Building a Project Ideas Web Application - Report

Introduction

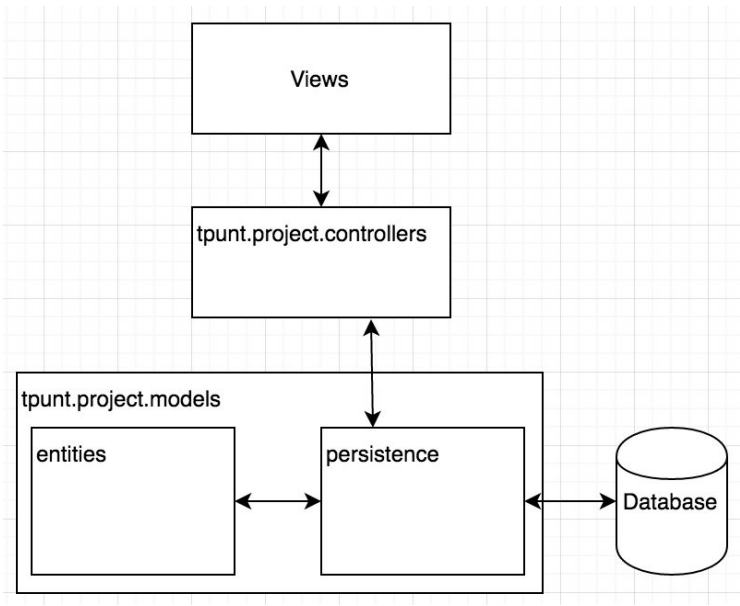
The primary aim of this coursework was to build a better version of the PUMS website[1]. To achieve this, I firstly looked at the deficiencies of the current website to see what could be improved about it. I then attempted to build a better system by giving additional capabilities to the current website and by enhancing the current user experience with realtime page updates using AJAX.

The secondary aim of this coursework was to look into Java EE 7 as a platform for developing web applications. This report therefore critically analyses and reflects upon my experiences of using this technology for this piece of coursework.

The GitHub repository can be found at: https://github.com/tpunt/ProjectIdeas

Design

Figure 1. Application architecture overview and the interaction between the MVC components.



The architecture of my application was subdivided into a number of packages. Broadly, the Java files were divided into two main categories:

- **controllers**. This package held the logic that responded to actions upon the user interface. It was responsible for holding ephemeral data, such as data pulled from form fields). It also performed actions against the database, including persisting ephemeral data and pulling data from the database to be displayed in the templates. It performed these database operations by

[1] http://www.sums.port.ac.uk/spi/

invoking methods from the **models** package, specifically the ***Facade** classes found in **models.persistence**.

Controllers followed a ***Handler** naming convention, because they were tasked with handling a specific part of the application.

- **models**. This package holds the application logic, including the business logic and domain logic. It is further subdivided into:
    - **entities**. These are domain model objects used to hold entity data and perform validations on their data.
    - **persistence**. The persistence classes perform actions against the database using the entity classes, including persisting them into the database and hydrating them from the database.
    - **business**. The business classes should have held the application and business logic of the application, but as mentioned below, this did not happen.

Other than the design decisions of the overall application architecture, the files that should not be directly accessed via the URI (such as the header.xhtml file) were placed into the WEB-INF directory.

Implementation and Testing

Development Tools

Netbeans was the IDE I used. Overall, it caused a rather frustrating experience because of the laggy behaviour. Netbeans is renownedly slow, where even simple scrolling to jolted. It also crashed a few times, and there seemed to be no correlation of when the server crashed (requiring a server restart), when the database connection failed (requiring a DB reconnection), and when the project simply needed to be cleanly built versus simply run.

Because of the above, I would not use Netbeans again. I used Eclipse in the past and found it to be a very pleasant experience, and so this is the IDE I will likely use in future for such work.

Implementation

Numerous implementation decisions were made, but I will only mention a few of the more important decisions here, along with the ones that highlight my thoughts, problems, and general experiences with Java EE 7.

Decisions

I avoided using JSF filtering because they would be applied to forms that would be used in multiple parts of the website. For example, the login form and the registration form both required a username and password to be entered (along with other fields for the registration form). If I had used JSF filtering, the validation logic would have been duplicated across two forms. At best, I could have composed these forms by making the login form a subpart of the registration form to keep validation

logic in a single place. This composability, however, can impact maintenance when parts of forms are subdivided all over the place.

I therefore chose to put the validation logic in with the domain model entities (package **tpunt.project.models.entities**), thereby coupling together the data with related processes. This decision also helped to avoid the anemic domain model anti-pattern (Martin Fowler, 2003). The primary drawback to this decision was that forms could no longer automatically populate fields of an entity object, since the respective mutator methods of domain model entities could now throw custom exceptions upon invalid input. The form data therefore had to be retrieved using the current **FacesContext** upon form submission.

When implementing the AJAX-driven search bar to filter projects on the landing page, I originally populated the project title, owner, and status fields via the **value** attribute for form fields. This, however, did not work as intended. When input was entered into one input box, when focus was put onto another input box, the field holding the value for the first input box reset its value. I suspect this was due to my bean having **RequestScoped**, and AJAX requests are seen as a new request cycle, thereby resetting the backing bean's fields. I could not, however, get the bean to function when using **ViewScoped**, and so I instead pulled the values from the form by the current **FacesContext** instance upon changing any of the form fields.

I attempted to keep components loosely coupled in my system. This was done by following the Law of Demeter (Karl Lieberherr, ND). Notably though, there were cases where I broke this because I wasn't sure how to best inject the dependency, and so I went for pragmatism over cleanliness. This can be seen, for example, in **LoginHandler.doLogin** when I fetch the **HttpServletRequest** request object.

Ugly Hacks

One of my biggest annoyances was not being able to leverage the URI. The URIs in their current form are not clean, and not bookmarkable either. This is because the project IDs are currently propagated via hidden form submissions using POST requests.

Also, when updating a project, the update action must perform 3 queries against the DB (which is highly inefficient). The first checks to see if the project owner is the same as the logged in owner, the second to updates the project, and the third re-fetches the updated project since the last updated timestamp field needs to be retrieved. This approach could be refactored by removing the first query with a slightly more complex UPDATE query, and removing the third query by generating the last updated timestamp on the client.

Unimplemented Functionality

Some functionality was not implemented in the end due to time constraints. Primarily, the user profiles feature was not implemented. This means that users cannot update their information, nor can they add, update, or delete addresses or organisations (classes for these can be seen in the source code, but they have no purpose at the moment). The admin functionality of being able to moderate users was also not implemented.

Implementation Regrets

My biggest regret was putting too much responsibility into the controllers. Because the controllers were so intimately close to the templates, I hacked out code in the controllers most. This created large unwieldy controllers that contained responsibilities beyond what they should be doing.

For example, much of the code in the **ProjectHandler** controller is application logic, with pieces of business related logic (such as admins only being able to set a project's status). This really should have been put into the **models** package, specifically in the **models.business** package.

This would have created a slightly different workflow to Figure 1 (above), where the controllers would have interfaced with the **models.business** package, and the **models.business** package would then interface with **models.persistence**. This follows the popular principle of "skinny controllers, fat models," (Jamis Buck, 2006) as well as the Single Responsibility Principle (Martin, R.C, 2002).

Frustrations

When building the application, my biggest frustration came from using JSF. The steep learning curve of this technology, in combination with its style of doing things, created a lot of confusion when first developing the web application. For example, when conditionally displaying forms to create or update a project, submitting the form always failed. This was apparently because when JSF decodes the form submission, it also checks if the component is rendered or not, and given that the backing bean is request scoped, the request was resetting the page's data, causing the old form to not be displayed.

JSF also made it difficult to:

- Debug - where it injected a lot of unreadable code into the generated HTML pages
- Go against convention - where, for example, the form field values got not simply be populated using the EL, they needed to be two-way getter/setter methods. This was not wanted in my case since the domain model objects could throw exceptions, and so populating the fields with default values from the entity, but not directly updating the entity fields was not easy to achieve
- Leverage the query string for bookmarkable content. I attempted many times to use the query string of the URI, but was not able to achieve using it. Instead, I had to resort to using the suboptimal hidden form submissions
- Make the page friendly to reloading. Because form submissions drove the navigation of the website, reloading the page caused undesired behaviour
- Non-obvious behaviour. The behaviour does not seem overly intuitive in many aspects, such as the conditional form submissions example given earlier.

Testing

Testing was primarily done through manual functional testing. This seemed like a good choice to begin with because it was quick and easy to do, but as the system grew in size and functionality, it became increasingly slower and more difficult to test everything. This lack of ability to regression test caused numerous bugs to be uncaught upon refactoring parts of the code base.

Taking an automated approach to testing, whether it is unit testing, integration testing, or functional testing, is therefore paramount to producing quality software. This is another regret I have when building the software, and had I had the chance to start over again, I would take a Test-Driven Development approach to enforce testing as part of my implementation process.

Summary

Overall, there was a lot more I wanted to do on the project, but simply didn't have the time for. The primary features that I didn't get around to implementing were a more sophisticated user management system (to enable for the moderation of users) and user profiles (to enable users to add, update, and delete addresses and organisations).

The implementations I did make that I would like to change predominantly revolve around how data is passed through the system (both on the server and client). Firstly, as mentioned before, many form submissions were simply used to propagate the project ID. It would have been far better to leverage the query string of the the URI to give bookmarkable pages, which would have been better for the user experience. Secondly, as also aforementioned, logic should have been refactored out of the controllers into the models, changing the flow of data through the system.

JSF, to me, felt like a massively over-engineered solution that provided an overall sub-optimal solution for building clients of web applications. I feel that it would have been far simpler and far more enjoyable to have simply used Java EE 7 to build a web service, and then use the usual HTML, CSS, and JavaScript technologies to build a client application that integrated into this web service. Given that JSF requires JavaScript anyway, using technologies that most web developers are already familiar with seems like a superior and simpler solution that having to learn a whole new technology (JSF) to build clients - of which are targeted only to web browsers (unlike web services, which can target any platform with integrating clients).

References

1. martinfowler.com, (2003). AnemicDomainModel. Retrieved from: http://martinfowler.com/bliki/AnemicDomainModel.html
2. Karl Lieberherr, (ND). Law of Demeter (LoD). Retrieved from: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html
3. Jamis Buck, (2006). Law of Demeter (LoD). Retrieved from: http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model
4. Martin, R.C., 2002. The single responsibility principle. The Principles, Patterns, and Practices of Agile Software Development, pp.149-154.