Bro], and OAuth [Oau]. It also supports the basic and more

```
getWallR :: Handler Html
getWallR = do
  wall <- runDB $ selectList [] []
  (widget, enctype) <- generateFormPost messageForm
  value <- lookupSession "recentPost"
  defaultLayout [whamlet|
    $maybe v <- value
      <p> Session value: #{v}
    <p> Post a new message!
    <form method=post action=@{PostR} enctype=#{enctype}>
      ^{widget}
      <button>Post
    <p> Messages!
      $forall Entity messageid message <- wall
        <p>Message: #{messageText message}
        <p>Author: #{messageAuthor message}
  |]

postPostR :: Handler Html
postPostR = do
  ((result, _), _) <- runFormPost messageForm
  case result of
    FormSuccess message -> do
      runDB $ insert message
      setSession "recentPost" message
      redirect $ WallR
    _ -> redirect $ WallR
```

Listing 4.24: Yesod sessions

common mechanism of username/password systems. While the latter provides more control over the application development, adding third-party authentication systems in Yesod is simple and users don't have to remember a new set credentials. The package providing these authentication plugins is called *yesod-auth*. For each plugin it is required that users are identified with a unique string, for example, in BrowserID an email address is used. Despite the mechanisms behind each plugin, at the end of a successful login process the plugins set a value in the session indicating the user's *AuthId*. This is usually persisted in a table for tracking users. Due to the use of the underlying session mechanism of Yesod, the stored authentication value is safe with the same encryption, as well as having the same timeout as the session for the period in which the user is authenticated.

To build an application using on of these plugins, a type class named `YesodAuth` is used to specify a number of settings as well as requiring six declarations:

- the `AuthId` representing the value that is returned when asking whether a user is logged in or not;
- the `getAuthId` function for fetching the `AuthId` which contains the used authentication backend, the actual identifier, and a list for storing extra information;
- a redirect route for a successful login named `loginDest`;
- a redirect route for a successful logout named `logoutDest`;

```
getWallR :: Handler Html
getWallR = do
  wall <- runDB $ selectList [] []
  (widget, enctype) <- generateFormPost messageForm
  message <- getMessage
  defaultLayout [whamlet|
    $maybe m <- message
      <p> Message: #{m}
    <p> Post a new message!
    <form method=post action=@{PostR} enctype=#{enctype}>
      ^{widget}
      <button>Post
    <p> Messages!
      $forall Entity messageid message <- wall
        <p>Message: #{messageText message}
        <p>Author: #{messageAuthor message}
  |]

postPostR :: Handler Html
postPostR = do
  ((result, _), _) <- runFormPost messageForm
  case result of
    FormSuccess message -> do
      runDB $ insert message
      setMessage "Post Success"
      redirect $ WallR
    _ -> do
      setMessage "Post Failed"
      redirect $ WallR
```

Listing 4.25: Yesod session messages

- an `authPlugins` list containing the plugins used in our application;
- an HTTP connection manager for allowing third-party login systems to share connections reducing the cost of restarting Transmission Control Protocol (TCP) connections with each request.

Listing 4.26 sets up an authentication application, and defines a route named `AuthR` to support the access to the sub-site for authentication. Defining the route `AuthR` requires an additional two parameters, the authentication sub-site, and a function that retrieves the sub-site value which Yesod automatically provides. If more than one plugin is used, Yesod automatically unfolds the login hyper links for each plugin and provides the route with the appropriate sub-site being requested.

Finally, we can query the user's `AuthId` with the `maybeAuthId` function. Listing 4.27 checks if the user is logged in when accessing the main page of the example application.

Now that our application can authenticate users, we can control the access of their requests. Yesod provides authorization in a simple and declarative manner through two methods: `authRoute` and `isAuthorized`. These methods are added to the `Yesod` type class instance. `authRoute` should point to the login page, which is almost always `AuthR LoginR`. The function `isAuthorized` takes a requested route and a `boolean` value

```
import           Data.Default                    (def)
import           Network.HTTP.Client.Conduit (Manager, newManager)
import           Yesod
import           Yesod.Auth
import           Yesod.Auth.BrowserId

-- ...

data App = App { httpManager :: Manager}

instance YesodAuth App where
  type AuthId App = Text
  getAuthId = return . Just . credsIdent
  loginDest _ = WallR
  logoutDest _ = WallR
  authPlugins _ =
    [authBrowserId def]
  authHttpManager = httpManager

mkYesod "App" [parseRoutes|
  /auth AuthR Auth getAuth
|]
```

Listing 4.26: Yesod Authentication

```
getWallR :: Handler Html
getWallR = do
  maid <- maybeAuthId
  defaultLayout [whamlet|
      $maybe id <- maid
        <p>You are logged in as: #{show id}
        <p> <a href=@{AuthR LogoutR}>Logout
      $nothing
        <p> <a href=@{AuthR LoginR}>Go to the login page
  |]
```

Listing 4.27: Yesod logged in status

indicating if the request is anything but a *GET*, *HEAD*, *OPTIONS*, or *TRACE* request. In it we can write code, such as accessing the file system or the database. As an example, consider listing 4.28, in which we define a route `AdminR` where only the user named `admin` is permitted. The `AuthenticationRequired` value will redirect the user to the login page as defined by `authRoute`, and the `Authorized` value will validate the user.

Although the code we have to write is quite extensive if we want more complex control over the resources, Yesod provides with a highly customizable authentication solution, with much of the boilerplate code being generated for the user.

4.2.5 Data Privacy in Current Frameworks

Although each one of the studied frameworks provide the tools to build applications that require strict logic data isolation and sharing data between groups of users, they involve extensive hand-written code to do so. This usually leads to security conditions not being

```
instance Yesod App where
  authRoute _ = Just $ AuthR LoginR
  isAuthorized AdminR _ = isAdmin
  isAuthorized _ _ = return Authorized

isAdmin = do
  maid <- maybeAuthId
  return $ case maid of
    Nothing -> AuthenticationRequired
    Just "admin" -> Authorized
    Just _ -> Unauthorized "You must be an admin"
```

Listing 4.28: Yesod Authorization

met, or simple errors going unnoticed. Consider the Meteor framework for example, writing the logic for controlling which data from each collection is visible to each client is very flexible but can become extensive and quite complex. Any mistake in this coding won't be detected immediately and can give rise to serious security issues.

There are frameworks that support multi-tenancy in some way, like the Athena framework [Ath]. Athena is an enterprise object-relational mapping framework that allows the development of applications with a shared schema, by automatically turning single tenant queries to the database into queries that take into account an organization identifier to distinguish the tenants. Although Athena makes it easy to map applications to multi-tenancy, it still requires logic behind each tenant to be written. Additionally, this mapping is not flexible and doesn't address our goal of creating other applications of the same nature with other, custom, views of the application besides tenancy.

4.3 Programming with Lenses

Propagating changes between connected structures (e.g. databases and materialized views) is usually done in ad-hoc fashion, that is, hand-written transformations from one structure to another and back. Naturally, when the structures involved are complex manual management and maintenance of such transformations becomes equally complex. Writing such bidirectional transformations is a problem in a vast set of domains, including data converters and synchronizers, picklers and unpicklers (serializing data), structure editors and constraint maintainers for user interfaces [Hof+15].

Lenses are bidirectional transformations between a set of inputs `C` ("concrete structures") and a set of outputs `A` ("abstract structures"). `A` *lens* **l** is comprised of three functions:

$$l.get \in C \to A$$
$$l.put \in A \to C \to C$$
$$l.create \in A \to C$$

```
let c : string =
  Alice Lopes, 28, FCT
  Bob Lopes, 21, UCP
  Charlie Martins, 21, FCT
```

Listing 4.29: Concrete state defined in Boomerang

```
Alice Lopes, 28
Bob Lopes, 21
Charlie Martins, 21
```

Listing 4.30: Abstract state from *get* transformation in Boomerang

```
let NAME : regexp = [A-Za-z ]+
let AGE : regexp = [0-9]{2}
let COLLEGE : regexp = [A-Z]+
```

Listing 4.31: Regular expressions in Boomerang

The *get* function is a forward transformation, a total function from `C` to `A`. The *put* function takes an old `C` with an updated `A` and produces a correspondingly updated `C`. The *create* function works like the *put* function, except that it only takes an `A` argument (if the only available structure is `A` then defaults are supplied).

4.3.1 Boomerang

In order to illustrate what *lenses* can do, we show a practical example in Boomerang [Boh+08], a programming language for writing lenses. Boomerang was developed to operate on ad-hoc, textual data with a set of *string lens combinators* based on familiar regular operators (union, concatenation, and Kleene-star), and to address issues in manipulation of ordered data (*dictionary lenses*).

Consider that we want write a lens whose *get* function takes as a concrete state newline-separated records, like the one defined in listing 4.29, with comma separated data about students, their age, and college name. We want the returned abstract state in listing 4.30. First we define the three regular expressions in listing 4.31 to make the writing of the lens simpler to read. The regular expressions match, respectively, student names, their age, and the college name. Next we define the lens. The lens is broken down into two parts for easier comprehension. The first declaration of the lens shows how each record line chunk is transformed. The second declaration goes over each line applying the first declaration.

```
let compA : lens = key NAME . ", " . AGE . del ", " . del COLLEGE
let compsA : lens =
  "" | <dictionary "":compA> . (newline . <dictionary "":compA>)*
```

Listing 4.32: Lens definition in Boomerang

```
test compsA.get c = ?

Test result:
"Alice, 28
Bob, 21
Charlie, 21"

let a1 : string =
  Bob Lopes, 22
  Alice Lopes, 28
  Charlie Martins, 21

test compsC.put a1 into c = ?

Test result:
"Bob, 22, UCP
Alice, 28, FCT
Charlie, 21, FCT"
```

Listing 4.33: Unit testing in Boomerang

```
Test result:
"Bob, 22, FCT
Alice, 28, UCP
Charlie, 21, FCT"
```

Listing 4.34: Effects of no dictionary lens in Boomerang

The lens composition *get* function uses the `del` lens to indicate that the college is to be removed. Note that the concatenation operator '.'" as well as other operators automatically promote their arguments, following the sub-typing relationships: string <: regexp <: lens. The `key` is an annotation used to indicate which part of the line chuck is used for alignment, so each line is iterated in an orderly fashion. This is complemented using a *dictionary lens* in `compsA`. Without this dictionary, the alignment would be positional and we would not get the expected results when the record lines have different positional alignment. The *put* function updates the old concrete structure with the given age updated abstract structure and returns the updated concrete structure.

Listing 4.33 shows how Boomerang allows for unit testing. We use the concrete structure defined previously to test the get function and an updated abstract structure `a1` (with Bob's age incremented) to test the put function. As expected, the update to Bob's age is propagated to the concrete structure returned by the test. However, the updated abstract structure `a1` has the first two lines swapped in relation to the concrete structure, meaning that, the update of the concrete structure leads to an unwanted update without a *dictionary lens*. If a *dictionary lens* is not used in the update, the updated concrete structure will as described in listing 4.34. As the test described result shows, without a *dictionary lens*, the college name will be swapped as well, because we are matching in positional alignment and the first line is different in each structure, thus updating Bob's college with Alice's college and vice versa.

Chapter 5

Validation

In this chapter we provide a small benchmark of our implemented language against the Meteor framework. As presented in section 4.2, Meteor has a synchronization process which automatically propagates changes with the DDP [Metb] that allows for rapid prototyping and incremental development, closely resembling the reactive and incremental language that we extend in this thesis. Additionally, Meteor is the most popular [Hot] and the most code efficient framework out of the ones studied in section 4.2. The target applications of the benchmark are the MTAs, as they are the target applications of the work done in the context of this thesis. We host the developed example applications on Heroku [Her]. The benchmark metrics we use are code succinctness, and development costs. We do not, however, compare the performances of both frameworks since the adopted prototype is visibly slower than Meteor when evaluating and providing an HTML page after any update. This visible performance discrepancy is due how the prototype handles HTML values, which are fully re-evaluated after each change as opposed to only re-evaluating the relevant parts. The low performance of the prototype is addressed as future work in the next chapter since it is a major factor in the viability and usability of any web framework.

First we will address the differences of authentication in a well established framework like Meteor and our language. Then, we look at a simple wall application where any user can post a message anonymously, and transform it into a multi-group wall application thus comparing the efforts required by both frameworks. Next, we take a closer look at the TodoMVC [Tod] multi-group application developed in both frameworks. Finally, we end the chapter with some conclusions.

5.1 Authentication

In this section, we focus on the development costs of both frameworks by analyzing code re-usability and maintenance with the example of authentication mechanisms in mind. Meteor, as a well established framework, has a package manager which provides around 590 packages [Atmb] with different implemented and ready to use authentication mechanisms. In section 2.2 we defined a working simple authentication mechanism in our language, which is later used in section 2.3.3 to control access to an authenticated user environment. The full simple authentication example code is listed in appendix B.1. Even thou our language allows the developer to build authentication from scratch as well, it lacks security aspects like cryptography to create more secure authentication mechanisms. Additionally, seeing as authentication mechanisms are at the core of most common web applications, it is crucial for a faster development that these mechanisms are easy to re-use and maintain.

Packaging and distributing smaller applications is essential for the faster development of applications. Our framework, being a prototype, still has no way of packaging and distributing applications, such as the one we developed in section 2.2. However, re-using previously written code for a new application in our framework can be simple due to its ability to receive large chunks of code. Utilizing this simple input of code is a good basis for a distribution method in the implementation of a future package manager for the current prototype. With a package manager and more security aspects implemented into the language, it is then possible to build cryptography modules and authentication mechanisms, package them and distribute through the package manager.

5.2 Simple Wall Application

The benchmark in this section is focused on understanding the cost of developing an application and then transforming it into a multi-group application. In listing 5.1 we define the wall application [1] in our language, where any user can post anonymously. In appendix B.2 we define the same application [2] in Meteor. The code is visibly a little more succinct in our prototype than in Meteor due to its file based nature and explicit division of client and server code.

Next, we want to transform the wall application into a multi-group application where users access a group wall and post anonymously. In our language, we need only to wrap the previously defined application with an indexed module. Listing 5.2 transforms the wall application in listing 5.1 into a multi-group wall application [3]. In Meteor, to develop the same transformation we require the definition of routes, a new collection for the groups, and some other template helpers. Additionally, in Meteor collections are

[1] A working example can be found at `http://live-programming.herokuapp.com/app/Br1ti/page`

[2] Working application at `https://meteor-wall.herokuapp.com/`

[3] Working example at `http://live-programming.herokuapp.com/app/aNRZd/Group/page/Family` for a family group wall. Change group by changing the last URL path (Family).

```
var posts = [{
  id: 0,
  msg: "Welcome to the Wall App!"
}]

def size = foreach (post in posts with y = 0) y+1

def post t = action {
  insert {id: size, msg: t} into posts
}
def deletePost id = action {
  delete post in posts where post.id == id
}

def page =
  <div class="container">
    <header>
      <h1>"Wall"</h1>
      <input type="text" name="text" placeholder="Post something"
             onenter=(post)/>
    </header>
    <ul>
    (map (p in posts)
      <li>
        <button class="delete" doaction=(deletePost p.id)>"X"</button>
        <span class="text">(p.msg)</span>
      </li>)
    </ul>
  </div>
```

Listing 5.1: Multi-user wall application in our language

```
module Group<string group> {
  // listing 5.1 application code
}
```

Listing 5.2: Multi-user wall application in our language

available on both the client and the server, thus we need to specify what information is published to the client. Appendix B.3 transforms the example wall application in appendix B.2 into a multi-group wall application [4]. The effort, as well as code written, to make the multi-tenant transformation is evidently greater with Meteor. Furthermore, since the security rules and routes are all written by hand, there are no guarantees that they are error free. Testing, as well as writing security rules and routes, require a greater development effort throughout the evolution of an application. The transformation of an application to an MTA is visibly more simple and less prone to error in our language.

[4] Working application at `https://wall-groups.herokuapp.com/group/Family` for a family group wall. Change group by changing the last URL path (Family)

5.3 TodoMVC Multi-Group

In modern web frameworks, as an application adds layers of multi-tenant traits, the effort and possible errors in development grows. The extended TodoMVC application [5] we implemented in chapter 3 involves most of the multi-tenant traits described. The code for our non styled and simplified TodoMVC for groups of users can be found in appendix B.4. The code of our styled[6] and complete TodoMVC application [7] for groups of users can be found at appendix B.6. We built the same application in Meteor [8]. The code for the Meteor version can be found in appendix B.5.

Writing the application in Meteor requires slightly more written code, and a more careful plan to segment group information without any errors. The simplest phase in the development of the application in Meteor is the use of the package manager to introduce a packaged authentication mechanism. In our language, we also re-use a previously defined authentication mechanism but it is not packaged and distributed automatically, requiring a manual copy of the code to the new application. Although it is quite simple to manually copy an application in our framework, it is important for the viability of any web framework to package and distribute applications.

Manually writing routes is a careful process as subscribing to required information needs to happen before the route delivers the desired result, or else some information might not be available on the client side when a piece of code requests data, which results in errors. In our language, data is segmented naturally and automatically with each module definition, and each segmented data has a route of its own provided through a REST interface. The automatic segmentation of data is effortless and guarantees the safe development of applications, as opposed to the possibility of errors in the manual segmentation of data in Meteor and other similar modern web frameworks.

Even thou there is a package for Meteor to associate roles to authenticated users from the already used authentication package, the example only required one role for one action (delete to-dos) which proved simpler to just define a condition through a helper function. Additionally, the package only creates roles for authenticated users, which leaves the creation and management of any other types of roles in any other application to be manual. In our language, creating and managing roles is equally simple but also guarantees a sound application without broken security rules. Further more roles are associated with defined nested modules which can represent anything, not just users.

[5] Working example in the work space at http://live-programming.herokuapp.com/dev/BluQ1, with the authentication page at http://live-programming.herokuapp.com/app/BluQ1/Public/page

[6] The TodoMVC CSS can be found at.https://github.com/tastejs/todomvc/blob/gh-pages/examples/backbone/node_modules/todomvc-app-css/index.css

[7] Working example at http://live-programming.herokuapp.com/dev/lCBCd, with the main login page at http://live-programming.herokuapp.com/app/lCBCd/Public/page

[8] Working example at https://todo-groups.herokuapp.com/ using the same users defined in appendix A.1

5.4 Conclusions

Meteor produces very efficient applications performance wise, and although our proposed language can very rapidly transform applications into MTAs, its performance is noticeably slower due to many factors, some of which are discussed and listed as future work in chapter 6. However, Meteor code is also written in JS, an untyped language that gives no guarantees of a sound application, which can increase the costs in the development phase when errors occur, moreover, in production there is always the possibility of errors occurring that were not caught in development. In contrast, the core language in this thesis is typed and is as code succinct as Meteor. Since our approach maps the introduced mechanisms to core language operations, the typing system of the language continues to provide the safe development of applications with guarantees that applications are sound. Additionally, there is an increased cost in deployment for applications developed in modern frameworks, which in our framework is cost free with an automatic deployment, with changes to an application being applied on the running application as soon as they are verified.

Chapter 6

Final Remarks

The work done in the context of this thesis resulted in a reactive and incremental language targeted at multi-user web applications, a runtime support system, and live development environment. Because most of the introduced mechanisms are mapped directly to core language constructs, the existing type system guarantees a safe development of sound applications. Additionally, for the same reason, the reactive and incremental nature of the core language carries over to the introduced mechanisms.

The language provides operations to define the three layers of a web application in different shared and isolated module environments: data scheme, business logic, and views. The runtime support system provides client identification allowing the developer to create authentication mechanisms with the language. The developer builds an application using an incremental approach in a live development environment, where each modification is statically verified and deployed into the running application without restarting or stopping the application. The reactive nature that allows for the immediate and continuous feedback in development is also present in every running web application, without the explicit effort of the developer.

We believe the introduced client identification mechanism provides the foundation on which authentication mechanisms can be built. We demonstrated this by developing our own user authentication from which an application can be built. We also believe that modules and introduced module mechanisms allow faster definition of sound web applications with multi-tenant traits, validated by our version of the TodoMVC [Tod] application[1] that provides the same application to multiple groups (tenants). We take the single user application provided by the TodoMVC project and apply a modular approach with our introduced abstractions, effectively indexing the same TodoMVC application

[1] Working example at `http://live-programming.herokuapp.com/dev/lCBCd`, with the main login page at `http://live-programming.herokuapp.com/app/lCBCd/Public/page`

with a different state for each group. The application provides each group with a different To-do list, and each user with a set of accessible groups. With our boilerplate authentication, we provide user authentication to the application and allow some users to have some extra actions over each group. The application shows that in a similar fashion a registration process for users can be added to the application, and users can also be granted the ability to create groups and manage them.

The system is not yet complete and ready for production applications due to the lack of some features which are described as future work in the next section. The overall performance of the system was improved during the development phase of the extended language, but there are still some areas in which performance should be revisited. Currently, the two main impairments of performance in the system are the lack of caching in lenses, and full re-evaluation of HTML pages when a single dependency changes. However, we believe that the introduced module mechanisms to the project provide a faster approach in developing sound web applications, and that more research in this area could decrease the costs in developing, verifying, and maintaining web applications.

6.1 Future Work

The next paragraphs present some of the possible features and improvements that can be added to the system in order to increase its performance and usability.

Filtered Lens Caching The current implementation of filtered lenses executes the same iterations for every `get` operation, most of the time unnecessarily because no dependency has changed. A simple cache system can be easily implemented to improve performance.

HTML Re-Evaluation When a dependency of an HTML definition changes, the page is fully re-evaluated to compare with the current value and find changes. This means that performance decreases as a page grows, which could be avoided by only re-evaluating the relevant parts of the page.

JS/CSS Associated with an HTML page The JS and CSS currently is associated with a workspace, rather than having a JS and CSS for each page to avoid collision of CSS selectors and JS event handlers.

Operations Queue In the current implementation of the language [Mat15], the operations are run sequentially. This obviously affects performance when there are multiple users using the framework. The operations queue should be separate from the interpreter to allow the system to schedule operations in the best way possible, and possibly executing different operations in parallel. Such a queue is described in the paper that introduces the core language [DS15].

Application Clones Currently verified changes to an application are deployed in the running application immediately. However, sometimes in development we want to try implemented features before making them available to users. To do achieve this, the framework could provide with a cloning feature, in which applications are cloned as if they were Git branches, and later the modifications can be merged with the running application or discarded. Cloning offers some challenges with merging data used when testing implemented features in the cloned application.

Improve Queries The available queries when filtering tables is limited to simple conditions with binary operators, but no ordering, grouping, aggregation or projections. Additionally, these queries need to be optimized in order to not have greater impact on the overall performance of an application.

Improved IDE The re-invented IDE can still be improved in multiple areas. Indexed values were originally planned to be displayed as records with easy to use lookup features. Imported names were also originally planned to show the cached values, but because the lens caching system wasn't implemented as described in section 6.1, the cached values are not displayed in the IDE. Displaying the graph of dependencies on demand for each name would also be a good development, because in a growing application it becomes harder to keep track of the already existing code. The language editor can also be changed to a full fledged text editor, with syntax highlighting, like the JS and CSS editors.

Garbage Collection The hoisting technique described in chapter 2 of previous framework thesis [Mat15] is used to store scopes in separate names. The system currently doesn't clean unused hoisted names, which in a long development process of a growing application can lead to performance issues in wasted disk space with unnecessary names being persistently stored.

Package Manager A common feature with modern frameworks is package managers. For example, AtmosphereJS[2] for Meteor or RubyGems[3] for Ruby. A central registry could be added to the system, in order to allow developers to publish new modular applications that could then be used in any application to speed-up the development process. A good example of such a modular application, is the authentication application we defined in sections 2.2 and 2.3.3, which could provide developers with an authentication package for their applications.

Security The current authentication mechanisms that can be built with the prototype lack security layers such as cryptography which are a fundamental layer for more secure authentication mechanisms.

[2] https://atmospherejs.com/
[3] https://rubygems.org/

Module Hooks The possibility of defining hooks for module access (similar to route hooks in modern web frameworks) in order to trigger certain actions can extend the number of applications that can be expressed in the framework. Additionally, the current language does not provide many event hooks for HTML elements when defining pages, thus a more generic event hook management for HTML pages could improve the expressiveness of the language.

Bibliography

[Bez+10] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t. Hart. "Enabling Multi-tenancy: An Industrial Experience Report". In: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–8. ISBN: 978-1-4244-8630-4. DOI: 10.1109/ICSM.2010.5609735. URL: http://dx.doi.org/10.1109/ICSM.2010.5609735.

[Boh+06] A. Bohannon, B. C. Pierce, and J. A. Vaughan. "Relational lenses: a language for updatable views". In: *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*. 2006, pp. 338–347. DOI: 10.1145/1142351.1142399. URL: http://doi.acm.org/10.1145/1142351.1142399.

[Boh+08] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. "Boomerang: resourceful lenses for string data". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 2008, pp. 407–419. DOI: 10.1145/1328438.1328487. URL: http://doi.acm.org/10.1145/1328438.1328487.

[CC12] N. Chapman and J. Chapman. *Authentication and Authorization on the Web (Web Security Topics)*. 2012. ISBN: 978-0956737052.

[DS15] M. Domingues and J. C. Seco. "Type Safe Evolution of Live Systems". In: *Workshop on Reactive and Event-based Languages Systems (REBLS 15)* (2015).

[Gro+08] D. M. Groenewegen, Z. Hemel, L. C. Kats, and E. Visser. "WebDSL: A Domain-specific Language for Dynamic Web Applications". In: *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA Companion '08. Nashville, TN, USA: ACM, 2008, pp. 779–780. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449858. URL: http://doi.acm.org/10.1145/1449814.1449858.

[Hof+15] M. Hofmann, B. C. Pierce, and D. Wagner. "Symmetric Lenses". In: *Journal of the ACM* (2015). To appear; extended abstract in POPL 2011.

[Kri01] D. M. Kristol. "HTTP Cookies: Standards, Privacy, and Politics". In: *ACM Trans. Internet Technol.* 1.2 (Nov. 2001), pp. 151–198. ISSN: 1533-5399. DOI: 10.1145/502152.502153. URL: http://doi.acm.org/10.1145/502152.502153.

[Mat15] J. P. C. Mateus. "Runtime Support System for an Incremental and Reactive Web Programming Language". MA thesis. Faculdade de Ciências e Tecnologia - UNL, Dec. 2015.

[MK11] C. Momm and R. Krebs. "A Qualitative Discussion of Different Approaches for Implementing Multi-Tenant SaaS Offerings". In: *Software Engineering 2011 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-25.02.2011, Karlsruhe*. 2011, pp. 139–150. URL: http://subs.emis.de/LNI/Proceedings/Proceedings184/article6330.html.

[Rod+12] J. Rodrigues, A. Leite, J. C. Damasceno, V. C. Garcia, P. Silveira, and S. R. L. Meira. "An Approach to Developing Multi-tenancy SaaS Using Metaprogramming". In: *Proceedings of the 18th Brazilian Symposium on Multimedia and the Web*. WebMedia '12. São Paulo/SP, Brazil: ACM, 2012, pp. 207–210. ISBN: 978-1-4503-1706-1. DOI: 10.1145/2382636.2382681. URL: http://doi.acm.org/10.1145/2382636.2382681.

[SR11] B. Sengupta and A. Roychoudhury. "Engineering Multi-tenant Software-as-a-service Systems". In: *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems*. PESOS '11. New York, NY, USA: ACM, 2011, pp. 15–21. ISBN: 978-1-4503-0591-4. DOI: 10.1145/1985394.1985397. URL: http://doi.acm.org/10.1145/1985394.1985397.

[Ste15] A. Steckermeier. "Lenses in Functional Programming". July 2015. URL: https://www21.in.tum.de/teaching/fp/SS15/papers/17.pdf.

[Tsa+14] W. Tsai, X. Bai, and Y. Huang. "Software-as-a-service (SaaS): perspectives and challenges". In: *SCIENCE CHINA Information Sciences* 57.5 (2014), pp. 1–15. DOI: 10.1007/s11432-013-5050-z. URL: http://dx.doi.org/10.1007/s11432-013-5050-z.

[Vel+10] T. Velte, A. Velte, and R. Elsenpeter. *Cloud Computing, A Practical Approach*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN: 0071626948, 9780071626941.

[You+11] A. J. Younge, R. Henschel, J. T. Brown, G. von Laszewski, J. Qiu, and G. C. Fox. "Analysis of Virtualization Technologies for High Performance Computing Environments". In: *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*. CLOUD '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 9–16. ISBN: 978-0-7695-4460-1. DOI: 10.1109/CLOUD.2011.29. URL: http://dx.doi.org/10.1109/CLOUD.2011.29.

Webography

[Ace] *Ace Editor*. URL: https://ace.c9.io/ (visited on 03/22/2017).

[Ama] *Amazon Web Services*. URL: https://aws.amazon.com/ (visited on 03/22/2017).

[Ath] *Athena Framework*. URL: http://athenasource.org/ (visited on 03/22/2017).

[Aut] *Authentication*. 2016. URL: https://en.wikipedia.org/wiki/Authentication (visited on 03/22/2017).

[Meta] *Build Apps with JavaScript*. URL: http://meteor.com (visited on 03/22/2017).

[Her] *Cloud Application Platform | Heroku*. URL: https://www.heroku.com/ (visited on 03/22/2017).

[Dle] *Distributed Lens Architecture*. URL: http://matrix.ai/2014/12/05/distributed-lens-architecture/ (visited on 03/22/2017).

[Has] *Haskell Language*. URL: https://www.haskell.org/ (visited on 03/22/2017).

[Rfc] *Hypertext Transfer Protocol – HTTP/1.1*. URL: http://www.w3.org/Protocols/rfc2616/rfc2616.html (visited on 03/22/2017).

[Htta] *Interface HttpServletRequest*. URL: https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletRequest.html (visited on 03/22/2017).

[Httb] *Interface HttpSession*. URL: http://docs.oracle.com/javaee/5/api/javax/servlet/http/HttpSession.html (visited on 03/22/2017).

[Bro] *Introducing BrowserID*. URL: http://identity.mozilla.com/post/7616727542/introducing-browserid-a-better-way-to-sign-in (visited on 01/22/2016).

[Kop11] A. Koprowski. *OPA Language Documentation*. 2011. URL: http://blog.opalang.org/2011/09/sessions-handling-state-communication.html (visited on 03/22/2017).

[Metb] *Meteor Distributed Data Protocol*. Jan. 2016. URL: http://www.meteor.com/ddp (visited on 03/22/2017).

[Metc] *Meteor Documentation*. Jan. 2016. URL: http://docs.meteor.com (visited on 03/22/2017).

[Atma] *Meteorite Account Packages*. URL: https://atmospherejs.com/ (visited on 03/22/2017).

[Atmb] *Meteorite Account Packages*. URL: https://atmospherejs.com/mrt?q=accounts (visited on 03/22/2017).

[Cmo] *Modular Programming in C*. URL: http://www.icosaedro.it/c-modules.html.

[Oca] *Modules - OCaml*. URL: https://ocaml.org/learn/tutorials/modules.html.

[Mon] *MongoDB*. URL: https://www.mongodb.com/.

[Nod] *Node.js*. URL: https://nodejs.org/.

[Oau] *OAuth Community*. URL: https://oauth.net/ (visited on 01/22/2016).

[Opaa] *OPA Language Documentation*. Jan. 2016. URL: http://github.com/MLstate/opalang/wiki/A-tour-of-Opa (visited on 03/22/2017).

[Ope] *OpenID Foundation*. URL: http://openid.net/ (visited on 01/22/2016).

[Xep] *Publish-Subscribe*. URL: http://xmpp.org/extensions/xep-0060.html (visited on 03/22/2017).

[Ses] *Session (Computer Science)*. URL: https://en.wikipedia.org/wiki/Session_(computer_science) (visited on 03/22/2017).

[Usi] *Session ID*. URL: https://en.wikipedia.org/wiki/Session_ID (visited on 03/22/2017).

[Opab] *Slicing*. Feb. 2016. URL: https://github.com/MLstate/opalang/wiki/Developing-for-the-web#slicing (visited on 03/22/2017).

[Sql] *SQLite*. URL: https://www.sqlite.org/ (visited on 03/22/2017).

[Tod] *TodoMVC Project*. URL: https://todomvc.com (visited on 03/22/2017).

[Hot] *Web Frameworks Ranking*. URL: http://hotframeworks.com/#rankings (visited on 03/22/2017).

[Weba] *WebDSL Manual*. Jan. 2016. URL: http://webdsl.org/page/Manual (visited on 03/22/2017).

[Webb] *WebSocket*. URL: https://en.wikipedia.org/wiki/WebSocket (visited on 03/22/2017).

[Yes] *Yesod Web Framework Book- Version 1.4*. 2016. URL: http://www.yesodweb.com/book (visited on 03/22/2017).

Appendix A

Seed Data for the To-do Application

A.1 Users Seed Data

```
var users = [{
  name: "Alice",
  password: "alpha"
}, {
  name: "Bob",
  password: "bravo"
}, {
  name: "Carol",
  password: "charlie"
}, {
  name: "David",
  password: "delta"
}, {
  name: "Eve",
  password: "echo"
}]
```

A.2 Groups Seed Data

```
var groups = [{
  name: "Family",
  users: [{ name: "Alice", admin: true },
          { name: "Bob", admin: true },
          { name: "David", admin: false },
          { name: "Eve", admin: false }]
}, {
  name: "School",
  users: [{ name: "Carol", admin: true },
          { name: "Alice", admin: true },
          { name: "David", admin: false },
          { name: "Eve", admin: true },
          { name: "Bob", admin: false }]
}, {
  name: "Gym",
  users: [{ name: "Bob", admin: false },
          { name: "David", admin: true },
          { name: "Carol", admin: true }]
}]
```

Appendix B

Code for Developed Applications

B.1 Simple Authentication

```
1  // Data structures in appendices A.1 and A.2
2
3  do action { delete a in authenticatedUsers where true }
4
5  def authenticate name password =
6    match
7      get user in users
8      where user.name == name and user.password == password
9    with
10     user::rest =>
11       action {
12         insert {
13           name: name,
14           token: usid
15         } into authenticatedUsers
16       }
17     | [] => action {}
18
19 def logout = action {
20     delete user in authenticatedUsers
21     where user.token == usid
22   }
23
24 def authenticated id =
25   match
26     get user in authenticatedUsers
27     where user.token == id
28   with
29     user::rest => true
```

```
30      | [] => false
31
32  module Public<usid> {
33    import authenticate
34    import authenticated
35    import logout
36
37    from user in authenticatedUsers where user.token == usid
38      import first user.name as currentUser
39      default {name:"",token:""}
40
41    def page =
42      <div>
43       (if not authenticated usid then
44         <div>
45           <div>
46             <input type="text" placeholder="username" id="name"/>
47             <input type="password" placeholder="password" id="password"/>
48           </div>
49           <button doaction=(authenticate #name #password)>"Log In"</button>
50         </div>
51       else
52          <div>
53            <h1>("Welcome " ++ currentUser)</h1>
54            <button doaction=(logout)>"Logout"</button>
55          </div>)
56      </div>
57  }
```

B.2 Simple Wall Application in Meteor

```
1  export const Posts = new Mongo.Collection('posts');
```

Listing B.1: posts.js

```
1  import { Meteor } from 'meteor/meteor';
2
3  import { Posts } from '../posts.js'
4
5  Meteor.startup(() => {
6    // code to run on server at startup
7  });
```

Listing B.2: server/main.js

```
1  <template name="post">
2    <li>
3      <button class="delete">&times;</button>
4      <span class="text">{{text}}</span>
5    </li>
```

```
</template>
```

Listing B.3: client/post.html

```
import { Template } from 'meteor/templating';
import { Posts } from '../posts.js';
import './post.html';

Template.post.events({
  'click .delete'() {
    Posts.remove(this._id);
  },
});
```

Listing B.4: client/post.js

```
<body>
  <div class="container">
    <header>
      <h1>Wall</h1>
      <form class="new-post">
        <input type="text" name="text" placeholder="Post something" />
      </form>
    </header>
    <ul>
      {{#each posts}}
        {{> post}}
      {{/each}}
    </ul>
  </div>
</body>
```

Listing B.5: client/body.html

```
import { Meteor } from 'meteor/meteor';
import { Template } from 'meteor/templating';
import { Posts } from '../posts.js';
import './body.html';

Template.body.helpers({
  posts() {
    return Posts.find({});
  },
});

Template.body.events({
  'submit .new-post'(event) {
    // Prevent default browser form submit
    event.preventDefault();
    // Get value from form element
    const target = event.target;
```

```
18      const text = target.text.value;
19      // Insert a task into the collection
20      Posts.insert({
21        text
22      });
23      // Clear form
24      target.text.value = '';
25    },
26  });
```

Listing B.6: client/main.js

B.3 Simple Groups Wall Application in Meteor

```
1  import { Meteor } from 'meteor/meteor';
2
3  export var Groups = new Mongo.Collection('groups');
4  export var Posts = new Mongo.Collection('posts');
5
6  Router.route('/group/:name', {
7    action: function () {
8      var group = Groups.findOne({ name: this.params.name });
9      var posts = Posts.find({ groupId: group._id });
10     this.render('Group', {
11       data: function() {
12         return {
13           group: group,
14           posts: posts
15         };
16       }
17     });
18   },
19   waitOn: function() {
20     Meteor.subscribe('groups', this.params.name);
21     return Meteor.subscribe('posts', this.params.name);
22   }
23 });
```

Listing B.7: groups.js

```
1  import { Template } from 'meteor/templating';
2  import { Groups, Posts } from '../groups.js';
3  import './main.html';
4
5  Template.Group.events({
6    'keypress input'(event, instance) {
7      var group = this.group;
8      if (event.which === 13) {
9        Posts.insert({ groupId: group._id, text: event.target.value });
10       event.target.value = '';
11     }
```

```
  }
});

Template.Post.events({
  'click .delete'(event, instance) {
    Posts.remove({ _id: this._id });
  }
});
```

Listing B.8: client/main.js

```
import { Meteor } from 'meteor/meteor';
import { Groups, Posts } from '../groups.js';

Meteor.publish('groups', function(name) {
  return Groups.find({ name: name });
});

Meteor.publish('posts', function(name) {
  var group = Groups.findOne({ name: name });
  if (!group) {
    var groupId = Groups.insert({
      name: name
    });
    return Posts.find({ groupId: groupId });
  } else {
    return Posts.find({ groupId: group._id });
  }
});

Meteor.startup(() => {});
```

Listing B.9: server/main.js

```
<template name="Group">
  <body>
    <div class="container">
      <header>
        <h1>{{group.name}} Wall</h1>
        <input type="text" data-group={{group.name}} name="post" placeholder="Post something" /
      </header>
      <ul>
        {{#each posts}}
          {{> Post}}
        {{/each}}
      </ul>
    </div>
  </body>
</template>
```

Listing B.10: client/main.html

```
<template name="Post">
  <li data-id={{_id}}>
    <button class="delete">&times;</button>
    <span class="text">{{text}}</span>
  </li>
</template>
```

Listing B.11: client/post.html

B.4 Simplified TodoMVC

```
// Data structures in appendices A.1 and A.2

var authenticatedUsers = [{
  name: "Dummy",
  token: "Dummy"
}]

do action { delete a in authenticatedUsers where true }

def authenticate name password =
  match
    get user in users
    where user.name == name and user.password == password
  with
    user::rest =>
      action {
        insert {
          name: name,
          token: usid
        } into authenticatedUsers
      }
    | [] => action {}

def logout = action {
    delete user in authenticatedUsers
    where user.token == usid
  }

def authenticated id =
  match
    get user in authenticatedUsers
    where user.token == id
  with
    user::rest => true
    | [] => false

module Public<usid> {
  import authenticate
  import authenticated
```

```
40    import logout
41
42    from user in authenticatedUsers where user.token == usid
43      import first user.name as currentUser
44      default {name:"",token:""}
45
46    def page =
47      <div>
48        (if not authenticated usid then
49          <div>
50            <div>
51              <input type="text" placeholder="username" id="name"/>
52              <input type="password" placeholder="password" id="password"/>
53            </div>
54            <button doaction=(authenticate #name #password)>"Log In"</button>
55          </div>
56        else
57          <div>
58            <h1>("Welcome " ++ currentUser)</h1>
59            <a href=(workspace_path ++ "User/page")>"Go to your page!"</a>
60            <button doaction=(logout)>"Logout"</button>
61          </div>)
62      </div>
63  }
64
65  def listContains list name =
66    match
67      get item in list
68      where item.name == name
69    with
70      u::us => true
71    | [] => false
72
73  module User<usid> when(authenticated usid) {
74    import logout
75    import listContains
76    from user in authenticatedUsers
77      where user.token == usid
78      import first user.name as username
79      default {name:"",token:""}
80    from group in groups
81      where (listContains group.users username)
82      import group as groups
83
84    def page =
85      <div>
86        <h1>("User: " ++ username)</h1>
87        <button doaction=(logout)
88                data-redirect=(workspace_path ++ "Public/page")>
89          "Log out"
```

```
90        </button>
91        <h2>"Groups"</h2>
92        <ul>
93          (map (group in groups)
94            <li>
95              <a href=(workspace_path ++ "Group/Member/page/" ++ group.name)>
96                (group.name)
97              </a>
98            </li>
99          )
100       </ul>
101     </div>
102 }
103 module Group<string groupName, *usid>
104   with User(usid)
105   when(listContains groups@User groupName)
106 {
107   from group in groups
108     where group.name == groupName
109     import first group.users as members
110     default {name:groupName,users:[]}
111
112   var todos = [{
113     id: 0,
114     text: "Welcome to group " ++ groupName,
115     done: false
116   }]
117
118   def isAdmin name =
119     match
120       get member in members
121       where member.name == name
122     with
123       m::ms => m.admin
124     | [] => false
125
126   module Member<usid> with User(usid) {
127     import todos
128     import isAdmin
129
130     module Admin<string *groupName, *usid> when(isAdmin username@User) {
131       import todos
132
133       def deleteTodo id = action {
134         delete todo in todos
135         where todo.id == id
136       }
137     }
138
139     def size = foreach(todo in todos with y = 0) y+1
```

```
140
141     def addTodo text = action {
142         insert { id: size, done: false, text: text }
143         into todos
144       }
145
146     def toggleComplete id = action {
147         update todo in todos
148         with { id: todo.id, done: not todo.done, text: todo.text }
149         where todo.id == id
150       }
151
152     def todoItem todo =
153       <li>
154         <checkbox type="checkbox" value=(todo.done)
155           docheck=(toggleComplete todo.id)
156           douncheck=(toggleComplete todo.id)
157         />
158         <label>(todo.text)</label>
159         (in Admin(groupName, usid) then
160           <button doaction=(deleteTodo todo.id)>"Delete"</button>
161          else <span></span>
162         )
163       </li>
164
165     def page =
166       <div>
167         <header>
168           <h1>groupName</h1>
169           <input placeholder="What needs to be done?" onenter=(addTodo) />
170         </header>
171         <section>
172           <ul>
173             (map (todo in todos) todoItem todo)
174           </ul>
175         </section>
176         <footer>
177           <p>("Logged as: " ++ username@User)</p>
178           <button doaction=(logout@User)
179                   data-redirect=(workspace_path ++ "Public/page")>
180             "Log out"
181           </button>
182         </footer>
183       </div>
184   }
185 }
```

B.5 TodoMVC for Groups in Meteor

```
1 import { Meteor } from 'meteor/meteor';
```

```
2
3  export var Groups = new Mongo.Collection('groups');
4  export var Todos = new Mongo.Collection('todos');
5
6  Todos.allow({
7    insert(userId, doc) {
8      var group = Groups.find({
9        _id: doc.groupId,
10       users: {
11         $elemMatch: {
12           username: Meteor.users.findOne({ _id: userId }).username
13         }
14       }
15     });
16     return userId && group.count();
17   },
18   remove(userId, doc) {
19     var group = Groups.find({
20       _id: doc.groupId,
21       users: {
22         $elemMatch: {
23           username: Meteor.users.findOne({ _id: userId }).username
24         }
25       }
26     }, {
27         fields: {
28           "users.$": 1
29         }
30       }
31     );
32     return userId && group.count() && group.fetch()[0].users[0];
33   },
34   update(userId, doc, fields, modifier) {
35     var group = Groups.find({
36       _id: doc.groupId,
37       users: {
38         $elemMatch: {
39           username: Meteor.users.findOne({ _id: userId }).username
40         }
41       }
42     });
43
44     return userId && group.count();
45   }
46 });
```

Listing B.12: groups.js

```
1  import { Meteor } from 'meteor/meteor';
2  import { Groups, Todos } from './groups.js';
3
```

```
4  Router.route('/group/:name', {
5    action: function() {
6      this.render('Group', {
7        data: function() {
8          return {
9            name: this.params.name
10         };
11       }
12     });
13   },
14   waitOn: function() {
15     if (Meteor.userId()) {
16       Meteor.subscribe('todos', this.params.name);
17       return Meteor.subscribe('groups', Meteor.userId());
18     }
19   },
20   onBeforeAction: function() {
21     if (!(Meteor.userId() || Meteor.loggingIn())) {
22       Router.go('home.show');
23     } else {
24       var groups = Groups.find({
25         name: this.params.name,
26         users: {
27           $elemMatch: {
28             username: Meteor.users.findOne(Meteor.userId()).username
29           }
30         }
31       });
32       if (groups.count()) {
33         this.next();
34       } else {
35         Router.go('user.show');
36       }
37     }
38   },
39   name: 'group.show'
40 });
41
42 Router.route('/', {
43   action: function() {
44     this.render("Home");
45   },
46   onBeforeAction: function() {
47     if (Meteor.userId()) {
48       Router.go('user.show');
49     } else {
50       this.next();
51     }
52   },
53   name: 'home.show'
```

```
54 });
55
56 Router.route('/user', {
57   action: function() {
58     this.render("User");
59   },
60   onBeforeAction: function() {
61     if (!(Meteor.userId() || Meteor.loggingIn())) {
62       Router.go('home.show');
63     } else {
64       this.next();
65     }
66   },
67   waitOn: function() {
68     if (Meteor.userId()) {
69       return Meteor.subscribe('groups', Meteor.userId());
70     }
71   },
72   name: 'user.show'
73 });
```

Listing B.13: routes.js

```
1  import { Template } from 'meteor/templating';
2  import { Session } from 'meteor/session';
3  import { Groups, Todos } from '../groups.js';
4  import './group.html';
5
6  Session.set('filter', 'all');
7
8  var filter = {
9      all: {},
10     active: {completed: false},
11     completed: {completed: true}
12 };
13
14 Template.Group.events({
15   'keypress .new-todo'(event, instance) {
16     var group = Groups.findOne({ name: this.name });
17     if (event.which === 13) {
18       Todos.insert({
19         groupId: group._id,
20         todo: event.target.value,
21         completed: false,
22         created_at: new Date().getTime()
23       });
24       event.target.value = "";
25     }
26   },
27   'click .clear-completed'(event, instance) {
28     Todos.find({ completed: true}).forEach(function(todo) {
```

```
29        Todos.remove(todo._id);
30      });
31
32    },
33    'click .toggle-all'(event, instance) {
34      var completed = true;
35      if (!Todos.find({ completed: false }).count()) {
36        completed = false;
37      }
38      Todos.find().forEach(function(todo) {
39        Todos.update(todo._id, { $set: { completed: completed } });
40      });
41    },
42    'click .filters > li > a'(event, instance) {
43      Session.set('filter', event.target.id);
44    }
45  });
46
47  Template.Group.helpers({
48    todos: function() {
49      return Todos.find(
50        filter[Session.get('filter')],
51        { sort: { created_at: 1 } }
52      );
53    },
54    size: function() {
55      return Todos.find().count();
56    },
57    leftTodo: function() {
58      return Todos.find({ completed: false }).count();
59    },
60    isAdmin: function() {
61      return Groups.find({ name: this.name }).fetch()[0].users[0].admin;
62    },
63    allComplete: function() {
64      return Todos.find().count() == Todos.find({ completed: true }).count();
65    },
66    singularTodo: function() {
67      return Todos.find({ completed: false }).count() == 1;
68    },
69    filters: function() {
70      return ['all', 'active', 'completed'];
71    },
72    clearCompleted: function() {
73      return Groups.find({ name: this.name }).fetch()[0].users[0].admin && Todos.find({ completed
74    },
75    filterSelected: function() {
76      return Session.equals('filter', this.valueOf()) ? "selected" : "";
77    }
78  });
```

```
79
80
81 Template.User.helpers({
82   groups: function() {
83     return Groups.find();
84   }
85 });
86
87 Template.Todo.events({
88   'click .destroy'(event, instance) {
89     Todos.remove(this._id);
90   },
91   'click .toggle'(event, instance) {
92     Todos.update(this._id, { $set: { completed: !this.completed } });
93   }
94 });
95
96 Template.Todo.helpers({
97   isAdmin: function() {
98     return Groups.find(this.groupId).fetch()[0].users[0].admin;
99   },
100  complete: function() {
101    return this.completed ? "completed": "";
102  }
103 });
```

Listing B.14: client/main.js

```
1  import { Meteor } from 'meteor/meteor';
2  import { Groups, Todos } from '../groups.js';
3
4  Meteor.publish('groups', function(userId) {
5    return Groups.find({
6      users: {
7        $elemMatch: {
8          username: Meteor.users.findOne({ _id: userId }).username
9        }
10     }
11   }, {
12       fields: {
13         name: 1,
14         "users.$": 1
15       }
16     }
17   );
18 });
19
20 Meteor.publish('todos', function(name) {
21   var group = Groups.findOne({ name: name });
22   return Todos.find({ groupId: group._id });
23 });
```

```
24
25 Meteor.startup(() => {});
```

Listing B.15: server/main.js

```
1  <head>
2    <title>Welcome!</title>
3  </head>
4  <template name="Home">
5    <body>
6      <div>
7        {{> loginButtons}}
8      </div>
9    </body>
10 </template>
```

Listing B.16: client/main.html

```
1  <template name="User">
2    <body>
3      <div>
4        <h1>{{> loginButtons}}</h1>
5        <h2>Groups</h2>
6        <ul>
7        {{#each groups}}
8          <li><a href="/group/{{name}}">{{name}}</a></li>
9        {{/each}}
10       </ul>
11     </div>
12   </body>
13 </template>
```

Listing B.17: client/user.html

```
1  <template name="Group">
2    <body>
3      <section class="todoapp">
4        <header class="header">
5          <h1>{{name}}</h1>
6          <input
7            autofocus=true
8            class="new-todo"
9            placeholder="What needs to be done?"
10         />
11       </header>
12       <section class="main">
13         {{#if size}}
14           {{#if allComplete}}
15             <input
16                   type="checkbox"
17                   checked="checked"
```

```
                    class="toggle-all"
                />
            {{else}}
                <input
                    type="checkbox"
                    class="toggle-all"
                />
            {{/if}}
        {{/if}}
        <label for="toggle-all">"Mark all as complete"</label>
          <ul class="todo-list">
            {{#each todos}}
              {{> Todo }}
            {{/each}}
          </ul>
    </section>
    {{#if size}}
      <footer class="footer">
        <span class="todo-count">
          <strong>{{leftTodo}}</strong>
          {{#if singularTodo}}
            item left
          {{else}}
            items left
          {{/if}}
        </span>
        <ul class="filters">
          {{#each filters}}
          <li><a id={{this}} class={{filterSelected this}}>{{this}}</a></li>
          {{/each}}
        </ul>
        {{#if clearCompleted}}
          <button class="clear-completed">Clear completed</button>
        {{/if}}
      </footer>
    {{/if}}
    </section>
  </body>
</template>
```

Listing B.18: client/group.html

```
<template name="Todo">
  <li class={{complete}}>
    <div class="view">
      {{#if completed}}
        <input class="toggle" type="checkbox" checked="checked" />
      {{else}}
        <input class="toggle" type="checkbox" />
      {{/if}}
      <label>{{todo}}</label>
```

```
10        {{#if isAdmin}}
11        <button class="destroy"></button>
12        {{/if}}
13      </div>
14    </li>
15  </template>
```

Listing B.19: client/todo.html

B.6 Full TodoMVC application for groups of users

```
1  // Data structures in appendices A.1 and A.2
2
3  var authenticatedUsers = [{
4    name: "Dummy",
5    token: "Dummy"
6  }]
7
8  do action { delete a in authenticatedUsers where true }
9
10 def authenticate name password =
11   match
12     get user in users
13     where user.name == name and user.password == password
14   with
15     user::rest =>
16       action {
17         insert {
18           name: name,
19           token: usid
20         } into authenticatedUsers
21       }
22     | [] => action {}
23
24 def logout = action {
25     delete user in authenticatedUsers
26     where user.token == usid
27   }
28
29 def authenticated id =
30   match
31     get user in authenticatedUsers
32     where user.token == id
33   with
34     user::rest => true
35     | [] => false
36
37 module Public<usid> {
38   import authenticate
39   import authenticated
```

```
40    import logout
41
42    from user in authenticatedUsers where user.token == usid
43      import first user.name as currentUser
44      default {name:"",token:""}
45
46    def page =
47      <div>
48        (if not authenticated usid then
49          <div>
50            <div>
51              <input type="text" placeholder="username" id="name"/>
52              <input type="password" placeholder="password" id="password"/>
53            </div>
54            <button doaction=(authenticate #name #password) id="log">"Log In"</button>
55          </div>
56        else
57          <div>
58            <h1>("Welcome " ++ currentUser)</h1>
59            <a href=(workspace_path ++ "User/page")>"Go to your page!"</a>
60            <button doaction=(logout) id="log">"Logout"</button>
61          </div>)
62      </div>
63  }
64
65  def listContains list name =
66    match
67      get item in list
68      where item.name == name
69    with
70      u::us => true
71    | [] => false
72
73  module User<usid> when(authenticated usid) {
74    import logout
75    import listContains
76    from user in authenticatedUsers
77      where user.token == usid
78      import first user.name as username
79      default {name:"",token:""}
80    from group in groups
81      where (listContains group.users username)
82      import group as groups
83
84    var filter = 0 // 0 -> "all" | 1 -> "active" | 2 -> "complete"
85    def allFilter = if filter == 0 then "selected" else ""
86    def activeFilter = if filter == 1 then "selected" else ""
87    def completedFilter = if filter == 2 then "selected" else ""
88
89    var changeFilter f = action { filter := f }
```

```
90
91    def page =
92      <div>
93        <h1>("User: " ++ username)</h1>
94        <button doaction=(logout) id="log"
95                data-redirect=(workspace_path ++ "Public/page")>
96          "Log out"
97        </button>
98        <h2>"Groups"</h2>
99        <ul>
100         (map (group in groups)
101           <li>
102             <a href=(workspace_path ++ "Group/Member/page/" ++ group.name)>
103               (group.name)
104             </a>
105           </li>
106         )
107       </ul>
108     </div>
109 }
110 module Group<string groupName, *usid>
111   with User(usid)
112   when(listContains groups@User groupName)
113 {
114   from group in groups
115     where group.name == groupName
116     import first group.users as members
117     default {name:groupName,users:[]}
118
119   var todos = [{
120     id: 0,
121     text: "Welcome to group " ++ groupName,
122     done: false
123   }]
124
125   def isAdmin name =
126     match
127       get member in members
128       where member.name == name
129     with
130       m::ms => m.admin
131     | [] => false
132
133   module Member<usid> with User(usid) {
134     import todos
135     import isAdmin
136
137     module Admin<string *groupName, *usid> when(isAdmin username@User) {
138       import todos
139
```

```
140       def deleteTodo id = action {
141         delete todo in todos
142         where todo.id == id
143       }
144       def clearCompleted =
145         action {
146           delete todo in todos
147           where todo.done
148         }
149     }
150
151     def size = foreach(todo in todos with y = 0) y+1
152     def activeTodos = get todo in todos where not todo.done
153     def completeTodos = get todo in todos where todo.done
154     def leftTodo = foreach(todo in activeTodos with y = 0) y + 1
155     def allComplete = leftTodo == 0
156
157     def toggleAllComplete =
158       action {
159         update todo in todos with {
160           id: todo.id,
161           done: not allComplete,
162           text: todo.text
163         } where true
164       }
165
166     def addTodo text = action {
167         insert { id: size, done: false, text: text }
168         into todos
169       }
170
171     def toggleComplete id = action {
172         update todo in todos
173         with { id: todo.id, done: not todo.done, text: todo.text }
174         where todo.id == id
175       }
176
177     def todoItem todo =
178       <li class=(if todo.done then "completed" else "")>
179         <div class="view">
180           <checkbox
181             class="toggle"
182             type="checkbox"
183             value=(todo.done)
184             docheck=(toggleComplete todo.id)
185             douncheck=(toggleComplete todo.id)
186           />
187           <label>(todo.text)</label>
188           (in Admin(groupName, usid) then
189             <button class="destroy" doaction=(deleteTodo todo.id) />
```

```
190           else
191             <div></div>)
192         </div>
193       </li>
194
195     def header =
196       <header class="header">
197         <h1>groupName</h1>
198         <input
199           autofocus=true
200           class="new-todo"
201           placeholder="What needs to be done?"
202           onenter=(addTodo)
203         />
204       </header>
205
206     def main =
207       <section class="main">
208         (if size > 0 then
209           <checkbox
210             value=(allComplete)
211             docheck=(toggleAllComplete)
212             douncheck=(toggleAllComplete)
213             class="toggle-all"
214             type="checkbox"
215           />
216         else <div></div>)
217         <label for="toggle-all">"Mark all as complete"</label>
218           <ul class="todo-list">
219             (if filter@User == 0 then
220               map (todo in todos)
221                 todoItem todo
222             else if filter@User == 1 then
223               map (todo in activeTodos)
224                 todoItem todo
225             else
226               map (todo in completeTodos)
227                 todoItem todo
228             )
229           </ul>
230       </section>
231
232     def footer =
233       (if size > 0 then
234         <footer class="footer">
235           <span class="todo-count">
236             <strong>
237               (leftTodo)
238             </strong>
239             (if (leftTodo == 1) then " item left" else " items left")
```

```
240           </span>
241           <ul class="filters">
242             <li>
243               <a class=(allFilter@User) doaction=(changeFilter@User 0)>
244               "All"
245               </a>
246             </li>
247             <span>" "</span>
248             <li>
249               <a class=(activeFilter@User) doaction=(changeFilter@User 1)>
250               "Active"
251               </a>
252             </li>
253             <span>" "</span>
254             <li>
255               <a class=(completedFilter@User) doaction=(changeFilter@User 2)>
256               "Completed"
257               </a>
258             </li>
259           </ul>
260             (if (leftTodo < size) then
261               in Admin(groupName, usid) then
262                 <button class="clear-completed"
263                   doaction=(clearCompleted)>
264                   "Clear completed"
265                 </button>
266               else <div></div>
267             else <div></div>)
268       </footer>
269     else <div></div>)
270
271   def page =
272     <div id="container">
273       <section class="todoapp">
274         header
275         main
276         footer
277       </section>
278       <footer class="info">
279         <p>("Logged as: " ++ username@User)</p>
280         <button doaction=(logout@User) id="log"
281                 data-redirect=(workspace_path ++ "Public/page")>
282           "Log out"
283         </button>
284         <p>"Created by "
285           <a href="https://bitbucket.com/tdlopes">"Tiago Lopes"</a>
286         </p>
287         <p>"To be part of "<a href="http://todomvc.com">"TodoMVC"</a></p>
288       </footer>
289     </div>
```

```
290     }
291 }
```