CS3754 Cloud Software Development

Semester Project Report

Fall 2017

**Thoughtware**

Casey J. Butenhoff, [eolxe@vt.edu](mailto:eolxe@vt.edu)

Thomas J. Corley, [tjcorley@vt.edu](mailto:tjcorley@vt.edu)

Shane J. Davies, [shane514@vt.edu](mailto:shane514@vt.edu)

Tan T. Doan, [td2304@vt.edu](mailto:td2304@vt.edu)

Kyle M. Dyess, [kdyess97@vt.edu](mailto:kdyess97@vt.edu)

Department of Computer Science

Virginia Tech

Blacksburg, VA 24061

*Date:* December 15, 2017

*Team Number:* 7

*Instructor:* Prof. Osman Balci

EXECUTIVE SUMMARY

Thoughtware is a cloud application hosted in the cloud using Virginia Tech’s cloud server. We thought of this application to be a collaborative software similar to GitHub and Trello. When browsing the web for team collaboration softwares, we saw a lack of calendar functionalities. So to remediate the deficit, we focused on implementing milestones and displaying them on coherent visualizations such as calendars and timelines.

The cloud application utilizes new features such as a RSS feed, real time chat using websockets, open authentication with google, password hashing, database triggers and UI improvements such as calendars and tiered drop-down menus. The RSS feed is dependent on database triggers, as if something changes in the database the RSS feed must refresh to accomodate for the changes. The real time chat uses websockets to have the server receive messages from the user, and then push said messages to their respective receivers. The open authentication with google allows you to use your google account to create and use accounts with Thoughtware. Rather than storing our password as strings in a database, we salt and hash the passwords so there is more security in our user information. We’ve added some components to our database to allow for “triggers” to occur, which essentially tell certain controllers that information has been updated in a database -- so that the controllers may display the new information immediately. The cloud application also has multi-user file sharing within the project, allowing for every user that is a member of the project to add / view / delete files, which updates whenever a user joins another project.

Ultimately we see this cloud application being used due to its enticing improvement of existing features, such as being able to chart milestones in projects, that other collaborative softwares lack.

**Functionalities and Features:**

1. **RSS Feed**
2. **Real Time Chat**
3. **Open Authentication with Google**
4. **Password Hashing**
5. **Database Triggers**
6. **Milestone Calendar Control**
7. **Activity Heatmap**
8. **Multi-User File Sharing** **(within a project)**
9. **File Tree Control**
10. **Dynamic “My Projects” menu resizing**

**TABLE OF CONTENTS**

[EXECUTIVE SUMMARY ii](#_Toc501092891)

[1. SOFTWARE LIFE CYCLE 1](#_Toc501092892)

[2. OUR OBJECTIVE 2](#_Toc501092893)

[3. PROBLEM SPECIFICATION 2](#_Toc501092894)

[3.1 Description of the Problem Context 2](#_Toc501092895)

[3.2 Cloud Software Features to be Implemented by the Proposed App 2](#_Toc501092896)

[3.3 New Cloud Features to be Implemented by the Proposed App 3](#_Toc501092897)

[3.4 Database Management System 3](#_Toc501092898)

[3.5 Complexity 4](#_Toc501092899)

[4. REQUIREMENTS SPECIFICATION 5](#_Toc501092900)

[4.1 Functional Requirements 5](#_Toc501092901)

[4.2 Non-Functional Requirements 5](#_Toc501092902)

[5. ARCHITECTURE SPECIFICATION 5](#_Toc501092903)

[6. DESIGN SPECIFICATION 6](#_Toc501092904)

[7. DELIVERED SOFTWARE FUNCTIONALITY 6](#_Toc501092905)

[8. STEP-BY-STEP INSTRUCTIONS FOR DEVELOPING NEW FEATURES 7](#_Toc501092906)

[8.1 Password Salting and Hashing 7](#_Toc501092907)

[8.2 Database Triggers 14](#_Toc501092908)

[8.3 RSS Feed 16](#_Toc501092909)

[8.4 Project Activity Heatmap 22](#_Toc501092910)

[8.5 Realtime Chat 32](#_Toc501092911)

[8.6 Google OAuth 2.0 and Account Inactivity Monitor 39](#_Toc501092912)

[8.7 Milestone Calendar 44](#_Toc501092913)

[8.8 Many-to-Many Relationships 45](#_Toc501092914)

[9. CONCLUSIONS 46](#_Toc501092915)

[10. SUBMISSION INSTRUCTIONS 47](#_Toc501092916)

[11. PERCENTAGES OF CONTRIBUTION 49](#_Toc501092917)

[12. CALCULATION OF GRADES BASED ON PERCENTAGES OF CONTRIBUTION 50](#_Toc501092918)

[13. TEAM PROJECT REQUIREMENTS 50](#_Toc501092919)

[14. GRADING SHEET 51](#_Toc501092920)

[REFERENCES 51](#_Toc501092921)

[APPENDIX A: MEETING MINUTES 52](#_Toc501092922)

# SOFTWARE LIFE CYCLE

A **good software engineer** develops software by following the software life cycle shown below.



A **programmer (hacker or ad-hoc developer)** develops software by looking at the problem and directly coding in an IDE. This approach is known as the **Build-and-Fix Approach**, which must never be used!

# OUR OBJECTIVE

The objective of our team project is to **demonstrate how capable the team members are** in engineering a cloud software application to solve a complex problem. The cloud app is created for the purpose of showing how learned and new complex functionalities and cloud features the team members are capable of developing.

# PROBLEM SPECIFICATION

This section explains the problem that we are planning to solve, and lists the features that will be built for our solution.

## Description of the Problem Context

Collaborative working environments allow for people to cooperate and communicate on a shared project. Collaborative software can become especially powerful when hosted on the Internet as a cloud application, such as Trello. With advancements in technology, using cloud-based programs to share files and communicate has become quite common.

Effective collaboration allows members’ perspectives to be shared and enables the brainstorming of ideas for the task. It also makes the team more cohesive and leads to a stronger sense of purpose in the group, causing the group to work more closely together. Additionally, it allows for equal contribution from each of the team members.

The general public needs an online collaborative working environment that will help organize teams, tasks, and projects, enable effective communication, and allow for file sharing. Every other system that we have researched only has a subset of these features and many lack a good implementation for zooming in and out of timelines.

## Cloud Software Features to be Implemented by the Proposed App

Our proposed cloud app shall implement the following 12 functionalities:

|  |  |  |
| --- | --- | --- |
| *Number* | *Description of the Functionality Planned to be Implemented* | *Implemented in the*  *Delivered App?* |
| 1 | Project file sharing using a Treetable control | YES |
| 2 | Project real-time chat using websockets | YES |
| 3 | Dynamically-generated header menu using a Menubar control | YES |
| 4 | Project calendar using a Schedule control | YES |
| 5 | Google Login | YES |
| 6 | Project activity visualization (heatmap) | YES |
| 7 | Project activity RSS feed | YES |
| 8 | Database triggers for automatically recording project activity | YES |
| 9 | Password salting and hashing | YES |
| 10 | Session timeout | YES |
| 11 | Many-to-Many database relationship (user-project) | YES |
| 12 | Project milestone timeline using a Timeline control | NO |

1. **Requirement 1:** We develop **cloud software**, not a website! Therefore, the emphasis of your application must be placed on the server side, application logic, JSON data processing, and storage and retrieval of data.
2. **Requirement 2:** You are **required** to use **PrimeFaces** to easily develop the user interface.
3. **Requirement 3:** Use of a **MySQL** database with JPA is **required**!

## New Cloud Features to be Implemented by the Proposed App

Functional Requirements

1. The system shall use websockets to provide real-time chat functionality between individual users and across entire teams and projects.
2. The system shall have a calendar visualization that shows important project milestones.
3. The system shall provide heat maps of all activity that occurs within each project.
4. The system shall allow sharing files among all members of a project.
5. The system shall provide an RSS feed for each project containing all activity that occurs on it.

Non-functional Requirements

1. The system shall invalidate a user’s session after 1 hour of inactivity.
2. The system shall allow users to authenticate to the system with their Google account.
3. The user shall be able to access only projects that they’re authorized to access.
4. The system shall salt and hash user passwords using a cryptographic hash function that is currently recognized as secure

## Database Management System

A database will be needed for various parts of the application including:

* Storage of messages for chat/forum
* Metadata for file-sharing
* User Account data
* Project Data(tasks, items that have been completed, etc)

## Complexity

Many aspects of the project will go beyond the scope of the tutorials in class. These include the use of websockets, advanced visualization techniques such as heat-maps and time-series, and an RSS feed service. They will also include API calls to google for open authentication, and adding unique features to the database such as database triggers.

# REQUIREMENTS SPECIFICATION

This section specifies the Functional and Non-Functional Requirements under which our cloud software application will be developed.

## Functional Requirements

1. The system shall use websockets to provide real-time chat functionality between individual users and across entire teams and projects.
2. The system shall have interactive time-series visualizations of project activity that show important milestones as major points in the timeline.
3. The user shall be able to click on milestone points within visualizations in order to see details about a particular milestone.
4. The system shall provide heatmaps of all activity that occurs within each project.
5. Users shall be able to filter the heatmap visualizations by an activity type and/or a particular user who was active in the project.
6. The system shall provide an RSS feed for each project containing all activity that occurs on it.

## Non-Functional Requirements

1. The UI template footer shall contain the following statement:  
     
   [CS3754 Cloud Software Development](https://manta.cs.vt.edu/cs3754/) course semester project application developed by studentName1, studentName2, studentName3, studentName4, and studentName5.  
     
   Course title and each student name above shall be hyperlinked to show the corresponding homepage *in a new window / tab*.
2. The cloud software application name in NetBeans and when deployed on the server shall be a meaningful name without course number or team number. The app name must reflect what the app does such as MeetingScheduler. Do **not** name it as “group”, “Team4App”, or “CS3754Team4Project”.
3. The system shall invalidate a user’s session after 1 hour of inactivity.
4. The system shall accept OpenID for authentication so users may login using accounts from other systems that issue OpenIDs such as Google accounts.
5. The user shall be able to access only projects that they’re authorized to access.
6. The system shall salt and hash user passwords using a cryptographic hash function that is currently recognized as secure

# ARCHITECTURE SPECIFICATION

TBD

Using a *detailed* architectural diagram / graphics, specify the Java EE technologies used under the Client-Server Architecture together with the Service-Oriented Architecture if calling upon web services provided by other cloud software applications and APIs.

Your architecture graphics must reflect the components of *your* application by specifying what is done on the client side, what is done on the server side, and which APIs and web services are used by the server side.

# DESIGN SPECIFICATION

TBD

Provide a graphical description (e.g., **storyboard**, **images, diagrams, drawings**, etc.) of the functionality of your cloud software application’s design.

(If your app uses a file storage directory, name it as TeamN-FileStorage, where N is your team number.)

# DELIVERED SOFTWARE FUNCTIONALITY

TBD

Describe the functionality of your deployed cloud software application by using screenshots of user interfaces. Start with the URL for accessing your software. Then, screenshot by screenshot describe how your software is used. The entire software functionality must be described in sufficient detail.

# STEP-BY-STEP INSTRUCTIONS FOR DEVELOPING NEW FEATURES

## Password Salting and Hashing

User account databases are hacked and otherwise compromised frequently and, even worse, users tend to use the same or similar passwords in multiple services, so we want to store passwords in a way that they are protected even if the database containing them is compromised, but while still retaining the ability to verify that an entered password is correct. Salting and hashing passwords allows our system to validate entered passwords without needing to store the original passwords to compare them against. Our specific implementation uses the bcrypt cryptographic hash function, which is one of the handful of hash functions (Argon2, bcrypt, scrypt, Catena, Lyra2, Makwa, yescrypt, and PBKDF2) currently considered to provide an acceptable level of security. We chose bcrypt because it has a number of benefits over the other algorithms in our use case and we can mitigate its main drawback, which is truncation at 72 characters and on NUL characters. In order to mitigate this drawback, we preprocess passwords by taking a SHA-384 hash then base64-encoding that hash value. This strategy allows us to accept passwords of arbitrary length and passwords that include the NUL character, and in exchange we drop from 573 bits of potential entropy to 384 bits of entropy, which is still more than enough to remain highly secure.

The actual implementation of this hash chain (SHA-384 -> base64 -> bcrypt) is apparent in the following PasswordUtil class which encapsulates it:

|  |
| --- |
| */\* \* Created by Casey Butenhoff on 2017.11.18  \* \* Copyright (c) 2017 Casey Butenhoff. All rights reserved. \* \*/* package com.mycompany.controllers.util;  import java.security.MessageDigest; import java.security.NoSuchAlgorithmException; import java.util.Base64; import org.mindrot.jbcrypt.BCrypt;  */\*\* \* Hashing is a one-way mathematical transformation which is easy to perform but \* difficult to reverse. This process allows the system to avoid storing the \* actual password. Because many users use the same passwords on multiple \* systems, a security breach that exposes the actual passwords they use puts \* their accounts at risk at many other sites. \* \* Note that encrypted passwords are not as secure as hashed passwords, due to \* encryption algorithms not being one-way functions. A compromised encryption \* key (which must necessarily be stored on the system in order to encrypt the \* passwords in the first place) instantly defeats the security of every \* encrypted password on the system. Also note that this drawback don't apply to \* password hashes, so encrypting password hashes can still gain some benefits \* from encryption while not suffering the security issues, especially if the \* cloud application (containing the encryption key) and database are on \* different servers. \* \* When hashing passwords, there are a couple further considerations. Not just \* any old hashing scheme should be used. Many hash functions, such as SHA-1 for \* example, are simply too easy to compute. This creates a security problem in \* that an attacker can simply generate a rainbow table - a hash of many \* possible passwords - and do reverse-lookup of the compromised hash. \* \* One way to mitigate this problem is to generate a unique salt for each \* password, combine it with the password before hashing it, then store the hash \* and salt in the database. This makes it harder to generate a single rainbow \* table that can be used on all the passwords because each one is effectively \* much longer and more complex due to having a unique salt combined with it. \* Effectively, a separate rainbow table would have to be generated for each \* salt. A salt also protects against guessing passwords based on the frequency \* of a particular hash in the database, since even if two users use the same \* password their salted hashes will be different. \* \* The situation can be made even better by using a hash function that was \* actually designed for passwords. Such hash functions are intentionally harder \* to compute than most hash functions, which makes generating rainbow tables \* vastly more impractical. They also generally include a cost factor which can \* be increased to make the hash even harder as technology progresses. \* \* There are only a handful of hash functions designed for password hashing and \* currently (as of 2017) recognized as secure and best-practice: Argon2, \* bcrypt, scrypt, Catena, Lyra2, Makwa, yescrypt, and PBKDF2. PBKDF2 is by far \* the weakest password hashing scheme in this group. \* \* Some of these schemes are available through the well-respected libsodium \* library or other C libraries via Java Native Access (JNA) or Java Native \* Interface (JNI) but setup and deployment with these libraries can be tricky. \* Additionally, many of the Java implementations fail to provide sane defaults \* for parameters so they leave room for programmer error in providing them. \* \* The hash method we chose is based on the bcrypt password hashing scheme. This \* scheme was chosen because it's currently the most accessible scheme in the \* Java EE environment. It depends only on a pure-Java library and doesn't \* require JNA or JNI so deployment is simple and reliable. It provides sane \* defaults for parameters so its security doesn't depend on the programmer \* avoiding configuration mistakes. \* \* Unfortunately, bcrypt also has a serious drawback. The bcrypt algorithm \* truncates input at 72 characters and on NUL bytes. In order to mitigate this \* drawback, we are preprocessing the password using a combination of SHA-384 \* hashing and base64-encoding. A base64-encoded SHA-384 hash is 64 characters, \* which theoretically reduces the upper limit of input entropy (SHA-384 output \* contains 384 bits of entropy vs bcrypt input has 573 bits of potential \* entropy) but few users are going to enter passwords that take advantage of \* anywhere near this maximum potential entropy. In exchange for this reduction \* in entropy headspace, we effectively eliminate the upper limit on the length \* of passwords that we can accept, which has a larger benefit to security in \* practice than a higher theoretical maximum entropy. Despite the entropy \* reduction, we use SHA-384 rather than something like SHA-512 because SHA-384 \* has the benefit of being the largest hash function in the SHA-2 family that \* is resistant to length extension attacks. \* \* @author CJ \*/* public class PasswordUtil {     */\*\*     \* Hashes the given password into a form suitable for storing in the     \* database.     \*     \* @param password The cleartext password.     \* @return The bcrypt hash of the given password in Modular Crypt Format.     \*     \* This format contains:     \*     \* 1. A prefix indicating the hash function used     \*     \* 2. A cost parameter     \*     \* 3. 128-bit salt Radix-64 encoded as 22 characters     \*     \* 4. 184-bit hash value Radix-64 encoded as 31 characters     \*     \* This format has two advantages:     \*     \* 1. We don't need to store the cost parameter, salt, and hash separately     \* in the database.     \*     \* 2. We can more easily upgrade the hash function in the future as     \* technology improves, either by increasing the cost parameter or by     \* completely replacing the hash function, without needing to modify the     \* databse structure. A password can simply be upgraded to a new format,     \* after verifying it against the existing format, by re-hashing it with the     \* new scheme and storing the new result in the databse. Given that all the     \* parameters for a particular hash are stored in the database, the database     \* can easily contain passwords hashed with multiple schemes while they are     \* progressively upgraded.     \*     \* @throws NoSuchAlgorithmException If an implementation of the SHA-384     \* hashing algorithm can't be found on the system.     \*/*    public static String hashpw(String password) throws NoSuchAlgorithmException {        *// Get an implementation of the SHA-384 hashing algorithm from*        *// MessageDigest and compute the SHA-384 hash of the cleartext password.*        *// This converts the password, regardless of its original length, to*        *// 384 bits. This compresses long passwords and allows us to accept*        *// passwords of arbitrary length and still hash them with bcrypt.*        MessageDigest md = MessageDigest.getInstance("SHA-384");        byte[] sha384Hash = md.digest(password.getBytes());         *// Bcrypt accepts a character string as input, so encode the binary*        *// 384-bit SHA-384 hash as a character string containing 64 characters.*        Base64.Encoder encoder = Base64.getEncoder();        String encodedString = encoder.encodeToString(sha384Hash);         *// Generate a brand new salt for this user and hash the base64-encoded*        *// character string with bcrypt. Return the result in Modular Crypt*        *// Format*        return BCrypt.hashpw(encodedString, BCrypt.gensalt());    }     */\*\*     \* Hashes the given password using the salt from the given hash     \*     \* @param plaintext The cleartext password.     \* @param hashed The hashed password from the database in Modular Crypt     \* Format.     \* @return True if the hash of the given plaintext password matches the     \* given hash.     \* @throws NoSuchAlgorithmException If an implementation of the SHA-384     \* hashing algorithm can't be found on the system.     \*/*    public static boolean checkpw(String plaintext, String hashed) throws NoSuchAlgorithmException {        *// Get an implementation of the SHA-384 hashing algorithm from*        *// MessageDigest and compute the SHA-384 hash of the cleartext password.*        *// This converts the password, regardless of its original length, to*        *// 384 bits. This compresses long passwords and allows us to accept*        *// passwords of arbitrary length and still hash them with bcrypt.*        MessageDigest md = MessageDigest.getInstance("SHA-384");        byte[] sha384Hash = md.digest(plaintext.getBytes());         *// Bcrypt accepts a character string as input, so encode the binary*        *// 384-bit SHA-384 hash as a character string containing 64 characters.*        Base64.Encoder encoder = Base64.getEncoder();        String encodedString = encoder.encodeToString(sha384Hash);         *// Using the salt from the given hashed password, hash the*        *// base64-encoded character string with bcrypt and compare it with the*        *// given hash.*        return BCrypt.checkpw(encodedString, hashed);    } } |

Take note that the value returned by BCrypt.hashpw(), and therefore PasswordUtil.hashpw(), is a combination of a code identifying the hash algorithm, a cost parameter, the salt, and the password hash all in a single string in Modular Crypt Format. We can store this one string in a single column of the database, and it provides us some potential future maintenance benefits. Since the hash algorithm and cost parameter are stored with the hash, in the future when technology has progressed enough that our password hashing scheme has become too weak, we can progressively upgrade the hash algorithm for individual users as they log in without needing to change the structure of the database. When our cloud application accesses a password hash in the database, it can simply detect if it was hashed using an outdated algorithm, still check the entered password against it, then re-hash the password using whatever the new algorithm is.

When creating new user accounts, the users enter cleartext passwords which our system receives. Then, in order to actually persist their account to the cloud application’s database, we simply hash the cleartext password they gave us with PasswordUtil.hashpw() before persisting the account to the database, as in the following method:

|  |
| --- |
| */\*    Create a new user account. Return "" if an error occurs; otherwise,    upon successful account creation, redirect to show the SignIn page.     \*/*    public String createAccount() throws NoSuchAlgorithmException {         *//-----------------------------------------------------------*        *// First, check if the entered username is already being used*        *//-----------------------------------------------------------*        *// Obtain the object reference of a User object with username*        User aUser = getUserFacade().findByUsername(username);         if (aUser != null) {            *// A user already exists with the username entered*            username = "";            statusMessage = "Username already exists! Please select a different one!";            return "";        }         *//----------------------------------*        *// The entered username is available*        *//----------------------------------*        if (statusMessage == null || statusMessage.isEmpty()) {            try {                *// Instantiate a new User object*                User newUser = new User();                 */\*                Set the properties of the newly created newUser object with the values                entered by the user in the AccountCreationForm in CreateAccount.xhtml                 \*/*                newUser.setFirstName(firstName);                newUser.setMiddleName(middleName);                newUser.setLastName(lastName);                newUser.setAddress1(address1);                newUser.setAddress2(address2);                newUser.setCity(city);                newUser.setState(state);                newUser.setZipcode(zipcode);                newUser.setSecurityQuestion(securityQuestion);                newUser.setSecurityAnswer(securityAnswer);                newUser.setEmail(email);                newUser.setUsername(username);                 *// When creating a new user, we must hash the cleartext password*                *// and persist the hash in the database instead of the cleartext*                *// password.*                newUser.setHashedPassword(PasswordUtil.hashpw(password));                 getUserFacade().create(newUser);             } catch (EJBException e) {                username = "";                statusMessage = "Something went wrong while creating user's account! See: " + e.getMessage();                return "";            }            *// Initialize the session map for the newly created User object (see the method below)*            initializeSessionMap();             */\*            The Profile page cannot be shown since the new User has not signed in yet.            Therefore, we show the Sign In page for the new User to sign in first.             \*/*            return "SignIn.xhtml?faces-redirect=true";        }        return "";    } |

The PasswordUtil class generates a brand new salt for the user and returns it along with the hashed password in Modular Crypt Format, which we store directly in the database.

When authenticating a user against an existing user account, the users again enter cleartext passwords which our system receives. But in this case, we don’t really care about storing the hash of the cleartext password, we simply want to know whether or not it matches the one stored in the database. So we simply pass the cleartext password and the password hash stored in the database for the corresponding user (again in Modular Crypt Format) into PasswordUtil.checkpw(), as in the following method:

|  |
| --- |
| *// Validate if the entered password and confirm password are correct*    public void validateUserPassword(ComponentSystemEvent event)            throws NoSuchAlgorithmException {        */\*        FacesContext contains all of the per-request state information related        to the processing of a single JavaServer Faces request, and the        rendering of the corresponding response. It is passed to, and        potentially modified by, each phase of the request processing lifecycle.         \*/*        FacesContext fc = FacesContext.getCurrentInstance();         */\*        UIComponent is the base class for all user interface components in        JavaServer Faces. The set of UIComponent instances associated with a        particular request and response are organized into a component tree        under a UIViewRoot that represents the entire content of the request or        response.         \*/*        *// Obtain the UIComponent instances associated with the event*        UIComponent components = event.getComponent();         */\*        UIInput is a kind of UIComponent for the user to enter a value in.         \*/*        *// Obtain the object reference of the UIInput field with id="password"*        *// on the UI*        UIInput uiInputPassword = (UIInput) components.findComponent("password");         *// Obtain the password entered in the UIInput field with id="password"*        *// on the UI*        String entered\_password = uiInputPassword.getLocalValue()                == null ? "" : uiInputPassword.getLocalValue().toString();         *// Obtain the object reference of the UIInput field with*        *// id="confirmPassword" on the UI*        UIInput uiInputConfirmPassword =                (UIInput) components.findComponent("confirmPassword");         *// Obtain the confirm password entered in the UIInput field with*        *// id="confirmPassword" on the UI*        String entered\_confirm\_password = uiInputConfirmPassword.getLocalValue()                == null ? "" : uiInputConfirmPassword.getLocalValue().toString();         if (entered\_password.isEmpty() || entered\_confirm\_password.isEmpty()) {            *// If the user didn't enter one or both of the password fields, we*            *// don't need to take any action because the required="true" in the*            *// XHTML file will catch this and produce an error message.*            return;        }         *// No need to hash these since we're not authenticating the user, just*        *// verifying that they entered the same password twice.*        if (!entered\_password.equals(entered\_confirm\_password)) {            *// If the user entered two different passwords then we set an error*            *// message in order to indicate that the password validation has*            *// failed.*            statusMessage = "Password and Confirm Password must match!";        } else {            *// Obtain the logged-in User's username*            String user\_name = (String) FacesContext.getCurrentInstance().                    getExternalContext().getSessionMap().get("username");             *// Obtain the object reference of the signed-in User object*            User user = getUserFacade().findByUsername(user\_name);             *// When authenticating a user to change their password, we must take*            *// the cleartext user password the user entered and compare it*            *// against the password that's stored in the database for that user.*            *// Since we're storing the hashed password in the database, we must*            *// salt the cleartext password with the same salt as the password*            *// stored in the database and hash it then compare the two hashes.*            *// This is all encapsulated within a single call to checkpw, and we*            *// only need to pass the cleartext password and the hashed password*            *// since the hashed password is stored in Modular Crypt Format which*            *// means it includes the cost parameter and salt that was used to*            *// hash it.*            if (PasswordUtil.checkpw(entered\_password, user.getHashedPassword())) {                *// entered password = signed-in user's password*                statusMessage = "";            } else {                *// If the user entered the wrong password, then we must set an*                *// error message in order to indicate that the password*                *// validation has failed.*                statusMessage = "Incorrect Password!";            }        }    } |

## Database Triggers

Several of our features including the RSS feed depend on being able to track when something in the system changes. One way to do this would be to make our client insert a new activity object into the database whenever it changes something, but this is too burdensome on the client and highly prone to programmer error. A better solution is to automatically record changes to the database as they are made, using database triggers. Triggers are little SQL scripts that automatically execute under certain conditions. A trigger can be set to execute either before or after insert, update, or delete operations occur on a particular table. It is automatically given the row(s) that were added, changed, or deleted and it can do whatever it wants with them. For example, the trigger created by this create statement executes after a new project is added to the Project table and inserts a new row into the Activity table containing a message that new project was created and the id of the project:

|  |
| --- |
| */\* Since the triggers may contain semicolon characters, we must change the delimiter to something else temporarily so that we can create the trigger with the correct contents. We are using a double dollar sign delimiter here since that is unlikely to conflict with anything else. \*/* DELIMITER $$  */\* Since this is the database creation script, we can assume that the trigger doesn't already exist and simple create it. \*/* CREATE TRIGGER     */\*    Each trigger must have a unique name, so we use a naming convention that    informs us which table the trigger is associated with, whether the    trigger is executed before or after the action, and which action the    trigger is actually listening for. There shouldn't be multiple triggers    on a particular table listening for the same thing so this convention    guarantees that we have a unique name while also being informative.    \*/*    `project\_after\_insert`     */\*    An INSERT trigger is only executed when a new row is inserted. We can choose    to have it executed either BEFORE or AFTER the insert operation occurs,    which would be an important distinction to make if the action the trigger    took involved foreign keys or something since a trigger is not allowed to    violate database integrity.    \*/*    AFTER INSERT     */\*    Define the name of the database table this trigger is attached to. This    table must already exist when we run this CREATE TRIGGER command so this    command should be below the CREATE TABLE Project command.    \*/*    ON `Project`     */\*    A database operation may have affected multiple rows, and our trigger    receives all of them at once, so it must iterate over each one of them    in turn.    \*/*    FOR EACH ROW BEGIN         */\*        MySQL triggers automatically create NEW and OLD row variables as        appropriate that contain the actual row data from the table the        trigger is associated with that was involved in whatever type of        change the trigger was watching for.         In an INSERT trigger, only NEW.col\_name can be used.         In this case, we are simply making a record of the change in the        Activity table, which is essentially functioning like an audit        log. This table is used as the data source for certain features        like the RSS feed.        \*/*        INSERT INTO `Activity` (                type,                message,                project\_id        ) VALUES (                'INSERT\_PROJECT',                concat('Created project ', NEW.name),                NEW.id        );    */\*    We must explicitly close the FOR EACH loop.    \*/*    END  */\* Since we previously changed the delimiter to a double dollar sign, we must terminate the CREATE TRIGGER command with a double dollar sign in order to execute it. \*/* $$  */\* After we've created the trigger, we must reset the delimiter back to a semicolon so that future commands are terminated and executed correctly. \*/* DELIMITER ; |

## RSS Feed

An RSS feed allows our users to access all activity that occurs within one of their projects in a standardized and machine-readable format. An RSS feed reader can automatically check the RSS feed address periodically and pull down the latest project updates, which include some potentially useful metadata such as the type of activity and the date and time that it occurred. This allows third-party tools to easily interface with our system in order to perform analysis of the activity that occurs within the project.

Since an RSS feed is a dynamic XML response and not a web page, providing an RSS feed required us to create a web service. In order to do that, we needed to first subclass Application and define a base ApplicationPath like so:

|  |
| --- |
| */\* Identifies the application path that serves as the base URI for all resource URIs provided by Path. We can't use the '/' path because that's already reserved for serving our actual web pages, so we must define a sub-path for web services. Basically, this means that all JAX-RS web services that our application exposes will be accessible at addressed of the form:   http://base\_url/webresources/endpoint\_address  JAX-RS is nice to use in the Java EE environment because it requires no particular configuration and provides a simple annotation-based interface to deploy web services with. \*/* @ApplicationPath("webresources")  */\* @ApplicationPath requires that we subclass Application. This class defines a JAX-RS application, a Java API for RESTful Web Services which uses annotations to easily turn plain old Java objects (POJOs) into REST endpoints. \*/* public class ApplicationConfig extends Application {     @Override    public Set<Class<?>> getClasses() {        Set<Class<?>> resources = new java.util.HashSet<>();        addRestResourceClasses(resources);        return resources;    }     */\*\*     \* Do not modify addRestResourceClasses() method.     \* It is automatically populated with     \* all resources defined in the project.     \* If required, comment out calling this method in getClasses().     \*/*    private void addRestResourceClasses(Set<Class<?>> resources) {        resources.add(com.mycompany.rss.ActivityFeed.class);    }  } |

As described in the documentation of that code snippet, this class exposes our web services under the ‘webresources’ sub-path of our application root path.

In order to implement the RSS feed, we used the ROME library, which has long been the gold-standard for generating RSS feeds using Java. You can see how easy it is to use in the following resource class, which actually generates our RSS feed and makes it available from a REST API endpoint:

|  |
| --- |
| */\* JAX-RS provides a number of potentially-useful annotations for mapping POJOs which represent resource classes to web services. These annotations include:   @Path - defines the path (relative to @ApplicationPath) of this web service  @GET - specifies the resource can be accessed using a HTTP GET request  @PUT - specifies the resource can be accessed using a HTTP PUT request  @POST - specifies the resource can be accessed using a HTTP POST request  @DELETE - specifies the resource can be accessed using a HTTP DELETE request  @HEAD - specifies the resource can be accessed using a HTTP HEAD request  @Produces - specifies which Internet media types the response may come in  @Consumes - specifies which Internet media types the request may use  In this case, we're using the @Path annotation because we want the ActivityFeed resource class to be mapped to a web service at the address:   http://base\_url/webresources/rss  By default, this resource class has a per-request lifecycle, so it's created and destroyed each time a request to its web service endpoint is made. \*/* @Path("/rss") public class ActivityFeed {     */\*    Since we have a different RSS feed for each project and the content of the    RSS feed contains details of the project, we must inject the ProjectFacade    in order to resolve the RSS key that was given to the web service to an    instance of an actual project.    \*/*    @EJB    private com.mycompany.FacadeBeans.ProjectFacade projectFacade;     */\*    Since the RSS feed pulls all of it's feed data from the Activity table, we    need to inject the ActivityFacade in order to get that data from the    database.    \*/*    @EJB    private com.mycompany.FacadeBeans.ActivityFacade activityFacade;     */\*    In order to generate some of the urls in our RSS feed, we need to access the    actual request paths, so we must inject the request context which will    provide them for us.    \*/*    @Context    private UriInfo context;     */\*\*     \* Creates a new instance of ActivityFeed     \*/*    public ActivityFeed() {    }     */\*\*     \* Retrieves the RSS feed for the project with the given RSS key.     \*     \* @return an RSS feed in XML format containing all the activity that has     \* occurred on the project associated with the given RSS key.     \*/*    */\*    We annotate this method with the @GET annotation to indicate that it    responds to HTTP GET requests.    \*/*    @GET    */\*    We annotate this method with MediaType.APPLICATION\_XML passed to the    @Produces annotation to indicate that its response conforms to the XML    Internet media type.    \*/*    @Produces(MediaType.APPLICATION\_XML)    */\*    The parameter to this method is annotated with @QueryParam("key") to    indicate that this method accepts an argument in the form of a 'key' query    parameter in the request path. In other words, requests conforming to a    path like this:     http://base\_url/webresources/rss?key=foobar     will pass the value 'foobar' as the value of the key parameter of this    method.    \*/*    public String getXml(@QueryParam("key") String key)            throws FeedException, NoSuchAlgorithmException {        List entries = new ArrayList();         *// Since we only have the project's RSS key, we must get the actual*        *// project instance in order to retrieve it's name and activity.*        Project project = projectFacade.findByRssKey(key);         *// In case we were given an incorrect project RSS key, we should still*        *// return a valid XML response containing a message.*        String feedTitle = "No Activity";        String feedDescription = "No rss key has been given " +                "or no activity has been found for the given RSS key";         if (project != null) {             *// If we actually found the project associated with the given RSS*            *// key, update the feed title and description to reflect it.*            feedTitle = project.getName() + " Project Activity";            feedDescription = "All activity that has occurred in the " +                project.getName() + " project since it was created.";             *// Iterate over all the activity associated with the project and add*            *// it to the RSS feed.*            for (Activity activity : activityFacade.findByProject(project)){                 *// This entry class comes from the ROME library, which is very*                *// mature, well-documented, and has been the gold-standard for*                *// Java RSS feed generation for many years.*                SyndEntry entry = new SyndEntryImpl();                 *// The title of each RSS entry is the type of change that the*                *// Activity row represents. i.e. INSERT\_PROJECT, UPDATE\_PROJECT,*                *// etc.*                entry.setTitle(activity.getType());                 *// Since there is no particular web page associated with this*                *// activity, we simply make the entry link back to the feed.*                entry.setLink(context.getRequestUri().toString());                 *// Use the timestamp representing when the row was added to the*                *// Activity table as the publish date for this entry.*                entry.setPublishedDate(activity.getTimestamp());                 *// The content of each RSS entry is simply the message we stored*                *// in the Activity table. It's nothing fancy, just plain text,*                *// so we set its content type accordingly.*                SyndContent description = new SyndContentImpl();                description.setType("text/plain");                description.setValue(activity.getMessage());                entry.setDescription(description);                 *// Add the entry to the collection of entries that the feed will*                *// contain.*                entries.add(entry);            }        }         *// This feed class comes from the ROME library, which is very mature,*        *// well-documented, and has been the gold-standard for Java RSS feed*        *// generation for many years.*        SyndFeed feed = new SyndFeedImpl();         *// ROME supports other feed types such as ATOM, but we're currently only*        *// interested in providing an RSS 2.0 feed at the moment.*        feed.setFeedType("rss\_2.0");         *// This title was generated above and either contains a short title with*        *// the project title or an error message depending on if the given RSS*        *// key is valid.*        feed.setTitle(feedTitle);         *// This description was generated above and either contains a short*        *// description with the project title or an error message depending on*        *// if the given RSS key is valid.*        feed.setDescription(feedDescription);         *// Since there is no particular web page associated with this RSS feed,*        *// we simply make the feed link back to itself.*        feed.setLink(context.getRequestUri().toString());         *// Actually add all the entries to the RSS feed and return it formatted*        *// as an XML document.*        feed.setEntries(entries);        return new SyndFeedOutput().outputString(feed);    } } |

You might notice that this endpoint requires an RSS key, which is a cryptographically-generated key that identifies a particular project. We generate and associate these keys with projects as the projects are created like so:

|  |
| --- |
| public void create() throws NoSuchAlgorithmException {        *// When creating a new project, we must hash the cleartext password and*        *// persist the hash in the database instead of the cleartext password.*        selected.setHashedPassword(PasswordUtil.hashpw(cleartextPassword));         *// Once we've hashed the cleartext password, we want to keep it from*        *// leaking into other code that uses this controller in the future, and*        *// we don't need it anymore so we can clear it.*        cleartextPassword = null;         *// we also derive a key from the password that identifies this project*        *// for the purposes of adding some security to its rss feed.*        String mcfHash = PasswordUtil.hashpw(selected.getHashedPassword());         *// We don't need to ever verify this RSS hash against anything, so we*        *// can discard all the Modular Crypt Format metadata.*        String[] splitMcfHash = mcfHash.split("$");        String payload = splitMcfHash[splitMcfHash.length - 1];         *// For the same reason, we can also discard the first 22 characters,*        *// which contain the salt.*        String hashOnly = payload.substring(22);         *// The hash can still contain some non-alphanumeric characters, and for*        *// the sake of simplicity, we only want alphanumeric characters, so we*        *// remove any non-alphanumeric characters by replacing them with an*        *// empty string.*        String alphaNumericHash = hashOnly.replaceAll("[^A-Za-z0-9]", "");         *// Save the RSS hash to the project. This project's activity will now be*        *// accessible by providing this RSS hash to the RSS Feed web service.*        selected.setRssKey(alphaNumericHash);         persist(PersistAction.CREATE, ResourceBundle.getBundle("/Bundle").getString("ProjectCreated"));        if (!JsfUtil.isValidationFailed()) {            items = null;    *// Invalidate list of items to trigger re-query.*        }    } |

As you can see, the RSS key is derived from the password hash by taking a second hash of it. This is just a nice shortcut to produce a string that should be extremely hard to guess. We actually discard the Modular Crypt Format metadata and salt since this we simply use this key as if it was a random string of characters. The key is made directly available to users who have joined a particular project (via the project’s RSS feed link) and it won’t be matched against anything so the metadata is not necessary in this case. So given an RSS key, how do we find the project? We simply perform a database query with the given RSS key as the parameter like so:

|  |
| --- |
| */\*\*     \* Given a project's RSS key, returns the project associated with that key,     \* if one exists. Otherwise, returns null.     \*     \* @param rssKey The RSS key of the project to retrieve.     \* @return The project associated with the given RSS key, or null.     \*/*    public Project findByRssKey(String rssKey) {        *// getSingleResult() throws an exception if the entity can't be found,*        *// so we must first make sure that it exists in order to avoid needing*        *// to catch and handle an unnecessary exception.*        if (em.createNamedQuery("Project.findByRssKey")                .setParameter("rssKey", rssKey)                .getResultList().isEmpty()) {            *// If we can't find even a single project with the given RSS key, we*            *// return null to indicate that it doesn't exist.*            return null;        } else {            *// There should only ever be at most one project with a particular*            *// RSS key, so simply return it. In the unlikely event that there is*            *// more than one result, this will throw a NonUniqueResultException.*            return (Project) em.createNamedQuery("Project.findByRssKey")                    .setParameter("rssKey", rssKey)                    .getSingleResult();        }    } |

Given the method we use to generate it, each project is almost certain to have a unique RSS key so we can use getSingleResult() and assume that it won’t throw an exception due to a duplicate key.

## Project Activity Heatmap

When managing a project, it can be helpful to know the distribution of when things are actually being worked on in order to manage risk. For example, if your team tends to work mostly right before deadlines, then your project has a larger risk of failure and missing deadlines since at that point it’s difficult to react to unforeseen problems. In order to help with that, we’ve implemented a heatmap visualization which takes all the activity that has occurred on a particular project and organizes it by day. The heatmap organizes the days into weeks since the project was started and colors each day according to the amount of activity that occurred on it, relative to the largest amount of activity that occurred in a single day on that project. This coloring allows the data to be quickly scanned by eye in a way that makes patterns more obvious than simply looking at a table of data. Alternatively, this data could be plotted in a graph, but it’s difficult for a single graph to match the flexibility and helpfulness towards pattern discovery that a heatmap has.

We are using a datatable with dynamic columns to actually construct the heatmap UI. We selected this type of control because we wanted the heatmap to show data in the form of both full and partial weeks. We consider Sunday to be the start of a week, and if a project was started on a later weekday we still wanted the heatmap to show the first several days of that week even though they contained no activity. Additionally, if there are days in the week that have not occurred yet, we want to skip those days since they would otherwise be misleading. If a project’s lifetime has not yet spanned into a second week and there are days in the week that have not occurred yet, those entire weekday columns which do not yet fall into the project’s timespan would ideally be entirely omitted. For this reason, we are using a datatable control with dynamic columns, which allows us to add or remove columns from the datatable at runtime. We demonstrate here how such a dynamic datatable is defined in an xhtml document:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> *<!-- Created by Casey Butenhoff on 2017.11.18 Copyright (c) 2017 Casey Butenhoff. All rights reserved. -->* <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> <html xmlns="http://www.w3.org/1999/xhtml"      xmlns:h="http://java.sun.com/jsf/html"      xmlns:ui="http://java.sun.com/jsf/facelets"      xmlns:p="http://primefaces.org/ui"      xmlns:f="http://xmlns.jcp.org/jsf/core">     *<!-- Do not enter tags before the composition line since they are ignored by JSF -->*     *<!-- This page is constructed based on the siteTemplate -->*    <ui:composition template="template/siteTemplate.xhtml">        <ui:define name="title">            *<!-- Set the page title -->*            <h:outputText value="#{heatmapController.project.name} Project Activity Heatmap"></h:outputText>        </ui:define>         *<!-- Create the content for this page below -->*        <ui:define name="editableContent">            <h:form>                <p:growl id="messages" showDetail="true" />                 <div align="center">                    <h3>#{heatmapController.project.name} Project Activity Heatmap</h3>                     *<!--                    Show a table of heatmap values using data from the                    controller and a dynamic number of columns which are also                    provided by the controller. Iterates over each row in                    heatmapMaps and stores the current one in the 'heatmapMap'                    variable.                    -->*                    <p:dataTable value="#{heatmapController.heatmapTable.heatmapMaps}"                                 var="heatmapMap"                                 styleClass="borderless headerless unpadded"                                 tableStyle="width:auto">                         *<!--                        Since we are generating the columns dynamically, we must                        retrieve column objects from the controller which                        contain the information necessary to set them up. The                        columns are iterated over and each column is assigned to                        the 'column' variable in turn for use by the enclosed                        code.                        -->*                        <p:columns value="#{heatmapController.heatmapTable.columns}" var="column">                             *<!--                            We define the header text for the column in a                            special named section called a facet. This prevents                            the header text from being repeated in each row even                            though most of the code within the p:datatable tag                            is repeated for each 'heatmapMap' row in                            heatmapController.heatmapTable.heatmapMaps.                            -->*                            <f:facet name="header">                                 *<!--                                The header simply contains the header text                                defined by the column object we retrieved from                                the controller                                -->*                                <h:outputText value="#{column.header}" />                            </f:facet>                             *<!--                            This defines the actual contents of the cell in the                            current column. heatmapMap contains the current row                            that p:datatable is iterating over, so we must                            index into it in order to find the actual data that                            corresponds to the current column that p:columns                            is iterating over. We also set the background color                            to the appropriate color in order to make it easy to                            glance at the chart and immediately know how the                            project activity was distributed over its lifetime.                            -->*                            <h:outputText value="#{heatmapMap[column.property].value}"                                          style="background-color: #{heatmapMap[column.property].color}" />                        </p:columns>                    </p:dataTable>                    <br /><br />                 </div>            </h:form>            <style type="text/css">                .value {                    width: 1000px;                }            </style>        </ui:define>    </ui:composition> </html> |

As you can see, we can get the column definitions, including the column headers, from the Java bean at runtime by using the p:columns and f:facet tags. In order to implement the controller that this control references, we again used the Activity table which is automatically populated by triggers with all activity that occurs on a project. We transform that data into a daily frequency distribution using a hashmap, then transform that frequency distribution hashmap into a data structure more suitable for a dynamic datatable. As you can see in the following code, this second data structure contains a list of column definitions (HeatmapColumns) and a list of data rows (HeatmapMaps). In the data rows, individual cells (HeatmapDays) reference the column definitions stored in the top-level data structure.

|  |
| --- |
| @Named("heatmapController") @SessionScoped public class HeatmapController implements Serializable {     @EJB    private ActivityFacade activityFacade;     private HeatmapTable heatmapTable;     private Project project;     public HeatmapController() {    }     public HeatmapTable getHeatmapTable() {        return heatmapTable;    }     public void setHeatmapTable(HeatmapTable heatmapTable) {        this.heatmapTable = heatmapTable;    }      public Project getProject() {        return project;    }     protected void setEmbeddableKeys() {    }     protected void initializeEmbeddableKey() {    }     public String generate(Project project) {        this.project = project;        if (project != null) {            LocalDate today = new LocalDate();             *// This keeps track of the earliest date of any of this project's*            *// activity. Since our heatmap displays the entire history of the*            *// project, it helps us later to determine which date out heatmap*            *// should begin on.*            LocalDate earliest = today;             *// This keeps track of the largest single day of activity for this*            *// project. It's used later to scale the coloring so that regardless*            *// of the actual magnitude of the activities, the day with the most*            *// activity will always be colored with the brightest color we have.*            int maxCount = 1;             *// This will contain the frequency of activity on each date that is*            *// represented in the activity table for this project. In other*            *// words, it's a sparse frequency table.*            HashMap<LocalDate, Integer> activityCounter = new HashMap<>();             *// Iterate over every piece of activity that was related to this*            *// project.*            for (Activity activity : activityFacade.findByProject(project)) {                 *// Java Date returns odd values such as years relative to 1900*                *// which we don't want to have to deal with, so we initialize a*                *// calendar instance with it instead, which lets us access the*                *// date in a way that makes a bit more sense.*                Date date = activity.getTimestamp();                Calendar calendar = Calendar.getInstance();                calendar.setTime(date);                 *// Java Calendar uses 0-indexing for month (but not for year*                *// or date), but LocalDate expects everything to be 1-indexed so*                *// we must add one to the month in order to convert it to*                *// 1-indexed form.*                LocalDate localDate = new LocalDate(                        calendar.get(Calendar.YEAR),                        calendar.get(Calendar.MONTH) + 1,                        calendar.get(Calendar.DATE));                 *// We need to keep track of the date of the earliest activity in*                *// this project so that we know which date to start the heatmap*                *// on.*                if (localDate.isBefore(earliest)) {                    earliest = localDate;                }                 *// We count each entry in the activity table as one unit of*                *// activity and simply sum them together. A more sophisticated*                *// version of this algorithm might assign each entry a weight*                *// depending on its significance or some other property.*                Integer count = activityCounter.containsKey(localDate)                        ? activityCounter.get(localDate) + 1                        : 1;                if (count > maxCount) {                    maxCount = count;                }                activityCounter.put(localDate, count);            }             *// Start on Sunday of the earliest week. Joda is 1-indexed and*            *// starts with Monday = 1 so we must take modulo 7 in order to make*            *// Sunday = 0.*            while (earliest.dayOfWeek().get() % 7 > 0) {                earliest = earliest.minusDays(1);            }             heatmapTable = new HeatmapTable();             *// Create a special column for a row label that will indicate which*            *// week that particular row represents.*            HeatmapColumn rowLabelColumn = new HeatmapColumn();            heatmapTable.getColumns().add(rowLabelColumn);             *// This special column is simply to label each row so no need to*            *// assign it a header value.*            rowLabelColumn.setHeader("");             *// The property field functions as the unique key that identifies*            *// this column in a datatable with dynamic columns. Actual row*            *// entries will need to reference this property field in order to*            *// become associated with this column.*            rowLabelColumn.setProperty("RowLabel");             *// Since this is for a datatable control with dynamic columns, we*            *// must supply the data for each row in map form, with the key of*            *// each entry being the column that entry corresponds to and the*            *// value being the actual data that should be displayed in that*            *// cell.*            HashMap<String, HeatmapDay> heatmapMap = null;             *// We want to include activity that occurred today, so we must loop*            *// until just before we reach tomorrow.*            LocalDate tomorrow = today.plusDays(1);            for (LocalDate current = earliest; current.isBefore(tomorrow);                    current = current.plusDays(1)) {                 *// we want to start a each week on Sunday but Joda is*                *// 1-indexed and starts with Monday = 1 so we must take modulo 7*                *// in order to make Sunday = 0.*                int currentDayOfWeek = current.dayOfWeek().get() % 7;                if (currentDayOfWeek == 0) {                      *// We want each row in the data table to represent a single*                    *// week, so we start a new HashMap for each new week we*                    *// encounter.*                    heatmapMap = new HashMap<>();                    heatmapTable.getHeatmapMaps().add(heatmapMap);                     *// In the first column of each week row, we want to insert*                    *// a label indicating which week that row represents. We*                    *// don't want this cell to contain a background color, so*                    *// we give it a completely transparent one (with the last*                    *// two bytes, which represent the alpha value, as 00 to make*                    *// the color completely transparent).*                    heatmapMap.put(rowLabelColumn.getProperty(),                            new HeatmapDay("#00000000",                                    "Week of " + current.toString()));                }                 *// Each day of the week must have a unique property string. The*                *// property field functions as the unique key that identifies*                *// the column in a datatable with dynamic columns. Actual row*                *// entries will need to reference a property field in order to*                *// become associated with the corresponding column. We aren't*                *// pre-computing these column properties before the LocalDate*                *// for loop because we don't want to show columns for days of*                *// the week beyond the current day if the project hasn't existed*                *// long enough to have a second row of activity data.*                String columnProperty = String.format("%d", currentDayOfWeek);                boolean found = false;                for (HeatmapColumn heatmapColumn : heatmapTable.getColumns()) {                    if (heatmapColumn.getProperty().equals(columnProperty)) {                        found = true;                    }                }                if (!found) {                    HeatmapColumn heatmapColumn = new HeatmapColumn();                    switch (currentDayOfWeek) {                        case 0:                            heatmapColumn.setHeader("Sunday");                            break;                        case 1:                            heatmapColumn.setHeader("Monday");                            break;                        case 2:                            heatmapColumn.setHeader("Tuesday");                            break;                        case 3:                            heatmapColumn.setHeader("Wednesday");                            break;                        case 4:                            heatmapColumn.setHeader("Thursday");                            break;                        case 5:                            heatmapColumn.setHeader("Friday");                            break;                        case 6:                            heatmapColumn.setHeader("Saturday");                            break;                    }                    heatmapColumn.setProperty(columnProperty);                    heatmapTable.getColumns().add(heatmapColumn);                }                 *// Get the actual frequency of activity that occurred on the*                *// current day of our LocalDate loop. If there was no activity*                *// on a certain day, we still want to put a zero there to make*                *// it obvious that there was zero activity.*                Integer count = activityCounter.containsKey(current)                        ? activityCounter.get(current)                        : 0;                 *// As mentioned before, we want to scale the magnitudes such*                *// that regardless of what they actually are, the day with the*                *// most activity will always be colored with the same maximum*                *// level of brightness, and lower magnitudes will be*                *// correspondingly scaled from there. We accomplish that by*                *// taking the ratio of activity count on the current day to the*                *// maximum count seen on any one day in this project. That*                *// fractional number is then multiplied by 255 in order to scale*                *// it two bytes.*                long colorLong = Math.round(((double) count / maxCount) \* 255);                 *// We use the two byte activity frequency number we scaled above*                *// to generate the color string. In this case, we are using a*                *// red background color for all cells in the heatmap and simply*                *// adjusting the alpha value in order to raise and lower the*                *// intensity according to the actual magnitude of the scaled*                *// value. %2x in the format string converts the two byte count*                *// value into a two character hexadecimal representation, which*                *// is placed in the alpha position of the eight-byte color. For*                *// the alpha value, FF corresponds to completely opaque while*                *// 00 corresponds to completely transparent.*                String color = String.format("#FF0000%2x", colorLong);                 *// We want to show the actual magnitude of activity on each day*                *// in the cell of the heatmap that it corresponds to, so we must*                *// convert the integer*                String value = String.format("%d", count);                 heatmapMap.put(columnProperty, new HeatmapDay(color, value));            }        }         *// Redirect to the xhtml page which will actually pull the data we just*        *// generated from this controller and display it in a dynamic datatable.*        return "ProjectHeatmap.xhtml?faces-redirect=true";    } } |

The classes that make up the dynamic table data structure are fairly obvious, but I will include them here anyway for the sake of completeness. You can see the implementation of the top-level structure, HeatmapTable, here:

|  |
| --- |
| */\*\* \* Represents all the data required to construct a single heatmap datatable \* containing dynamic columns. \* \* @author CJ \*/* public class HeatmapTable {     */\*\*     \* Contains all the rows represented in the table. The string key in*  *\* this collection must match the property field of the column that*  *\* day is associated with, because this is how days are actually*  *\* associated with the dynamic columns.     \*/*    private final List<HashMap<String, HeatmapDay>> heatmapMaps = new ArrayList<>();     */\*\*     \* The dynamic columns that should be displayed by the data table. Only     \* columns in this list and data that references these columns will     \* ultimately be shown in the datatable.     \*/*    private final List<HeatmapColumn> columns = new ArrayList<>();     public List<HashMap<String, HeatmapDay>> getHeatmapMaps() {        return heatmapMaps;    }     public List<HeatmapColumn> getColumns() {        return columns;    } } |

Clearly, this structure contains both HeatmapColumn and HeatmapDay objects. The definition of HeatmapColumn contains only the header text that the column should display and a key value that HeatmapDay objects must be associated with (which is done via the heatmapMaps collection in HeatmapTable above) in order to be included in the column. The implementation is as follows:

|  |
| --- |
| */\*\* \* Represents a single dynamic column in a heatmap datatable. \* \* @author CJ \*/* public class HeatmapColumn {     */\*\*     \* The text to display in the header of this column.     \*/*    private String header;     */\*\*     \* A key that uniquely identifies this column in the HeatmapTable's columns     \* collection.     \*/*    private String property;     public String getHeader() {        return header;    }     public void setHeader(String header) {        this.header = header;    }     public String getProperty() {        return property;    }     public void setProperty(String property) {        this.property = property;    } } |

Similarly, HeatmapDay must contain the data actually shown in the cell. In our case, this means the magnitude of the activity count that occurred on that day and a color string generated from the relative size of that count compared to the project’s most active day. The implementation of that class is simply like so:

|  |
| --- |
| */\*\* \* Represents a single day in a heatmap datatable. \* \* @author CJ \*/* public class HeatmapDay {     */\*\*     \* The background color (in hex format, such as #FFFFFFFF) that should be     \* used for this day.     \*/*    private final String color;     */\*\*     \* The string to actually display in the cell for this day.     \*/*    private final String value;     public HeatmapDay(final String color, final String value) {        this.color = color;        this.value = value;    }     public String getColor() {        return color;    }     public String getValue() {        return value;    } } |

## Realtime Chat

For any project a key component is communication. Without it projects easily can fall behind and stagnate. Realizing this we decided to add a realtime chat option that would be the center of every projects home page. In the past a realtime chat would have to be implemented using asynchronous AJAX queries. This could become quite inefficient however as many HTTP requests would have to be sent just to poll for messages that not might even be there. However in modern times we are able to leverage websockets using PrimeFaces Push Servlet technology.

For the implementation of this technology we are making use of multiple third party libraries.

The following libraries should be added to the lib folder like any other jar file throughout the semester:

atmosphere-runtime

slf4j-api

We are making use of the Primefaces Push Servlet. This must be configured to load on startup of the application, and we must tell the servlet what directory is mapped to. In the web.xml file the following was added:

|  |
| --- |
| *<!-- Tells us to load the PrimeFaces push servlet on startup -->* <servlet>    <servlet-name>Push Servlet</servlet-name>    <servlet-class>org.primefaces.push.PushServlet</servlet-class>    <load-on-startup>1</load-on-startup> </servlet> *<!-- The url path where the websockets will be connected -->* <servlet-mapping>    <servlet-name>Push Servlet</servlet-name>    <url-pattern>/primepush/\*</url-pattern> </servlet-mapping> |

It was decided that the chat would be implemented all inside one package. This was named *com.mycompany.chat*. An overview of each file is below:

ChatMessage.java - Used to encapsulate the messages that a user is sending/receiving.

ChatResource.java - PushEndpoint that handles what actions should be done when a user enters a room, leaves a room, receives a message, or sends a message.

ChatView.java - Interfaces with the JSF front end to transfer messages to be sent.

MessageEncoder/Decoder.java - Encode/Decode to/from JSON. Used by ChatResource

ChatMessage is a very simple class that simply has two constructors and setters and getters for two fields. The username of the sender and the actual message contents. The implementation is below.

|  |
| --- |
| */\* \* The ChatMessage class is a simple wrapper class that stores the User and \* and the Message that is being sent. It gets encoded and decoded into/from \* JSON when interacting with the front end. \*/*    public class ChatMessage {      *// Only two things we care about in a message*  private String text;  private String user;   public ChatMessage() {  }   *// It is possible for a message to not have a username.*  *// For example when someone enters a room a message is sent announcing*  *// their presence.*  public ChatMessage(String text) {     this.text = text;  }   *// When a user is sending a message to the chat this constructor will be*  *// used.*  public ChatMessage(String user, String text) {     this.text = text;     this.user = user;  }      *// Getter and Setter methods for the fields.*  public String getText() {     return text;  }   public ChatMessage setText(String text) {     this.text = text;     return this;  }   public String getUser() {     return user;  }   public ChatMessage setUser(String user) {     this.user = user;     return this;  } } |

ChatResource is used whenever an action dealing with websockets is happening. For example when a user connects to a room or leaves a room handler functions are invoked and we push a message that states the user has just entered or left. The implementation can be seen below.

|  |
| --- |
| */\* \* This is the endpoint class that the websocket will open. \* In the browser it will look like /Thoughtware/primepush/project/user \* The PrimePuser servlet uses this class to determine what to do when \* Messages are recieved/sent and when users join and leave. \* \*/* @PushEndpoint("/{project}/{user}") @ApplicationScoped public class ChatResource {  *//The eventBus bean is the bean that handles the actual connection.*  *// When we want to send a message we use its publish message to send the*  *// content to the clients.*  @Inject  private EventBus eventBus;      *// We determine both the username and project based on the url that the*  *// the socket is opened on.*  @PathParam("user")  private String username;  @PathParam("project")  private String projectName;   *// When a socket is opened(and thus a user has gone to the chate page)*  *// we publish to all the open sockets on that project that the user has*  *// joined*  @OnOpen  public void onOpen(RemoteEndpoint r, EventBus e) throws InterruptedException {     if (username == null) {         return;     }     eventBus.publish("/" + projectName + "/\*", new ChatMessage(String.format("%s has entered the %s room", username, projectName)));  }      *// Likewise when a socket is closed we publish to everyone that the person has*  *// left the room.*  @OnClose  public void onClose(RemoteEndpoint r, EventBus e) {     if (username == null) {         return;     }     eventBus.publish("/" + projectName + "/\*", new ChatMessage(String.format("%s has left the room", username)));  }      *// onMessage is called both when a message is going to be sent and when*  *//it is going be received.*  *// We do not need to modify the message in transit so we just return it*  *// without changing anything. However we do specify the encoder and decoders*  *// for the class. We simply encode/decode the class to/from JSON.*  @OnMessage(decoders = {MessageDecoder.class}, encoders = {MessageEncoder.class})  public ChatMessage onMessage(ChatMessage message) {     return message;  } } |

Since ChatView handles the interfacing between the JSF frontend it contains a message field for the front end to populate and then publishes using the eventBus to all the websocket listeners in the room when a button is clicked. The implementation is below.

|  |
| --- |
| */\* \* ChatView interfaces with the jsf template to provide the the real time chat \* functionality. \*/* @ManagedBean @SessionScoped public class ChatView implements Serializable {   *// The eventBus is the bean that we use to publish the messages to all*  *// web socket listners.*  private final EventBus eventBus = EventBusFactory.getDefault().eventBus();   *// The accountManager is needed to determine which user this session is a*  *// a part of so we can determine who is sending the messages.*  @ManagedProperty("#{accountManager}")  private AccountManager accountManager;   *// The projectViewManager is used to determine which project the user is*  *// currently viewing.*  @ManagedProperty("#{projectViewManager}")  private ProjectViewManager projectViewManager;   *// Every message that is sent is saved in the database, this is used for*  *// RSS feed, and could also allow for message playback feature to happen.*  @EJB  private com.mycompany.FacadeBeans.MessageFacade messageFacade;   *// When a message is to be sent this field is populated by the jsf template*  private String globalMessage;   *// Triggered by the front end. We simply save the message contents to the*  *// database and then publish it to everyone that has a websocket connection*  *// to the current project path.*  public void sendGlobal() {     *// Message we are saving to the db*     Message create = new Message();     create.setMessageText(globalMessage);     create.setProjectId(projectViewManager.getSelected());     create.setUserId(accountManager.getSelected());     *// Default constructor date defaults to the current time.*     create.setTimestamp(new Date());     *// Actually save the message to the db.*     getMessageFacade().create(create);     *// Send message to /projectName/\* so everyone in the room gets the*     *// message*     eventBus.publish("/" + projectViewManager.getSelected().getName() + "/\*", getAccountManager().getSelected().getUsername() + ": " + globalMessage);     globalMessage = null;  }      *// When we leave the room we announce to everyone we have left*  public void disconnect() {     *//push leave information*     eventBus.publish("/" + projectViewManager.getSelected().getName() + "/\*", getAccountManager().getSelected().getUsername() + " left the channel.");  }     *// Below just contains setters and gettters.*      public String getGlobalMessage() {     return globalMessage;  }   public void setGlobalMessage(String globalMessage) {     this.globalMessage = globalMessage;  }   */\*\*  \* @return the accountManager  \*/*  public AccountManager getAccountManager() {     return accountManager;  }   */\*\*  \* @param accountManager the accountManager to set  \*/*  public void setAccountManager(AccountManager accountManager) {     this.accountManager = accountManager;  }   */\*\*  \* @return the messageFacade  \*/*  public com.mycompany.FacadeBeans.MessageFacade getMessageFacade() {     return messageFacade;  }   */\*\*  \* @param messageFacade the messageFacade to set  \*/*  public void setMessageFacade(com.mycompany.FacadeBeans.MessageFacade messageFacade) {     this.messageFacade = messageFacade;  }   */\*\*  \* @return the projectViewManager  \*/*  public ProjectViewManager getProjectViewManager() {     return projectViewManager;  }   */\*\*  \* @param projectViewManager the projectViewManager to set  \*/*  public void setProjectViewManager(ProjectViewManager projectViewManager) {     this.projectViewManager = projectViewManager;  } } |

The frontend aspect of the chat is all implemented in the ProjectView.xhtml file. There are three additions to this file for the chat.

First off a form is created that holds the text box where the sent and received messages will be displayed. Also when the “Send” button is clicked in the form a ChatView function is invoked and the message is sent to all users.

|  |
| --- |
| <h:form id="form">                 <p:fieldset id="container" legend="Group Chat" toggleable="true"  style="width:550px">                     *<!--Only render the chat box if we are logged in-->*                     <h:panelGroup rendered="#{accountManager.isLoggedIn()}" >                         <h:panelGrid columns="1" columnClasses="publicColumn,usersColumn">                             <p:outputPanel style="height:300px; width:500px" id="public" layout="block" styleClass="ui-corner-all ui-widget-content chatlogs" />                         </h:panelGrid>                          <p:separator />                         *<!--This is where the messages are populated-->*                         <p:inputText value="#{chatView.globalMessage}" styleClass="messageInput" />                         <p:spacer width="5" />                         <p:commandButton value="Send" actionListener="#{chatView.sendGlobal}" oncomplete="$('.messageInput').val('').focus()" />                         <p:spacer width="5" />                         <p:commandButton value="Disconnect" actionListener="#{chatView.disconnect}" global="false" update="container" />                     </h:panelGroup>                  </p:fieldset>             </h:form> |

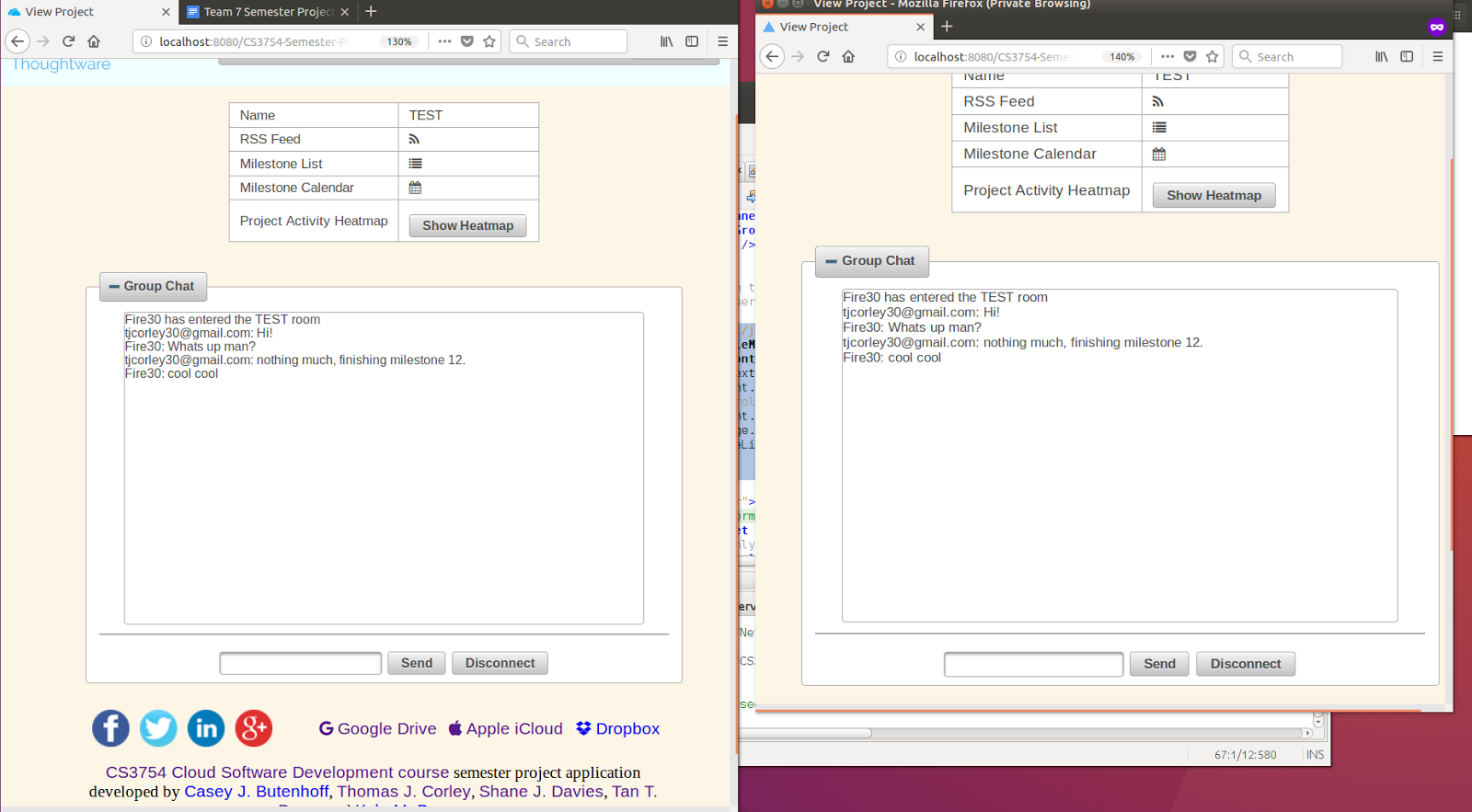
The next aspect is that the actual socket object needs to be embedded in the page. This is a simple one liner.

|  |
| --- |
| <p:socket onMessage="handleMessage" channel="/#{projectViewManager.selected.name}/#{accountManager.selected.username}"/> |

The last needed functionality is a javascript function that is called when a message is sent or received. It simply appends the received text to the chatbox text.

|  |
| --- |
| <script type="text/javascript">             function handleMessage(message) {                 var chatContent = $(PrimeFaces.escapeClientId('form:public')),                         text = (message.user) ? message.user + ':' + message.text : message.text;                 chatContent.append(text + '<br />');                 *//keep scroll*                 chatContent.scrollTop(chatContent.height());                 if (message.updateList) {                     updateList();                 }             }         </script> |

That is  all that is needed to implement a realtime chat using PrimeFaces Push and JSF.  When finished you should be able to have multiple users chat with each other seamlessly. To test this out open one browser window normally and another in incognito, login into two different accounts, go to the same project, and then start chatting. It should look like the following.



## Google OAuth 2.0 and Account Inactivity Monitor

Another feature implemented in our application that can be useful to help streamline the user login and account creation process is the use of Google’s OAuth library to allow users to sign in with an existing Google or G Suite account.  Holding true to the methodologies learned in this class, we implemented this by using allowing the managed Java Beans to receive the information from Google’s Javascript authentication API and then parsing all user information server-side.  There were two main ways to accomplish this task, each having their own advantages and disadvantages.  The first of the two, having the user authenticate using Google, parsing the returned information, and then sending it to the Java Bean was the method that we chose to implement for our project.  The advantages of this implementation include the seamless integration with the currently existing user login already available in applications developed in this class and its overall lack of dependencies required to run the code.  Below I will outline the basic procedure to implement Google OAuth on top of existing local user login abilities (further instructions can be found on [Google’s Developer site](https://developers.google.com/identity/sign-in/web/).  The tutorial found below draws heavily from this information):

Log on to [Google Developer Console](https://console.developers.google.com), create a project, and create a client\_id that will be used by your application when making calls to any Google API’s.  Further information about this ever changing process can be found [here](https://developers.google.com/identity/sign-in/web/devconsole-project).

Once a client\_id has been created Google’s auth API can be included in the <head> section of the SignIn.xhtml file. like so:

|  |
| --- |
| <h:head>                *<!-- Initializes the Google API JavaScript files in order to user OAuth -->*                <script src="//ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.min.js">                </script>                <script src="https://apis.google.com/js/client:platform.js?onload=start" async="defer">                </script>                <script>                    function start() {                        gapi.load('auth2', function () {                            auth2 = gapi.auth2.init({                                client\_id: '152469586034-0gsehfd87qe5b58gr7jlvtopq2usreeb.apps.googleusercontent.com',                                *// Scopes to request in addition to 'profile' and 'email'*                                *//scope: 'additional\_scope'*                            });                        });                    }                </script>            </h:head> |

3. Once the Google API auth2 instance has been instantiated using the client\_id created in step 2, the developer has multiple options on how to implement and stylize the actual Google Sign in button.  For this project, the decision was made to utilize Google’s [signin2.render library](https://developers.google.com/identity/sign-in/web/reference#gapisignin2renderid-options).  This option gives the ability to change how the button looks and how it should react to different user inputs.  An example of this from our code that is responsible for handling successful login and transferring the user’s basic profile information into the Java Bean is found below:

|  |
| --- |
| <div id="my-signin2"></div>    <script>    *//JavaScript function that is called upon successful authentication*    function onSuccess(googleUser) {        *//Hides the sign-in button after a successful login*        $('#my-signin2').attr('style', 'display: none');        *//Populates the hidden Primefaces fields with the information*       *//that is generated by Google from a successful authentication*                            document.getElementById("googleForm:googleUsername").value = googleUser.getBasicProfile().getEmail();                            document.getElementById("googleForm:googleFirstName").value = googleUser.getBasicProfile().getGivenName();                            document.getElementById("googleForm:googleLastName").value = googleUser.getBasicProfile().getFamilyName();                            document.getElementById("googleForm:googleImageUrl").value = googleUser.getBasicProfile().getImageUrl();                            document.getElementById("googleForm:googleId").value = googleUser.getBasicProfile().getId();     *//Makes use of a new feature called Primefaces RemoteCommand*     *//This functionality allows for JavaScript to interact*     *//directly with the Java Beans located on the server*                            document.getElementById("googleForm:sendGoogleButton").click();     *//Disconnects from google once account info has been obtained*         googleUser.disconnect();     }     *//JavaScript function that is called upon failed authentication*     function onFailure(error) {         console.log(error);     }     *//JavaScript function that makes use of Google's own CSS*     *//styling to render the log in button*     function renderButton() {         gapi.signin2.render('my-signin2', {             'scope': 'profile',             'width': 240,             'height': 50,             'longtitle': true,             'theme': 'dark',             'onsuccess': onSuccess,              'onfailure': onFailure          });                        } </script> <script src="https://apis.google.com/js/platform.js?onload=renderButton" async="defer"></script> |

4.  This is where our implementation differs from a pure server-side implementation because the decision was made to send the user’s information itself to the server rather than generate an auth token upon successful login and send that to the server where it would be utilized to make calls to Google API’s.  This made sense in our application because of the limited use of other Google functionality and allowed us to further reduce unnecessary packages and overhead in our code.  For a more in-depth tutorial on a pure Java server-side implementation of Google OAuth, please see [Google Sign-In for server-side apps](https://developers.google.com/identity/sign-in/web/server-side-flow).

5.  The last step done in the XHTML file before sending it off to the Java Bean for processing, is to create the form fields that will contain the user’s information being sent to the server like so:

|  |
| --- |
| <p:commandButton id="sendGoogleButton" ajax="false"  style="display: none;" action="#{loginManager.signedInWithGoogle()}" form="googleForm"/>                    *<!--                        Hidden fields that are used to transfer the information retrieved                        from the JavaScript Google login to the Java Beans on the server                    -->*                    <h:inputHidden id="googleUsername" value="#{loginManager.googleUsername}"/>                    <h:inputHidden id="googleFirstName" value="#{loginManager.googleFirstName}"/>                    <h:inputHidden id="googleLastName" value="#{loginManager.googleLastName}"/>                    <h:inputHidden id="googleImageUrl" value="#{loginManager.googleImageUrl}"/>                    <h:inputHidden id="googleId" value="#{loginManager.googleId}"/> |

6. Now that the information has been sent to the server, the Java Bean is able to parse the information and determine if a user account already exists for that user.  If a user account is found, login proceeds as normal with the exception of the difference to a Google user and a local user’s Profile pages.  The code that is responsible for logging in a user or creating their account is found below:

|  |
| --- |
| */\*\*     \* Logs the user into the application much like the loginUser() method above.     \* The main difference between the two methods is this method will create a      \* new user if the user has not logged into the application before.     \* This allows for a seamless experience for Google users as they are not forced     \* to fill out repetitive information during the account creation process.     \* @return a redirect String to the profile page     \* @throws NoSuchAlgorithmException      \*/*    public String signedInWithGoogle() throws NoSuchAlgorithmException {         *// Obtain the object reference of the User object from the entered username*        User user = getUserFacade().findByUsername(getGoogleUsername());         *//System.out.println("goog: " + user.toString());*        if (user == null) {            accountManager.setFirstName(googleFirstName);            accountManager.setLastName(googleLastName);            accountManager.setUsername(googleUsername);            accountManager.setEmail(googleUsername);            accountManager.setAddress1("Google account, please update!");            accountManager.setCity("Google account, please update!");            accountManager.setState("AK");            accountManager.setZipcode("00000");            accountManager.setSecurityQuestion(0);            accountManager.setSecurityAnswer("Google account, please update!");            accountManager.setPassword("Google account, please update!");            accountManager.setGoogleImageUrl(googleImageUrl);             accountManager.createAccount();                        User newUser = getUserFacade().findByUsername(getGoogleUsername());            initializeSessionMap(newUser);        } else {            errorMessage = "";             *// Initialize the session map with user properties of interest*            initializeSessionMap(user);             FacesContext.getCurrentInstance().getExternalContext().                    getSessionMap().put("isGoogleAccount", true);            *// Redirect to show the Profile page*            return "Profile.xhtml?faces-redirect=true";        }         *// Redirect to show the Profile page*        return "Profile.xhtml?faces-redirect=true";    } |

One field was also added to the account creation process to allow for the storage, retrieval, and use of the user’s Google profile image.  Shown below is a snippet taken from the AccountManager.java createAccount() method:

|  |
| --- |
| *//Sets the user image displayed by google*                if (googleImageUrl != null) {                    newUser.setGoogleImageUrl(googleImageUrl);                } else {                    newUser.setGoogleImageUrl("N/A");                } |

Following these steps allows for the addition of Google Signin into any pre-existing JavaEE application developed in this course that makes use of profile creation and storage.

Once Google Authentication was successfully integrated into the application, the next step to securing Thoughtware further was adding session timeouts when a user was inactive for more than an hour long period.  This was done by making use of a new Primefaces feature called IdleMonitor.  This smart bit of code can be placed in the siteTemplate.xhtml file like so to monitor the user’s activity and trigger an automatic logout to secure a user’s account after an hour:

|  |
| --- |
| *<!--            Makes use of the Primefaces IdleMonitor tag to create a listener that            logs the user out after an hour of activity and redirects them to the            home page.            -->*            <p:idleMonitor timeout="3600000" >                <p:ajax event="idle" listener="#{loginManager.timeout()}" oncomplete="if (#{accountManager.isLoggedIn()}) alert('Session invalidated due to inactivity.')"/>            </p:idleMonitor> |

By pairing this XHTML element with a timeout() method in LoginManager.java to invalidate the current session and redirect the user to the homepage, Thoughtware is able to attain a greater level of security when used in a public environment.

|  |
| --- |
| */\*\*     \* Invalidates user session if there is a user currently logged in and redirects      \* them to the home page.  Used by Primefaces IdleMonitor to log the user out     \* after 1 hour of inactivity as per project specification.     \* @throws IOException      \*/*    public void timeout() throws IOException {        if (accountManager.isLoggedIn()) {            FacesContext.getCurrentInstance().getExternalContext()                    .invalidateSession();            FacesContext.getCurrentInstance().getExternalContext()                    .redirect("index.xhtml");        }    } |

## Milestone Calendar

A major goal for this project was to be able to visualize milestones in an interesting manner. One way this was done by displaying the milestones in a calendar view based on their due date.  This visualization can be extremely useful to see spots when many things are due and allow members to form a plan of action to get everything finished on time.

This feature was implemented using the ScheduleView that comes included in PrimeFaces. The ScheduleView is passed a ScheduleModel which contains a list of ScheduleEvent. We simply lazily loaded every incomplete milestone into this list on page load, and PrimeFaces handled the rest. This was handled in the ScheduleController.java file. The implementation is below:

|  |
| --- |
| */\* \* The ScheduleController class handles the values that are passed to the \* milestone calendar \*/* @ManagedBean @SessionScoped public class ScheduleController implements Serializable {      *// We are loading everything lazily on page load so we use a lazy event model.*  private ScheduleModel lazyEventModel;  *// Need the projectViewManager to determine which project we are in.*  @ManagedProperty("#{projectViewManager}")  private ProjectViewManager projectViewManager;  *// Need the milestoneController to get all Milestones from the database*  @ManagedProperty("#{milestoneController}")  private MilestoneController milestoneController;  *// This is called at the beggining of the session. All it does is create*  *// a LazyScheduleModel that will do things late.*  @PostConstruct  public void init() {      lazyEventModel = new LazyScheduleModel() {         *// This will be called everytime the calendar is loaded.*         *// Aka on page load*         @Override         public void loadEvents(Date start, Date end) {             *// Sometimes there have been reports of null pointer exceptions*             *// so we cautiously check for them.*             if(milestoneController == null || projectViewManager == null)                    return;             *// Get all milestones, and put all of the ones that are not*             *// completed into the database.*             List<Milestone> mstones = getMilestoneController().projectMilestones(getProjectViewManager().getSelected());             for (Milestone m : mstones) {                 if (m != null && m.getCompletedDate() == null) {                     addEvent(new DefaultScheduleEvent(m.getDescription(), m.getDueDate(), m.getDueDate()));                 }             }         }     };  }   *// The rest of the functions are just setters and getters* } |

This code was very simple, and the front end part is even simpler. All we need to is include a schedule element and PrimeFaces will render it for us. Below is the implementation.

|  |
| --- |
| <h:form>             <div align="center">                 <h3>Milestone Calendar</h3>                 <h:panelGrid columnClasses="value">                     <p:schedule id="schedule" resizable="false" draggable="false" widgetVar="myschedule" value="#{scheduleController.lazyEventModel}">                     </p:schedule>                 </h:panelGrid>             </div>         </h:form> |

## Many-to-Many Relationships

Another cloud feature that was instrumental to the project was the fact that we used a Many-to-Many relationship in the database. To do this in MySQL an association table must be created that holds a reference to the two tables that are trying to be linked. In our case we needed to link Projects and Users. The table was defined as:

|  |
| --- |
| CREATE TABLE UserProjectAssociation (    id INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT NOT NULL,    project\_id INT UNSIGNED,    FOREIGN KEY (project\_id) REFERENCES Project(id) ON DELETE CASCADE,    user\_id INT UNSIGNED,    FOREIGN KEY (user\_id) REFERENCES User(id) ON DELETE CASCADE ); |

We also wrote helper methods that would get all Users for a project, and vice versa.

|  |
| --- |
| public List<User> getUsersForProject(Project p) {        return (List<User>) em.createQuery("SELECT c.userId FROM UserProjectAssociation c WHERE c.projectId = :project")                .setParameter("project", p)                .getResultList();    }     public List<Project> getProjectsForUser(User u) {        return (List<Project>) em.createQuery("SELECT c.projectId FROM UserProjectAssociation c WHERE c.userId = :user")                .setParameter("user", u)                .getResultList();    } |

# CONCLUSIONS

TBD

# SUBMISSION INSTRUCTIONS

Please carefully follow the instructions below in submitting your project:

1. Create a directory called lib in your NetBeans project folder. Place all of your JAR files in this lib directory. Right click the Libraries directory in NetBeans and select Add JAR/Folder... Import the JAR files into the Libraries from the lib directory by using *Relative Path*. This is critically important to run your project on another computer or location.
2. If your app uses a file storage directory, name it as TeamN-FileStorage, where N is your team number.)
3. Deploy your cloud software application to the server computer.
4. Using the deployed software, each team member must create an account with   
     
    *username* = last name (first letter capitalized), and   
    *password* = course website password.   
     
   If last name is too short, then use *FirstnameLastname* as the username.   
     
   Each team member must use the software application and create *meaningful content*, which is very important for evaluation and demo for your application.
5. Export (dump) your populated MySQL database content into a SQL script file using # mysqldump -u root -p NameOfDB > NameOfDBwithDBcontent.sql
   1. Move all DROP TABLE IF EXISTS 'TableName' statements in the exported SQL script file to the top of the file and *list them in a logical order* based on foreign key and delete cascading dependencies.
   2. *Reorder all of the table creations* with CREATE TABLE 'TableName' in a logical order.
   3. Correct problems with apostrophe as being the same as the single quote. MySQL exports character strings enclosed within single quotes. If there is a string like Dave’s it conflicts. So, if you have 'These are Dave's files.' change it to "These are Dave's files." to prevent single quote conflict.
   4. Test the corrected exported SQL script file on a computer and make sure that it can be imported to populate the database under the NetBeans IDE.
6. Create a ZIP file containing the following and copy it to a USB flash drive:
   1. Team N Semester Project Report.docx (Do not change this filename and do not remove the spaces in the filename!)
   2. Your NetBeans project folder containing your entire application.
   3. Database SQL file original version.
   4. Database SQL file exported from the database on the server in Step 5 above.
   5. TeamN-FileStorage directory copied from the server computer containing the files stored during use by the team members in Step 4.
7. One or more team members go to Dr. Balci’s office with the USB flash drive to copy the ZIP file to his iMac, open the project under NetBeans IDE, change file paths, create and populate the database, and make sure that the software is fully functional and running on his iMac. *The delivered source code and its documentation will be examined for grading*.

# PERCENTAGES OF CONTRIBUTION

We hereby certify that the list of contributions and the corresponding percentages of contribution specified below truly reflect the actual contributions of the team members.

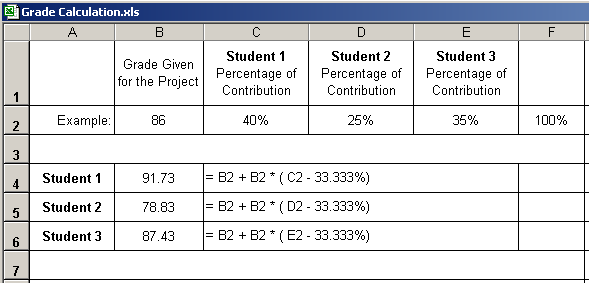
(Write your name as your signature)

|  |  |  |
| --- | --- | --- |
| *Student Name & List of Contributions* | *% Contributed* | *Signature* |
| Casey Butenhoff’s Contributions:   1. Wrote functional and non-functional requirements 2. Hand-sketched wireframes 3. Opened most of the Github issues for project coordination 4. Set up Slack channel for project coordination 5. Created project skeleton code 6. Researched, implemented, and documented password salting and hashing 7. Researched, implemented, and documented database triggers 8. Researched, implemented, and documented RSS feed 9. Researched, implemented, and documented project activity heatmap 10. Researched OAuth 2.0 and OpenID Connect 11. Fixed various minor implementation issues 12. Created favicon | a% |  |
| Thomas Corley’s Contributions:   1. Created Balsmiq wireframe diagrams 2. Created database tables 3. Implemented creation and joining of projects 4. Implemented creation of milestones 5. Researched, implemented, and documented real time chat feature 6. Researched, implemented, and documented milestone calendar feature 7. Implemented and Documented Many-to-Many database relationships 8. Created Thoughtware Logo | b% |  |
| Shane Davies’ Contributions:   1. Researched and documented OAuth 2.0, OpenId Connect, Google SignIn2 API 2. Implemented Google SignIn capabilities 3. Partial creation of Entity Relationship Diagrams 4. Researched, implemented, and documented inactivity monitor and automatic sign out 5. Implemented and improved user’s profile page to include dynamic Google information 6. Responsible for organizing group meetings and report planning with Casey’s help | c% |  |
| Tan Doan’s Contributions:   1. Wrote the executive statement 2. Updated Architecture Diagram 3. Created Design Specification 4. Added delivered software functionality 5. Wrote conclusion | d% |  |
| Kyle Dyess’ Contributions:   1. Created original Architecture Diagram 2. Initial few meetings’ minute-taker 3. Implemented Primefaces menubar with a dynamically updating “My Projects” menu 4. Implemented multi-user file sharing within created projects 5. Implemented the Primefaces Basic Tree Table to display a given Project’s files 6. Created the ProjectViewManager to track which Project the current client will view on ViewProject, ViewProjectFiles, etc. 7. Organization of project progression via creating new issues on GitHub / updating and closing old issues 8. Feature testing and specification/rubric consistency checking | e% |  |
| Sum of Percentages: | 100% |  |

The Virginia Tech Honor Code is fully in effect for the above declaration. The percentage of contribution must be justified by the list of contributions specified in detail.

If you disagree with your team members, you can independently email Dr. Balci what you think the fair percentages of contribution are.

# CALCULATION OF GRADES BASED ON PERCENTAGES OF CONTRIBUTION



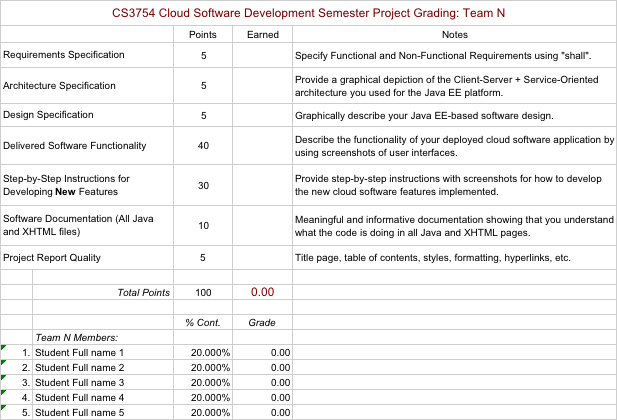
**Additional percentage of contribution cannot exceed 10%.**

**If a situation arises where a student is doing more than 10% extra work, Dr. Balci must be informed immediately.**

# TEAM PROJECT REQUIREMENTS

1. Each team member is expected to contribute equally.
2. You shall submit percentages of contribution together with a list of each student’s individual contributions with signatures of all team members. In case of disagreement, you shall submit it separately with your rationale. (Write your name as representing your signature.)
3. Grades shall be determined based on the percentages of contribution.
4. If you contribute more than your equal share, then your grade shall be increased based on the extra percentage of contribution, which cannot be more than 10%.
5. If you contribute less than your equal share, then your grade shall be decreased accordingly.
6. The extra percentage of contribution shall not be more than 10%. Dr. Balci shall be notified immediately if a situation arises where a student needs to contribute more than 10% extra.
7. A team member who does not cooperate with other team members for conducting the project with equal contribution shall be penalized. Doing more work than agreed upon by the team and claiming extra contribution shall not be acceptable. Cooperation is essential!

# GRADING SHEET



REFERENCES

TBD

Balci, O. (2017), “CS3754 Cloud Software Development Course Website,” <https://manta.cs.vt.edu/cs3754>

GlassFish (2017), “GlassFish Application Server,” <https://glassfish.java.net/>

MySQL (2017), “MySQL Open Source Relational Database Management System,” <http://www.mysql.com/>

NetBeans (2017), “NetBeans IDE,” <https://netbeans.org/>

Paragon Initiative (2016), “How to Safely Store Your Users' Passwords in 2016,” <https://paragonie.com/blog/2016/02/how-safely-store-password-in-2016>

PrimeFaces Documentation (2017), “Documentation | PrimeFaces,” <https://www.primefaces.org/documentation/>

PrimeFaces ShowCase (2017), “PrimeFaces ShowCase,” <https://www.primefaces.org/showcase/>

APPENDIX A: MEETING MINUTES

Meeting Number 1

|  |  |
| --- | --- |
| *Date & Duration:* | October 5, 2017 from 11:00 a.m. to 12:15 p.m. |
| *Location:* | Classroom |
| *Members Present:* | CJ, TJ, Shane, Kyle, Tan |
| *Members Absent:* | None |
| *Discussions:* | Collaboration |
| *Decisions Made:* | Slack channel and GitHub repository created |
| *Work Assignments:* |  |
| *Minutes Prepared By:* | CJ |

**Meeting Number 2**

|  |  |
| --- | --- |
| *Date & Duration:* | November 7, 2017 from 11:00 a.m. to 12:15 pm |
| *Location:* | Classroom |
| *Members Present:* | CJ, TJ, Shane, Kyle |
| *Members Absent:* | Tan |
| *Discussions:* | * What tool for wireframes? * What do we need to get done for the first presentation? * Should we encrypt data/passwords? * Focus on chat, file sharing, or project organization? * What will file sharing look like? |
| *Decisions Made:* | * Use Balsamiq for wireframes * Encrypt data/passwords * Focus mainly on chat feature * Sidebar with file tree of files shared in a project (use pf treetable) * Focus on being able to share files between users * Owner creates project with password, users add project to their account with password * Dynamic hamburger nav menus in header * Active users / members list |
| *Work Assignments:* | * Architecture diagram * Wireframes * Prepare the first presentation |
| *Minutes Prepared By:* | Kyle Dyess |

**Meeting Number 3**

|  |  |
| --- | --- |
| *Date & Duration:* | November 14th, 2017 from 11:00 a.m. to 12:15 p.m. |
| *Location:* | Classroom |
| *Members Present:* | CJ, TJ, Shane, Kyle |
| *Members Absent:* | Tan |
| *Discussions:* | * Discuss the next steps needed for the project/presentation |
| *Decisions Made:* | * Design specification: design and creation of databases * OpenIDConnect for Google Signin (OAuth 2.0) * Designed Entity Relationships diagram for DB design |
| *Work Assignments:* | * Prepare the second presentation * Class diagram * Create DB / Entities |
| *Minutes Prepared By:* | Kyle Dyess |

Meeting Number 4

|  |  |
| --- | --- |
| *Date & Duration:* | November 28th, 2017 from 11:00 a.m. to 12:15 p.m. |
| *Location:* | Classroom |
| *Members Present:* | CJ, TJ, Shane, Kyle |
| *Members Absent:* | Tan |
| *Discussions:* | Presentation and implementation planning |
| *Decisions Made:* |  |
| *Work Assignments:* |  |
| *Minutes Prepared By:* | CJ |

Meeting Number 5

|  |  |
| --- | --- |
| *Date & Duration:* | December 6th, 2017 from 5:00 p.m. to 6:15 p.m. |
| *Location:* | Library |
| *Members Present:* | CJ, TJ, Shane, Kyle, Tan |
| *Members Absent:* | None |
| *Discussions:* | Progress report, presentation and implementation planning |
| *Decisions Made:* |  |
| *Work Assignments:* | Slides assigned to respective members who are working on features |
| *Minutes Prepared By:* | CJ |

Meeting Number 6

|  |  |
| --- | --- |
| *Date & Duration:* | December 13th, 2017 from 5:00 p.m. to 6:15 p.m. |
| *Location:* | Library |
| *Members Present:* | CJ, TJ, Shane, Kyle |
| *Members Absent:* | Tan |
| *Discussions:* | Progress report, presentation and implementation planning |
| *Decisions Made:* |  |
| *Work Assignments:* | Report sections |
| *Minutes Prepared By:* | CJ |

Meeting Number 7

|  |  |
| --- | --- |
| *Date & Duration:* | December 14th, 2017 from 12:00 p.m. to 1:15 p.m. |
| *Location:* | Cloud Lab |
| *Members Present:* | CJ, TJ, Shane, Kyle, Tan |
| *Members Absent:* | None |
| *Discussions:* | Presentation deployment |
| *Decisions Made:* |  |
| *Work Assignments:* | Test data generation |
| *Minutes Prepared By:* | CJ |