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

Chris O'Brien

March 11, 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 Architectures of Web Applications

2.2 Technologies

There are a number of components needs 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.

2.2.1 Web Application Framework

The WAF chosen for this project is Spring MVC [Model View Controller]. Shan and Hua defined a WAF as a defined support structure in which other software applications can be organized and developed. (Shan and Hua 2006). Model-View-Controller is a software pattern that facilitates the use of a user interface. The Model manages the behaviour and data of the application. The View will manage the information obtained from the model and display it to the user. The Controller takes user input, such as key strokes, mouse movements or a touch display, and can interact and invoke functionality within the Model and/or View.

```
@RequestMapping("/contactus")
public String contactUs(Model model){
    model.addAttribute("admins", userService.getAdmins());
    model.addAttribute("committee", userService.getCommittee());
    return "contactus";
}
```

Figure 2.1: Contoller adding Model to View

2.2.2 Application Server

2.2.3 Project Management Tool

2.2.4 Database Model

2.2.5 Source Control

2.2.6 Integrated Development Environment

2.2.7 Logging

2.2.8 Web Page Creation

2.3 Software Engineering

2.3.1 Requirements

2.3.2 Design

2.3.3 Testing

2.3.4 Software Quality

Chapter 3

Requirements

Chapter 4

Design

Chapter 5

Implementation and Testing

5.1 Implemention

5.1.1 Spring

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 reponsible 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.

```
<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>
```

Table 5.1: Spring Context File Defintion

Of particular importance are the definition of the context parameters. In this project, there were three main context files. These files are important, as they are the foundation for Springs dependency injection facilities. Dependency Injection is a software design pattern. It allows an application to be configured so that hard coded dependencies are removed and

the necessary objects can be inserted during run time or at compile time. In Spring, this is done with the *@Autowired* annotation. This instantiates a reference with a relevant bean, and removes having to use the *new* keyword.

- 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.

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

Table 5.2: DAO Context Configuration

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.

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

Table 5.3: Service Context Configuration

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>
```

5.1.2 Security

A core part of the Spring platform is the Security support. Security is an important aspect for any application, but more-so for one that stores user data, particularly sensitive data such as names, addresses, phone numbers, and payment details. While there is no payment infrastructure within this application, there is scope for a system to be implemented, and it could certainly be a future requirement.

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.

- 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.

Table 5.4: Security Roles within the Web Application

To ensure that the correct user is logged in, the framework provides a SecurityContextHolder class which can be used by the Controllers in order to ensure that that any actions performed by the system are attributed to the correct user. In this regard, it was important that the system ensures that the duplication of a user-name is restricted. In this project, the user table had two keys: an integer id, which was the primary key, and the user-name was an email address. The validity of the email address was enforced at class level with the use on annotations on the user-name attribute within the User class.

```
//Excerpt from the User class
@NotNull(groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Pattern(regexp=".+\\@.+\\..+", message="This does not appear to be a valid email
    address", groups={PersistenceValidationGroup.class, FormValidationGroup.class})
@Column(name="username")
String username;
```

Another aspect of the Spring Security platform was Form Validation. When registering a user, there are a number a validation constraints that are placed upon 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 BindResult object for errors, and can respond appropriately. In order for this to work, both the Controller and the form need to be defined clearly. The form needs to be creating using the Spring Framework form tag library, and errors needs to be specified for each input within the form. These inputs, when used with Hibernate, all need to match with the attribute names given in the class they represent. This example references the User class.

```
<!-- 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>

//Method from the MembersController class
//This method is responsible for validating the form that users complete to
    register.
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String doRegister(Model model,
@Validated(FormValidationGroup.class) @ModelAttribute("member") User member,
    BindingResult result) {
if (result.hasErrors()) {
       return "createmembers"; // if the result has errors, go back to create page
}
if (userService.exists(member.getUsername())) {
       result.rejectValue("username", "Duplicate Key",
       "This email address has already been used");
       return "createmembers";
       //if the email address already exists, return with this message.
}
       else {
              try {
                     member.setAuthority("ROLE_MEMBER");
                     userService.create(member);
                     return "registerSuccess";
                     //successful creation of member
                     } catch (Exception e) {
                            return "error";
                     }
              }
}
```

Table 5.5: User Registration Form

The UserDAO class then encodes the password using the already configured Password Encoder bean prior to saving it to the database. It was this process that introduced an issue with the validation process. When defining the constraints placed upon the password at-

tribute within the User class, the encoding of the password at a later stage was not taken into account. When encoded, the length of the password increased well beyond the scope of the initial constraint. When Hibernate attempted to persist the User object, it found that the password was now in violation of the constraints placed upon this attribute within the User class. This issue did not occur in the application when using a JDBC database, and is a result of the close links that Hibernate has with its entity classes. In order to resolve this issue, Validation Groups were introduced. A Validation Group is a mechanism in which a class is used to define the validity of attributes within a class. Using these groups, the web application can enforce validation on attributes depending on the validation group specified in the Controller. As respects the application developed for this project, the constraint for the password only had to be enforced within the scope of the form. Once the form validation had passed, enforcement of the constraint was not a concern. By using a separate group for Hibernate persistence, the issue was resolved. (This needs revising! Bit wobbly and lacks structure, also previous code excerpt shows this in action also. Move up to there?)

The Security Context XML file is key to maintaining the integrity of the application security. This file defines access rights to the URL mappings within the application, and also enforces security for the service class methods. By default, the configuration denies access to all parts of the application, and access rights must be explicitly stated. This system relies on the developers to ensure that proper testing is completed to ensure that all access works as designed. This is a better solution than allowing access to all the site and restricting certain parts, such as the administrator panel, however.

```
<security:http use-expressions="true">
        <security:intercept-url pattern="/static/**" access="permitAll" />
        <security:intercept-url pattern="/images/**" access="permitAll" />
        <security:intercept-url pattern="/creatememberes" access="permitAll" />
        <security:intercept-url pattern="/approveMembers"
            access="hasRole('ROLE_ADMIN')" />
        <security:intercept-url pattern="/tournamentRegister"
            access="isAuthenticated()" />
</security:http>
```

This code fragment from the *Security-Context.xml* file displays how access for the web application is configured. While there are five roles within the system, it is not necessary to explicitly define what actions a role can perform. For example, all registered users should be able to register for an existing tournament. In this case, once the framework detects that a user is authenticated, that is sufficient to allow access to that part of the application. Other examples are the display of static images, such as banners and advertising. These should be visible to all visitors to the site, regardless of their authentication status. Accordingly,

this area of the application is set to permitAll allowing such access.

5.1.3 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>
```

Table 5.6: Dependency XML Structure for Maven

5.1.4 Hibernate

Hibernate requires a SessionFactory object in order to connect to the database. Spring provides a class in order to facilitate this requirement, AnnotationSessionFactoryBean. This bean must be defined and configured within the *DAO-Context.xml* file explained previously.

```
<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>
                ...
        </list>
</property>
</bean>
```

Table 5.7: Hibernate SessionFactory Configuration

Once the SessionFactory is configured, it needs to be called from within the DAO classes in the web application. In this web application, the SessionFactoryBean was derived from the AnnotationSessionFactoryBean class, and needed to be configured via annotations. The annotation used in the DAO classes is *@Autowired*. This allows the object to be set using Springs dependency injection facilities.

```
@Autowired
private SessionFactory sessionFactory;

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

Table 5.8: Declarion of SessionFactory in DAO Class

Once this SessionFactory object is configured within the DAO class, it can be used to perform queries on the database. There are four operations that this application is concerned with: *Create*, *Read*, *Update* and *Delete*, commonly known as CRUD. As discussed earlier, Hibernate is an Object Relational Mapping persistence solution. In order to ensure the correct mapping of classes to database schema, some preparatory work needs to be undertaken upon the entity classes.

5.1.5 Class Configuration

When defining the class, it is necessary to do one of two things with the class attributes. The attribute names much match both the type and name of the database schema, or they attributes must be annotated with the column name within the same schema. The number of attributes within a class must also not be less than the number of attributes on the database table. If an attribute is an auto-incrementing primary key, it needs to be stated within the class. It is also important to define getters and setters for each attribute, as these will be called by Hibernate without the application having to explicitly use them.

```
@Component
@Entity
@Table(name="users")
public class User {
        @Id
        @GeneratedValue
        int id;

        @NotNull(groups={PersistenceValidationGroup.class,
            FormValidationGroup.class})
        @Pattern(regexp=".+\\@.+\\..+", message="This does not appear to be a valid
            email address", groups={PersistenceValidationGroup.class,
            FormValidationGroup.class})
        @Column(name="username")
        String username;

        @Size(min=5, max=45, message="Named must be between 5 and 45
            characters",groups={PersistenceValidationGroup.class,
            FormValidationGroup.class})
        @Column(name="name")
        String name;

        @Column(name="password")
        @Size(min=5, max=15, message="Password must be between 5 and 15
            characters", groups=FormValidationGroup.class)
        String password;

        // rest of class truncated, getters and setters defined
```

Table 5.9: Hibernate Entity Class Configuration

Once the entity classes are configured correction, the SessionFactory object can be used to

perform the CRUD operations upon the database.

Create

In order to create an entry in the database, an object is passed to the save() method within the SessionFactory object. Due to the mapping defined within the entity class, there is no need to specify parameters.

```
public void createNewEvent(I_Event e){
        logger.info("Creating new event....");
        session().save(e);
}
```

Table 5.10: Hibernate Create

Read

Hibernate can return an individual object, or a list of those objects, to the Service class. Restrictions can also be applied to retrieve objects based on certain values within the attributes. This is specified using Hibernate Query Language (HQL).

The below example shows how a list of all User objects would be returned to the Service layer. The criteria, such as table name and column names, are specified within the User class.

```
public List<User> getAllUsers() {
        logger.info("Selecting All Members....(Hibernate)");
        return session().createQuery("from User").list();
}
```

Table 5.11: Hibernate Retrieve All Objects

In order to retrieve a unique object, Hibernate provides a Criteria interface for representing a query against an entity class, as outlined below. This will throw an error if there is more than one unique result.

```
public User getUserByName(String name) {
        Criteria crit = session().createCriteria(User.class);
        crit.add(Restrictions.eq("name", name));
        User user = (User) crit.uniqueResult();
        return user;
        }
```

Table 5.12: Hibernate Retrieve Specific Objects

It is also possible to specify attributes within the SessionFactory query using HQL.

```
public List<User> getCommittee() {
        logger.info("Selecting All Admin Members....");
        return session().createQuery("from User where authority =
            'ROLE_COMMITTEE'").list();
}
```

Table 5.13: Hibernate Retrieve Subset Objects using HQL

Update

Update operations are identical to Save operations in Hibernate. An object is passed to the SessionFactory update() method, and Hibernate will save any changes to the object in the database. Hibernate checks the indentifier of the object, and if the identifier does not exist, it will throw an exception. This can be circumvented by using the saveOrUpdate() method of the SessionFactory object. In the event that a record does not exist to update, Hibernate will persist that object as a new record.

```
public void disableMember(String username) {
        Criteria crit = session().createCriteria(User.class);
        crit.add(Restrictions.eq("username", username)); // Restrictions.idEq for
            primary key integer
        User user = (User) crit.uniqueResult();
        user.setEnabled(false);
        session().update(user);
        logger.info("User Disabled");
        }
```

Table 5.14: Hibernate Update Object

Delete

Delete operations are implemented in a very similar fashion to Update queries.

```
public void deleteTournament(Tournament t){
        session().delete(t);
}
```

Table 5.15: Hibernate Update Object

Comparison with JDBC Queries

A considerable advantage of Hibernate over JDBC is the reduction in complexity in creating queries. Since Hibernate encapsulates the table structure of the database table within the entity class, there is no need to explicitly state column names within the query structure. This was made clear during the conversion from a JDBC database to a Hibernate one during the development of this project.

```
@Transactional
public void createUser(User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        user.setBookings_left(0);
        session().save(user);
}
\caption{Hibernate Create User Query }
```

The JDBC query for the above Hibernate query was much larger, and much more needed to be explicitly stated. While the configuration of the User class was not necessary for the JDBC connection, the risk of possible errors introduced by the length of the SQL query contain within, made it a very difficult solution to manage.

```
@Transactional
public boolean createUserJDBC(User user) {
        MapSqlParameterSource params = new MapSqlParameterSource();
        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 into users (username, password, name, gender,
            member_type, grade, ad_line1, ad_line2, ad_city, ad_county,
        contact_num, em_con_name, em_con_num, enabled. authority) values
            (:username, :password, :name, :gender, :member_type, :grade,
        :ad_line1, :ad_line1, :ad_city, :ad_county, :contact_num, :em_con_name,
            :em_con_num, false, :role)",params) == 1;
}
```

Table 5.16: JDBC Create User Query

5.1.6 Apache Tiles

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 TilesConfigurer object takes one parameter: a location of the template that the default tile, and its subsequent children, will use.

```
<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>
```

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.

```
<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>
```

Table 5.17: Apache Tiles Configuration

5.1.7 Services

In this application, Services are the layers between the Controllers and the DAO classes and enable communication between the two sets of objects. This allows us to encapsulate the logic of what a service needs to do within a set of classes, rather than allowing the Controllers to access to DAO directly. Any changes required to the DAO need only be changed in the Service classes, allowing the Controllers to remain unchanged.

5.1.8 Logging

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 5.18: 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.

```
log4j.rootLogger=INFO, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
log4j.logger.org.hibernate.SQL=DEBUG
log4j.logger.org.hibernate.type=TRACE
log4j.logger.org.springframework.security=DEBUG
```

Table 5.19: 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.

```
private static Logger logger = Logger.getLogger(UserDAO.class);

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

public List<User> getUsers() {
        logger.info("Selecting All Enabled Members....");
        return session().createQuery("from User where enabled = '1'").list();
}

public User getUserByName(String name) {
        Criteria crit = session().createCriteria(User.class);
        crit.add(Restrictions.eq("name", name));
        try{
        User user = (User) crit.uniqueResult();
        }
        catch(Exception e){
        logger.error{"Must be unique result : Thrown from
            UserDAO.getUserByName(String name));
        }
        return user;
}
```

Table 5.20: Logger Usage within UserDAO.class

5.1.9 JSP Standard Tag Libraries

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. The following statement 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"%>
```

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, and user actions based on the information contained within. The example below is taken from the Timetable display page from the application.

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 5.21: Model Attributes

```
@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";
}
```

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>
```

Table 5.22: Code Showing Display of Timetable

(another example maybe? Can show how JSTL used to hide null attributes within the models to ensure page looks well)

5.1.10 External Code

There are two aspects of external code being used within this web application. The first is the CSS file that the site is using. This was a free template obtained from Skylines Templates

This template was modified in order to fit in with certain aspects of the site, but the look, feel and images remain consistent with those of the template.

The other code used in this application that was not original is the Google Maps script, visible in the Contact Us page of the web application. This is provided at Google Maps Simple Map Example and is available to use freely. The code was slightly modified to change both the GPS co-ordinates and the initial zoom level of the map. The page is included in the Contact Us tile.

```
//code to include maps.jsp in contactus.jsp
<div align="center"><%@include file="maps.jsp"%></div>

//code of the maps.jsp change with modified Google javascript
<script src="http://maps.googleapis.com/maps/api/js?sensor=false">
</script>
<script>
function initialize()
{
var mapProp = {
center:new google.maps.LatLng(52.6565176,-8.5537577),
zoom:18,
mapTypeId:google.maps.MapTypeId.ROADMAP
};
var map=new google.maps.Map(document.getElementById("googleMap"),mapProp);
}
google.maps.event.addDomListener(window, 'load', initialize);
</script>
```

Table 5.23: Code Showing Google Maps Integration

5.1.11 Application Breakdown

The following section is a breakdown of the application into its various packages and files. (Move to appendices perhaps?)

Package: beans

File Name	File Type	Function
beans.xml	XML	Context Component Scan, Datasource Definition
dao-context.xml	XML	Hibernate Configuration, Database Exception Translator
security-context.xml	XML	Security Configuration, Access Control
service.xml	XML	Service Configuration

Table 5.24: beans package

Package: controllers

File Name	File Type	Function
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 5.25: controllers package

Package: dao

File Name	File Type	Function
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 5.26: dao package

Package: email

File Name	File Type	Function
Email	JAVA	Create and Send Email Message
IEmail	JAVA	Interface for Email class

Table 5.27: email package

5.2 Testing

5.2.1 JUnit Testing

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
}
```

Table 5.28: 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");
}
```

Table 5.29: 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"));
}
```

Table 5.30: 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());
}
```

Table 5.31: JUnit Create and Size Test

Chapter 6

Software Quality

Chapter 7

Evaluation

Chapter 8

Conclusions

Bibliography

[Martin James] Martin, James (1983) Managing the Database Environment J. Martin, MA: Prentice Hall PTR.