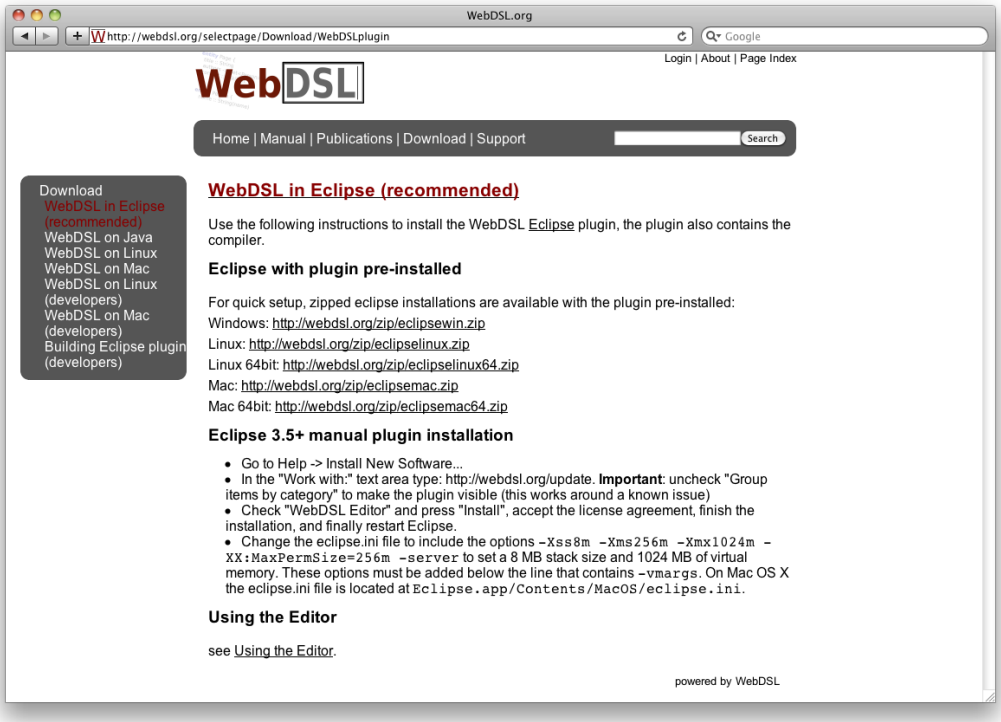
Weaving Web Applications with WebDSL Tutorial

Danny M. Groenewegen, Eelco Visser

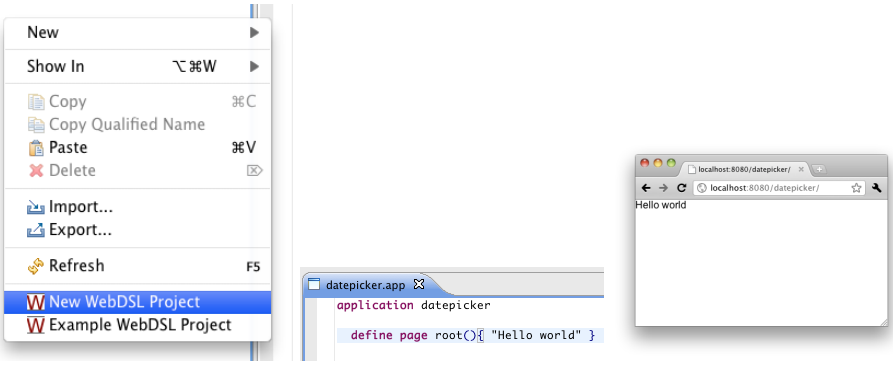
Software Engineering Research Group, Delft University of Technology, The Netherlands
d.m.groenewegen@tudelft.nl, visser@acm.org

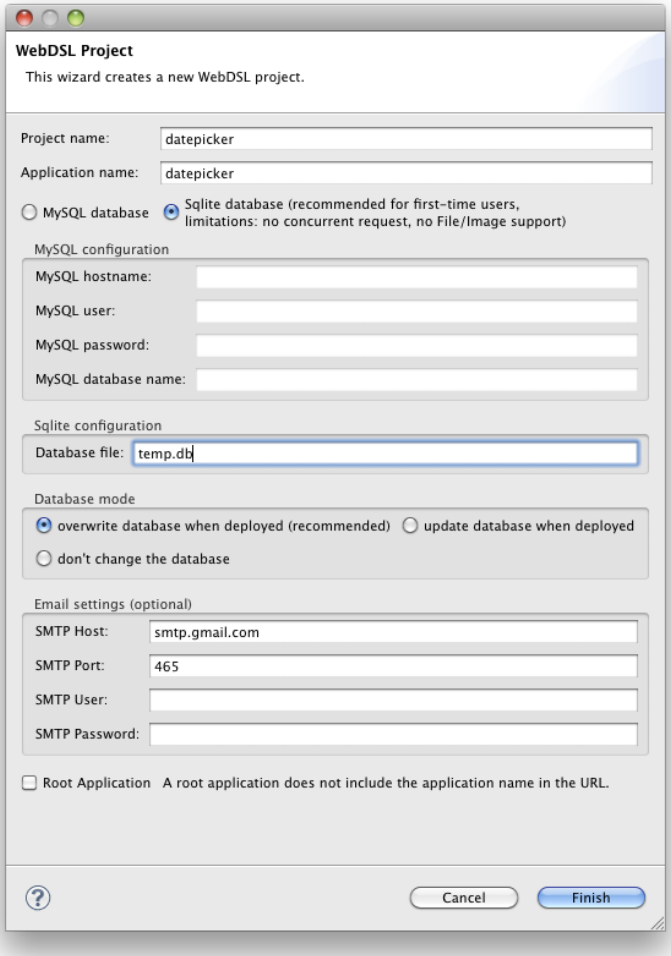
1 Installing the Editor

– Extract eclipse zip for your platform, get it from http://webdsl.org or the tutorial USB stick.



– Create a new empty project using the `New WebDSL Project` wizard. Name the application 'meeting', choose Sqlite as database, and set it to 'overwrite when deployed'. The project will be automatically build and deployed, a browser window is opened when the application is ready.

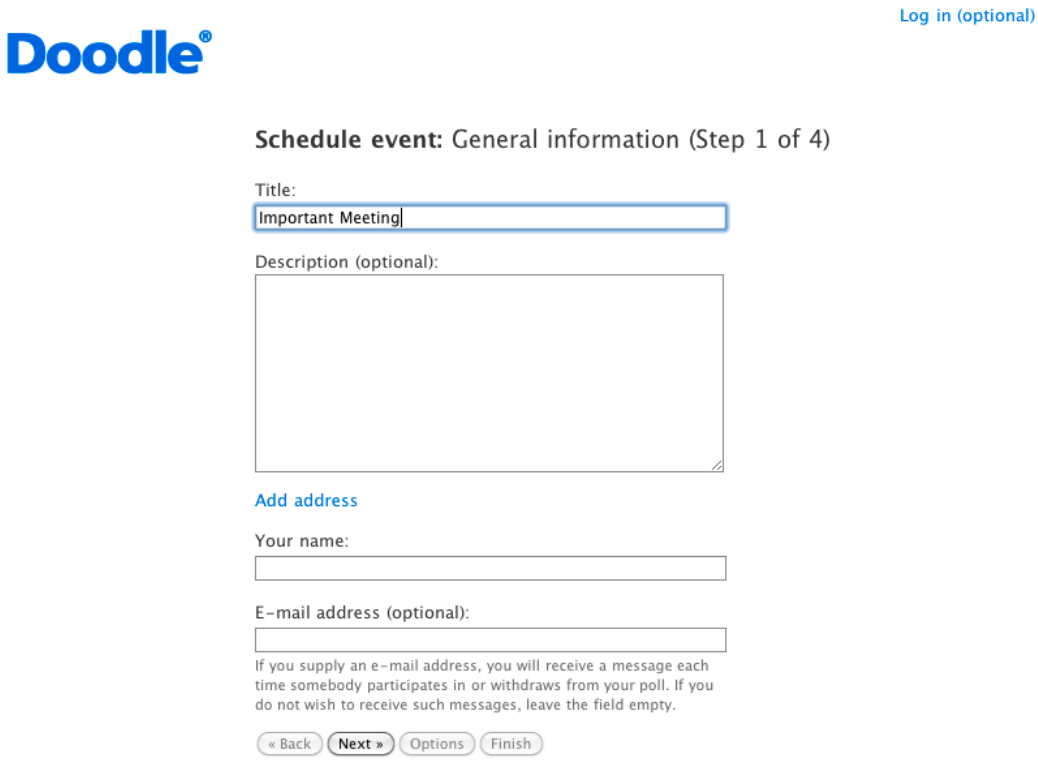

New
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Import...
Export...
Refresh F5
New WebDSL Project
Example WebDSL Project
datepicker.app
application datepicker
define page root(){ "Hello world" }
localhost:8080/datepicker/
Hello world


WebDSL Project
This wizard creates a new WebDSL project.
Project name: datepicker
Application name: datepicker
MySQL database
Sqlite database (recommended for first-time users, limitations: no concurrent request, no File/Image support)
MySQL configuration
MySQL hostname:
MySQL user:
MySQL password:
MySQL database name:
Sqlite configuration
Database file: temp.db
Database mode
overwrite database when deployed (recommended)
update database when deployed
don't change the database
Email settings (optional)
SMTP Host: smtp.gmail.com
SMTP Port: 465
SMTP User:
SMTP Password:
Root Application A root application does not include the application name in the URL.
Cancel
Finish

– Change the “hello world” text to “hello web” and rebuild using ctrl+b (on Mac cmd+b).

2 Tutorial Application

In this tutorial we will be creating an event scheduling application similar to Doodle



and Datumprikker (Dutch).

Feature list:

- Allow creation and managing of events without requiring a registration or login, when an event is created the user receives unique URLs for administration and participation.
- Events can have multiple time slots for which participants can enter their preferences.
- Notify user when input is incorrect with appropriate messages.
- Support emailing the participants when email addresses are provided.
- Support access control for event administration.

3 Data Model

If you are attending a live tutorial you can skip to the assignments in 3.2.

3.1 Entity Definitions

The first part of the application that you will be creating is the data model. The data model consists of entity definitions, which represent the data that is persisted to the database. The first entity that we will create is `Slot` for timeslots.

```
entity Slot {
  time :: String
  event -> Event (inverse = Event.slots)
  prefs -> List<Preference>
}
```

Notes:

```
entity Slot
```

An entity declaration starts with an entity name.

```
time :: String
```

Entity `Slot` has a `time` property which is of primitive type String.

```
event -> Event (inverse = Event.slots)
```

Property `event` is a reference to an Entity with name Event. `Slot.event` and `Event.slots` are inverses, these are synchronized automatically.

```
prefs -> List<Preference>
```

Property `prefs` is a list of Preference entities.

This example covers all the elements you need to know in order to complete the first assignment (3.2). The next part of this section explains the code that is imported into the assignment in order to exercise the data model in the web application.

The `PrefOption` entity contains the options for indicating the preference of a user for a given slot. Add the following to the application:

```
var p_yes := PrefOption{ name := "yes" }
var p_no := PrefOption{ name := "no" }
var p_maybe := PrefOption{ name := "maybe" }
```

A top-level variable declaration in WebDSL is globally visible in the application. This type of variable is automatically instantiated once and added to the database.

Next we're going to populate the database with some test data. Add the following to the application:

```
init{
  var e := Event{
    name := "Important meeting"
  };
  var s1 := Slot{
    event := e
    time := "16 Oct 8:30"
  };
  var s2 := Slot{
    event := e
    time := "16 Oct 13:30"
  };
  var p1 := Preference{
    slot := s1
    option := p_no
  };
  var p2 := Preference{
    slot := s2
    option := p_yes
  };
  var up := UserPreference{
    prefs := [p1,p2]
    event := e
    user := "some user"
  };
  e.save();
}
```

A top-level init block can be used for application initialization. This is performed once after regenerating the database, after application global variables are initialized.

Finally, we need some user interface to visualize our created entities. At this point we will generate most of it:

```
define page root(){
  navigate event((from Event)[0]) {"view event"}
  " "
  navigate manageEvent() {"manage events"}
}
derive crud Event
derive crud Slot
derive crud UserPreference
derive crud Preference
```

Notes:

```
derive crud Event
```

This definition will generate CRUD pages, namely `createEvent` (create), `event` (read/view), `editEvent` (update/edit), `manageEvent` (delete/list all).

```
navigate event((from Event)[0]) {"view event"}
```

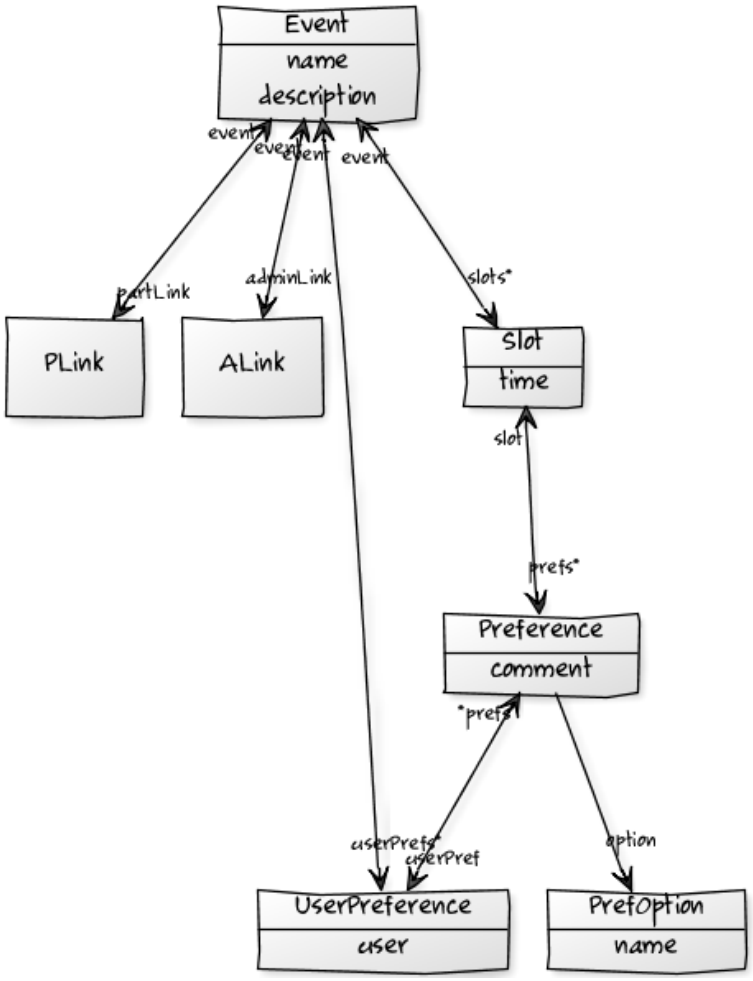
Creates a navigation link to the `event` page with argument `(from Event)[0]` and text `view event`. `from Event` is an HQL query that retrieves all entities from the database, and `[0]` gets the first element of a list.

3.2 Data Model Assignments

Copy the files inside tutorial-base-v1/ to the root of your project. Import the app files in the meeting.app file:

```
imports data
imports rootpage
```

Complete the data model shown below (use type `String` for the non-reference fields).



The complete solution for this part of the assignment can be found under solution/v1/.

4 User Interface

If you are attending a live tutorial you can skip to the assignments in 4.2.

4.1 Template Definitions

Presentation is implemented by pages and templates (partial pages) in WebDSL. These pages and templates are not just for output of data, they can also handle inputs, form actions, and data validation. A `root` page must always be defined, it is the main page of your application. The URLs of pages consist of the page name and the arguments

separated by '/'. We will now discuss a number of page/templates that will be part of our application (no need to copy them from this document, the files are provided in the assignments 4.2)

```
define page root(){
  form{
    submitlink action{
      var e := Event{ slots := [ Slot{ } ] };
      e.save();
      return new(e);
    } { "Create new event" }
  }
  showAllEvents()
}
```

This root page contains a `form` and a call to the `showAllEvents` template. The form consists of a single `submitlink`. The action the link will perform when clicked creates a new `Event` instance, sets its `slots` property to a list with 1 `Slot`, saves it to the database, and redirects to the `new` page with the new event `e` as argument.

```
define page new(e:Event){
  form{
    eventEdit(e)
    submit save() { "create event" }
  }
  action save(){
    e.aLink := ALink{};
    e.pLink := PLink{};
    return completed(e);
  }
}
```

The `new` page takes an `Event` argument. It will show a form to edit this event argument (inputs will be defined in the `eventEdit` template) and a submit button that saves the changes, as well as filling the `aLink` and `pLink` properties and redirecting to the `completed` page.

```
define eventEdit(e:Event){
  label("Name:"){
    input(e.name){
      validate(e.name.length()>0,"name required")
    }
  }
  label("Description:"){
    input(e.description)
  }
  label("Slots:")
    for(slot : Slot in e.slots) {
      input(slot.time)
      <br />
    }
    validate(
      And[s.time.length()>0 | s:Slot in e.slots]
     ,"time slot description required for each slot")
  submit action{ e.slots.add(Slot{}); }
         [ignore-validation] {"add slot"} }
```

The `eventEdit` template takes an Event argument. It consists of labeled inputs for the `name`, `description`, and `slot.time` properties. For the `slots` property all slots in the list will be represented by an input field. The name and slot inputs are validated, they may not be empty. The validation for slots uses some syntactic sugar for working with lists in expressions [1]. A `submit` button is presented that adds an extra time slot input field. The `ignore-validation` allows this particular submit to work even though the other inputs have validation errors, so a user can first add a few time slots and then enter the descriptions for them (the other button that finishes the editing will still perform the validations).

In our application, URL's will be generated for each `Event` to edit the event and let a user enter preferences. The page that allows adding preferences is discussed next.

[1] http://webdsl.org/selectpage/Manual/ActionCode#ListComprehension

```
define page event(e:PLink){
  title{"participants"}
  var up := UserPreference{}
  form{
    label("Name:"){
      input(up.user)
    }
    <br />
    label("Slots:"){
      t{
        for(slot : Slot in e.event.slots) {
          r{
            pickOption(up,slot)
          }
        }
      }
    }
    submit action{
      e.event.userPrefs.add(up);
      return participants(e.event);
    } { "save preference" }
  }
}
```

The `event` page takes a `PLink` argument. Since the default representation of this argument in the URL is the UUID database identifier, it will not be easy to guess. An event organizer can simply email the link to anyone who's invited. This page starts by initializing a new `UserPreference` instance. It contains labeled inputs for the user name and a radio button for each of the slots (implemented in `pickOption`).

```
define pickOption(up:UserPreference,slot:Slot){
  var p := Preference{}
  c{
    output(slot.time)
  }
  c{
    radio(p.option, [p_yes,p_no,p_maybe])
  }
  databind{
    up.prefs.add(
      Preference{slot := slot option := p.option});
  }
}
```

`pickOption` shows a time slot description and radio buttons for selecting yes/no/maybe. The `databind` fragment is executed when it is inside a form that is submitted, it creates a `Preference` instance and connects it to the `UserPreference` and `Slot`.

The `t`, `r`, and `c`, construct an HTML table for layout (although frowned upon by many web designers, in this case it's convenient for getting default alignment). These templates wrap the contents in table, row, and column tags. XML syntax can be used in template definitions, attribute values are WebDSL expressions. The `elements` template call will include any template elements that were passed in the call to the template, e.g. `t{''inside a table''}`. Currently, the WebDSL compiler adds a span to a template render, which we don't want in this case, hence the `no-span` modifier.

```
define no-span t(){
  <table>
    elements()
  </table>
}
define no-span r(){
  <tr>
    elements()
  </tr>
}
define no-span c(){
  <td>
    elements()
  </td>
}
```

4.2 User Interface Assignments

Copy the files inside tutorial-base-v2/ to the root of your project (also include the stylesheet in stylesheets/common_.css). Import the app files in the meeting.app file:

```
imports data
imports lib
imports ui
imports rootpage
```

In these assignments you will be implementing the following definitions, first copy these empty definitions to your meeting.app file:

```
define page admin(a:ALink)
define page completed(e:Event)
define showEvent(e:Event)
define page participants(e:Event)
define showAllEvents()
```

- add `validate(user.length()>0,"name required")` to the UserPreference entity, this will validate each form input for UserPreference.name.
- create a `define page admin(a:ALink)` page, using the `eventEdit` template

- add the `completed` page which shows the navigation links to pages `admin` and `event` for the given Event parameter.
- create a `showEvent` template for showing an event and important data related to it such as user's preferences Add the following method to UserPreference:

```
function getPrefForSlot(s:Slot):Preference{
  for(pr:Preference in prefs){
    if(pr.slot == s){
      return pr;
    }
  }
  return null;
}
```

  With this you can iterate over both the Slot and UserPreference instances and show for each user their preference in a table.
- add a `participants` page that uses the `showEvent` template to show the results of the poll
- to debug our system add to the root page an iteration over all Events and call the `showEvent` template for each of them, by implementing the `showAllEvents` template.

The complete solution for this part of the assignment can be found under solution/v2/.

5 Email

If you are attending a live tutorial you can skip to the assignments in 5.2.

5.1 Email Templates

Email templates can be defined using `define email`. They need some extra template calls to set up sender and receiver.

```
define email sendInvite(e:Event, u:User){
  to(u.email)
  from("pick@slot.com")
  subject("event invitation")

  "You are invited to event" output(e.name) "!"
  " Pick a time slot that suits you here: "
  navigate event(e.pLink) {
    output(navigate(event(e.pLink))) }
}
```

`sendInvite` is an email template that invites a user to go to the `event` preference page of a particular Event. Navigates in emails automatically become absolute links. In order to send this email, use `email sendInvite(e,u)`. Since you probably haven't configured email, we're going to use the internal `renderemail(sendInvite(e,u))` function instead, which renders the email but doesn't send it.

```
define invitees(e:Event){
  <br />
    "Invitees:"
  <br />
  form{
    for(u: User in e.invitees) {
      label("Name: "){ input(u.name) }
      label("Email: "){ input(u.email) }
      <br />
    }
    submit action{ e.invitees.add(User{}); }
      {"add invitee"}
    submit action{
      for(u:User in e.invitees){
        //fake email
        var email := renderemail(sendInvite(e,u));
        log("to: "+email.to);
        log("from: "+email.from);
        log("subject: "+email.subject);
        log("body: "+email.body);

        //send email
        //email sendInvite(e,u);
      }
    } {"send invites"}
  }
}
```

The `invitees` template shows the list of invitees and allows sending all of them an email. For this tutorial, instead of actually sending the email we're logging it using textttrenderemail. You can also configure email settings in application.ini or use the convert to webdsl project wizard to enable real emailing using the `email` call.

5.2 Email Assignments

Copy the files inside tutorial-base-v3/ to the root of your project. Import the app files in the meeting.app file:

```
imports data
imports lib
imports ui
imports invite
imports rootpage
```

- Add an `invitees` property to Event which contains a List of User entities.
- Add a user entity with properties `name` of type String and `email` of type Email.
- In the `new` page add to the `save` action `e.invitees := [User];` (the previous version of this page can be found in the comments in invite.app)
- Add an `invitees(e)` template call to the `completed` page. (the previous version of this page can be found in the comments in invite.app)

The complete solution for this part of the assignment can be found under solution/v3/.

6 Access Control

If you are attending a live tutorial you can skip to the assignments in 6.2.

6.1 Access Control Rules

The final addition we're going to make to the application is access control. We will add a login form, and if a user creates an event while logged in, we're going to protect the admin page from being accessed by other users.

Extend the existing User entity with a password and create an initial user (registration is skipped here for conciseness).

```
entity User{
  name :: String
  password :: Secret
  email :: Email
}
var user_admin :=
  User{ name := "admin" password := "admin" }
```

Access control is enabled by declaring a principal entity. If access control is enabled all pages and templates are blocked by default. They can be opened up using access control rules. In this example we will open everything except the admin page. Note that rules matching the same resource must all evaluate to true in order for the resource to be accessible (conjunction of ac rules).

```
principal is User with credentials name,password

access control rules
  rule page *(*){ true }
  rule page admin(e:ALink)
    { e.event.organizer == principal }
  rule template *(*){ true }
```

Note: Currently the syntax separates regular sections and access control rules. If you want to write regular definitions below these rules add "`section`" between them.

For quick creation of a login form use `authentication()`. This form will be based on the specified credential properties of the principal entity.

You can also customize the login form:

```
define auth(){
  if(loggedIn()){
    signout()
  }
  else{
    signin()
  }
}

define signout(){
  "Logged in as: " output(securityContext.principal.name)
  " "
  form{
    submitlink action{logout();} {"Logout"}
  }
}

define signin(){
  var name := ""
  var pass : Secret := ""
  form{
    label("name: "){ input(name) }
    label("password: "){input(pass)}
    <br />
    submit action{
      validate(authenticate(name,pass)
               ,"The login credentials are not valid.");
      message("You are now logged in.");
    } {"login"}
  }
}
```

The `authenticate` function is generated based on the credentials specified for the principal.

6.2 Access Control Assignments

Copy the files inside tutorial-base-v4/ to the root of your project. Import the app files in the meeting.app file:

```
imports data
imports lib
imports ac
imports ui
imports invite
imports rootpage
```

– Add an access control rule that enforces that only the creator of an Event can view the vote results (`showEvent` template). If there is no creator for an event, it should be visible to anyone. Note: Start an access control section with 'access control rules'.

The complete solution for this part of the assignment can be found under solution/v4/.