CS3733 Team Project Report

Ohtaki

Haley Hauptfeld
Sophiya Litovchick
Rebecca Markowitz
Nick St. George

Purpose

The purpose of this document is to provide evidence for the different roles of the team Members, as well as outline the project and explain how it was completed by the team.

Introduction

The project created by team Ohtaki is called Meeting Scheduler. The purpose of Meeting Scheduler is to provide a simple website to schedule meetings. There are three main functions of this site, the first is to allow an organizer to create a schedule that will be be automatically populated with open meeting slots during the time and date period specified upon creation. The site also allows the organizer to go back and edit the schedule they created to either close or open slots, or cancel meetings. The second function is to allow a participant to enter into a specified organizers schedule in order to book a meeting. They can also cancel meetings they previously created. Finally the System Administrator can upkeep the entire system by either retrieving recently created schedules, or deleting schedules that are older than a specified amount of time.

This project was completed over the course of four weeks and utilized many areas of computer science. The website front end was displayed and created using JavaScript and HTML. The back end logic was created using Java. In order to store, retrieve, and send information between the back end and front end, Amazon Web Services (AWS) was utilized. The data created was also stored in the mySQL database hosted on AWS.

This project provided many challenges due to the nature of working with new programming languages and services. However, through research and time, the team was able to create a completed working product.

Team organization, members, and responsibilities

Ohtaki is a four member team, which allowed the team to be flexible with its leadership positions and roles. All members took on leadership responsibilities equally. Most of these responsibilities included calling meetings when they felt necessary, booking rooms or finding space to work together as a group. One major component in the organization of the group was making sure each member has an up-to-date outlook calendar, which makes it easy to schedule meetings when comparing everyone's schedule at once. This was decided upon by the team because we felt that, with a small group, giving everyone equal leadership would facilitate the best communication and collaboration. Each member took on a leadership roles in their particular area of the project. The group communicated by send Outlook meetings to each others when necessary and using the app, GroupMe, which each team member could access on their phone and computer. GroupMe also provided us with a secondary side channel to relieve stress and send funny Memes to each other.

Meetings were held about 3 times a week when planning out the project. When the group moved onto implementation and testing, an hourly meeting was held every work day in order to allow members to ask questions and communicate daily. Additional many meetings were called between members of the team whenever a new feature needed to be tested or implemented. Most of our implementation meetings were held during the last two weeks of the project.

The Entity Boundary Controller (EBC) design was created by all team member equally, by splitting up the initial problem statement into use cases and assigning each team member an equal amount. The Unified Modeling Language (UML) diagram was collaboratively created by the entire team. The mock API was created by Rebecca and Sophiya. The database and database access objects (DAOs) where created and maintained by Sophiya. The lambda functions were led by Rebecca with the help of Nick. The front end, HTML and Javascript, was created and maintained by Haley. Connecting the front end and back end was implemented by Rebecca and Haley, along with troubleshooting AWS. Nick took lead with version control on Git and making sure everyone else in the group understood how to use Git. Sophiya wrote the group report, and Haley wrote the README file, which includes all of the instructions on how to use the GUI properly.

The team kept track of the HTML code created by keeping on the most updated and functioning version of it in the AWS Simple Storage Service (S3) bucket. The back end code was organized with the help of Git and GitKraken, a program that can connect to Eclipse and visually keep track of code branches. Each team member created their own branch and each was tested, and the functional release was pushed to the master. There were also additional branches that were added, when merges were unsuccessful and version control was needed.

Process

The processes of creating Meeting Scheduler started from base analysis using the EBC design paradigm. The problem statement was first broken into a total of around 15 use cases that would become the functionality of the website.These use cases made up the controller aspect of EBC. Once these use cases were solidified, the entity and boundary were discussed. Initially, the model consisted of many classes in order to allow for flexibility (Fig. 1). However, as the team got further into the project, the model was edited to reflect the functionality needed by the team. The final UML diagram is shown in (Fig. 2).

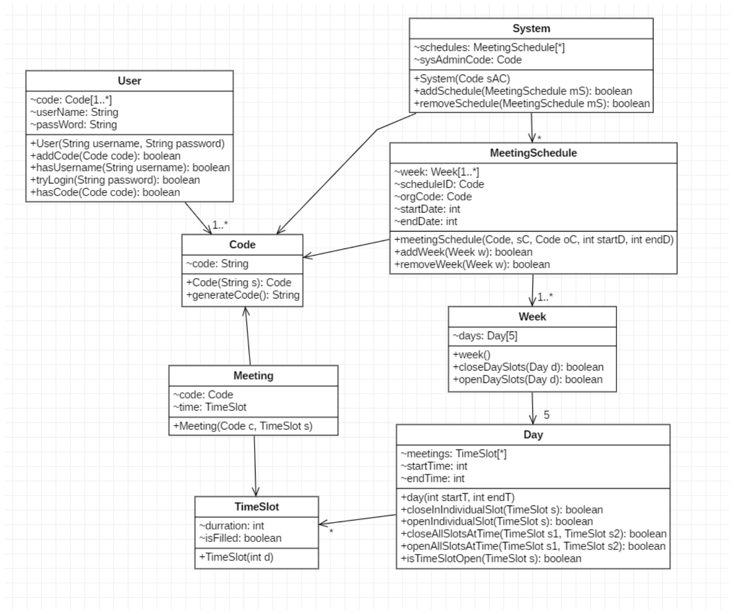
System
~schedules: MeetingSchedule[*]
~sysAdminCode: Code
+System(Code sAC)
+addSchedule(MeetingSchedule mS): boolean
+removeSchedule(MeetingSchedule mS): boolean
User
~code: Code[1..*]
~userName: String
~passWord: String
+User(String username, String password)
+addCode(Code code): boolean
+hasUsername(String username): boolean
+tryLogin(String password): boolean
+hasCode(Code code): boolean
*
MeetingSchedule
~week: Week[1..*]
~scheduleID: Code
~orgCode: Code
~startDate: int
~endDate: int
+meetingSchedule(Code, sC, Code oC, int startD, int endD)
+addWeek(Week w): boolean
+removeWeek(Week w): boolean
1..*
Code
~code: String
+Code(String s): Code
+generateCode(): String
1..*
Week
~days: Day[5]
+week()
+closeDaySlots(Day d): boolean
+openDaySlots(Day d): boolean
Meeting
~code: Code
~time: TimeSlot
+Meeting(Code c, TimeSlot s)
5
Day
~meetings: TimeSlot[*]
~startTime: int
~endTime: int
+day(int startT, int endT)
+closeInIndividualSlot(TimeSlot s): boolean
+openIndividualSlot(TimeSlot s): boolean
+closeAllSlotsAtTime(TimeSlot s1, TimeSlot s2): boolean
+openAllSlotsAtTime(TimeSlot s1, TimeSlot s2): boolean
+isTimeSlotOpen(TimeSlot s): boolean
TimeSlot
~durration: int
~isFilled: boolean
+TimeSlot(int d)
*

Fig. 1

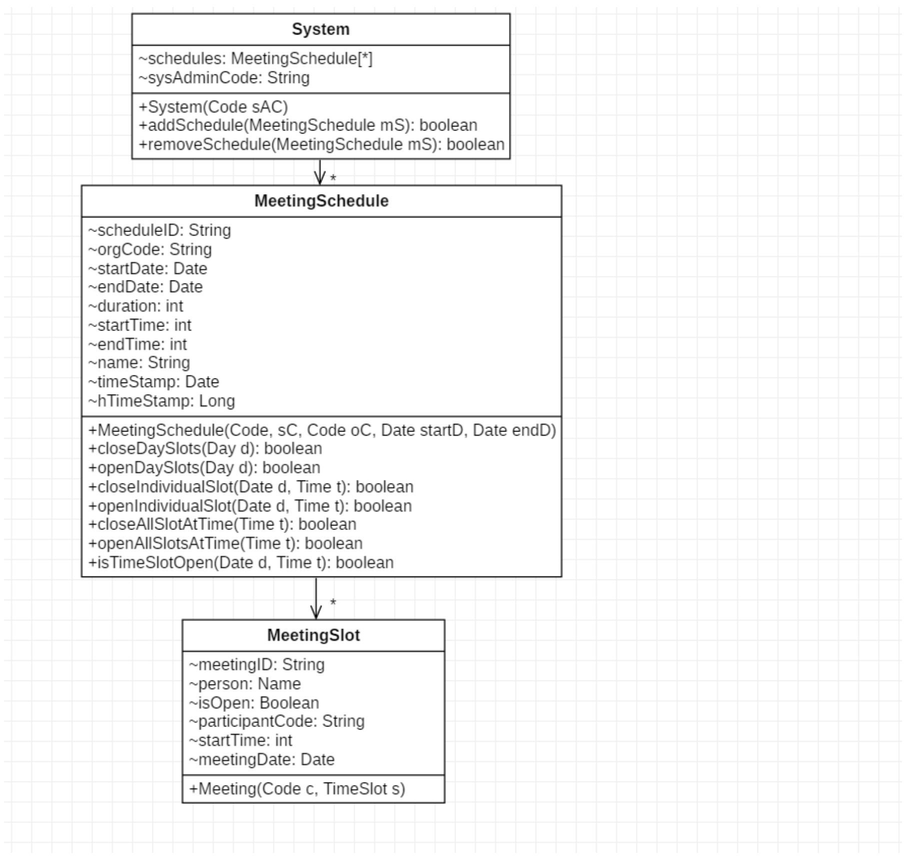
System
~schedules: MeetingSchedule[*]
~sysAdminCode: String
+System(Code sAC)
+addSchedule(MeetingSchedule mS): boolean
+removeSchedule(MeetingSchedule mS): boolean
*
MeetingSchedule
~scheduleID: String
~orgCode: String
~startDate: Date
~endDate: Date
~duration: int
~startTime: int
~endTime: int
~name: String
~timeStamp: Date
~hTimeStamp: Long
+MeetingSchedule(Code, sC, Code oC, Date startD, Date endD)
+closeDaySlots(Day d): boolean
+openDaySlots(Day d): boolean
+closeIndividualSlot(Date d, Time t): boolean
+openIndividualSlot(Date d, Time t): boolean
+closeAllSlotAtTime(Time t): boolean
+openAllSlotsAtTime(Time t): boolean
+isTimeSlotOpen(Date d, Time t): boolean
*
MeetingSlot
~meetingID: String
~person: Name
~isOpen: Boolean
~participantCode: String
~startTime: int
~meetingDate: Date
+Meeting(Code c, TimeSlot s)

Fig. 2

Once the analysis was completed, the team began laying down the base for implementation. First, a mock API was created using SwaggerHub. This allowed the team to get a better understanding about how to interact between the lambda functions and the front end. This API created the basis of request and return types that the front end would send and receive. The team then started work on creating the AWS set up necessary to run the program. The team used the S3 buckets and Lambda functions, and Relational Data Service (RDS). The team connected to the AWS and created the S3 bucket along with the database.

Once all the set up for the project was completed, the team began implementation. Initially, only one lambda function was created and used to test out how the system would work. Additionally, the database was created using mySQL and DAO functions. The structure of the database is very different from how entities are structured using java code, thus many changed to the initial entity classes that were made when creating the new UML seen above (fig 2). Once the first function was tested and properly working, the rest of the lambda functions were written based upon this. The DAO were utilized as helper functions to retrieve data from the database in an organized fashion as specified by the lambda functions. It was integral to communicate between the person creating the DAO and the lambda functions in order to return the correct data types that the lambda function needed.

Once all the lambda functions were written, they were connected to the front end. At first, the group found difficulty connecting the lambda functions. This is due to the fact that there are multiple ways to connect the lambda through AWS. In the sample project created by the professor, wrote the lambda functions in a different way in Java, and used lambda proxy integration default settings in AWS. Since the Ohtaki group set up the lambda functions differently, they had to manually write headers into the API integration. This took a good amount of time to figure out, but once this was realized, connecting the backend to the frontend was smooth sailing. A large reason for the amount of time spent on this problem was because the group did not know this was an AWS setup problem and instead they were trying to debug their own project code.

Additionally, the HTTP methods outlined in the API were not entirely correct for the needs of the group. Many changes were made to the API model through AWS at this time. The group decided to use a different method than the example shown in class and thus had to go through troubleshooting before figuring out the proper method. The group only used Post methods in the project, because the other methods did not work when trying to be implemented. However, once this was discovered, the subsequent lambda functions were able to be connected in an efficient manner.

Finally the group worked on debugging the front end. The existing javascript embedded within the HTML was mostly untested due to the fact that it is hard to test without the data received from the lambda functions. The group spent the last week working on connecting lambda functions and then debugging the javascript for each one. This was very efficiently done by having Rebecca, the lead lambda function coder, and Haley, the front end coder, pair program each API connection and request and response.

The project's main functionality is complete and all the javascript is debugged. Due to the set-back experiences with AWS integration, there were a few additional lambda functions that were unable to be implemented by the end. These functions were determined by the team to be non-essential, or special features of the system. Due to the well defined base code, these features could easily be implemented within a day or two, if given the time. However, gaining an in-depth understanding of AWS was beneficial to the whole team and will allow the team to utilize these skills more efficiently in the future.

Tools

The team used many tools to complete the project. StarUML was used to create the UML diagram. This tool was very helpful for creating visuals, but did not function very well and was tedious to work with when creating the diagram. In order to keep track of and share code, GitHub was used, along with the desktop app, GitKraken, that provided a better interface when connecting git to the Eclipse IDE used by the team. GitKraken was extremely helpful because it allowed the team to use git more efficiently. The Eclipse IDE does not have good built-in git integration. Atom was used for the HTML and JavaScript. It is a simple text editor that made it easier to write the front-end. Additionally, Amazon Web Services was used to upload the controller functions (called lambda functions) to a server along with the HTML code. AWS also provided a server for the database created. The database was maintained and created with MySQL, using the MySQL Workbench. For team communication, Microsoft Outlook and GroupMe were used.

Accomplishments

Use Cases implemented:

- Create Meeting Schedule
- Close Meeting Slot
- Open Meeting Slot
- Cancel Meeting Organizer
- Send Meeting Schedule
- Delete Meeting Schedule
- Create Meeting Participant
- Cancel Meeting Participant
- Delete Past Scheduled Meeting
- Review Newly Created Meetings. T

Use cases written but not implemented in front end:

- Search Open Time Slot
- Extend Schedule End Date
- Extend Schedule Start

Use cases not yet written:

- Close/Open All Slots At Time
- Close/Open All Slots On Day

Deliverables

The deliverables are the website itself, Meeting Scheduler. Along with the final README, also called Manual. This Manual outlines all the slightly unconventional formatting and wait times when using the Meeting Scheduler, along with a general outline on the functionality of each button and input.

Reflection

The group put in a lot of work. However due to a lack of experience with AWS and web development, there were a couple places where the group failed. This caused overall set backs. Having good communication worked for our team. However, sometimes members would not fully articulate when they were having issues or not fully understanding how to write a function. This also caused problems when other members had to rewrite someone else's code in order to have it work properly. This definitely added a lot of time, that could have been spent implementing and planning new lambda functions or working on JavaScript. The root cause of this was miscommunication on how to write and structure the lambda functions to be consistent with the API, which caused a lot of re-writing and lost time.

Our biggest mistake

The biggest challenges faced by the team were creating the display of the schedule and updating the view. We only had one person creating the HTML and JavaScript on the front end. However, the team ended up relying on a lot more logic within the front end. Having only one person being proficient in HTML and JavaScript was not enough. Although the rest of the team members started learning JavaScript in attempt to aid Haley, there was not enough time. Additionally, there was no easy way to share the JavaScript and HTML code among the members, thus it was harder for team members to split up the front end portion of the code.

Changes we would make

Some changes that would improve the Meeting Scheduler include adding an additional person to work on JavaScript, as well as better communication between the structure and purpose of the lambda functions. More people working on JavaScript would allow for better functionality of the front end while creating a better blueprint for the lambda functions and connecting the functions. This way, there would be less issues when connecting the lambda to the front end.

Lessons learned

Clear and constant communication is very important.

Advice to future teams

Have at least one member that knows JavaScript well.