WIS project report

Drinking Buddies

EVENING 2: Tom De Bie & Pieter Libin

Deliverables

As submitted in a zip file named: **WIS-EVENING_2.zip**
Report: report.pdf (this document)
Source code: WIS-project directory
Install file: INSTALL.html (generated from INSTALL.md, which is part of the source code directory)
Video: demo.mp4 (in case there are any problems with the playback of this video, we also added this video to YouTube: http://www.youtube.com/watch?v=FYh1l-0qiRM)

Functional Overview

Start page

On the start page the user has the possibility to log in and search for a beer or bar.

Both search boxes have auto-complete functionality. When the full name of a beer or bar is entered and search is clicked, the user will be redirected to the corresponding beer or bar page respectively. If the text entered is not a full name the user is redirected to a search page with the results of the search.

For the log in functionality we use OAuth. Currently the application supports Facebook as an OAuth provider. When the user logs in for the first time, he registers by logging in to Facebook and gives permission to our application to access his Facebook profile and friends. The application uses the Facebook Graph API (this is a RESTful web service Facebook provides) to retrieve the basic information of the user. When a user is logged in, the application header at the top of the page shows the user's name. The user-name can be clicked to go to the user's page. This application header is shown on every page of the application, this way the user is always aware that he is logged in.

Beer search page

The beer search page provides the functionality to search for a beer. The search box has auto-complete functionality. If a user searches for a full name of a beer he will be redirected to the beer page of that beer. Otherwise a list of beers that match the search will be shown on the page. If no beers are found a message is shown that there are no beers found.

Our application doesn't allow users to add new beers. This is because all the beers are retrieved from the BreweryDB REST web service. Currently the application doesn't automatically update the list of beers. A future development could be to fetch the beer from BreweryDB when a beer is searched for but not found in our database.

Beer page

On the beer page you can see some basic information about the beer. The brewery, alcohol percentage and the picture of the label shown on this page are retrieved from the BreweryDB web service. There is a list of tags shown on this page. These tags are styled using CSS3 rounded corners. There is also a Facebook like and share button. Because we use a unique (but readable) url for every beer it is possible to share or like a specific beer on Facebook.

The page also shows how many users added the beer to their favorites and the score of the beer. This score is the average score of all the reviews in the database. At the bottom of the page all the reviews are shown. Only the scores, poster and the first part of the review are shown. The first part that is shown is the first 60 characters (plus the rest of the last sentence; to make sure no incomplete sentence is shown). We used 60 characters because this corresponds to more or less 2-3 sentence(s)[1]. If the last sentence ends in an ellipsis this ellipsis is also added. If the users clicks the expand/comment button the review's full text is shown and the comments on the review are visualized. The user can collapse the review when it is expanded. When logged in the user can add a new comment on a review. The post time of a comment is shown as a text (e.g. moments ago, 8 hours ago, 10 days ago); this text is formatted by the PrettyTime library.

[1] http://strainindex.wordpress.com/2008/07/28/the-average-sentence-length/

There is also a link to the "Search nearby bars" page. This page allows the user to find bars that sell this beer.

If the user is logged in, several options are added on the page. There is a button to add the beer to the favorites of the user. A button to add tags is also available when logged in. If the user adds a tag he can choose to add a new tag or a tag that already is in the system. If the user chooses to add an existing tag there is a drop down list with all the tags that are in the database. The user can also add a new review when logged in. The form to add a new review has four fields to add scores for color, smell, taste and feel. These input fields have client- and server-side validation so that only a floating-point number between 0 and 5 can be entered. If something is entered that is not correct, a tooltip on the input field is shown with a helpful message about what is wrong. Once the review is added the score of the beer is automatically updated on the page.

Bar search page

The bar search page provides the functionality to search for a bar. On this page there is a button to add a new bar. The search box has auto-complete functionality. If the user enters the full name of the bar, he is redirected directly to the bar page. If the user only enters a partial name a list of bars that match this input is shown. If no bars are found a message is displayed to inform the user. When the user clicks the "Add new bar" button a dialog is shown to allow the user to add a new bar. This form also has some basic validation to check if required fields are filled in. This validation is checked both client- and server-side. When "Save" is clicked, the address entered on this form is passed to Google's geocoding RESTful web service to find the coordinates of the bar. The retrieved coordinates are stored in the database.

Bar page

The bar page contains the information about a bar. The bar name and address are shown. There is also an image of the bar and it is possible to change this image (when the user is logged in). The image is uploaded from the user's computer to the server and stored in the database (as a binary blob). The bar page also shows the score of the bar and a chart with the evolution of the score.

The score is shown in a spin box and when the score is changed a button to save the new score appears. The score input is disabled when the user is not logged in. The chart is automatically updated when the score is saved. The chart shows the score evolution over the last 10 months. An improvement could be to change the range of the chart if there are no scores available for the last 10 months. This improvement is possible because we store every score change in the database and calculate averages per month to visualize in the chart. If there are no scores in the database, the chart is not shown. There is a list of the beers available in the bar on the page. If the user is logged in, there is a button to add a beer to the list. This button shows a small dialog where the user can enter the name of the beer. This field also has auto-complete functionality. The bar page also has a Facebook like and share button. It is possible to add a comment to a bar. Here the same comment widget is used as the one for beer reviews. There is also a button to add the bar as favorite.

User page

The user page shows some information about the user. The name of the user and his picture are shown. This picture is the profile image from Facebook and is retrieved from Facebook. The image is not stored in our database; this means that if the user changes his profile picture, the new picture is also shown in our application. The page contains 5 favorite beer and 5 favorite bars. If the user is logged in, he can remove one of this favorite bars or beers from his favorites. When logged in the users has the possibility to share his location. When the user clicks the share location button, the current location of the user is fetched using the HTML5 location sharing API. The coordinates are used to find the bar the user is currently occupying. If the user shared his location, the bar is shown on the user page. It is only possible to share your location if you are at a bar. At the bottom of the page there are two links: one to search for nearby bars and one to search for nearby friends.

We planned to provide a page to show a list of friends and a page to find other users, but we decided to drop this functionality when we had to reconsider our feature set (see “Organization of the project”). The possibility to show all the favorites is also to be developed.

There is also a button to import your Facebook friends on the user page. By allowing the user to import his friends, he doesn't need to add all his friends manually to our application.
Currently all of the user's friends are imported at once, we are aware that it would be better to allow the user to control which friends should be imported, but we did not had the time left to implement this.

Search nearby bars page

This page provides the functionality to search for nearby bars. When the user enters the page his location is fetched by using the HTML5 geolocation API. This location is then converted to an address using the Google Geocoding API. The address is filled in the location search box. The user can change this location and enter the range of the search. If the user clicks the search button the location entered is converted to coordinates using the Google Geocoding API and all the bars within the range are listed and shown on a map. The map shown uses the JWt Google maps widget. The user can also provide the name of a beer; only bars serving that beer will be shown.

Search nearby friends page

This page allows a user to search for bars where his/her friends currently reside. Like the search nearby bar page the user's current location is shown when he enters the page. The user can specify the range of the search. If the user clicks the search button a list of bars within the range of the specified location are shown and for each bar there is a list of which friends are currently at the bar.

Location search algorithm

To search for nearby bars we have to query for a location in the database. We based our query on an article written by Jan Philip Matuschek[2]. We also used the Java class provided with this article. The idea is to first query on a bounding rectangle where we can use a database index on the coordinates. And then we use the results of this query to find the locations that are within the specified radius.

[2] http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates

Responsive application

Our application is responsive and can be used on mobile devices. We present some screenshots to demonstrate this.

Screenshot 1: Our application on a mobile browser

Screenshot 2: our application on a mobile browser

Internationalization

We implemented our web application with internationalization in mind. All strings that we used throughout the user interface are externalized in a resource bundle. To demonstrate this, we translated the application to Dutch.

The language will be chosen based on the user's browser settings; this is demonstrated in the video.

Paths

Every page in our application is directly accessible by using the correct URL. The URL's of our pages are:
Start page: "/db"
Beer search page: "/db/beers"
Beer page: "/db/beers/$specific-beer$"
Bar search page: /db/bars"
Bar page: "/db/bars/$specific-bar$"
User page: "/db/users/$specific-user$"
Search nearby bars page: "/db/find_nearby_bars"
Search nearby friends page: "/db/find_nearby_friends"

Our web service

We developed a REST web service that has 2 functions:

- serve a list of all bars in the system as JSON text
- serve the details of one bar as JSON text

For the detailed description of a bar, we include the comments on the bar and the bar's image (the image is stored as a data URI, the image blob is encoded as base64).

The web service is built as a simple servlet that parses the URL, if the URL equals "/bars" we return a list of all bars in the system, if the URL equals something like "/bars/$specific-bar$" we only return that specific bar.

Technological overview

Development tools

We used Eclipse as our IDE.

To be able to build the software without Eclipse, we created an ANT script ($PROJECT$/build.xml) that can build a WAR file; such a file can be deployed in any servlet container.

To collaborate, we used git; our project is available as a public github project on this URL: http://github.com/plibin-vub/WIS-project

JWt

We used the Java Web Toolkit (JWt) to develop our web interface. The website of this toolkit is: http://webtoolkit.eu/jwt.

The main goal of the JWt project is to build an abstraction on today's web technologies and provide the users of the library with a widget-centric API. This allows the users of JWt to build web applications in a similar way in which desktop applications are built, without trying to emulate a desktop in a browser.
JWt builds on the Servlet 3.0 API, so an application written on top of JWt, can be used in any Servlet container.
JWt will keep a server-side DOM tree of the application; based on the widget structure the user programmed. JWt will render this DOM tree in a way that is most suitable for the client;

- if the client has JavaScript support: AJAX will be used to render the DOM tree
- if the client has no JavaScript support: the DOM tree will be rendered using plain HTML

This approach solves some of the problems with AJAX that were discussed during the lecture of the WIS course:

- AJAX applications are difficult to crawl by bots
    - JWt solves this by serving the web application as plain HTML to such client
- AJAX applications do not work or work only partially on older browsers
    - JWt "gracefully degrades" the application to match the client optimal conditions
- in AJAX applications it is difficult to bookmark states
    - JWt uses internal paths to make AJAX "pages" accessible

While solving a lot of the issues commonly found with AJAX, it is important to note that when AJAX **is** available JWt makes use of it extensively without requiring any explicit commands by the programmer.
A nice example of JWt's abstraction is the event system, let's take a look at this code that creates a button and connects a listener to the clicked signal:

```
WPushButton button
      = new WPushButton("Greet me.", getRoot());
final WText greeting = new WText(getRoot());
button.clicked().addListener(this, new Signal.Listener() {
public void trigger() {
```

```
        greeting.setText("Hello there, " +
                                 nameEdit.getText());
        }
});
```

This code will trigger the anonymous function trigger() when the button is clicked. However, when AJAX is available this button click will be propagated to the server via JavaScript, if AJAX is not available this button click will be propagated to the server via form submission.

JWt applications are Single Page Applications, but it is possible to reference to a particular state using internal paths (an internal path represents a URL), a feature which we frequently use in our project.

JWt's widget-centric approach also stimulates the developer to divide his/her application in different widgets. The advantage of this is that such widgets can be easily re-used in different parts of the project, or even in different projects.

In our project a clear example of such widgets are the widgets involved in showing and adding comments.

We extensively made use of JWt's template system; we formatted most of our user interface with the combination of XHTML templates and Bootstrap CSS style classes. The advantage of this approach is that the styling of our application can be easily outsourced to web designers.

By design, JWt always checks user input (from text fields, or other input widgets) for HTML/JavaScript code injection.

JWt has proven to be a nice library to develop our web application; it allowed us to build a very responsive AJAX-enabled application that can also be crawled by bots without any extra effort. Our application is also accessible by older browsers.

We found some small problems with the library:

- a missing feature to use auto-completion out of the box: we had to write some JavaScript code to fix this
- cookies could not be set from the OAuth event loop

We will propagate these issues to the authors of the library.

JWt has a Google maps widgets (WGoogleMap) that wraps most of the functionality of Google maps. We had to extend the default widget to add some functionality that we needed in our project.

JWt Charts

JWt has a charting module that allows charts to be included in a web application. JWt abstracts charts in a way that the chart can be rendered in different ways (depending on what the browser supports): HTML5 canvas, inline VML, inline SVG, or a raster image.

JWt Auth

We used the JWt auth library (this library is part of the JWt software distribution) to implement OAuth-based authentication, currently we only support Facebook's OAuth service.

Using this library as an abstraction layer allows us to easily integrate other OAuth backend (such as Google's OAuth service).

Bootstrap

We used Bootstrap (http://getbootstrap.com) to style our application and to make use of its "responsive design" feature, to ensure that our web application scales well on mobile clients.

We tried to style as much as possible using Bootstrap, but we also needed to make some style classes by ourselves.

jOOQ

jOOQ (http://jooq.org) is Java library that implements the "active record" design pattern. The active record design pattern maps database records more or less directly to objects that can be used in an object oriented programming language.

jOOQ offers a database abstraction; most of the code written in jOOQ can be used with different database backends.

Since jOOQ maps records to objects, we need to specify classes that model these objects. These classes can be generated from a database schema, to do this; we need to invoke the program "org.jooq.util.GenerationTool", which is part of the jOOQ software distribution. We wrapped this program with an appropriate

configuration in an ant task “generate-jooq-classes”, this ant task is part of our buid.xml file.

jOOQ allows the programmer to write SQL in type safe Java code, rather than SQL strings. The advantage of this approach is that if your database schema changes, and some of the queries do not correspond with the new types user in your schema, a compile error will be reported. Another advantage of this approach is that a query can be auto-completed by IDE’s such as Eclipse.

By using jOOQ consistently (without mixing in SQL strings), we ensure that we are safe for SQL injection.

Using jOOQ was a pleasant experience, which saved us a lot of time that would otherwise be spent on debugging SQL strings.

We also noticed that the queries that were written in jOOQ’s DSL looked really nice. The SQL code looks like it is really part of the application, rather than that it is simply a string that is to be parsed at runtime.

PrettyTime

PrettyTime (http://ocpsoft.org/prettytime/) is a time formatting Java library; we used it to format the times for comments. It allows you to generate human-friendly strings such as “a minute ago”, “2 days ago”.

Gson

Google’s gson (http://code.google.com/p/google-gson/) is a Java library to convert JSON objects to JSON text and vice versa. We used this very convenient library to parse the results of the REST services we invoke to objects. We also used this library to generate JSON text based on objects to expose data in our own REST webservice.

XStream

XStream (http://xstream.codehaus.org) is a Java library to read an XML file into a tree of Java objects. We used this library to read our configuration XML file to an appropriate object structure.

Sqlite

Sqlite (https://www.sqlite.org) is a lightweight database system that is written in C. We used it as our database backend, since it is easy to setup on Windows

and MacOS (our development platforms). In order to scale the database to a larger number of users, it would probably be necessary to use a more advanced database solution, such as Postgresql for example.

HttpClient

HttpClient (http://hc.apache.org/httpcomponents-client-ga/) is a Java library that we use to invoke RESTful web-services and download the results of such an invocation.

BreweryDB

We use the BreweryDB RESTful web service to retrieve information about the beers. We used a development account to access BreweryDB. This account is free but has some limits. One of the limits is that only 400 requests per day are allowed. A second limitation is that you can't retrieve all the beers; you have to specify a beer or search criteria.

Dependencies

All our dependencies are included in the project's lib directory. The dependencies are explicitly defined in the Eclipse classpath file ($PROJECT$/.classpath).

Ant automatically includes all necessary dependencies when building a WAR file.

The database software sqlite is not included in the project directory; we tested the software with version 3.7.12.

Our database schema and an example database are included in the project source, details on this can be found in the installation document ($PROJECT$/INSTALL).

Code metrics

~4700 lines of Java code (generated code not included)

~450 lines of XHTML code

~140 lines of XML resource bundle

~25 lines of JavaScript code

~105 lines of CSS code

We present these metrics to show that with the JWt and jOOQ framework we were able to build a very responsive AJAX-enabled web application without

writing much JavaScript and without writing any explicit SQL (all our SQL was written using the jOOQ Java API).

Almost all of our code is written in Java, which is a compiled statically typed language. The advantage of such languages is that a lot of errors can be detected at compile time, rather that at run time (what would be the case in interpreted languages such as JavaScript).

A quick guide trough the source code

The main class that directs what is shown to the user is the Application (com.github.drinking_buddies); this class parses the URL and shows the appropriate form.

All forms can be found in the "com.github.drinking_buddies.ui" package, this package also contains the templates.xml file, where all XHTML template code resides.

To familiarize new developers with our application, we thoroughly documented the com.github.drinking_buddies.ui.BeerForm class. The documentation in this class explains some of the features of JWt that are required to build forms and how to deal with events.

Some specific widgets:

- comment widgets: can be found in the "com.github.drinking_buddies .ui.comments" package
- autocomplete widget and model: can be found in the "com.github.drinking_buddies .ui.autocomplete" package

Some basic widgets that should be part of the JWt distribution but we had to implement ourselves can be found in com.github.drinking_buddies.jwt.

All code related to the webservices we use can be found in the following subdirectories of the package "com.github.drinking_buddies.webservices": brewerydb, facebook, google and rest. Our own bar web service can be found in this package: "com.github.drinking_buddies.webservices.servlet".

All jOOQ related class can be found in the package "com.github.drinking_buddies.jooq", most of these classes are generated.

Our resource bundles can be found in the package "com.github.drinking_buddies.i18n".
The classes that deal with the configuration of our web application reside in the package "com.github.drinking_buddies.config".
Our CSS file is located in "WebRoot/style/".

Organization of the project

We started this project with 3 students: Tom De Bie, Pieter Libin and Ingrid Nijs. Shortly after the presentation of the prototype, Ingrid decided to stop with the project. At this time Ingrid had only contributed the first version of the database schema.
After Ingrid left, our team only had 2 members, so we tried to change the specifications in such a way that we could still implement everything with 2 developers while still maintaining the gist of our project.
Over the course of the semester, several meetings were organized were Tom and Pieter discussed technical issues and implemented large parts of the application.
The other communication was held largely via e-mail and Skype.
The work on the project was nicely balanced, and we both did some work on different aspects of the project. Pieter setup an initial JWt and jOOQ project and spend most time on the beer-form and user-form. Tom spent most of his time on the bar-form, the search-on-location forms.
Our project is a public project on github; so it is possible to get a detailed overview on which code Pieter and Tom worked on specifically.
Since we use JWt as a GPLv2 library (JWt is available under a commercial or GPLv2 license); we licensed our project GPLv2.

Conclusion

We enjoyed working on this project a lot. We both learned a lot during the course of this project about web development, using web services, setting up web services, sqlite, jOOQ, …

It was a bit difficult to decide which features to drop (without dropping the gist of our project) when Ingrid left the project, so we tried to implement most of it anyway.

I think we succeeded in implementing all features we considered interesting, however, the final completion in terms of icons and CSS could have been a bit more elaborate. On the other hand, this is something that with the architecture we provided can be easily done by web designers.

We think the final project is a nice and usable web application that we, and a lot of our friends see as an application they might use.