Hibernate 'til Spring
Benefits of Spring MVC, Hibernate and Struts for the Development of a Web Application

Chris O'Brien

March 27, 2014

Abstract

Web development is one of the fastest growing areas in software development, with new tools being developed yearly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Introduction

This project concerns the development of a web application using a web framework in conjunction with a number of other tools. Throughout development, there is a particular cognisance towards the support of Non-Functional Requirements [NFRs] by both the web framework and the supporting tools throughout the development process.

1.1.1 General Introduction

The main goal of this project is to reflectively analyse a WAF [Web Application Framework], and architecture stack, in the creation of a website. This will be analysed in respect to both functional and non-functional requirements. Two key requirements are extensibility and maintenance. Extensibility refers to the ability of the framework to allow added functionality to the web application without having to modify the core workings of the application. Maintenance refers to the upkeep of the code, and facilitates the modification of the source code after the product is deployed. This may be to correct faults, improve attributes such as performance and security. The creative driver of the project is the development of a website to meet the requirements and needs of Monaleen Tennis Club, for both members of the club and of the committee. These needs will overlap as all committee representatives are all club members, but not all members are on the committee. From this, it was important to identify the precise requirements for each type of user. The main focus of this project was for the club to be able to perform their core functions through the website. This extended to the registration of members, a timetable for the courts, the creation and distribution of tournament schedules, the organisation and timetabling of training sessions, a method to contact all members and a news section to update and advise members of changes and upcoming events .

- Member Management
- Timetable Management
- Tournament Management

1.2 Objectives

1.3 Scope

1.4 Methodoloy

The methodology chosen as the foundation for this project is the Russo and Graham (1998) design methodology. It focuses on 9 iterative steps, each with feedback loops. The steps are outlined below

- Identification of the problem
- Analysis
- Design of the Application
- Resource Gathering
- Coding
- Testing
- Implementation
- Post Implementation Review and Maintainance

Other methodologies that were examined such as Balasubramanin and Bashian (1997), Siegel (1997), Iskawitz et al (1995) and Cranford-Teague (1998). The pros and cons of these methodologies were examined by Howcroft and Carroll (Howcroft and Carroll 2000), and after an examination of their findings, the Russo and Graham methodology best suited the nature and scale of this project. While the other methodologies are strong, they are geared towards large scale web development projects, or towards document-centred websites, and would not suit this project.(Howcroft and Carroll 2000) Using these as a guide, the following methodology was established.

- Identification of the problem
- Structured Literature Review
- Statement of the FYP Objectives
- Design of the Test Suite
- Development of the Prototype

  Analysis

  Design of the Application

  Resource Gathering

  Design Review

Coding

Testing

Implementation

Post Implemetation Review and Maintainance

- Emperical Study
- Critical Evaluation of the Results

1.5 Overview of Report

1.6 Motivation

The motivation behind this project for me was to examine, understand and work with software frameworks and methodologies that would be commonly used in industry, and to develop a software application from them. The module, Distributed Systems, touched on some of the tools and technologies, Netbeans and EJB respectively, used in relation to Java Enterprise development, and this formed the foundation of my interest in the area. I felt the FYP was a perfect vehicle to supplement my knowledge of this subject, with particular attention being paid to popular and in demand technologies.

Chapter 2

Background

2.1 Introduction

There are a number of components needed to build the architecture of a web application. The nature of these components is explored below, and their contribution to the creation of a web application is analysed. A more detailed breakdown on their usage within the application is explored in subsequent chapters.

(bit about usability to go here)

2.2 Technologies

2.2.1 Web Application Framework

The Web Application Framework [WAF] chosen for this project is Spring MVC. Shan and Hua define a WAF as a defined support structure in which other software applications can be organized and developed. (Shan and Hua 2006). MVC, or Model-View-Controller, is a software pattern that facilitates the use of a user interface, shown in its classic form in Figure 2.1. The intention of this pattern is to form a clear division between domain objects and presentation objects. The Model manages the behaviour and data of the application. The View manages the information obtained from the model and displays it to the user. The Controller manages user input, such as key strokes, mouse movements or a touch display, and can interact and invoke functionality within the Model and/or View.

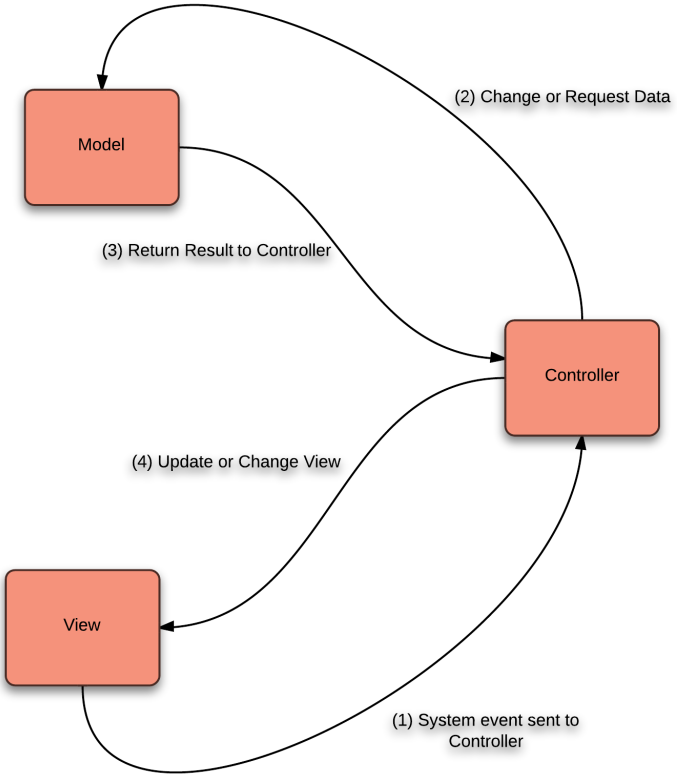

Figure 2.1: Classic MVC

Spring MVC has a *DispatcherServlet*. This is defined within the *web.xml* file, shown in Figure 2.3. This file is located in the WEB-INF folder. Its purpose is to load the application context from a servlet file, defined within this application as *member-servlet.xml*. An Application Context is an interface within the framework. This interface provides the configuration for the application.

Spring MVC provides a clear separation of roles. Each role, such as a View, Controller, Mapper, Validator, Model and View Resolver, can be encapsulated within a relevant object. It is a request driven framework designed around the *DispatcherServlet*.

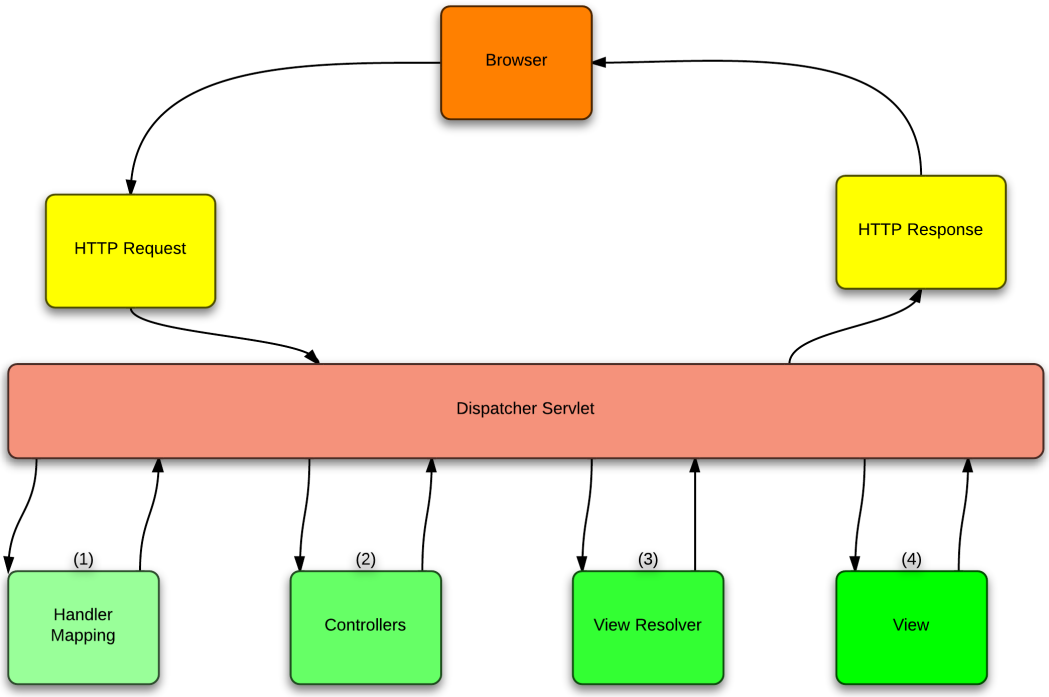

Figure 2.2: DispatcherServlet

The *DispatcherServlet* is responsible for handling HTTP requests from the browser. Once it receives these requests, it consults the HandlerMapper, which calls the appropriate Controller. A HandlerMapper takes a value, such as "/admin", and checks which controller handles this mapping. The Controller will take this request, and call the appropriate method, or methods, and interact with the Service layer of the application, if necessary. The View name is then returned to the *DispatcherServlet*, which in turn passes the value to the ViewResolver.

A ViewResolver provides a mapping between a *view name* and the *view*, that is to say, the web page requested. Once this view is finalised, the *DispatcherServlet* passed any model created within the Controller to the View, which is then rendered by the browser. This process is shown in Figure 5.2

```
1  <web-app
2  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xmlns="http://java.sun.com/xml/ns/javaee"
4  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5  version="2.5">
6    <display-name>monaleen-tennis</display-name>
7    <welcome-file-list>
8      <welcome-file>index.jsp</welcome-file>
9    </welcome-file-list>
10   <servlet>
11     <description></description>
```

```
12      <display-name>members</display-name>
13      <servlet-name>members</servlet-name>
14      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
15      <load-on-startup>1</load-on-startup>
16    </servlet>
17    <servlet-mapping>
18      <servlet-name>members</servlet-name>
19      <url-pattern>/</url-pattern>
20    </servlet-mapping>
21    <description>Database</description>
22    <resource-ref>
23      <description>DB Connection</description>
24      <res-ref-name>jdbc/mtc</res-ref-name>
25      <res-type>javax.sql.DataSource</res-type>
26      <res-auth>Container</res-auth>
27    </resource-ref>
28    <context-param>
29      <param-name>contextConfigLocation</param-name>
30      <param-value>
31          classpath:beans/dao-context.xml
32          classpath:beans/service-context.xml
33          classpath:beans/security-context.xml
34          </param-value>
35    </context-param>
36  </web-app>
```

Figure 2.3: Spring DispatcherServlet Configuration

Figure 2.3 is the configuration file needed for the *DispatcherServlet*. The file has a number of responsibilities within the application.

- Define DispatcherServlet
  - Line 9: This defines what class the DispatcherServlet implements.
- Define ApplicationContext
  - Line 17-19: This specifies the file that defines the application context of the application
- Define DataSource
  - Lines 22-27: This defines the reference to the database, and the DataSource class
- Define Context Config Location
  - Lines 28-35: This defines the files that contain the configuration for the DAO, Service and Security context files.

Table 2.1: DispatcherServlet Code

The *Controller*, within the Spring MVC framework, is designed for preparing a model with data, and selecting a view which will represent that data. This is done through the use of a *RequestMapping* annotation, which is discussed in further detail in Section 5 of this report.

The default *ViewResolver* within the Spring MVC is the InteralResourceViewResolver, depicted in Figure 2.4. This is defined with the *members-servlet.xml* file in the application, which is the structure of the *DispatcherServlet*. This class takes the value that is returned by a Controller, and passes a View to the DispatcherServlet. The browser can then render this view. It is important that any views, such as JSP files within the scope of this application, are stored within the *WEB-INF* folder. This is to ensure that the files are treated as an internal resource, and as such, are only accessible by the servlet, or the Controller classes within the application.

```
1 <bean id="jspViewResolver"
2       class="org.springframework.web.servlet.view.InternalResourceViewResolver">
3       <property name="prefix" value="/WEB-INF/jsps/"></property>
4       <property name="suffix" value=".jsp"></property>
5 </bean>
```

Figure 2.4: Default ViewResolver Configuration

This *ViewResolver* was not used within this application. Instead, Apache Tiles provides its own *ViewResolver*. This is due to the changes in how JSP pages are constructed and displayed by Apache Tiles, as discussed in the next section.

2.2.2 View Resolver

The framework that provided the *ViewResolver* for this application was Apache Tiles. This framework allows for the composition of a template for a JSP page. Apache Tiles allows the application developer to define page fragments, which are assembled into one page at run time, based on a template. This allows the application to reduce duplication of common page elements, such as headers, footers, link bar and advertising. The defined templates allow for a consistent look and feel across the application, and a change in one place, such as modifying a link in a header, will change across all Views within the application.

In order to use the Apache Tiles *ViewResolver*, it must be defined within the *DispatcherServlet* XML file, see Figure 2.5, in lieu of the default ViewResolver. This ViewResolver is part of the Spring Framework, and allows for interoperability between both the Spring and Apache Tiles frameworks. The reason that *TilesViewResolver* is used instead of *SimpleTilesListener* is for the support of JSTL within the JSP pages, as discussed within the Implementation.

```
<bean id="tilesViewResolver"
      class="org.springframework.web.servlet.view.tiles2.TilesViewResolver">
</bean>
```

Figure 2.5: Default ViewResolver Configuration

2.2.3 Application Server

The application server, or web server, used for this project was Tomcat 7. Tomcat is an open source project by Apache, which is a software implementation of Java Servlet and JavaServer Pages technologies. This application provides an environment in which Java code can run. Tomcat has a servlet-container called Catalina. A servlet container is the part of the web server that interacts with the servlets created by the application. It manages the life-cycle of the servlets, and is responsible for specifying the run time environment for the components within the application, and delivering this content. This includes security, transaction management, deployment and other services.

2.2.4 Project Management Tool

The project management tool used for this project was Maven. Maven was used within the scope of this project to manage the dependencies required by the web application. Maven came pre-installed and configured within the Spring Tool Suite IDE. Dependency Management can be handled one of two ways. Dependencies can be added using the GUI interface provided by an IDE, in this case, Spring Tool Suite. This GUI links to the repository located at http://mvnrepository.com/, and the user searches for the required files. Otherwise, the *pom.xml* file may be edited to define dependencies manually. Below is an example of the Apache Tiles v3.0.3 dependency.

```
<dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-core</artifactId>
        <version>3.0.3</version>
</dependency>
```

Figure 2.6: Dependency XML Structure for Maven

Maven also provided the archetype, or structure, for the application. It defined the folder structure for both the production and tests environments. It also sets up JUnit within the project to support unit testing throughout the development phase.

2.2.5 Database Model

The database framework used within this project was the open source framework, Hibernate. Hibernate is an Object/Relational Mapping [ORM] solution, and is concerned with relational databases, and more importantly for programmers, objects. Programmers generally "prefer to work with persistent data held (for the moment, anyway) in program objects, rather than use SQL directly for data access" (Bauer and King 2005).

Hibernate implements an Entity Data Model, and "sits between the object world of applications and the underlying database(s)" (Bauer and King 2005). Hibernate is derived from the Java Persistence API (JPA) and can be used in any environment that supports JPA, such as Java EE, Java SE and Enterprise applications.

It manages objects that need to be persisted, known as Entity Classes [EC], using annotations, which are detailed in subsequent sections. Each EC has a unique identifier "whose value is not important to the application apart from its use as an identifier" (Bauer and King 2005).

A rudimentary examination of Hibernate with Java Database Connectivity [JDBC] will be completed with regards towards the CRUD operations of each ORM database. This is examined within the Evaluation section of this report.

2.2.6 Integrated Development Environment

The IDE used for this project was Spring Tool Suite [STS], a modified version of the open source IDE, Eclipse. The advantages of using STS over Eclipse are the pre configured services within the application. Tomcat, Maven, egit, and the core Spring dependencies themselves come pre-packaged within the application. Java EE and web application support are also present. One clear advantage of using STS over Eclipse is that if a organisation were using Eclipse as as IDE, they could be running a variety of different versions of application servers or plug ins. A pre-packaged solution like STS reduces the risks of bugs being introduced, or not being able to reproduce bugs, on different development environments.

2.2.7 Source Control

Source Control is the management of changes to the source code of an application. In this day and age, it is not unusual for a program to be worked on by a number of different persons. In fact, it is more likely to be a globally dispersed team of programmers, so management of changes to the code base is very important. Essentially, it is a system that "provides facilities for storing, updating and retrieving all versions of modules, for controlling updating privileges, for identifying load modules by version number, and for recording who made each software change" (Rochkind 1975).

The source control system used within this project was GitHub, a free open source solution located at www.github.com. It provides integration with STS through the use of the *egit* plugin, as well as GUI and Shell user interfaces for a variety of user systems. It was also used to manage the different versions of the report you are reading now. Within the scope of GitHub, each project is called a repository.

GitHub also provides graphs and statistics about each repository, such as which days are the busiest for commits as shown in Figure 2.7, the growth of the code base over time and many more.

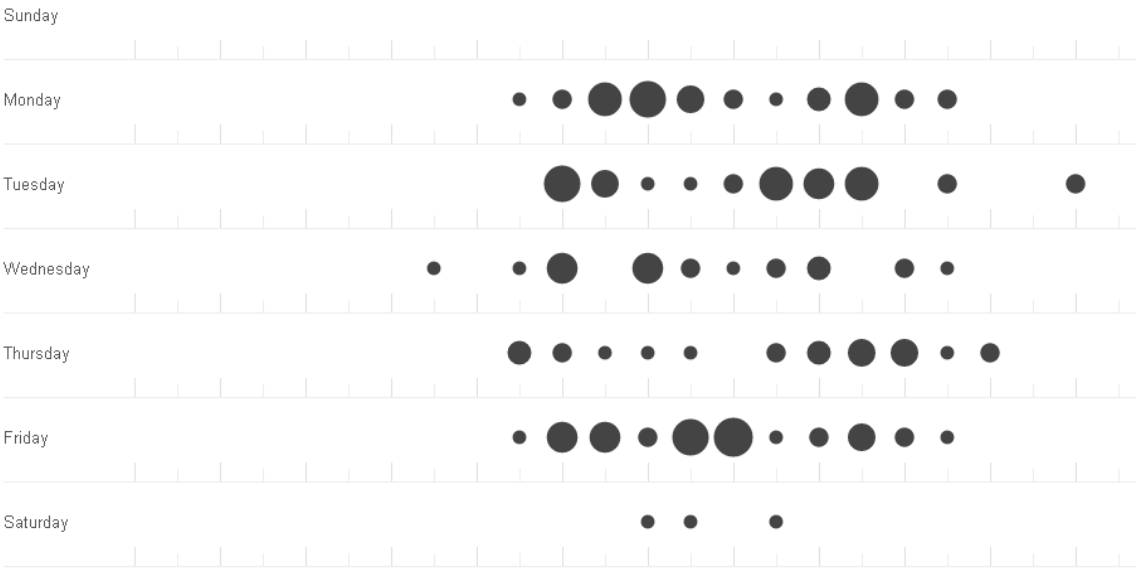

Figure 2.7: GitHub Visualisation of Commits/Day

2.2.8 Logging

Logging was used within the application to check both the flow of the application, as well as to pin point where certain lines of code were being called. The logging implementation used was log4j, an open source logging solution created by Apache. Log4j is designed with the possibility of enabling or disabling logging at run time. This is important in an application

may have thousands of logging instances within it. This would be difficult to remove from production code, and would increase the risk of introducing bugs were it to be attempted.

Logging can also be used to analyse usability and a "log will contain statistics about the frequency with which each user has used each feature in the program and the frequency with which various events of interest (such as error messages) have occurred." (Holzinger 2005)

2.3 Usability Studies

The important criteria that govern usability were looked at within this project. In doing this, two other websites were examined: the current website for the club, and a site that was recommended to me by a club member, Tralee Tennis club. For this task, usability studies by Jakob Nielson were examined.

2.3.1 Case Study: Monaleen GAA Tennis Club

The first site examined was the existing site for the club. This site is located at *http://www.monaleengaatennisclub.com/*. The current site is a basic HTML site with a CSS stylesheet, that occasionally uses a PHP script to facilitate users to register for tournaments. It uses a basic architecture stack consisting of these HTML pages coupled with an Apache HTTP Server. The reasons for this choice that the club just needed a presence on the internet, there was no need for much functionality at the time, and simplicity.

This architectural choice places a number of constraints on the potential growth of the site however. The definition of roles within the system is not possible. An example would be that committee members cannot add news stories themselves, but must contain the web-master to perform this action on their behalf. There is also no scope to add secure features to the site, such as a members only feature. Any introduction of these features would require a considerable overhaul of the existing architecture.

The only NFR looked at during the development of the site was availability, which is fulfilled by the choice of architecture and lightweight implementation.

(usability stuff will go here, leading onto sites that were admired by the club (ie Tralee))

2.3.2 Case Study: Tralee Tennis Club

Chapter 3

Requirements

3.1 Introduction

In eliciting possible requirements for the site, there was a discussion with three club members with varying backgrounds and experience within the club.

Name	Age Bracket	Club Role	Club Membership	Work Background
'Larry'	35 - 45	Committee Member	5 years	Senior Software Engineer
'Moe'	18 - 25	New Member	1 year	Graduate Software Engineer
'Curly'	55+	Senior Member	10+ years	Retired Public Servant

Table 3.1: Stakeholders for Requirements Elicitation

1. Book Court
    - Allow user to book a slot on a timetable for a court
2. Remove Booking
    - Allow user to book a slot on a timetable for a court
3. Register for Tournament
    - Allow users to register for a tournament
4. Contact Members
    - Easy way to contact all members
5. Member Directory
    - A list of all members, contact details, roles
6. News Section
    - Create new items to display for members
7. Members Area
    - A secure area that only members could access
8. Member Application
    - Automated registration, replace old paper form
9. Club Map
    - Directions to the club for new members and non-local visitors
10. Contact Details
    - Information on how to contact within the club for specific needs
11. Statistics
    - Such as games played, Win/Loss ratio

Table 3.2 refers to each numbered requirement, and whether it was brought up by a stakeholder during the elicitation process.

Name	1	2	3	4	5	6	7	8	9	10	11
'Larry'	N	N	Y	Y	Y	N	Y	Y	Y	Y	N
'Moe'	Y	Y	Y	Y	N	N	N	N	N	N	Y
'Curly'	Y	Y	N	N	N	N	N	Y	Y	N	N

Table 3.2: Requested Feature Breakdown

3.2 Method for Requirements

3.2.1 Storyboarding

At an early stage of the application, rough storyboards were prepared for the FYP presentation. These storyboards were used to demonstrate how a page, such as the timetable shown in Figure 3.1, would be displayed by the application.

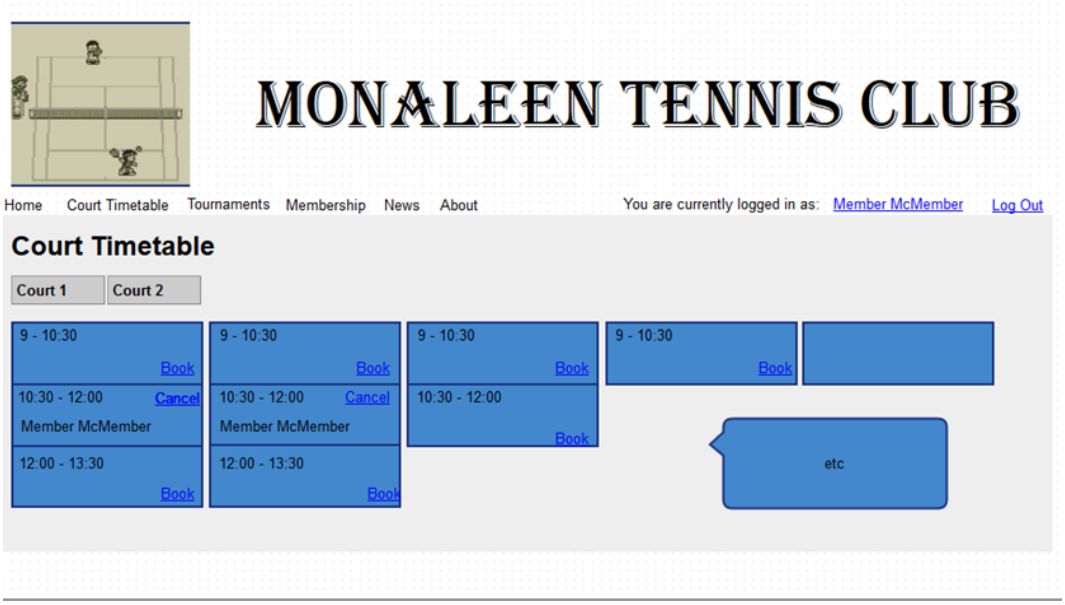

Figure 3.1: Timetable Storyboard, October 2013

The storyboarding visualised aspects of the site, and gave a rough idea of functionality that would be needed within the application.

3.3 Application

3.4 Functional Requirements

3.4.1 User

There are a number of actions that a user needs to be able to accomplish within the system. The user should be able to register their details within the system to create an account. Once the account is created, the user should be able to impact other site features, such as the timetable, based on their authority within the application.

1. Register
2. Login
3. Register for a tournament
4. View members registered for a tournament
5. View all member contact details
6. Book a slot on timetable
7. Remove a booking that they placed on a timetable
8. Report a no show for a booked slot

3.4.2 Timetable

The timetable is the core aspect of the application, and one that would be most likely to be used by all members, not just those involved competitively. While the regular member would only be concerned with the booking of slots, there are a number of requirements defined for use by the administrator in order to configure a relevant timetable not the club. The timetable needs to be flexible to allow the administrator full control at all stages.

1. Flexible
2. Edit individual slots
3. Define a template for a timetable
4. Reset timetable
5. Define look ahead for timetable (how many weeks in advance a user can see)
6. Delete timetable
7. Enable and disable timetable
8. Timetable analysis (No slots free, booked etc)

3.4.3 Tournament

3.4.4 Administration

From an administration point of view, there are many aspects that the application needs to support.

3.5 Use Cases

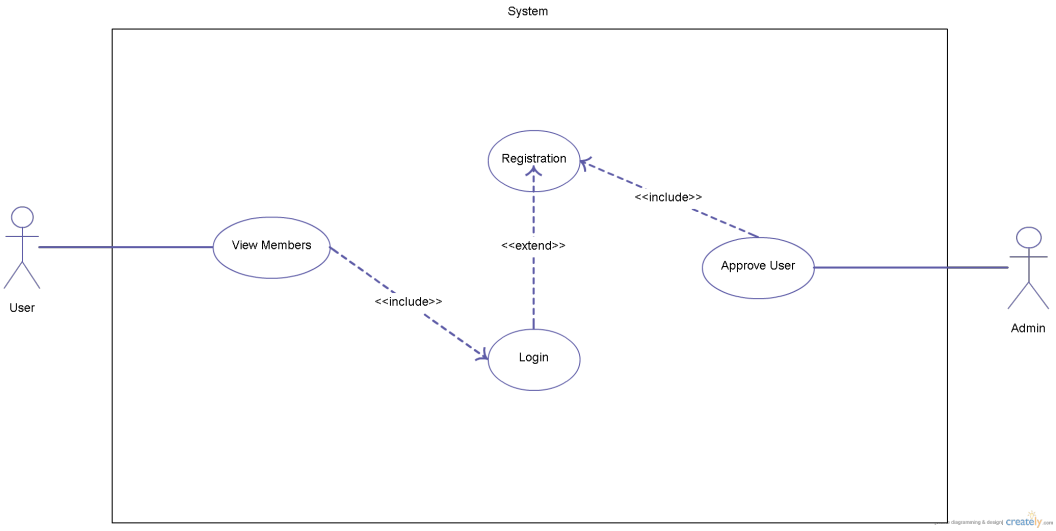

Figure 3.2: Use case for User login

Use Case 1	**View All Members**
Scope:	System-wide
Level:	User can view a list of all registered, and approved, members of the club
Primary Actor:	All Registered and Authenticated Users
Stakeholders and Interests:	• All Users: contact information for club members
Preconditions:	User is registered and approved
Postconditions:	User must be authenticated by the framework
Main Success Scenario:	

1. User logs in
2. User clicks on View Members
3. System displays member information

Extensions:

1.a Invalid login data:

   1. System shows failure message
   2. User returns to step 1

1.a User not approved:

   1. System shows failure message
   2. Admin is emailed about attempted access by unapproved member

---

Frequency of Occurrence: High

3.6 Non Functional Requirements

3.6.1 Security

3.6.2 Extensibility

3.6.3 Usability

Chapter 4

Design

4.1 Introduction

4.2 Key Features

4.2.1 Users

The User class was designed with the existing application form of Monaleen Tennis Club as a foundation. Spring handles security a number of ways. Firstly, it uses an *authority* hierarchy to separate different levels of users. For this web application, there were three main levels of authority, with one level containing three different branches.

Roles

- ROLE ADMIN
  - This refers to the main administration group. The group retains full rights across the web application
- ROLE COMMITTEE
  - This refers to the committee, as defined by the club themselves. This group with have the ability to perform some administrator privileges, but only those directly related to club activities, not site activities.
- ROLE MEMBER
  - The default user state. This group can perform actions such as booking slots in a timetable, registering for a tournament, and will have access to parts of the site unavailable to non-registered users.
- ROLE WARNING
  - A restriction placed upon a member. For example, a member who books time slots, but does not attend.
- ROLE SUSPEND
  - A further restriction placed upon a member.

4.2.2 Tournaments

Events

4.2.3 Timetable

Events

4.2.4 Administration

Logs

Analysis

4.2.5 News

4.2.6 Look and Feel

Chapter 5

Implementation and Testing

5.1 Introduction

This chapters deals with the implementation of the application, with the focus on the application entities, and how they were configured.

5.2 Application Entities

5.2.1 Users - Persistence

The *User* class represents every user account within the application. (Link to appendices showing class attributes). This section will focus on a regular user, how it is configured within the application in terms of bean definition and persistence.

Firstly, as shown in line one of Figure 5.1, the class needs to be configured as a *Component* for the application. This ensures that the Spring framework considers the User class as one for auto-detection, through the use of class path scanning and annotations prevalent within this application. The framework instantiates this bean, or object, automatically, without the developer having to use the *new* keyword.

```
1  @Component
2  @Entity
3  @Table(name="users")
4  public class User {
5          @Id
6          @GeneratedValue
7          int id;
8
9          @NotNull(groups={PersistenceValidationGroup.class,
               FormValidationGroup.class})
10         @Pattern(regexp=".+\\@.+\\..+", message="This does not appear to be a valid
               email address", groups={PersistenceValidationGroup.class,
               FormValidationGroup.class})
11         @Column(name="username")
12         String username;
13
14         @Size(min=5, max=45, message="Named must be between 5 and 45
               characters",groups={PersistenceValidationGroup.class,
               FormValidationGroup.class})
15         @Column(name="name")
16         String name;
17
18         @Column(name="password")
19         @Size(min=5, max=15, message="Password must be between 5 and 15
               characters", groups=FormValidationGroup.class)
20         String password;
21
22         @Column(name="gender")
23         String gender;
24
25         @Pattern(regexp="08[35679]([0-9]{7})", message="Number must be in the
               format 083, 085, 086, 087, 089 and 7 additional numbers eg 0851234567",
               groups={PersistenceValidationGroup.class, FormValidationGroup.class})
26         @Column(name="contact_num")
27         String contact_num;
28         //Class truncated. Some repetitive attributes omitted
29         //Getters and Setters below here.
```

Figure 5.1: User Class Definition and Configuration

The *Entity* and *Table* annotations of lines 2 and 3 respectively belong to the javax.persistance package. These annotations are used by Hibernate in order to manage and persist the class. The *@Table* annotation has a 'name' attribute that refers to the schema table the class maps to. There are two ways that an attribute can be assigned to a table column by Hibernate. Both methods are shown in Figure 5.1. An annotation may be placed on the attribute in order to specify a column name. Line 12 in Figure 5.1 shows the username attribute being mapped to the username column within the User database schema. The other way of specifying where an attribute should be persisted is to ensure that the attribute name

matches the column name within the table. This implicitly allows Hibernate to map a class, without having to explicitly define the mapping for the persistence framework.

The User class has a number of attribute constraints placed upon it. There are two types of constraints within this application: *FormValidationGroup* and *PersistenceValidationGroup*. These are interface classes with no attributes that serve as identifiers. As shown in lines 10, 14, 19 and 25 of Figure 5.1, an attribute may be constrained by one or more groups. An annotation, from the javax.validation.constraints, is applied to the attribute. The annotations used within this application were as detailed in Table 5.1.

Constraint Name	Description
NotNull [Line 9]	Ensures the value within the attribute does not have a null value
Pattern [Line 10]	Ensures the value within the attribute conforms to a regular expression
@Size - Min [Line 14]	Ensures the value within the attribute has a minimum length
@Size - Max [Line 14]	Ensures the value within the attribute has a maximum length

Table 5.1: Class Constraints

These validation package interfaces provide a *groups* attribute, which is an array of objects. The *FormValidationGroup* and *PersistenceValidationGroup* are passed to this attribute. These groups allow attributes to have different constraints at different stages in the application. When using this attribute within the application, such as the creation of a user within a form, the Controller classes apply the validation to the user input and persisted data. The reason for having two groups of validation within this application is due to security. In every application, it is advisable to perform encryption on sensitive data, such as passwords. Within the scope of this application, user passwords were defined as being between 5 and 15 characters long, with no restriction to the content of the password. This application flow on taking in user input and persisting it is demonstrated below in Figure **??**.

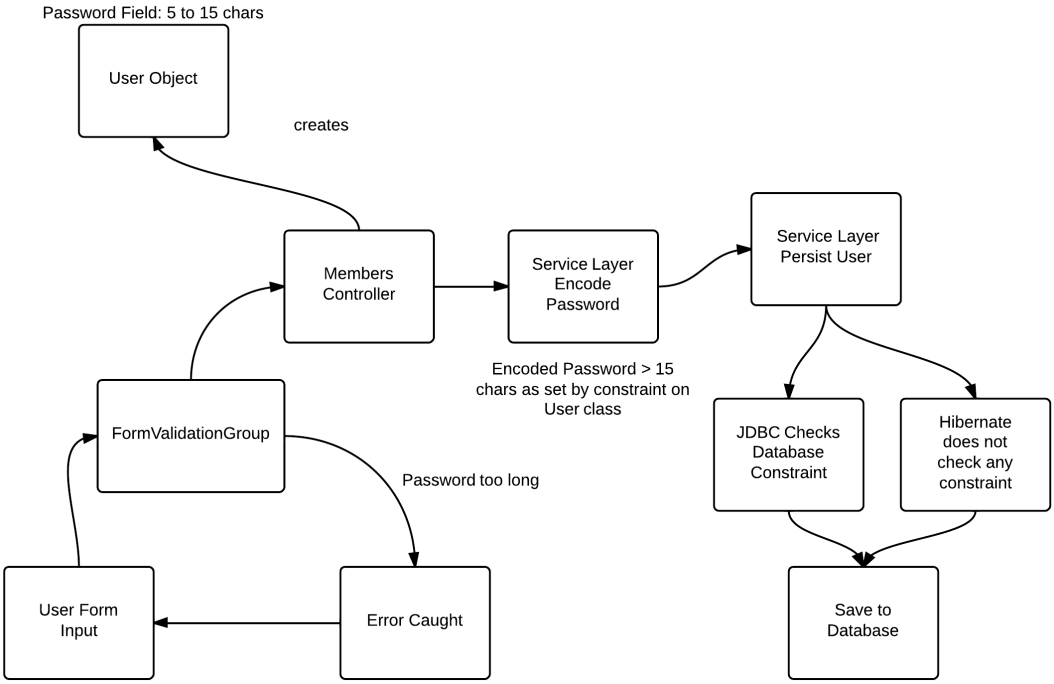

Figure 5.2: DispatcherServlet

For example, a user password with 8 characters would pass form validation with no issues. When the PasswordEncoder bean is applied to the string prior to persistence, it will result in a value like *'acb172137243c0b931321d7645dc31c2efb8346385ae0547d11b1d8de333215b'*, a value much longer than 15 characters. This will cause a failure with Hibernate persistence. This is because Hibernate works at a class level, and does take note of the constraints placed upon the class, while JDBC does not. The constraints that JDBC takes note of are those taken directly from the database itself. In this application, passing form validation is sufficient, as there are security annotations placed on the Service classes that manage data persistence. An example of how the controller handles validation is shown later in Figure 5.6.

Alongside class configuration, it is also necessary to configure Hibernate to scan the packages that contain entities, as detailed in Figure 5.3. This is done through the creation of a sessionFactory bean, which uses the AnnotationSessionFactoryBean class. This bean is responsible for the creation of session instances within the application, though each application usually only has one session. It is an immutable object, and cannot be changed once it is created, so proper configuration of classes to facilitate object-relational mapping is important. The Session object created by the factory is responsible for creating the connection between the application and the database.

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
<property name="dataSource" ref="dataSource"></property>
<property name="hibernateProperties">
        <props>
                <prop
                    key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
        </props>
</property>
<property name="packagesToScan">
        <list>
                <value>users</value>
                <!-- user refers to a package that contains user related classes --!>
        </list>
</property>
</bean>
```

Figure 5.3: Hibernate SessionFactory Configuration

The HibernateProperties property defined in Line 4 refers to the SQL dialect used by Hibernate. This dialect provides the functionality to use human readable expressions to define query statements. Hibernate does not use SQL to structure these statement. It uses a variant called Hibernate Query Language [HQL]. HQL syntax is based has its basis in object oriented programming. The class itself defines the table, not the syntax. The different in implementation is shown in Figure 5.4.

```
SQL [JDBC]: jdbc.query("select * from users", new UserRowMapper());
// UserRowMapper is a class that maps rows to objects

HQL [Hibernate]: session().createQuery("from User").list;
```

Figure 5.4: HQL-SQL Comparison

The Form Validation is provided by the Spring Security file, which will be examined in detail in the Administration Implementation section on page 42. As discussed previously, there are a number of validation constraints placed on the *User* class. Spring provides a facility to ensure these constraints are enforced, and to also provide a positive user experience. It does this through the use of a BindingResult object. This object holds a record of any errors from the form that the user populates. The controller that deals with the form will check the BindingResult object for errors, and can respond appropriately. In order for this to work, both the Controller and the form, as depicted in Figures 5.5 and 5.6, need to be defined clearly. The form needs to be created using the Spring Framework form tag library, and errors needs to be specified for each input within the form.

```
<!-- Excerpt from the User registration form. Formatting removed for clarity --!>
<sf:form id="details" method="post"
    action="${pageContext.request.contextPath}/register" commandName="member">
Name <sf:input name = "name" path="name" type="text"/>
<sf:errors path="name" cssClass="error"></sf:errors>
Password <sf:input id="password" name = "password" path="password" type="password"/>
<sf:errors path="password" cssClass="error"></sf:errors>
</sf:form>
```

Figure 5.5: User Registration Form

The form structure, illustrated partially in Figure 5.5, contains a number of important attributes. The form used is an extended version of the HTML ¡form¿ object. The Spring MVC version adds extra functionality such as mapping to objects within the controller classes and error validation checking. The *commandName* is a variable within the form that contains the information within the form. This temporarily persists data between the form and the controller. This benefits the user as the application repopulates valid fields in the form should an error be made. The *sf:error* tags allow the controller to identify specific errors within the form. These also reference attributes within the class the form is linked with. In this example, the *User* class is being created with this form.

Figure **??** Line 4 shows the declaration of a BindingResult object. This object holds the results of the form validation. Line 5 declares that if the result object has any errors, the controller returns to the form, where the *sf:error* tags will highlight the error message relation to any attributes that broke their constraints. Lines 9 through 13 show home the application ensures that the primary key remains unique.

```
1  //Method from the MembersController class
2  //This method is responsible for validating the form that users complete to
       register.
3  @RequestMapping(value = "/register", method = RequestMethod.POST)
4  public String doRegister(Model model,
5  @Validated(FormValidationGroup.class) @ModelAttribute("member") User member,
       BindingResult result) {
6  if (result.hasErrors()) {
7          return "creatememberss"; // if the result has errors, go back to create page
8  }
9  if (userService.exists(member.getUsername())) {
10         result.rejectValue("username", "Duplicate Key",
11         "This email address has already been used");
12         return "creatememberss";
13         //if the email address already exists, return with this message.
14 }
15         else {
16                 try {
17                         member.setAuthority("ROLE_MEMBER");
18                         userService.create(member);
19                         return "registerSuccess";
20                         //successful creation of member
21                         } catch (Exception e) {
22                                 return "error";
23                         }
24                 }
25 }
```

Figure 5.6: User Registration Controller

Line 13 in Figure 5.6 shows the use of a *ModelAttribute*. A *ModelAttriute* is a property of the model that is supplied by Spring MVC to the controller. This object is created using the data mapped from the preceding form. This object is being validated using the *FormValidationGroup* as specified using the *Validated* annotation.

Within this application, the Service layer is responsible for the Controller communicating with the DAO layer to persist objects like the User class. Since Hibernate is configured at a class level, in relation to attributes and column names, there is no need for any INSERT or UPDATE statements. The current session, see Figure 5.7 is returned to the DAO object via the configured bean, and the necessary methods, such as save() and delete(), are called upon it. An object must be passed to the *save()* method of the current sessionFactory object, detailed in 5.3.

```
public Session session(){
        logger.info("Session Factory returning current session.....");
        return sessionFactory.getCurrentSession();
}
```

Figure 5.7: UserDAO getSession()

In the case of the User object, the password needs to be encoded prior to the object being persisted by the session(). In order to encode password, a bean responsible for the encoding must be defined within the application context, illustrated in Figure 5.8. Spring provides a class that allows passwords to be encoded, and the bean for this class is defined within a security-context file.

```
<bean id="passwordEncoder"
class="org.springframework.security.crypto.password.StandardPasswordEncoder">
</bean>
```

Figure 5.8: Password Encoder Definition

This Spring defined class provides an implementation for encoding data using SHA-256 hashing with 1024 iterations, with a random 8 byte salt value. This object then calls *encode()* on the value passed from the form filled by the user. As a result, the actual password is never stored in the database, just an encrypted form of it, as shown in Figure 5.9. Once the password is encoded (Line 4, 5.9), the session() object can save the object. Due to the class configuration, there is no need to specify any database schema information within the DAO classes.

```
@Transactional
public void createUser(User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        session().save(user);
}
```

Figure 5.9: Persisting User object with encoded password

5.2.2 Timetable - Persistence Changes

One of the most difficult features to implement within the application was the Timetable. While the goal was to create a timetable suitable for Monaleen Tennis Club, it was desirable that the timetable retain the portability and not rely on hard coded attributes. One issue this raised was how to handle a varying number of slots in each day. If a user could define 10 slots a day, how would be store this in such a way that a user could also define a timetable with 20 slots a day? Hibernate was able to facilitate this design with considerably less input from a developer.

The solution implemented in this application was to use List objects to store the values for each day. There would be seven lists in the Timetable class, one for each day, as per Figure 5.10.

```
@Entity
@Component
@Table(name = "timetable")
public class MonaleenTTV1 implements Timetable {
	@Id
	@GeneratedValue
	private int id;
	/**
	* Other Attributes Here
	**/

	@ElementCollection
	@CollectionTable (name = "monday", joinColumns=@JoinColumn(name="id"))
	private List<String> monday;

	@ElementCollection
	@CollectionTable (name = "tuesday", joinColumns=@JoinColumn(name="id"))
	private List<String> tuesday;

	@ElementCollection
	@CollectionTable (name = "wednesday", joinColumns=@JoinColumn(name="id"))
	private List<String> wednesday;

	@ElementCollection
	@CollectionTable (name = "thursday", joinColumns=@JoinColumn(name="id"))
	private List<String> thursday;

	@ElementCollection
	@CollectionTable (name = "friday", joinColumns=@JoinColumn(name="id"))
	private List<String> friday;

	@ElementCollection
	@CollectionTable (name = "saturday", joinColumns=@JoinColumn(name="id"))
	private List<String> saturday;

	@ElementCollection
	@CollectionTable (name = "sunday", joinColumns=@JoinColumn(name="id"))
	private List<String> sunday;

	//getters and setters
}
```

Figure 5.10: Timetable Class List Configuration

pid	monday
1	Free Court
1	Free Court
2	Tournament
2	Free Court
3	User Booking
3	Free Court
4	Training
4	Free Court

Table 5.3: Timetable Storage Table for Monday Collection

This results in the Timetable object being made up of 8 database tables. The core of the class is stored in the 'Timetable' database table. This table contains the primary keys and all other attributes, such as name, number of slots, timetable series. The seven other tables each represent a collection within the Timetable class. Each of these tables has a non unique, foreign key that ties it back to the central table. This is illustrated in Figure 5.10 with use of the *CollectionTable* annotation. This annotation defines the table that each collection belongs to. It also specifies a column for the HQL and SQL JOIN statement in the *JoinColumn* annotation. For example, if we were to create a Timetable with 10 slots, with the primary key being 1, each of the collection tables would have 10 entries. Each of these entries would have an id of 1, to match the primary key in the core table, plus a value, such as 'Free Court'. The position in the table corresponds to the position within the collection class.

The following example, in Figure 5.2, shows a timetable configured with two slots. Upon creation, a series is created for each timetable. Each series contains 52 timetables, one for each week. The currently displayed timetable is determined by its position in the series in relation to the current week of the year, as shown by the Java Calendar class. The 'preview' attribute determines how far ahead the application will allow the user to browse in a timetable series. By default, the application displays the current weeks timetable, and does not allow a user to go backwards. If the current week, as defined by the Java Calendar class, is 13, the user will be able to see weeks 14, 15 and 16, owing to the preview with a value of 3. In Table 5.2, the next three weeks of timetables are available for viewing and booking. This allows the administrator to dynamically restrict how far in advance users can book slots.

pid	name	slots	startTime	endTime	enabled	preview	series	total
1	Court One Week 1	2	8	22	1	3	1	52
2	Court One Week 2	2	8	22	0	3	1	52
3	Court One Week 3	2	8	22	0	3	1	52
4	Court One Week 4	2	8	22	0	3	1	52

Table 5.2: Timetable Core Database Table

While the table structure of the *User* and *Timetable* differ considerably, since Hibernate deals with objects, that have table structures defined within their classes, there is no change to the way the Timetable is saved and updated, as per Figure **??**. Using JDBC, there would have been 8 SQL queries to save to each of the 8 table, increasing the risk of bugs within the application, or invalid data being saved or retrieved.

```
1 @Transactional
2 public void createTimetable(Timetable t) {
3         session().save(t);
4 }
```

Figure 5.11: Hibernate Save Timetable

The timetable is displayed in the application as shown in Figure 5.12.

Time	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
0 to 1	Chris O'Brien Report User	Chris O'Brien Report User	ChanMan Report User	Book	Book	ChanMan Report User	Book
1 to 2	Book	Book	Book	ChanMan Report User	Book	Book	Book

Figure 5.12: Timetable display within application

5.2.3 Tournaments and Timetable Events

The third major area of the site were the *Tournament* and *Event* classes. The structure and configuration of these classes are not different from the *User* or *Timetable* classes. In the context of this report and application, an Event is any object that can populate a field within a Timetable class.

Events

Within the scope of this application, an *Event* is any item that can be scheduled within the Timetable object. As with the *Tournament* class, the *Event* class is configured much like the *User* and *Timetable* classes. One event that is required by all versions of the Timetable is the 'Free Court' event. This event is created by default (see Figure 5.13) either the first time an administrator attempts to create an Event, or to create a Timetable. An Event has two attributes: a name and an author. In the case of a system event, the name will be the login of the person who booked the timetable slot, while the author is defined as *BOOKING SYSTEM*. For an administrator event, the name is defined by that administrator, while the author is listed as the administrator who created the event. Example events for Monaleen Tennis would be:

- Training Sessions
- Club Social Events

- Ladies Tennis
- Recurring Tournaments
- General Club Activities

```
1  @RequestMapping(value = "/saveEvent", method = RequestMethod.POST)
2  public String saveEvent(Model model, @ModelAttribute("event") Event e,
3          BindingResult result) {
4          if (eventService.getEventById(1).equals(null)){
5                  e.setName("Free Court");
6                  e.setAuthor(userService.emailToName(SecurityContextHolder
7                  .getContext().getAuthentication().getName()));
8                  e.setEnabled(true);
9                  eventService.createEvent(e); // see Line 29-32 for creation of Event.
10         }
11         logger.info("Event Save Method....");
12         if (result.hasErrors()) {
13                 return "createEvent";
14         }
15         if (eventService.exists(e.getName())) {
16                 result.rejectValue("name", "Duplicate Key",
17                 "This Event Name has already been used.");
18                 return "createEvent";
19         } else {
20                 e.setAuthor(userService.emailToName(SecurityContextHolder
21                 .getContext().getAuthentication().getName()));
22                 e.setEnabled(false);
23                 eventService.createEvent(e);
24                 return viewEvent(model);
25         }
26 }
27
28 //Event DAO
29 public void createNewEvent(I_Event e){
30         logger.info("Creating new event....");
31         session().save(e);
32 }
```

Figure 5.13: Event Creation

Lines 4 through 10 are necessary to ensure no NullPointerExceptions are introduced when creating a timetable. If no events exist within the system either the first time a timetable is created or, as illustrated above in Figure 5.13, an event is created, the system will create a *Free Court* event which is needed for the timetable to function correctly.

In order for an event to show up as an option when creating, or editing, a timetable, it must be enabled, as shown in Figure 5.14. This implementation allows the administrator to pre-create events for later use without overcrowding the timetable.

```
@RequestMapping("/changeEventStatus")
public String changeEventStatus(Model model, HttpServletRequest request) {
        Event e = (Event) eventService.getEventById(Integer.valueOf(request
        .getParameter("eventID")));
        if (e.isEnabled()) {
                e.setEnabled(false);
                eventService.updateEvent(e);
                return viewEvent(model);
        } else {
                e.setEnabled(true);
                eventService.updateEvent(e);
                return viewEvent(model);
        }
}
```

Figure 5.14: Change Event Status

Tournament

The *Tournament* class is configured similar to the *Timetable* class, in that it has a secondary table which contains a list of members who have registered for the tournament. The key area for tournament implementation was for the application to differentiate between the various states of the tournament, such as registration being enabled, yet the tournament not having started yet. It was also important that the tournament could link up with the Timetable system, in order to book slots to play the tournament. In this regards, a tournament was required to manage an Event object, see Figure 5.15, and to ensure that no ambiguity was present when dealing with multiple tournaments and possible multiple timetables. It was also necessary for a Tournament object to manage its own event in order to minimise both the risk of ambiguity and administration involvement.

```
@RequestMapping(value = "/registerTournament", method = RequestMethod.POST)
public String doCreateTournament(
        Model model, BindingResult result,
        @Validated(FormValidationGroup.class) @ModelAttribute("tournament")
            Tournament t) {

        if (result.hasErrors()) {
                return "createTournament";
        }
        if (tournamentService.exists(t.getTournamentName())){
                if (eventService.exists(t.getTournamentName())){
                result.rejectValue("tournamentName", "Duplicate Key",
                "An event of this name already exists");
                return "createTournament";
        }
        result.rejectValue("tournamentName", "Duplicate Key",
                "A tournament of this name already exists");
```

```
                return "createTournament";
                }
        else {
                try {
                        tournamentService.create(t);
                        eventCreation(t);
                        logger.info("Tournament Created");
                        return "tournamentSuccess";
                } catch (Exception e) {
                return "error";
        }
}
public void eventCreation(Tournament t){
        Event e = new Event();
        e.setName(t.getTournamentName());
        e.setAuthor(userService.emailToName(SecurityContextHolder.getContext()
        .getAuthentication().getName()));
        eventService.createEvent(e);
}
```

Figure 5.15: Tournament Creation

Deleting a tournament needed similar logic, as displayed in Figure 5.16. Similar to the *User* class, the Tournament name, much like the username, is used as a primary key, and can be recycled once a tournament is deleted.

```
        @RequestMapping("/confirmDelete")
        public String deleteTournament(Model model, HttpServletRequest request){
                Tournament t = tournamentService.getTournamentById(request
                                .getParameter("tournamentID"));
                tournamentService.deleteTournament(t);
                eventService.deleteEvent(eventService.getEventIdByName(t.getTournamentName()));
                model.addAttribute("tour", tournamentService.getAllTournaments());
                return "deleteTournament";
        }
```

Figure 5.16: Delete Tournament

Due to constraints on time within the final year project, it wasn't possible to fully flesh out this area of the site, though it was implemented in such a way to allow for extensibility such as an interface for the sorting of registered members into teams. This interface would allow the specification of business logic to allow the system to, for example, sort tournament teams by experience and ranking.

5.2.4 Administration - Security and Session Management

The *Administration* section of the application deals with the implementation, and configuration, of the security and session management aspects of the application. An administrator has the same structure as a *User* and is defined by the *Role* it has, as seen previously in Figure **??**. These roles are configured through Spring Security, in which a specific database schema must be adhered to. By default, there should be two tables: *users* and *authorities*, with a foreign key constraint. In this application, as shown in Line 3, Figure 5.17, this was modified to keep the user data within the same table.

```
1 <security:authentication-provider>
2 <security:jdbc-user-service data-source-ref="dataSource"
3 id="jdbcUserService" authorities-by-username-query="select username, authority from
      users where binary username = ?" />
4 <security:password-encoder ref="passwordEncoder"></security:password-encoder>
5 </security:authentication-provider>
```

Figure 5.17: Spring Role Configuration

The security within the application is controlled by the *security-context.xml* file, which used Expression Based Access Control [EBAC] in order to restrict site access to relevant roles. EBAC allows complicated boolean logic to be encapsulated within a single expression, such as whether a user is authenticated or not. The base class used within this application is *SecurityExpressionRoot*. These expressions are contained within the *security-context.xml* file, as seen in Figure 5.18. This configuration file is partly responsible for controller access to the mappings within the application.

```
1 <security:http use-expressions="true">
2 <security:intercept-url pattern="/static/**" access="permitAll" />
3 <security:intercept-url pattern="/images/**" access="permitAll" />
4 <security:intercept-url pattern="/creatememberes" access="permitAll" />
5 <security:intercept-url pattern="/tournamentSuccess"
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_COMMITTEE')" />
6 <security:intercept-url pattern="/createNews" access="hasAnyRole('ROLE_ADMIN',
      'ROLE_COMMITTEE')" />
7 <security:intercept-url pattern="/members" access="isAuthenticated()" />
8 </security:http>
```

Figure 5.18: Security Context File Excerpt

An issue that arises with this implement ion is that all resources, such as images, would automatically be blocked from appearing on the site unless explicitly defined. In the case of images, this would be very time consuming, especially when new images are added to the application frequently. In order to overcome this, Lines 2 and 3 are implemented.

This syntax specifies that all files and folders within the *static* and *images* folder are to be granted access to all pages within the application. Site access can also be granted on an authentication basis, rather than a role based system, and this is implemented as show in Line 7 of Figure 5.18.

In addition to the expression level security, service-level security annotations are also in place. This gives the application an extra layer of security, as it restricts the use of methods within the service layer to defined roles. It is possible to use either expression level security or service level security independently. This is configured within the *security-context.xml* file, Figure 5.19, and is implemented, as shown in Figure 5.20, within the service layer. These annotation use the same roles as defined previously within the application.

```
1 <security:global-method-security
      secured-annotations="enabled"></security:global-method-security>
```

Figure 5.19: Security Context Service Layer Annotation Configuration

```
1  @Service("timetableService")
2  public class TimetableService {
3
4  private TimetableDAO timetableDAO;
5
6          @Autowired
7          public void setTimetableDAO(TimetableDAO timetableDAO) {
8                  this.timetableDAO = timetableDAO;
9          }
10
11         @Secured("ROLE_ADMIN")
12         public void create(Timetable t){
13                 timetableDAO.createTimetable(t);
14         }
15
16         @Secured({"ROLE_ADMIN", "ROLE_MEMBER", "ROLE_COMMITTEE", "ROLE_WARNING",
               "ROLE_SUSPEND"})
17         public void update(Timetable t){
18                 timetableDAO.updateTimetable(t);
19         }
20
21         public List<Timetable> getAllTimetables(){
22                 return timetableDAO.listTimetables();
23         }
24 }
```

Figure 5.20: Security Context Service Layer Annotation Implementation

In the implementation shown above, there are some items to note: the *Service* and *Autowired* annotations. The *Service* annotation allows implementation classes to be auto-detected through class-path scanning. The package that contains the Service classes is defined within another XML file, *service-context.xml* within this application, Figure 5.21. This file configures Spring to look for Service classes via annotations, and where to look for them.

```
1 <context:annotation-config></context:annotation-config>
2 <context:component-scan base-package="service"></context:component-scan>
```

Figure 5.21: Service Context Configuration

The *Autowired* annotation is used within the framework to mark a constructor or setter. Spring then passes the needed dependencies into the application with its dependency injection facilities.

5.3 Model View Controller

5.3.1 Controller Layer

A Controller is a class within the application that creates and modifies a model, and passes it onto a View. A View will also pass information to a Controller, which will communicate with the Service layer to persist any relevant data within the application.

A Controller is annotated, as illustrated in Figure 5.22, Line 1. This allows the DispatcherServlet to identify the controllers and inject them into the application. This behaviour is displayed in Figure 5.23, which shows the DispatcherServlet being configured to scan the controllers package looking for annotations.

```
1 @Controller
2 public class TimetableController {
3       //.........
4 }
```

Figure 5.22: Service Context Configuration

```
1 <context:component-scan base-package="controllers, email"></context:component-scan>
2 <mvc:annotation-driven></mvc:annotation-driven>
```

Figure 5.23: Service Context Configuration

In order for the application to choose the correct method within a specific controller, it is necessary to define a mapping. The *RequestMapping* annotation is used for this purpose. This annotation maps a web request onto a handler method within a controller, as depicted in Figure 5.24. It is also important, within the scope of the Spring Security configuration, to ensure that this mapping has access rights defined within the application or access will be denied.

```
1 @RequestMapping("/")
2 public String showHome(Model model) {
3         logger.info("Showing Home Page....");
4         News news = newsService.getLatestStory();
5         model.addAttribute("newsHeader", news.getSummary());
6         model.addAttribute("newsContent", news.getContent());
7         return "index";
8 }
```

Figure 5.24: Request Mapping

This mapping responds to a request with the value '/', Line 1. On Tomcat servers, this the home page, as the URL will be *localhost:8080/¡application-name¿/*. In this example, the latest news story is retrieved on Line 4, and values from it are added to the model displayed on the returned view.

Any value specified after *localhost:8080/¡application-name¿/* will search for a mapping within all controllers in the application. If a mapping does not exist, a resource not found will be displayed. This is default behaeviour for a denied access attempt. In order to ensure the site can deal with invalid requests, an access denied handler can be defined within the Spring Security context file, as depicted in Figure 5.26. This allows a mapping to be specified for invalid requests,

```
1 <security:access-denied-handler error-page="/denied" />
```

Figure 5.25: Access denied Handler

In this example, the *denied* mapping, shown in Figure **??** is called, and the corresponding view displayed.

```
1 @RequestMapping("/denied")
2 public String showDeny(Model model){
3         model.addAttribute("message", "The page you are looking for either does not
              exist, or you do not have access. Please contact an administrator if
              you believe this is an error.");
4 return "error";
5 }
6
```

```
7  //Error Page
8
9  <center>${message}</center>
```

Figure 5.26: Error Page Mapping

Line 3 shows a message being added to a model, which is added to a standard error page. This message will change depending on what error is being caught by the application. The message is then displayed by the 'error' View for the user, and the site appearance is not disrupted by a generic error page.

5.3.2 Models within the Spring MVC Framework

A model is a map containing a key-value pair that can be modified by a Controller and displayed by a View. A model is implicitly created by the framework by including a reference to it within the method signature in a controller, as illustrated in Figure 5.27. The below example adds two lists to a model: a list of disabled events and a list of enabled events, shown in Lines 5 and 6. The Controller then returns the value *viewEvents* to the ViewResolver. The ViewResolver displays the relevant page that displays the information held within the model for the user, as depicted in Figure 5.28

```
1  @RequestMapping("/viewEvents")
2  public String viewEvent(Model model) {
3          List<Event> eventsDisabled = eventService.listDisabledEvents();
4          List<Event> eventsEnabled = eventService.listEnabledEvents();
5          model.addAttribute("eventsEnabled", eventsEnabled);
6          model.addAttribute("eventsDisabled", eventsDisabled);
7      return "viewEvents";
8          }
```

Figure 5.27: Model Definition

```
<c:if test="${not empty eventsEnabled }">
<table class="members" align="center">
<tr><th>Enabled Events</th></tr>
<tr>
<th>ID</th><th>Name</th><th>Author</th><th>Action</th></tr>
<c:forEach var="row" items="${eventsEnabled}">
<sf:form method="post"
action="${pageContext.request.contextPath}/changeEventStatus"
commandName="eventsEnabled">
<tr>
        <td><input type="hidden" value="${row.id}" name="eventID" />${row.id}</td>
        <td>${row.name}</td>
        <td>${row.author}</td>
        <td><input value="Disable" type="submit" name="${row.id}" />
</sf:form>
</c:forEach>
</table>
<center><font color="red">${message }</font></center>
</c:if>
```

Figure 5.28: JSP View of model

The page contains a conditional loop, Line 1, that checks if the model attribute, eventsEnabled, is empty. In the event that it is not, it will create a table to display the information to the user, as shown in Lines 2 through 17. The *c:forEach* tag is the JSTL version of a for loop, which will be looked at in a later section. Line 18 is a message, that will display if the value is not null. This is used to display a confirmation or error message to the user when an action is performed on an event. An example would be an attempt to disable the *Free Court* event. This will display a message, shown below in Figure 5.29, that the action cannot be performed.

```
if (e.getName().equalsIgnoreCase("Free Court")){
        model.addAttribute("message", "You cannot modify the Free Court event");
        return viewEvent(model);
}
```

Figure 5.29: User message added to model

5.3.3 View Layer

Apache Tiles Configuration and Implementation

Apache Tiles is configured within the web application core XML file. There are two classes that the configuration is concerned with: TilesViewResolver and TileConfigurer. Both are declared as beans within the configuration file and automatically created when the application is launched. The primary function of the ViewResolver is to take in a String value,

and return the relevant *RequestMapping* value within the application. These mappings are defined within the Controller classes of the application. The TilesConfigurer object, see Figure 5.30, takes one parameter: a location of the template that the default tile will use. The default tile will then be used by other pages as a template.

```
<bean id="tilesViewResolver"
        class="org.springframework.web.servlet.view.tiles2.TilesViewResolver">
</bean>

<bean id="tilesConfig"
        class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
        <property name="definitions">
        <list>
                <value>/WEB-INF/layout/default.xml</value>
        </list>
        </property>
</bean>
```

Figure 5.30: Apache Tiles Configuration

The default tile consists of a number of sections identified by a specific tag. These tags correspond to values within the tile layout configuration file. Using a version of inheritance, these can be overwritten and replaced with other pages in order to change the content of a page, while maintaining cohesion across the design of the application.

The following examples shows the implementation within the configuration file. The first section of code is the overall template. This specifies the default values that make up a JSP page within the application. The second segment of code is the the definition for the initial home page for the web application. By the inclusion of the *extends="users.base"* within the definition tags, it is defining the index as a sub class of the users.base definition. Consequently, it is possible to override any of the attributes within the users.base definition. In this example, the title and content of the default page are being overridden with different values in order construct a more suitable page. The header, links and footer however remain the same, and will do so will all pages following this format, as shown in Figure 5.31.

```
<definition name="users.base" template="/WEB-INF/templates/default.jsp">
        <put-attribute name="title" value="Monaleen Tennis Club - Default
            Template"></put-attribute>
        <put-attribute name="header"
            value="/WEB-INF/tiles/header.jsp"></put-attribute>
        <put-attribute name="links"
            value="/WEB-INF/tiles/links.jsp"></put-attribute>
        <put-attribute name="content"
            value="/WEB-INF/tiles/content.jsp"></put-attribute>
        <put-attribute name="footer"
            value="/WEB-INF/tiles/footer.jsp"></put-attribute>
</definition>

<definition name="index" extends="users.base">
        <put-attribute name="title" value="Monaleen Tennis Club - Home
            Page"></put-attribute>
        <put-attribute name="content"
            value="/WEB-INF/tiles/index.jsp"></put-attribute>
</definition>

<definition name="admin" extends="users.base">
        <put-attribute name="title" value="Monaleen Tennis Club -
            Admin"></put-attribute>
        <put-attribute name="content"
            value="/WEB-INF/tiles/admin.jsp"></put-attribute>
</definition>
```

Figure 5.31: Apache Tiles Configuration

JSTL

JSTL is used within the application to manage how information was displayed. It was preferred, during the development of the application, that all of the logic be handled at the Controller level, and that the JSP pages would resolve the models passed to it into the view seen by the user. It was not desirable for the pages to contain JSP directives, or to use the implicit objects contained within JSP pages.

The main tags used within the application were the JSTL Core tags. These tags allow the usage of conditional statements and the definition of parameters within the JSP page. In order to use this technology, the relevant jar must be made available in the build path or within the Maven dependencies of the project. A declaration, as shown in Figure 5.32 must be included in all JSP pages that wish to make use of the tags also.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

Figure 5.32: JSTL Tag Library Declaration

Within the application, the controller will create a model and pass it to the JSP page. The page uses the JSTL tags to manage and display relevant information from the model, as shown in Figure **??**, and user actions based on the information contained within. The example below is taken from the Timetable display page, Figure 5.33 from the application.

```
@RequestMapping(value = "/gotoCourt", method = RequestMethod.POST)
public String chooseCourt(Model model, HttpServletRequest request) {
        //abbreviated method to display court, logic removed
        //highlighting the attributes within the model
        model = addDateToTimetable(model, id));
        model.addAttribute("series",timetableService.getById(id).getSeries());
        model.addAttribute("name",
            SecurityContextHolder.getContext().getAuthentication().getName());
        model.addAttribute("court", current);
        model.addAttribute("realname", name);
        model.addAttribute("bookings", left);
        if (seriesMatch(courtID, nextCourt)) {
                model.addAttribute("next", (current.getId() + 1));
        }
        if (seriesMatch(courtID, prevCourt)) {
                model.addAttribute("prev", (current.getId() - 1));
        }
        return "court";
}
```

Figure 5.33: Timetable Controller chooseCourt() method

Model Name	Summary
name	The username of the currently authenticated user
realName	The real name of the currently authenticated user
bookings	The number of remaining bookings of the currently authenticated users
date	The current week of the year and the current date. Calculated using separate method.
next	The id number of the court following the current court, if applicable
prev	The id number of the court preceding the current court, if applicable
court	The current court, determined by the current week, provided by the java.util.Date class

Table 5.4: Model Attributes

This example is an excerpt from the TimetableContoller class. The logic determining the values has been removed. This is to highlight how attributes are added to the model from within the controller. This is the information that the JSP page will have access to once it has been displayed.

The above code deals with the display of *Monday* within the Timetable display page. In the *c:forEach* tags, it loops through each value in the *court.monday* list that has been passed

to it by the controller. The size of this list is determined by the user when the timetable is created, and the number of slots per day is specified. If the current value being examined in the loop is equal to the value "Free Court", it will display a link to the Book Form mapping. This aspect of the Timetable Controller will check that a user has any remaining bookings left and respond as appropriate. In the event that the value in the list does not equal "Free Court", it will make a choice. If the currently authenticated user made the booking, it will display an option to remove their booking from the slot. Otherwise, it will give any other user an option to report the user as a "no show" should a user fail to show for a previously booked slot.

```
<c:forEach var="row" varStatus="loop" items="${court.monday}">
<c:choose>
<c:when test='${row eq "Free Court"}'><tr>
<td class="inner"><form action="${pageContext.request.contextPath}/bookCourt"
method="POST">
<input type="hidden" value="${loop.index}" name="position" />
<input type="hidden" value="monday" name="day" />
<input type="hidden" value="${court.id }" name="ttid" />
<input type="submit" value="Book">
</form></td></tr>
</c:when>
<c:otherwise><tr><td class="inner">${row}
<c:choose>
<c:when test="${name eq pageContext['request'].userPrincipal.name && row eq
    realname }">
<form action="${pageContext.request.contextPath}/unbookCourt" method="POST">
<input type="hidden" value="${loop.index}" name="position" />
<input type="hidden" value="monday" name="day" />
<input type="hidden" value="${court.id }" name="ttid" />
<input type="submit" value="Unbook">
</form></c:when>
<c:otherwise>
<form action="${pageContext.request.contextPath}/reportNoShow" method="POST">
<input type="hidden" value="${row}" name="bookedUser" />
<input type="hidden" value="monday" name="day" />
<input type="hidden" value="${court.id }" name="ttid" />
<input type="submit" value="Report User">
</form></c:otherwise>
```

Figure 5.34: Code Showing Display of Timetable

5.4 Logging the Application

The logging for this application was provided by *log4j*. Logging became very useful for tracking down, and isolating bugs, throughout the application. Since there were a considerable number of dependencies and different technologies working together, it rapidly became very

difficult to see where errors originated from. Stack-traces quickly became unmanageable. *Log4j* works by allowing the developer to view a number of logs of varying types within the application.

Log Type	Description
INFO	Messages that highlight the progress of the application at coarse-grained level
DEBUG	Fine-grained informational events that are most useful to debug an application
TRACE	Finer-grained informational events than the DEBUG
WARN	Potentially harmful situations
ERROR	Error events that might still allow the application to continue running
FATAL	Very severe error events that will presumably lead the application to abort

Table 5.5: Log Types

Log4j is configured with a properties file that allows you to see the various levels of logs displays by the applications running. Implementation of a logging system resolved a number of Spring Security issues within the web application. Spring Security, concerned with access rights to mappings within the application, did not output failed access attempts to the console. This made it very difficult to debug. When configuring *Log4j* to catch the logs created by the security components, the application became much easier to debug. The properties file for this web application is detailed below.

```
1 log4j.rootLogger=INFO, CONSOLE
2 log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
3 log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
4 log4j.logger.org.hibernate.SQL=DEBUG
5 log4j.logger.org.hibernate.type=TRACE
6 log4j.logger.org.springframework.security=DEBUG
```

Figure 5.35: Log4j Configuration

Logging can be implemented on a class by class basis. Within this application, it was used to display informational messages to the developer. These included items such as database access, objects being created and updated. In order to enable logging on a class, a logger must be instantiated with reference to the class that requires logging. The logger object then is called with a method corresponding to the type of log you wish to throw along with a message.

```
1  private static Logger logger = Logger.getLogger(UserDAO.class);
2
3  public Session session(){
4          logger.info("Session Factory returning current session.....");
5          return sessionFactory.getCurrentSession();
6  }
7
8  public List<User> getUsers() {
9          logger.info("Selecting All Enabled Members....");
10         return session().createQuery("from User where enabled = '1'").list();
11 }
12
13 public User getUserByName(String name) {
14         Criteria crit = session().createCriteria(User.class);
15         crit.add(Restrictions.eq("name", name));
16         try{
17         User user = (User) crit.uniqueResult();
18         }
19         catch(Exception e){
20         logger.error{"Must be unique result : Thrown from
               UserDAO.getUserByName(String name));
21         }
22         return user;
23 }
```

Figure 5.36: Logger Usage within UserDAO.class

5.5 Configuration of the Application

In order to begin implementation with the Spring MVC framework, there are a number of configuration files that are necessary. The core file is the *web.xml* file. This file is responsible for the configuration for the framework. One of the key responsibilities is the definition of the context xml files, whose purpose will be elaborated on later. Different development profiles can be configured within this file in order to produce different development environments, such as production and testing environments. The configuration of this file within the application is shown in Figure 5.37

```
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
</props>
</property>
```

Figure 5.38: Hibernate Configuration

```
<context:annotation-config></context:annotation-config>
<context:component-scan base-package="service"></context:component-scan>
```

Figure 5.39: Service Context Configuration

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
        classpath:beans/dao-context.xml
        classpath:beans/service-context.xml
        classpath:beans/security-context.xml
</param-value>
 </context-param>
```

Figure 5.37: Spring Configuration

Of particular importance are the definition of the context parameters. In this project, there were three main context files.

- Data Access Object Context
- Service Context
- Security Context

The DAO Context file specifies the packages that contain the various DAO classes within the application. It also contains configurations for both the database connection details, and Hibernate configurations. Packages containing entity classes for Hibernate are specified within this context also.

The Service Context file is responsible for specifying the base package containing the Service classes necessary to facilitate the collaboration between the Controller classes and the DAO classes. This file specifies that annotations will be used to configure the Service classes.

The Security Context file is the larger of the three files, and is responsible for the security configuration of the web application. There are four main areas within the file that were

used to configure the web application created in this project.
The User Service aspect of the configuration file is responsible for retrieving users and their authority within the scope of the web application.
The URL access configuration ensures that only users who are authorised to access certain portions of the site are allowed access.
The Security Annotations allow the creation of an extra level of security into an application. At class level, annotations can be placed on methods to further ensure that proper access is enforced throughout the application.
Lastly, the Security Context is responsible for creating the password encoder bean in which passwords are encoded, and decoded, upon account creation and login. This ensures that no passwords in plain text form are ever stored on either the server or the database within the web application

- User Service

```
<security:authentication-manager>
        <security:authentication-provider>
        <security:jdbc-user-service data-source-ref="dataSource"
                id="jdbcUserService" authorities-by-username-query="select
                    username, authority from users where binary username = ?" />
        <security:password-encoder
            ref="passwordEncoder"></security:password-encoder>
        </security:authentication-provider>
</security:authentication-manager>
```

- URL Access

```
<security:intercept-url pattern="/timetable" access="permitAll"/>
<security:intercept-url pattern="/reportNoShow" access="permitAll"/>
<security:intercept-url pattern="/admin" access="hasRole('ROLE_ADMIN')"/>
<security:intercept-url pattern="/approveMembers"
    access="hasRole('ROLE_ADMIN')"/>
```

- Security Annotation for Service Class

```
<security:global-method-security
    secured-annotations="enabled"></security:global-method-security>
//Java Code from TimetableService class.
//This code is invoked when booking a slot on the timetable and is only
    accessible by registered members.
@Secured({"ROLE_ADMIN", "ROLE_MEMBER", "ROLE_COMMITTEE", "ROLE_WARNING",
    "ROLE_SUSPEND"})
        public void update(Timetable t){
                timetableDAO.updateTimetable(t);
        }
```

- Password Encoding

```
<bean id="passwordEncoder"
class="org.springframework.security.crypto.password.StandardPasswordEncoder">
</bean>
```

5.6 Test Driven Development

The primary method of testing was implemented using JUnit. A Test Suite of JUnit tests were prepared to test the key features of the application. A separate test database was constructed. It was important to ensure that the testing environment was using the same context files as the production environment. The test class had to be annotated to enforce this. While the context files were the same, the DataSource file has changed as a different database is being using in this environment.

```
@ActiveProfiles("dev")
@ContextConfiguration(locations = { "file:src/main/java/beans/dao-context.xml",
"file:src/main/java/beans/security-context.xml",
"classpath:test/config/datasource.xml" })
@RunWith(SpringJUnit4ClassRunner.class)
public class HibernateTests {

        @Autowired
        private UserDAO userDAO;

        @Autowired
        private TournamentDAO tournamentDAO;

        @Autowired
        private DataSource dataSource;

        //Class truncated
}
```

Figure 5.40: JUnit Test Example

The database is then initialised to ensure the tests are being run against the same database, and that repeat tests are consistent.

```
@Before
public void init(){
        JdbcTemplate jdbc = new JdbcTemplate(dataSource);
        jdbc.execute("delete from users");
}
```

Figure 5.41: JUnit @Before Test Configuration

In these example tests, the UserDAO is being tested to ensure that it returns true when the exists() method is called on it. This is important within the scope of the application to ensure that primary keys are not duplicated. The method is annotated with *@Test*. The methods *assertTrue* and *assertFalse* expect a return value of true and false respectively. They take two parameters: an error message and a boolean value, or a method that returns a boolean value. In the *assertTrue* method below, the UserDAO will return true if the user exists. In the event that the user does not exist, it will fail the test and return the message "User should exist".

```
@Test
public void testExists(){
        userDAO.createUser(user1);
        assertTrue("User should exist", userDAO.exists(user1.getUsername()));
        assertFalse("User should not exist",
            userDAO.exists("jkljfksakfjahghdsopjclkhfkjafhkjdshFHajhgouwe"));
}
```

Figure 5.42: JUnit UserDAO Exists() Test

Another test with the UserDAO was to ensure that users were being saved correctly. In this example, users are being created and saved to the database. The method *assertEquals* checks two interger values and returns an error message if they do not match.

```
@Test
public void testCreateRetrieve(){
        userDAO.createUser(user1);
        List<User> users1 = userDAO.getAllUsers();
        assertEquals("One user should have been created and retrieved", 1,
            users1.size());
        assertEquals("Inserted user should match retrieved", user1, users1.get(0));
        userDAO.createUser(user2);
        userDAO.createUser(user3);
        userDAO.createUser(user4);
        List<User> users2 = userDAO.getAllUsers();
        assertEquals("Number of users should be four", 4, users2.size());
}
```

Figure 5.43: JUnit Create and Size Test

5.7 Conclusion

Chapter 6

Software Quality

6.1 Introduction

The use of software metrics within this project was to guide the development process and measure the quality of the application. Measurement is concerned with 'capturing information about *attributes* of *objects*' (Fenton 1991).

6.2 Application Summary

This was done through the use of a metrics plugin for STS called *Metrics 1.3.6*.

(Discssion on software metrics) - Software Metrics - A Rigorous Approach - Norman Fenton (ref)

The following is a summary of the various metrics of the application. A detailed breakdown by package in contained within the appendices.

Metric	Total	Mean	Std. Deviation	Maximum
McCabe Cyclomatic Complexity	n/a	1.262	.862	8
No. Parameters per method	n/a	0.894	1.115	15
Nested Block Depth	n/a	1.126	.588	5
Number of Classes	59	4.538	3.456	10
Method Lines of Code	1944	3.88	6.732	66

Table 6.1: Application Metrics

6.3 Software Quality Tools and Visualisations

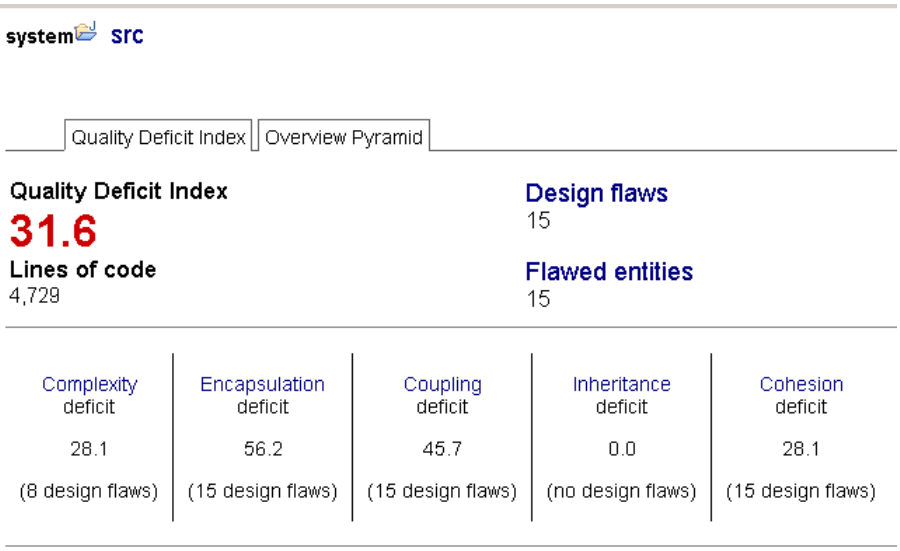

Figure 6.1: Pre Re-factoring using Infusion

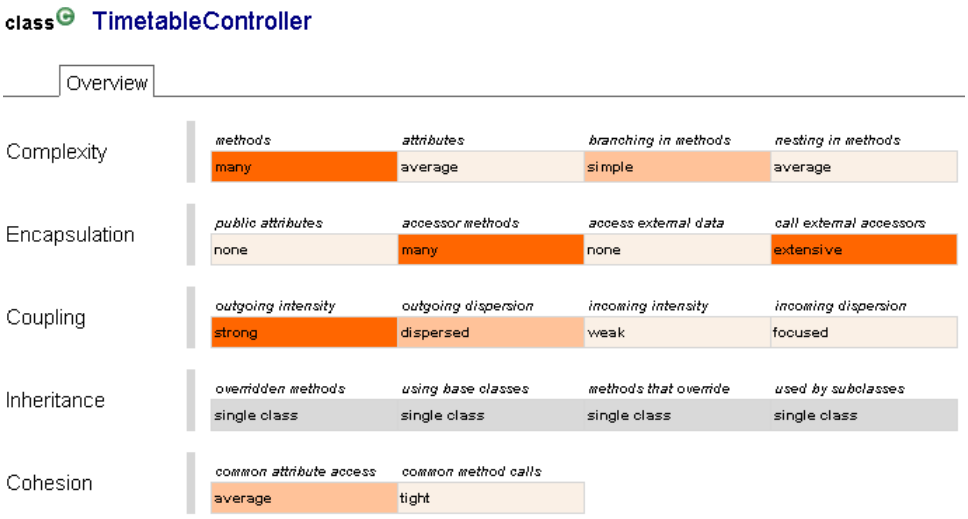

Figure 6.2: Example of Class 'Bad Code Smell' Breakdown using Infusion

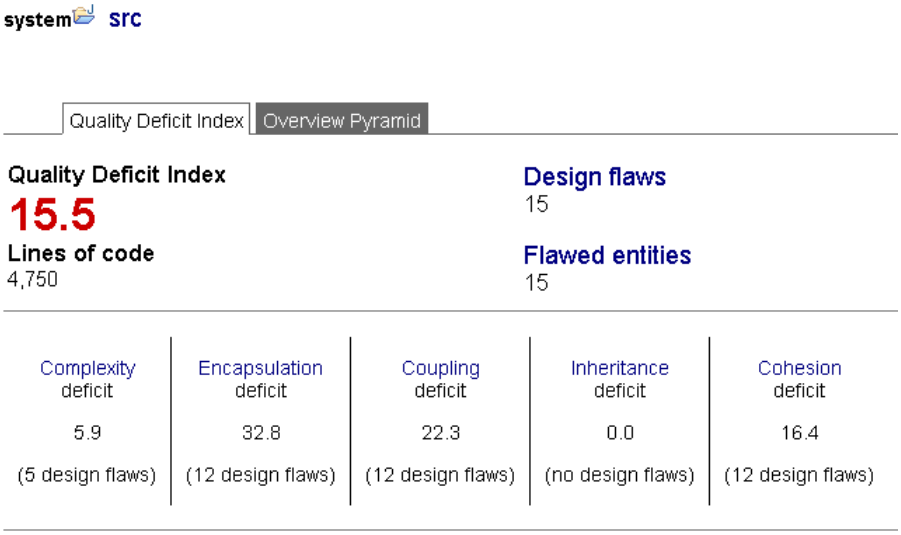

Figure 6.3: Post Re-factoring using Infusion

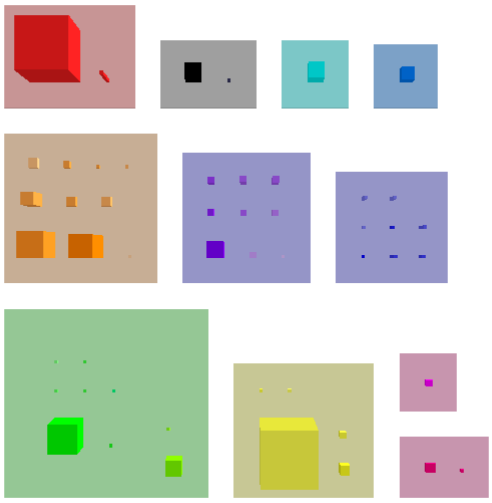

Figure 6.4: CodeCity 2D Visualisation of Application

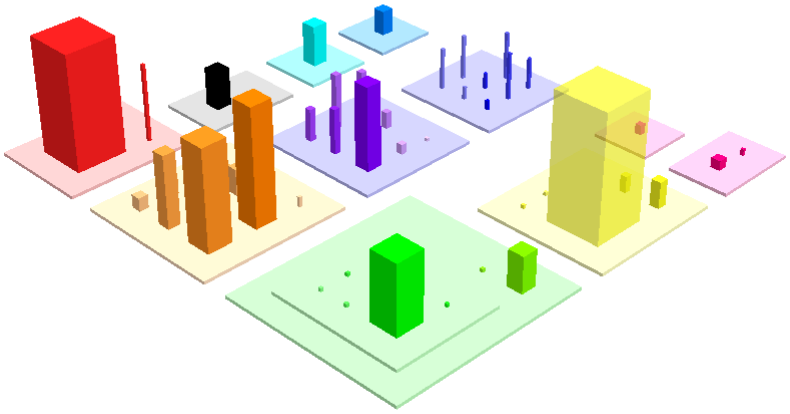

Figure 6.5: CodeCity 3D Visualisation of Application

6.4 Sample Refactorings

Chapter 7

Evaluation

7.1 Usability

7.1.1 Productivity

Hibernate

One of the core technologies used within the application was Hibernate. One of the main benefits Hibernate provided to the project was simplifying the code in order to persist objects. When developing the application, the only concern was with objects as a whole, not on their individual attributes. With JDBC, there was a need to use SQL syntax within the application. In the case of the User class, there were 16 attributes which made the SQL INSERT statement for that class, shown in Figure 7.1, difficult to read and troubleshoot.

```
@Transactional
public boolean createUserJDBC(User user) {
	MapSqlParameterSource params = new MapSqlParameterSource();
		//other params cut out to save space
		params.addValue("role", "ROLE_MEMBER");
		return jdbc
			.update("insert into users (username, name, password, gender,
			member_type, grade, ad_line1, ad_line2, ad_city, ad_county,
			contact_num, em_con_name, em_con_num, enabled, authority,
			bookings_left) values (:username, :name, :password,
			:gender, :member_type, :grade, :ad_line1, :ad_line1,
			:ad_city, :ad_county, :contact_num,:em_con_name, :em_con_num,
			false, :role, 0)",params) == 1;
	}
```

Figure 7.1: JDBC SQL INSERT

However, it could be arguing by supplying a framework, such as Hibernate, to automate the creation of a query language that performance of an application could be affected. To

examine this briefly, a small test was created. This this test, the application would use JDBC to create 10,000 records, update those records and then delete them. The application would then perform the same actions with the same data using Hibernate. The time taken to perform these tasks as a whole would be recorded in nanoseconds. The test is illustrated in Figure 7.2 below.

```
@RequestMapping("/testdatabases")
public String testdatabases(Model model){
        User user = new User(); // create a new user
        long jStart = System.nanoTime();
        model.addAttribute("jdbcStart", jStart);
        for (int i = 0; i < 10000; i++){
                user.setUsername(i + "@test.ie");
                userService.createJDBC(user);
                userService.enableUserJDBC(user);
                userService.deleteUserJDBC(user);
        }
        model.addAttribute("jdbcFinish", (System.nanoTime() - jStart));

        long hStart = System.nanoTime();
        model.addAttribute("hibernateStart", hStart);
        for (int i = 0; i < 10000; i++){
                user.setUsername(i + "@test.ie");
                userService.create(user);
                user.setEnabled(true);
                userService.updateUser(user);
                userService.delete(user);
        }
        model.addAttribute("hibernateFinish", (System.nanoTime() - hStart));
        return "testdatabases";
}
```

Figure 7.2: JDBC Vs Hibernate Test

The results are displayed in Table 7.1 below. NS refers to NanoSeconds, and MM:SS refers to the total in minutes and seconds. The test was run three times and an average was created. The test machine was an Amazon EC-2 server, which was a 32-bit micro instance, with 650mb of RAM with one virtual CPU. This is running a modified version of Red Hat Enterprise Linux.

Run	Database	No Queries	Total (NS)	Total (MM:SS)
1	Hibernate	10,000	198469890740	03:20
1	JDBC	10,000	211883464327	03:30
2	Hibernate	10,000	188055485374	03:07
2	JDBC	10,000	200521721809	03:22
3	Hibernate	10,000	188423897510	03:06
3	JDBC	10,000	193541735201	03:20
Avg	Hibernate	10,000	191649757878	03:12
Avg	JDBC	10,000	201982307112	03:21

Table 7.1: Hibernate Vs JDBC Results

```
//Excepts from the UserDAO for Hibernate and JDBC actions
//Hibernate
return sessionFactory.getCurrentSession(); //return current session
user.setPassword(passwordEncoder.encode(user.getPassword()));
session().save(user);
session().update(user);
session().delete(user);

//JDBC
MapSqlParameterSource params = new MapSqlParameterSource();//start create
params.addValue("username", user.getUsername());
params.addValue("name", user.getName());
params.addValue("password", passwordEncoder.encode(user.getPassword()));
params.addValue("gender", user.getGender());
params.addValue("member_type", user.getMember_type());
params.addValue("grade", user.getGrade());
params.addValue("ad_line1", user.getAd_line1());
params.addValue("ad_line2", user.getAd_line2());
params.addValue("ad_city", user.getAd_city());
params.addValue("ad_county", user.getAd_county());
params.addValue("contact_num", user.getContact_num());
params.addValue("em_con_name", user.getEm_con_name());
params.addValue("em_con_num", user.getEm_con_num());
params.addValue("role", "ROLE_MEMBER");
return jdbc.update("insert", params) == 1; //end create
MapSqlParameterSource params = new MapSqlParameterSource(); //start update
params.addValue("enabled", enabled);
params.addValue("username", username);
jdbc.update("update",params); // end update
MapSqlParameterSource params = new MapSqlParameterSource(); //start delete
params.addValue("username", user.getUsername());
jdbc.update("delete",params); //end delete
```

Figure 7.3: Hibernate-JDBC Code

In all tests, Hibernate took less time to perform all actions, though there was not much different between both technologies. However, due to the way Hibernate is configured, it is certainly much easier to develop with. Hibernate required 5 lines of code, shown in Figure 7.3, where JDBC requires 23 lines of code, with 3 different SQL queries, one for INSERT, UPDATE and DELETE.

By defining and mapping attributes in a database to the class with Hibernate, the DAO classes become much smaller and easier to manage with no impact on performance within the application. In this application, for example, there are more than 70 DAO operations, which would mean 70 or more SQL statements. Use of JDBC would also require the definition of SQL JOINs in relation to both the Timetable and Tournament classes, thereby again increasing complexity within the application. There is more configuration to be undertaken with Hibernate, but the end result is a much easier to manage persistence solution that reduces code complexity.

Session and Cookie Management

One benefit of using a framework such as Spring is that the developer does not need to be overly concerned with defining and checking a session in each controller. In a previous project, the developer was responsible to creating and managing the session, as well as ensuring proper access to the site was maintained. Figure 7.4 shows how a session must be manually retrieved from a request object within a servlet.

```
1  @Override
2  protected void doGet(HttpServletRequest request, HttpServletResponse response)
3  throws ServletException, IOException {
4      PrintWriter out = response.getWriter();
5      // Print Generic header then check authenitcation and print HTML
6      printHeader(out);
7      HttpSession currentSession = request.getSession(false);
8      checkAuthentication(out, currentSession);
9      // Print search form
10 }
```

Figure 7.4: Manual Session Creation

In the *checkAuthentication()* method, depicted in Figure 7.5, the developer must check the session for any attributes that match a certain value, and act accordingly. In this example, the id of a user. The id then needs to checked against a database to ensure that it is a valid id, and to determine what permissions the account has.

```
private void checkAuthentication(PrintWriter out, HttpSession currentSession) {
//Check Authentication
        if (currentSession.getAttribute("ID") == null) {
                //print registration screen
    } else {
        // Use ID and Username to identify users. At this point we know there is a
            session with an ID, verify it is a valid session
        int intID = (Integer) currentSession.getAttribute("ID");
        String username = (String) currentSession.getAttribute("username");
        if (customerBean.checkCustomerId(intID) == true &&
            customerBean.checkCustomerUsername(username) == true) {
                        Customer currentCustomer = customerBean.getCustomer(intID);
                         // Print what customer should see
            }
                else if (adminBean.checkAdminId(intID) == true &&
                    adminBean.checkAdminUsername(username) == true) {
            Admin currentAdmin = adminBean.getAdmin(intID);
                        // Print what admin should see
            }
        }
        }
```

Figure 7.5: Session Authentication based on user level

An issue with that is that it is reliant on the developer to identify the correct attribute. This is a user defined attribute that may not be unique to the application. Spring manages this through the *security-context.xml* file, which encapsulates security for the application to one file, and the framework itself ensures that the correct session is created for a user. There is no need to specify any of this within the codebase of the application, which again reduces complexity, and minimises the possibility of security leaks being introduced through developer errors.

7.2 Architectural Quality

Performance, Security and Evolution Support

7.3 Functional and Non Functional Requirement Evaluation

7.4 FYP Process and Learnings

The FYP has been certainly been significant part of the overall college journey. Looking back, and knowing what I know now, there are a number of avenues that I would have liked to explore.

7.4.1 Hibernate in Design

Hibernate was a very powerful framework throughout the development of this project. It was very much a case of learning by doing. Having accumulated experience with the framework, I would like to apply it more in the design stage on an application. One idea that came to me late in the project was the idea of having a top level class that could build the entire application, as depecited in Figure 7.6.

```
@Entity
public class WebApplication(){

@ElementCollection
@CollectionTable (name = "tournament", joinColumns=@JoinColumn(name="id"))
private Tournaments tournaments;

@ElementCollection
@CollectionTable (name = "timetable", joinColumns=@JoinColumn(name="id"))
private Timetable timetable;

@ElementCollection
@CollectionTable (name = "news", joinColumns=@JoinColumn(name="id"))
private News news;

@ElementCollection
@CollectionTable (name = "events", joinColumns=@JoinColumn(name="id"))
private Events events;

@ElementCollection
@CollectionTable (name = "users", joinColumns=@JoinColumn(name="id"))
private User users;

public WebApplication(){
       create();
}

private void create(){
       tournaments = getTournaments();
       etc.....
}

//getters and setters
```

Figure 7.6: 'Top Level' Class

(This section needs to be rewritten. I'm tired right now so just doing stream of conciousness)

It would be possible to build the application so that all classes were linked through a hierarchy to one root class. This may impact performance, but it would allow a number of benefits such as ease of deployment, and the ability to take snapshots of the application at

any stage.

I also would have like to use Hibernate with more complicated objects rather than just collection classes, but time constraints did not allow this.

7.4.2 Tournament Business Logic

7.4.3 Web Services and HTML5

Other aspects of a web application, such as web services and HTML5 were omitted from the final submission due to time constraints, but are definitely something that I would have an interest in looking at later on.

Chapter 8

Conclusions

Bibliography

Bauer, Christian and Gavin King (2005). *Hibernate in action*. Manning Greenwich CT.

Fenton, Norman E. (1991). *Spftware Metrics: A Rigorous Approach*. Kluwer Academic Publishers.

Holzinger, Andreas (2005). "Usability engineering methods for software developers". In: *Communications of the ACM* 48.1, pp. 71–74.

Rochkind, Marc J. (1975). "The source code control system". In: *Software Engineering, IEEE Transactions on* 4, pp. 364–370.

Appendix A

Appendices

A.1 Application Breakdown

The following section is a breakdown of the application into its various packages and files.

A.1.1 Java Source

Package: beans

File Name	File Type	Function	No. Lines
beans.xml	XML	Context Component Scan, Datasource Definition	0
dao-context.xml	XML	Hibernate Configuration, Database Exception Translator	0
security-context.xml	XML	Security Configuration, Access Control	0
service.xml	XML	Service Configuration	0

Table A.1: beans package

Package: controllers

File Name	File Type	Function	No. Lines
ErrorHandler	JAVA	Handles application errors	
EventController	JAVA	Handles Event actions	
HomeControler	JAVA	Displays Home Page	
LoginController	JAVA	Manages user login	
MembersController	JAVA	Handles Member actions	
NewsController	JAVA	Handles display and creation of News	
SiteController	JAVA	Displays site pages with static data	
TimetableController	JAVA	Controls the Timetable Creation and Display	
TournamentController	JAVA	Handles Creation and Management of Tournaments	

Table A.2: controllers package

Package: dao

File Name	File Type	Function	No. Lines
EventDAO	JAVA	DAO for IEvent	
LogDAO	JAVA	DAO for ILog	
NewsDAO	JAVA	DAO for News	
RoleDAO	JAVA	DAO for Role	
TimetableDAO	JAVA	DAO for Timetable	
TournamentDAO	JAVA	DAO for Tournament	
UserDAO	JAVA	DAO for User	
UserRowMapper	JAVA	JDBC Row Mapper to handle multiple User objects	

Table A.3: dao package

Package: email

File Name	File Type	Function	No. Lines
Email	JAVA	Create and Send Email Message	
IEmail	JAVA	Interface for Email class	

Table A.4: email package

Package: events

File Name	File Type	Function	No. Lines
Event	JAVA	Defines an event used by the Timetable	
IEvent	JAVA	Interface for Event class	

Table A.5: event package

Package: events.tournaments

File Name	File Type	Function	No. Lines
Tournament	JAVA	Defines an Tournament object	

Table A.6: events.tournament package

Package: logs

File Name	File Type	Function	No. Lines
Log	JAVA	Defines the structure of a system log file	
ILog	JAVA	Interface for Log class	

Table A.7: log package

Package: news

File Name	File Type	Function	No. Lines
News	JAVA	Defines the structure of a News object	

Table A.8: news package

Package: properties

File Name	File Type	Function	No. Lines
jdbc	PROPERTIES	Holds log in values for the JDBC connection	
mail	PROPERTIES	Holds the log in values for the email system	

Table A.9: properties package

Package: reports

File Name	File Type	Function	No. Lines
IReport	PROPERTIES	Interface for Reports	
CSVCreator	PROPERTIES	Creates a CSV file for User Data	

Table A.10: reports package

Package: service

File Name	File Type	Function	No. Lines
EventService	JAVA	Layer between EventController and EventDAO	
LogService	JAVA	Layer between Controllers and LogDAO	
NewsService	JAVA	Layer between NewsController and NewsDAO	
RoleService	JAVA	Layer between Controllers and RoleDAO	
TimetableService	JAVA	Layer between TimetableController and TimetableDAO	
TournamentService	JAVA	Layer between TournamentController and TournamentDAO	
UserService	JAVA	Layer between MemberController and UserDAO	

Table A.11: service package

Package: timetable

File Name	File Type	Function	No. Lines
Timetable	JAVA	Interface for Timetable	
MonaleenTTV1	JAVA	Defines structure and behaviour of Timetable object	

Table A.12: timetable package

Package: users

File Name	File Type	Function	No. Lines
FormValidationGroup	JAVA	Form Validation Class	
PersistenceValidationGroup	JAVA	Hibernate Validation Class	
Grade	JAVA	Defines structure of a Grade object. Used in User class.	
User	JAVA	Defines structure of a User object	
Role	JAVA	Defines structure of Role object, and its attributes	

Table A.13: reports package

A.2 Apache Struts and JSP Pages

layout

File Name	File Type	Function	No. Lines
default	XML	Defines the structure for each JSP page in the application	

Table A.14: struts layout

templates

File Name	File Type	Function	No. Lines
default	JSP	The default JSP page that is used as the template for all others	

Table A.15: templates layout

tiles

File Name	File Type	Function	No. Lines
accessdenied	JSP	Access Denied page	
admin	JSP	Displays administrator page with admin options	
adminAnalysis	JSP	Displays site analytics	
adminEditProfile	JSP	Allows admin to edit user accounts	
alreadyReg	JSP	Error page when attempting to re-register for tournament	
approveMembers	JSP	Admin approve members page	
blockMembers	JSP	Admin suspend members page	
bookingExists	JSP	Error page to handle duplicate bookings	
checkRegistered	JSP	Displays users registered for a selected tournament	
chooseEdit	JSP	Choice for which Timetable to edit	
confirmEdit	JSP	Detailed layout for editing individual slots in Timetable	
contactus	JSP	Contact Us page for application	
content	JSP	Place-holder page for default JSP template	
court	JSP	Displays selected Timetable and available options	
createEvent	JSP	Admin Create Event page	
createmembers	JSP	User Registration page	
createNewRole	JSP	Admin create new User Role	
createNews	JSP	Admin Create News page	
createTimetable	JSP	Admin Create Timetable page	
createTournament	JSP	Admin Create Tournament page	
deleteNews	JSP	Admin Delete News entry	
deleteTimetable	JSP	Admin Delete Timetable object	
deleteTournament	JSP	Admin Delete Tournament object	
displayUsers	JSP	Admin Displays all users to choose which one to edit	
editDetails	JSP	User Edit Profile	
emailSent	JSP	Email Sent Confirmation Page	
error	JSP	Default Error Page. Displays Class Error.	
fillTimetable	JSP	Page that allows admin to create Timetable template for series.	

File Name	File Type	Function	No. Lines
index	JSP	Default Home page	0
footer	JSP	Place-holder page for default JSP template	
header	JSP	Place-holder page for default JSP template	
links	JSP	Displays Site Navigation links	
loggedout	JSP	Logout Confirmation page	
login	JSP	Login Page - Linked to Spring Security	
maps	JSP	Displays Google Maps Location for Club	
members	JSP	Displays Members Address Book for authenticated users	
membership	JSP	Displays Membership Information	
news	JSP	Displays News Page	
profile	JSP	Displays User Profile	
profileUpdated	JSP	Confirmation of profile being updated	
registerSuccess	JSP	Confirmation that user has successfully registered	
resetAllTimetable	JSP	Removes all bookings for a Timetable	
seriesChoice	JSP	Choose which Timetable series to edit/reset	
timetable	JSP	Displays enabled timetables for users	
timetableStatus	JSP	Allows admin to enable or disable timetables	
tournaments	JSP	Displays Enabled Tournaments for Users	
tournamentStatus	JSP	Allows admin to enable/disable/activate/deactivate tournaments	
tournamentSuccess	JSP	Displays success upon successful tournament creation	
viewAllMembers	JSP	Admin View of Members	
viewEvents	JSP	Admin Event Management	

Table A.16: templates layout