Software Architecture Document

caproomster

Zhipeng Bruce Cai 21346482
Adrianna Diaz 27184778
Lance Lafontaine 26349188
Lenz Petion 26775837
Arek Manoukian 21710389
Taimoor Rana 26436110

Instructor: Constantinos Constantinides

April 4, 2017

Original credits

Software Architecture Design

Version 0.03

for

Bookme

Prepared by

Emir Bozer	26424724	emir.bozer@gmail.com
Nikolas De Vigne Blanchet	27189877	nikdvb@gmail.com
Ahmad Hyjaz Loudin	27179294	hyjaz.loudin@gmail.com
Mary Psaroudis	27209193	marypsaroudis@gmail.com
Leo Yu	27036736	yuleo@outlook.com

Instructor: Constantinos Constantinides

Course: SOEN343: Software Architecture and Design I

Date: 23/11/2016

Document history

Date	Version	Description	Author
	0.01	All the sections added	Everyone
20/11/2016	0.02	Architectural requirements	Mary Psaroudis
20/11/2016	0.03	Formatting	Leo Yu
23/03/2017	1.00	Reformatting	Adrianna Diaz
04/04/2017	1.01	Finalization of submission	Zhipeng Cai Adrianna Diaz Lance Lafontaine Arek Manoukian Taimoor Rana

Contributor Log

Zhipeng Cai	Update of documentation New client-side architecture and implementation Client-side and server-side system tests Refactoring of existing solution
Adrianna Diaz	Extensive update of documentation Implementation of all equipment features Implementation of capstone priority requirement Design and implementation of server architecture Server-side unit tests Refactoring of existing solution State machine diagram
Lance Lafontaine	Update of documentation Design and implementation of server architecture Implementation of REST API Server-side system and unit tests Refactoring of existing solution
Arek Manoukian	Extensive update of documentation Design and implementation of server architecture Implementation of AOP feature migration Server-side system and unit tests Refactoring of existing solution
Lenz Petion	Refactoring of existing solution (room lock deletion)
Taimoor Rana	Extensive update of documentation Implementation of new reservation logic and constraints Design and implementation of server architecture Server-side unit tests Refactoring of existing solution

Modifications Log

Feature Change or Refactoring for Corrective Maintenance	Reasoning
Previous implementation made no use of TDGs, was completely stateless	- The previous implementation did not use any TDGs or any concepts learned in this class to make a stateful application - Once the server was restarted, the application would restored itself to a ***manually hardcoded*** state that did not involve TDGs or the database. - Removed this hardcoding in #42 (https://github.com/lancelafontaine/caproomster/pull/42/files)
Removal of the concept of room locks and write sessions from the system	- The previous implementation was locking an entire room as opposed to a specific timeslot. - The concept of room locks and write sessions impeded the project's goals of supplying a ***safe, live and fair*** system. - This position is justified in more depth within this document. - Fixed across multiple pull requests
Removal of the templated client-side subsystem of the application	- The previous implementation of the server-rendered HTML used a different template for every month of an arbitrary year. - Each of these templates contains nearly identical information and hardcoded the days and weeks of month - Beginning of new client to REST API shown in pull request #23 (https://github.com/lancelafontaine/caproomster/pull/23/files) - Removal of old templates in pull request #60 (https://github.com/lancelafontaine/caproomster/pull/60/files)
Removed hardcoded platform-specific Python bytecode from our repository	- These files were needlessly included in the repository - They were compiled files specific to the Windows platform - These files serve no purpose other than to run directly from Python bytecode on the windows platform. - Fixed in pull request #10 (https://github.com/lancelafontaine/caproomster/pull/10/files)
Removed hardcoded plain-text passwords from the repository	- These passwords, stored in plain text across dozens of files within the repository, were both a vital security concern and a maintenance burden. - Fixed in pull request #11 (https://github.com/lancelafontaine/caproomster/pull/11/files)
Fixed the indirection of mappers and TDGs in the previous implementation	- The previous implementation would have the system's mappers pass an object as a parameter to a TDG object - This implies that the TDG has implicit knowledge of how each object works - This was fixed by having the mapper maintain the knowledge

	of each object and only pass the necessary values to the TDG - Fixed in pull request #11 (https://github.com/lancelafontaine/caproomster/pull/11/files)
Registry controller interacted directly with TDGs	- In the previous implementation, the Registry controller would call an object's TDG directly, which coupled the state of the application with the controller as opposed to being maintained by an object's mapper - Fixed in pull request #21 (https://github.com/lancelafontaine/caproomster/pull/21/files) - Also fixed in pull request #42 (https://github.com/lancelafontaine/caproomster/pull/42/files)
Mappers of one type would interact with the TDGs of a different type	- In the previous implementation, the ReservationMapper would directly call the TDGs of other types such as Room, User and Timeslot. This leads to a very highly coupling between reservations and TDGs of other types. - Fixed in pull request #19 (https://github.com/lancelafontaine/caproomster/pull/19/files)
Removal of the Directory class	- In the previous implementation, There was a Directory class which kept track of all rooms. - This class performed very little function, and duplicated all the functionality of the RoomMapper class - Fixed in pull request #31 (https://github.com/lancelafontaine/caproomster/pull/31/files)
Previous functionality for month of March was completely non-functional	- This was caused by a typo in the template that had been copied and pasted to every of the 12 hardcoded month templates - Fixed in pull request #34 (https://github.com/lancelafontaine/caproomster/pull/34/files)
Refactored all database connections into one common module	- In the previous implementation, every TDG implemented its own logic to configure the database and make connections to it. This involved lots of code repetition and resulted in a higher maintenance cost - Fixed in pull request #24 (https://github.com/lancelafontaine/caproomster/pull/24/files)

Table of contents

List of figures

1. Introduction

Purpose

The purpose of the Software Architecture Design (SAD) document is to provide a comprehensive architectural views of the caproomster software, an online conference room reservation system. The document is composed of several sections: Architectural Representation, Architectural Requirements, multiple different Views, Size and Performance, and Quality. The Architectural Representation section provides a top-level architectural style of the caproomster system. The Architectural Requirements section describes how key functional and non-functional requirements from the Software Requirement Specification (SRS) affect the architecture. There will be multiple sections to describe of each individual view models provide to the 4+1 architectural view model. The Size and Performance section details how architecture supports the key sizing and performance requirements. The Quality section evaluates how the software architecture contributes to the quality attributes of the caproomster System based on the ISO 205010. This document is intended for the developers of the caproomster software as it is possible to map the design of the software architecture into lines of codes.

Scope

The scope of this SAD is to depict the overall architecture of the caproomster software. The caproomster software architecture follows the 4+1 architectural view model, which is composed of five individual views: Logical View, Use Case View, Development View, Process View, Physical View. From these views, developers can code the actual software product by mapping the view models into code. Thus, the product's code is heavily influenced by the SAD. If there is any change in the design of the software architecture, the product's code must be updated in order to reflect on the new changes.

Definitions, acronyms, and abbreviations

RUP	Rational Unified Process
UML	Unified Modeling Language
SAD	Software Architecture Document
SRS	Software Requirements Specification

2. Architectural Representation

The overall top-level architectural style of the caproomster system is model using the 4+1 architectural view model illustrated in Figure 1. Each views describe the system in the point of the view of the different stakeholders: Designers, programmers, deployment manager and integrators. The four views are the logical view, the deployment view, the process view and the physical. Also, critical use cases shall be used to demonstrate the system architecture, thus serving as the use case (scenario) view.

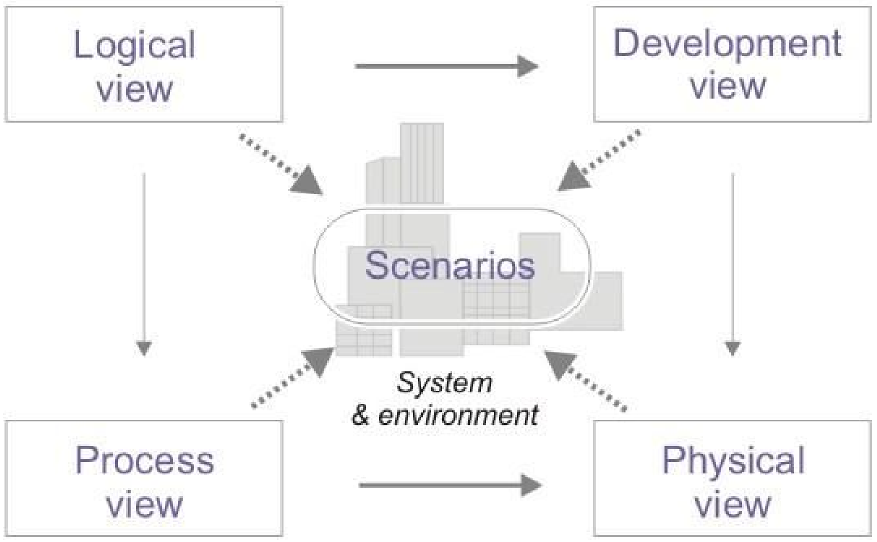


Figure 1. The 4+1 View Model

1. Logical view : Audience: Designers. The logical view is concerned with the functionality that the system provides to end-users. UML Diagrams used to represent the logical view include Class diagram, and interaction diagrams (communication diagrams, or sequence diagrams).

2. Development view (also known as Implementation view): Audience: Programmers. The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view. It uses the UML Component diagram to describe system components. UML Diagrams used to represent the development view include the Package diagram.

3. Process view: Audience: Integrators. The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML Diagrams to represent process view include the Activity diagram.

4. Physical view (also known as deployment view): Audience: Deployment managers. The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components. UML Diagrams used to represent physical view include the Deployment diagram.

5. Use case view (also known as Scenarios): Audience: all the stakeholders of the system, including the end-users. The description of an architecture is illustrated using a small set of use cases, or scenarios which become a fifth view. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. Related Artefacts: Use-Case Model.

Note: This document will only cover the Logical View and the Use case view.

6. Architectural requirements: goals and constraints

There are some key requirements and constraints that have a significant impact on the architecture of the system.

- The caproomster system must access the PostgreSQL database to retrieve/add/modify user and reservation information.
- The system must be secure. Authentication and authorization of users are required to be able to perform system operations.

- Data persistence will be applied using the object-relational database system PostgreSQL.

Functional requirements (Use case view)

The significant use cases of the caproomster system are:

- Reserve a room
- Reserve a Room on a weekly occurrence
- Cancel an existing reservation
- Modify a reservation
- View my reservations
- View schedule

Source	Name	Architectural relevance	Addressed in:
U3, UC4, UC9	Reserve Room, Add to Waiting List	Allows the user to reserve a room at a given date and time. Should the room selected not be available, the user will have the option to be added to a waitlist. The user may also be allowed to repeat their reservation for up to three weeks in total.	Section 9 - Logical view: Use case realizations

Source	Name	Architectural relevance	Addressed in:
U5, UC6, UC7	View my reservations, Modify Reservation, Cancel Reservation	Allows the user to view their reservations. The user can also modify or cancel an existing reservation.	Section 9 - Logical view: Use case realizations

Source	Name	Architectural relevance	Addressed in:
U8	View schedule	Allows the user to view a calendar of reservations with the occupied time slots of rooms.	Section 9 - Logical view: Use case realizations

Non-functional requirements

Source	Name	Architectural relevance	Addressed in:
SRS	Design Constraints	Implementation of an object-relational structural and behavioral patterns for the database.	Section 9 - Logical view: Layer, tiers etc.

Source	Name	Architectural relevance	Addressed in:
SRS	Aspect-Oriented Programming component	Implementation of a component of the system must be re-architected to include elements of the Aspect-Oriented programming paradigm.	Section 8 - Logical view: Layers, tiers, etc.

7. Use case view (Scenarios)

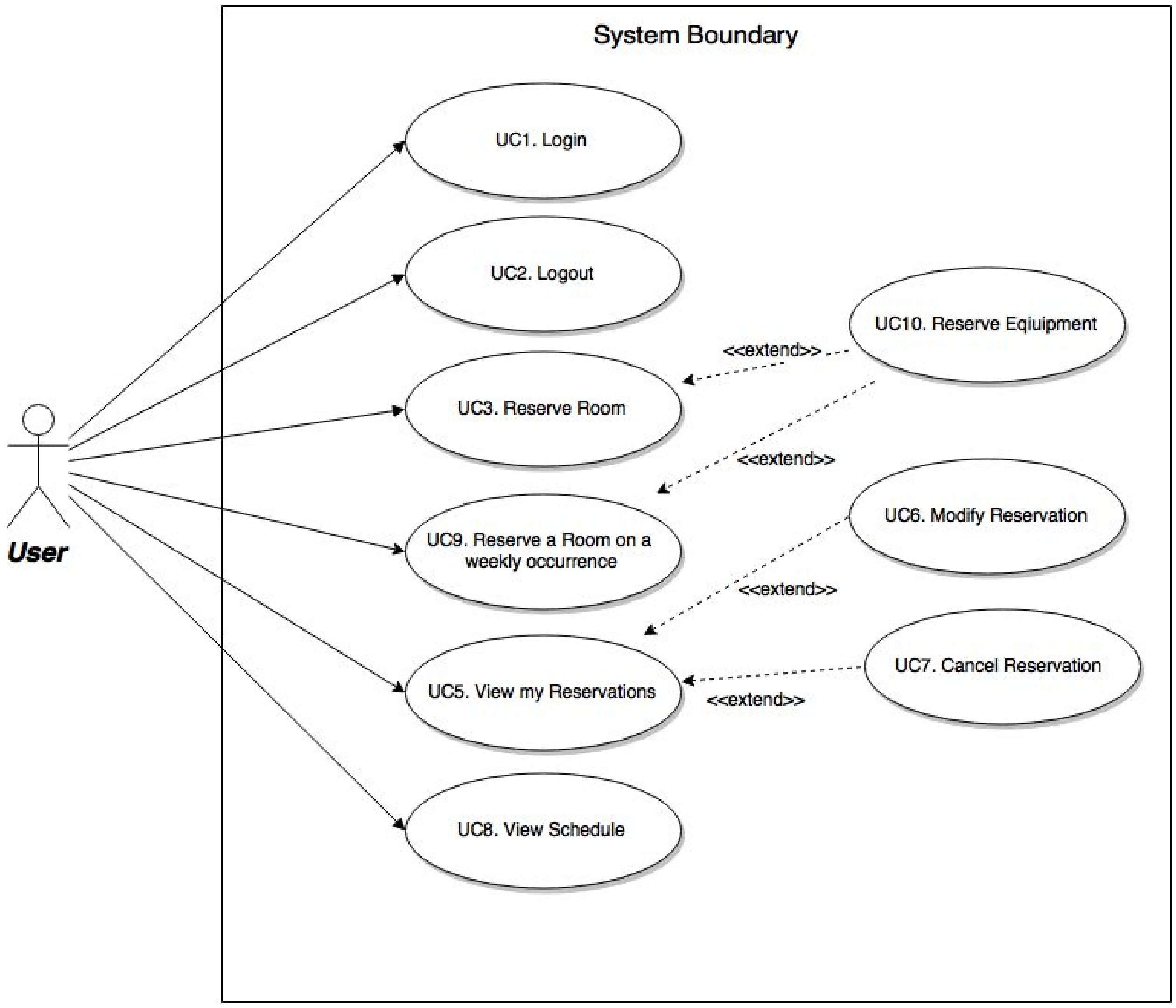


Figure 2. Use Case Model

8. Logical view

Layers, tiers, etc.

In terms of the logical view, the top-level architecture style of the caproomster software follows the three-layered system of the Enterprise Application pattern. It is composed of a presentation layer, a domain layer and a data source layer. The Presentation layer contains the graphical user interface (GUI) of the caproomster application. The Domain layer contains the logic that governs the software. The Data Source layer contains the persistent storage mechanism of the caproomster system.

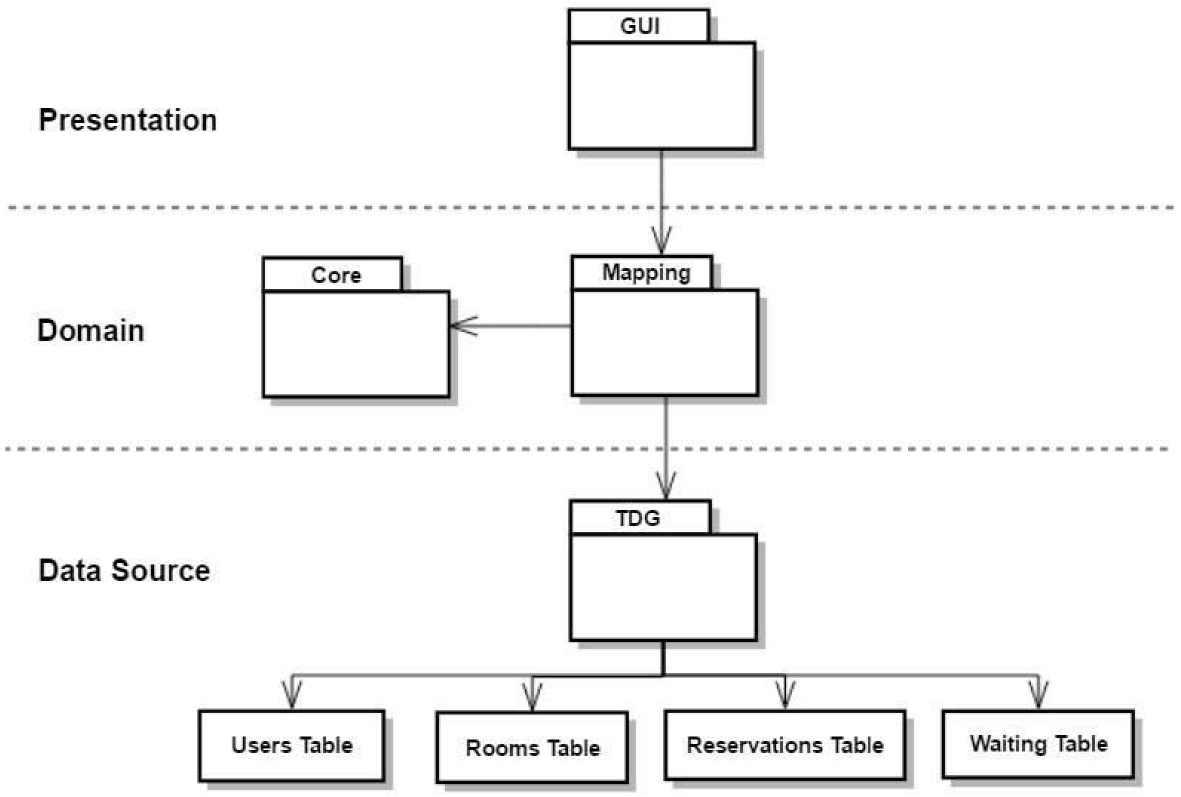


Figure 3. Caproomster Layered Class Diagram

Subsystems

The decomposition of the caproomster system is based on the three different layers and the specific functions that the subsystem provides. All GUIs that the user utilizes to interact with the caproomster software will be inside the GUI package. The Mapping system is a package composed of all classes related to three architectural patterns: Data Mapper, Identity Map and Unit of Work. Included in this subsystem is the use of aspect-oriented programming, to reduce coupling among objects involved in the Data Mapper and Identity Map patterns. Core subsystem is a package that contains all the logic and classes needed to execute the application. The Table Data Gateway (TDG) subsystem is a package that contains all the TDG of the system. The relation between these subsystems are described below.

To start with, the GUI package sends inputs to the system by communicating with the Mapping package to execute a process. The Mapping package is set up in between the Core package and TDG package in order to move data between them while reducing the amount of coupling and database call. The TDG packages have access to the database tables for when the Mappers packages sends messages to interact with the databases.

Architecturally significant design packages

Notes: In order to improve readability and comprehension of the class diagrams, methods that are not pertinent to the communication diagrams or architecture of the system may not appear in the class diagrams. This includes accessors, mutators, auxiliary and generic methods.

Core Subsystem

The Core subsystem is the main subsystem of the caproomster web application. The package contains all the methods and the classes needed to satisfy the requirements and the constraints of the software project. Despite creating, modifying and deleting instances of reservation objects, the Core subsystem contains no code related to the interaction of the database

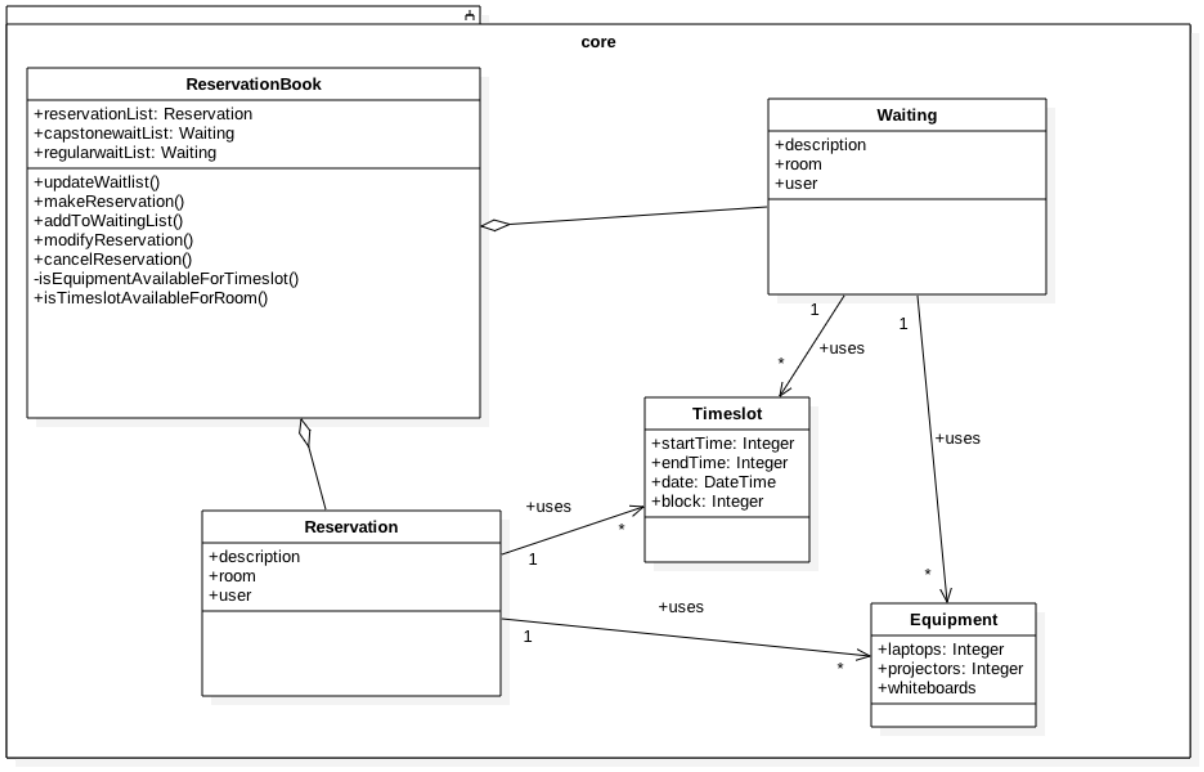


Figure 4. Core Class Diagram

Constraints Expressed in OCL

Requirement: Capstone students should have priority over regular students.

Translation: The system maintains two waitlists, one for regular students, and the other one for capstone students. When updating the waitlists, the system will first iterate over the capstone waitlists to search for an eligible reservation, and only if none is found, then search will continue on the regular wait list. This method has side effects.

```
context ReservationBook::updateWaitlist(current:Timeslot)
pre: self.capstoneList.notEmpty() or self.regularList.notEmpty()
self.capstoneList.exists(w:Waiting|w.timeslot=current)@pre or
self.regularList.exists(w:Waiting|w.timeslot=current)@pre
```

Requirement: There is a limited amount of equipment to be allocated for each reservation.

```
    context ReservationBook
inv: laptopsAvailable=2,projectorsAvailable=2,whiteboardsAvailable=2
self.reservationList.forAll(r:Reservation|r.getEquipment()[‘laptops’]<=
laptops and r.getEquipment()[‘projectors’]<= projectorsAvailable and
r.getEquipment()[‘whiteboards’]<=whiteboardsAvailable)
```

Mapping Subsystem

The Mapping package contains three architectural patterns: Data Mapper, Identity Map and Unit of Work. The Data Mapper pattern is represented as Mapper objects that are in charge of setting up a communication between the Core package and the TDG package in order to move data between them while keeping them independent from each other. The Identity Map pattern is represented as IdentityMap objects that prevent loading the same data from the database twice. Four pointcuts are included within the Identity Map pattern, with their respective advice, in an Aspect. This aspect is used to decouple the Mapper objects from their respective Identity Maps, as well as simplifying the logic that is included within Mapper classes. The Unit of Work pattern is represented as a single class instance that is in charge of keeping track of every changes during a process and committing those changes at the end of a process.

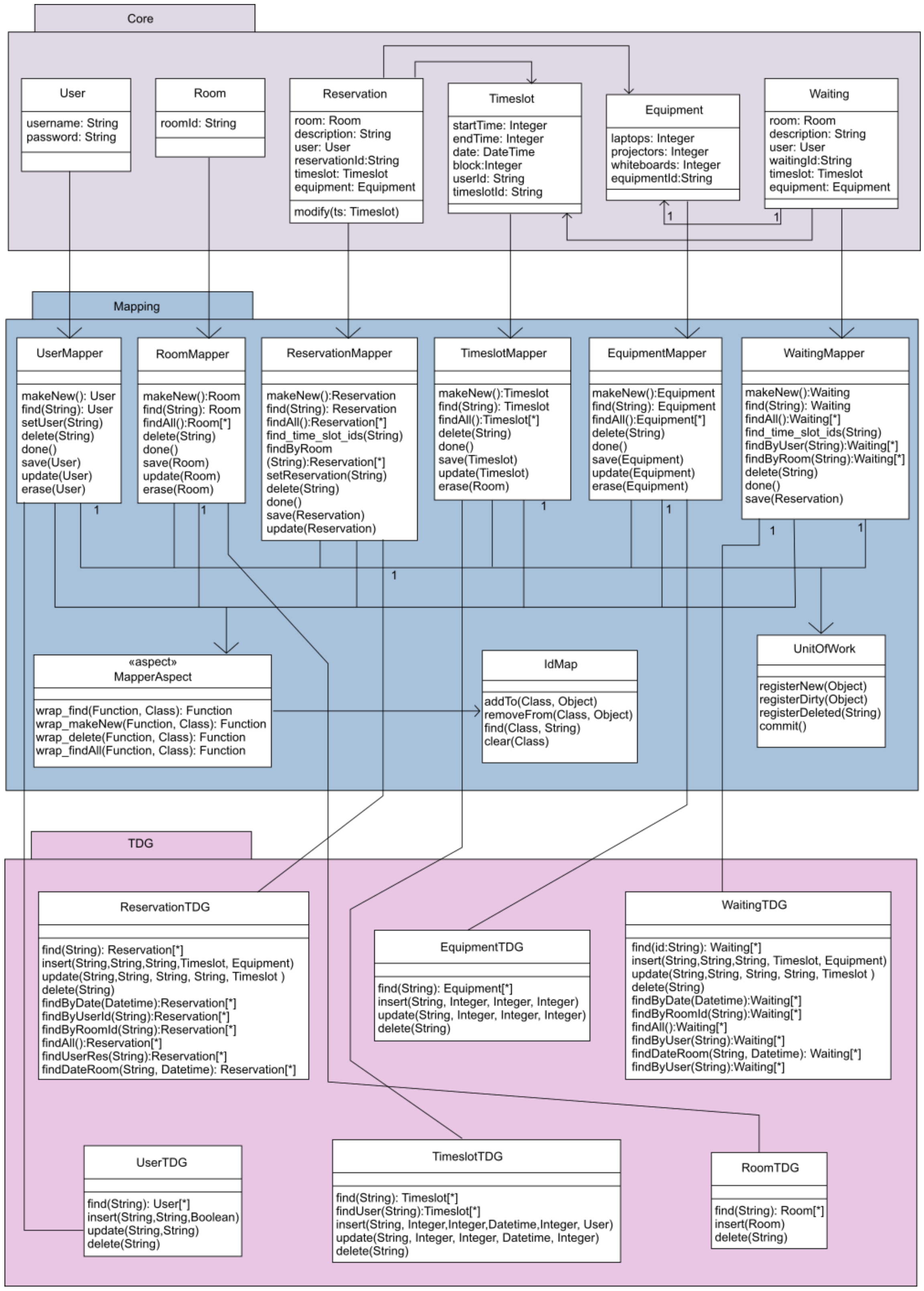
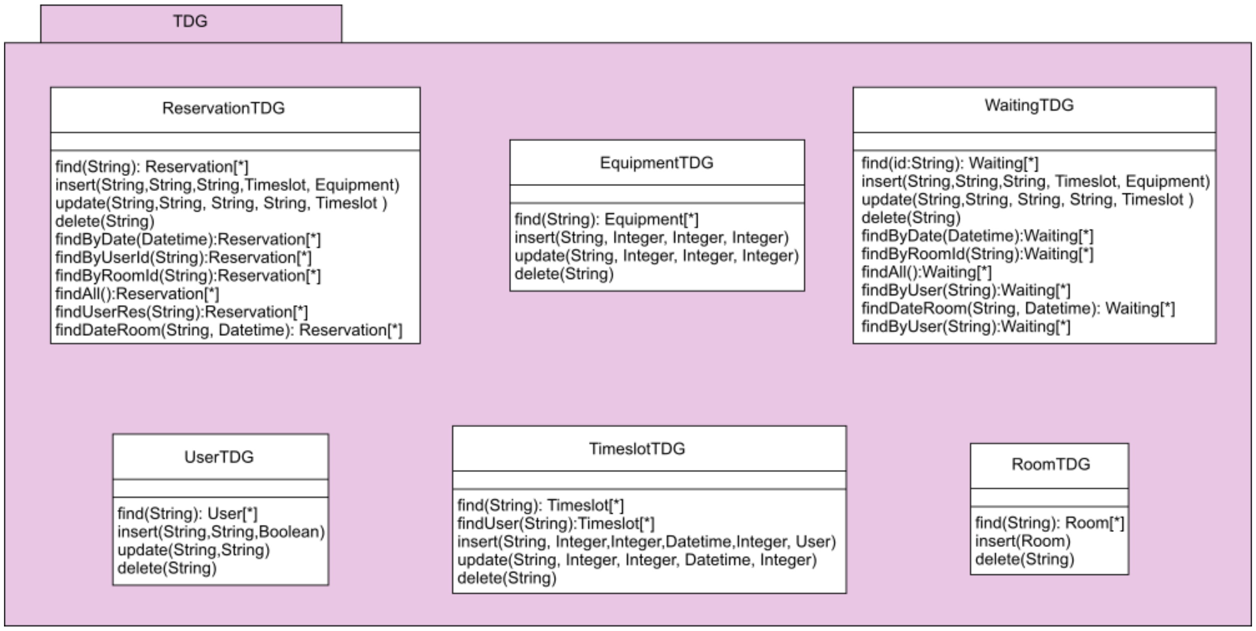

Core
User
username: String
password: String
Room
roomId: String
Reservation
room: Room
description: String
user: User
reservationId:String
timeslot: Timeslot
equipment: Equipment
modify(ts: Timeslot)
Timeslot
startTime: Integer
endTime: Integer
date: DateTime
block:Integer
userId: String
timeslotId: String
Equipment
laptops: Integer
projectors: Integer
whiteboards: Integer
equipmentId:String
Waiting
room: Room
description: String
user: User
waitingId:String
timeslot: Timeslot
equipment: Equipment
1
1
Mapping
UserMapper
makeNew(): User
find(String): User
setUser(String)
delete(String)
done()
save(User)
update(User)
erase(User)
RoomMapper
makeNew():Room
find(String): Room
findAll():Room[*]
delete(String)
done()
save(Room)
update(Room)
erase(Room)
ReservationMapper
makeNew():Reservation
find(String): Reservation
findAll():Reservation[*]
find_time_slot_ids(String)
findByRoom
(String):Reservation[*]
setReservation(String)
delete(String)
done()
save(Reservation)
update(Reservation)
TimeslotMapper
makeNew():Timeslot
find(String): Timeslot
findAll():Timeslot[*]
delete(String)
done()
save(Timeslot)
update(Timeslot)
erase(Room)
EquipmentMapper
makeNew():Equipment
find(String): Equipment
findAll():Equipment[*]
delete(String)
done()
save(Equipment)
update(Equipment)
erase(Equipment)
WaitingMapper
makeNew():Waiting
find(String): Waiting
findAll():Waiting[*]
find_time_slot_ids(String)
findByUser(String):Waiting[*]
findByRoom(String):Waiting[*]
delete(String)
done()
save(Reservation)
1
1
1
1
1
1
1
«aspect»
MapperAspect
wrap_find(Function, Class): Function
wrap_makeNew(Function, Class): Function
wrap_delete(Function, Class): Function
wrap_findAll(Function, Class): Function
IdMap
addTo(Class, Object)
removeFrom(Class, Object)
find(Class, String)
clear(Class)
UnitOfWork
registerNew(Object)
registerDirty(Object)
registerDeleted(String)
commit()
TDG
ReservationTDG
find(String): Reservation[*]
insert(String,String,String,Timeslot, Equipment)
update(String,String, String, String, Timeslot)
delete(String)
findByDate(Datetime):Reservation[*]
findByUserId(String):Reservation[*]
findByRoomId(String):Reservation[*]
findAll():Reservation[*]
findUserRes(String):Reservation[*]
findDateRoom(String, Datetime): Reservation[*]
EquipmentTDG
find(String): Equipment[*]
insert(String, Integer, Integer, Integer)
update(String, Integer, Integer, Integer)
delete(String)
WaitingTDG
find(id:String): Waiting[*]
insert(String,String,String, Timeslot, Equipment)
update(String,String, String, String, Timeslot)
delete(String)
findByDate(Datetime):Waiting[*]
findByRoomId(String):Waiting[*]
findAll():Waiting[*]
findByUser(String):Waiting[*]
findDateRoom(String, Datetime): Waiting[*]
findByUser(String):Waiting[*]
UserTDG
find(String): User[*]
insert(String,String,Boolean)
update(String,String)
delete(String)
TimeslotTDG
find(String): Timeslot[*]
findUser(String):Timeslot[*]
insert(String, Integer,Integer,Datetime,Integer, User)
update(String, Integer, Integer, Datetime, Integer)
delete(String)
RoomTDG
find(String): Room[*]
insert(Room)
delete(String)

Figure 5. Mapping Class Diagram

TDG Subsystem

The TDG subsystem follows the Table Data Gateway (TDG) pattern. The package contains TDG objects that acts as an intermediary between the data mapper and the database. Each data mapper objects have a corresponding TDG objects that will provide an interface for accessing the database. Hence, its role is to push data back and forth between the data mapper and the data base.



TDG
ReservationTDG
find(String): Reservation[*]
insert(String,String,String,Timeslot, Equipment)
update(String,String, String, String, Timeslot)
delete(String)
findByDate(Datetime):Reservation[*]
findByUserId(String):Reservation[*]
findByRoomId(String):Reservation[*]
findAll():Reservation[*]
findUserRes(String):Reservation[*]
findDateRoom(String, Datetime): Reservation[*]
EquipmentTDG
find(String): Equipment[*]
insert(String, Integer, Integer, Integer)
update(String, Integer, Integer, Integer)
delete(String)
WaitingTDG
find(id:String): Waiting[*]
insert(String,String,String, Timeslot, Equipment)
update(String,String, String, String, Timeslot)
delete(String)
findByDate(Datetime):Waiting[*]
findByRoomId(String):Waiting[*]
findAll():Waiting[*]
findByUser(String):Waiting[*]
findDateRoom(String, Datetime): Waiting[*]
findByUser(String):Waiting[*]
UserTDG
find(String): User[*]
insert(String,String,Boolean)
update(String,String)
delete(String)
TimeslotTDG
find(String): Timeslot[*]
findUser(String):Timeslot[*]
insert(String, Integer,Integer,Datetime,Integer, User)
update(String, Integer, Integer, Datetime, Integer)
delete(String)
RoomTDG
find(String): Room[*]
insert(Room)
delete(String)

Figure 6. TDG Class Diagram

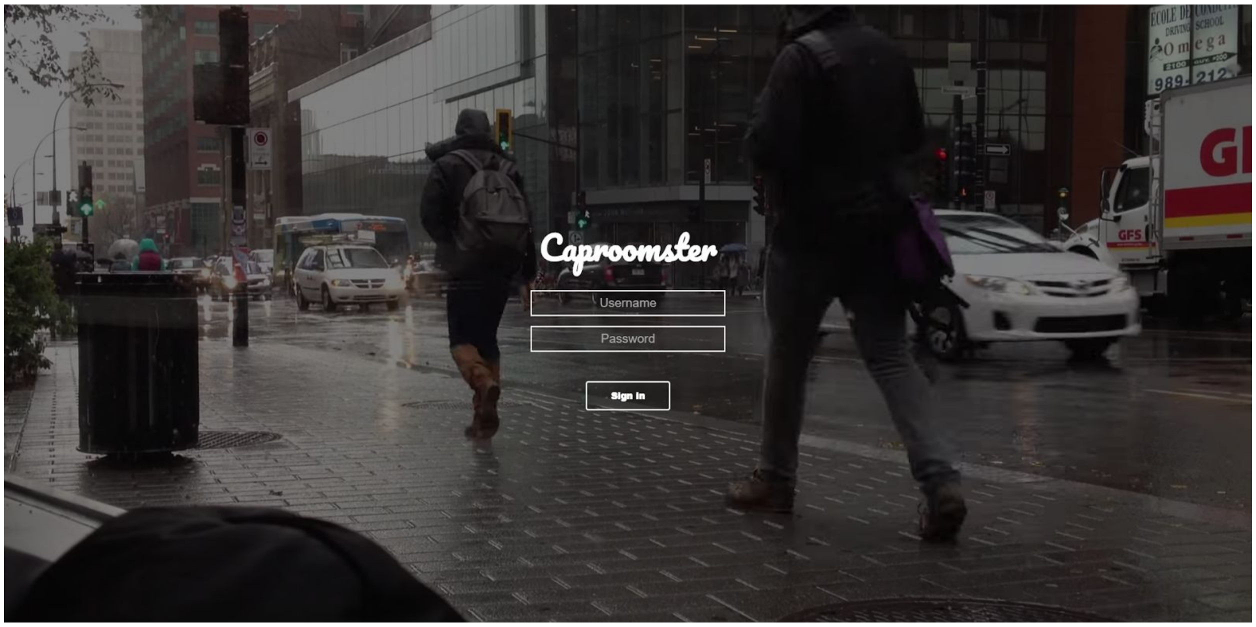
GUI Subsystem

The GUI subsystem in previous deliverable was implemented with Python templates. The register behaves as a controller and communicates with mapper subsystem, and represent the data in the templates. However, the GUI subsystem was not functional as desired. Multiple features were not able to realized in UI for end users.

The GUI subsystem is now refactored from Python templates to a client-server application with RESTful APIs and an independent front-end application. The RESTful APIs behaves as a controller and calling methods from core subsystem. The front-end application is implemented with AngularJS, and communicate with the RESTful APIs by sending requests over HTTP. The biggest advantages of this design are:

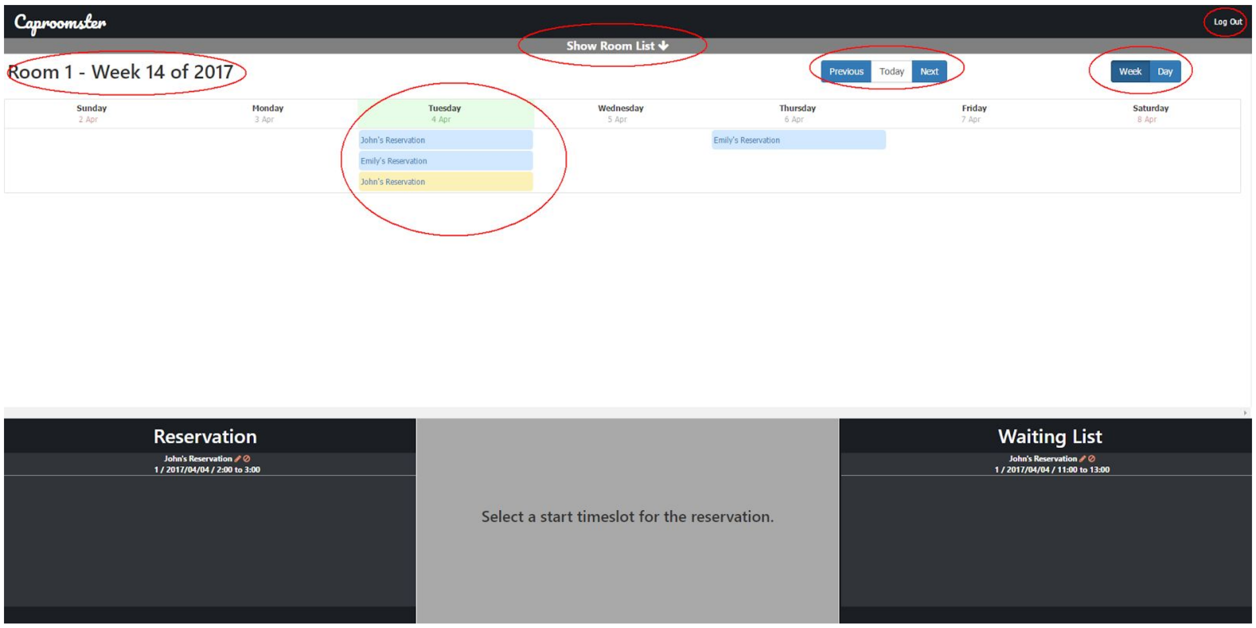
- Significantly less coupling comparing to the old implementation. Data controller now is separated into front-end application.

- Significantly improve the performance of the application in terms of speed. AngularJS modifies the page DOM directly instead of adding inner HTML code, which is faster.
- Using the latest front-end technology and RESTful APIs to keep the application up to date. This will reduce the future cost of maintenance.
- Better UX including user friendly UI and workflow and mobile friendly responsive design. The new UI improves the usability and efficiency for users to make reservations faster.



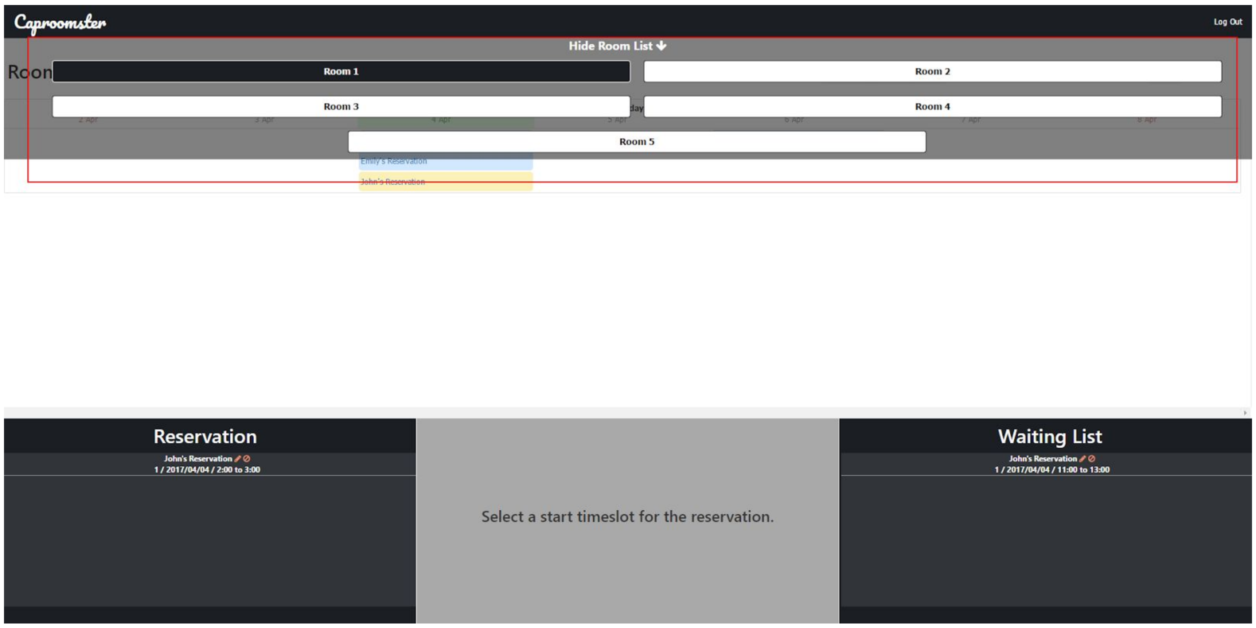
1) login page:

- user enters credentials and login to dashboard.
- If credential is authenticated, user will be redirected to dashboard
- if credential is not authenticated, an error message will show.



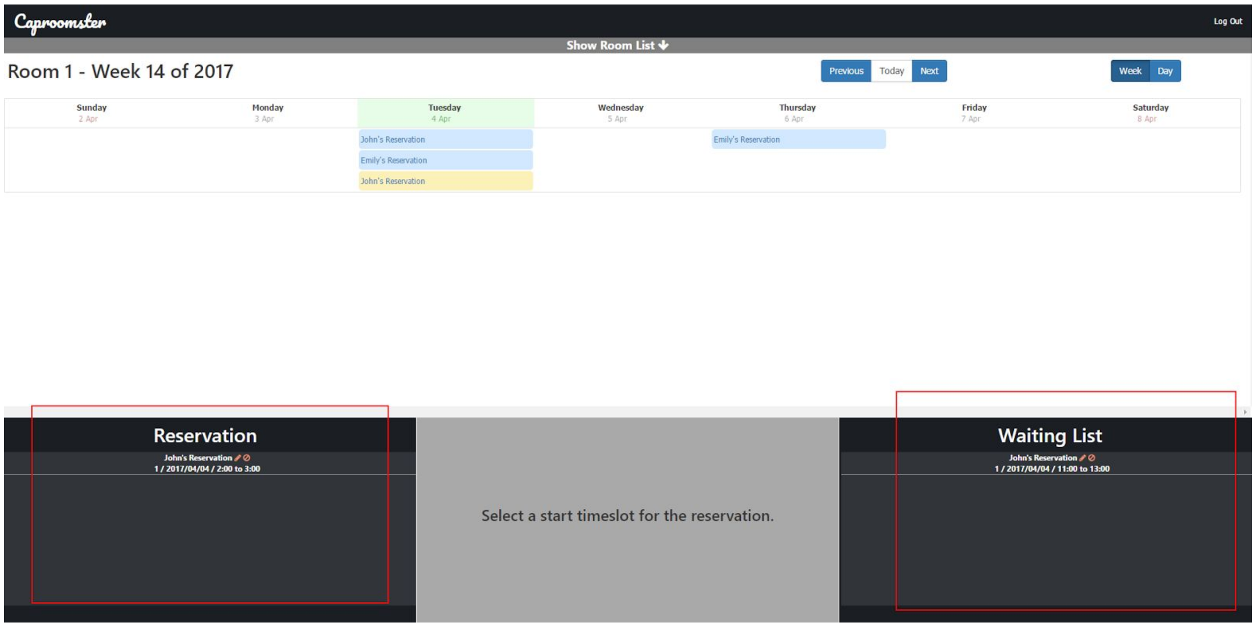
2) Dashboard page - calendar:

- On top left corner, the current shown date or week of the current calendar view is shown.
- On top right corner, a logout button is shown. User can logout by clicking this button
- In the top middle, a hidden toggle menu is shown. User can open the menu and select the calendar view of different rooms.
- Above the calendar view, users have options select from either weekly or daily view. Users can also go to previous day or week and next day or week depends on the calendar view. Users can click on `Today` button to quickly navigate to current date.
- In calendar view, users can see all reservations of a selected room. Blue means reservations, and yellow means waiting list.



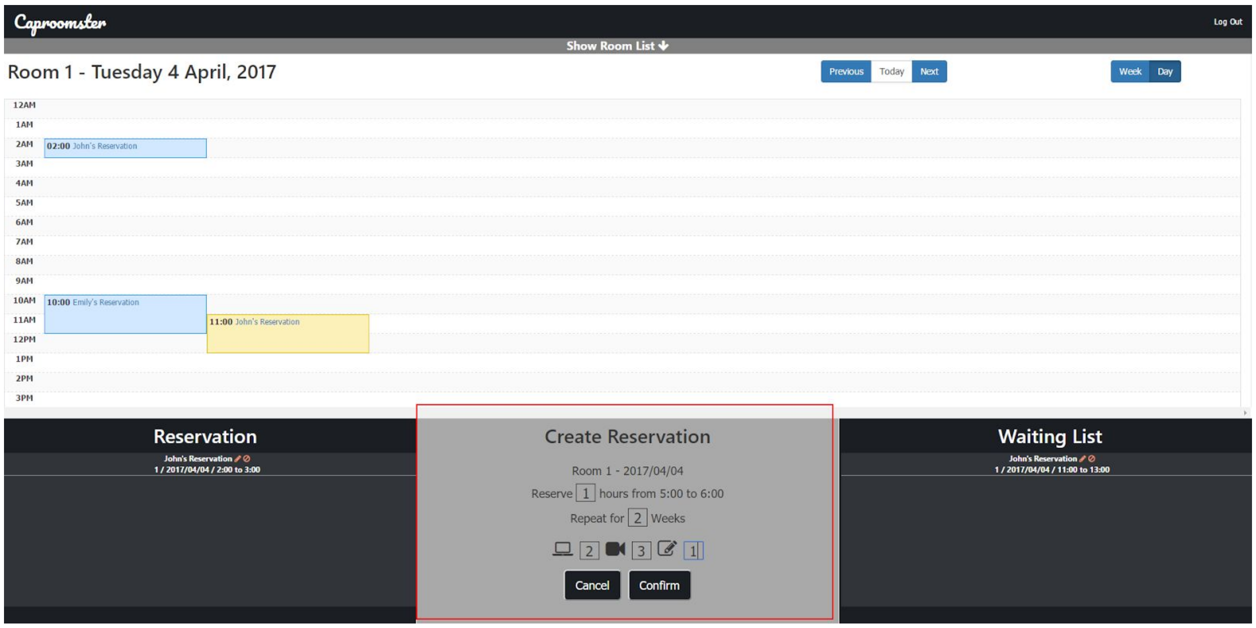
3) Dashboard - toggle menu:

- Users can select a room in order to show the corresponding reservations in calendar view



4) Dashboard - user information:

- On bottom left, a list of all reservation of a logged in user is shown
- On bottom right, a list of all waiting reservations of a logged in user is shown


Caproomster
Log Out
Show Room List
Room 1 - Tuesday 4 April, 2017
Previous Today Next
Week Day
12AM 1AM 2AM 3AM 4AM 5AM 6AM 7AM 8AM 9AM 10AM 11AM 12PM 1PM 2PM 3PM
02:00 John's Reservation
10:00 Emily's Reservation
11:00 John's Reservation

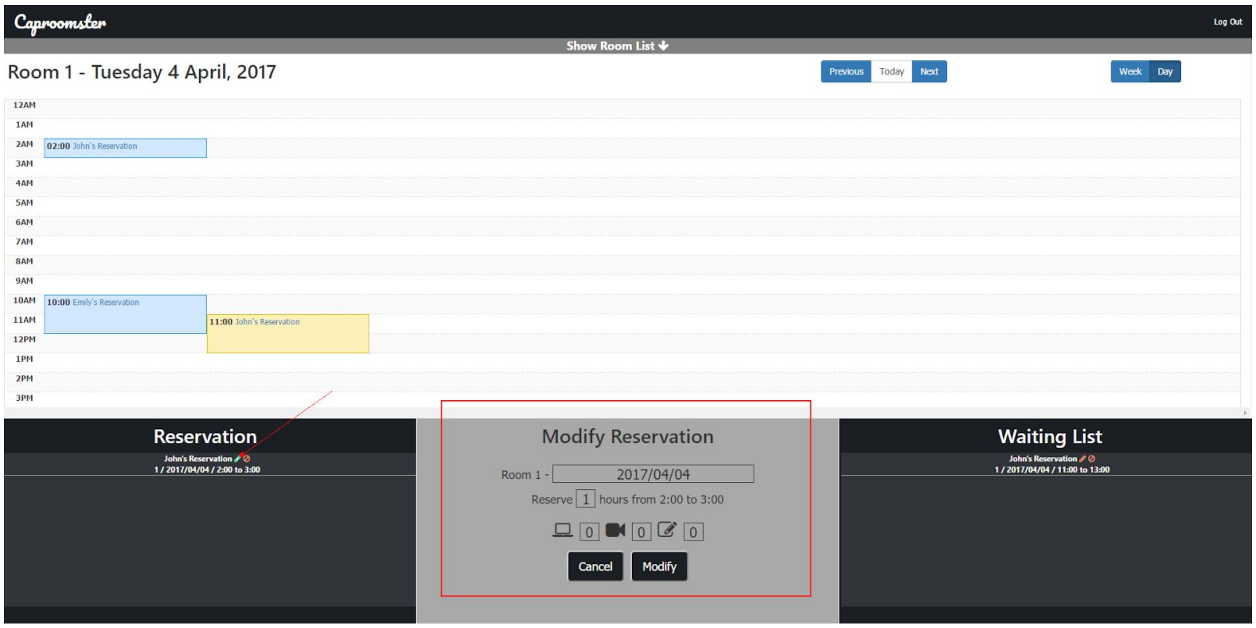
Reservation
John's Reservation
1 / 2017/04/04 / 2:00 to 3:00

Create Reservation
Room 1 - 2017/04/04
Reserve 1 hours from 5:00 to 6:00
Repeat for 2 Weeks
2 3 1
Cancel Confirm

Waiting List
John's Reservation
1 / 2017/04/04 / 11:00 to 13:00

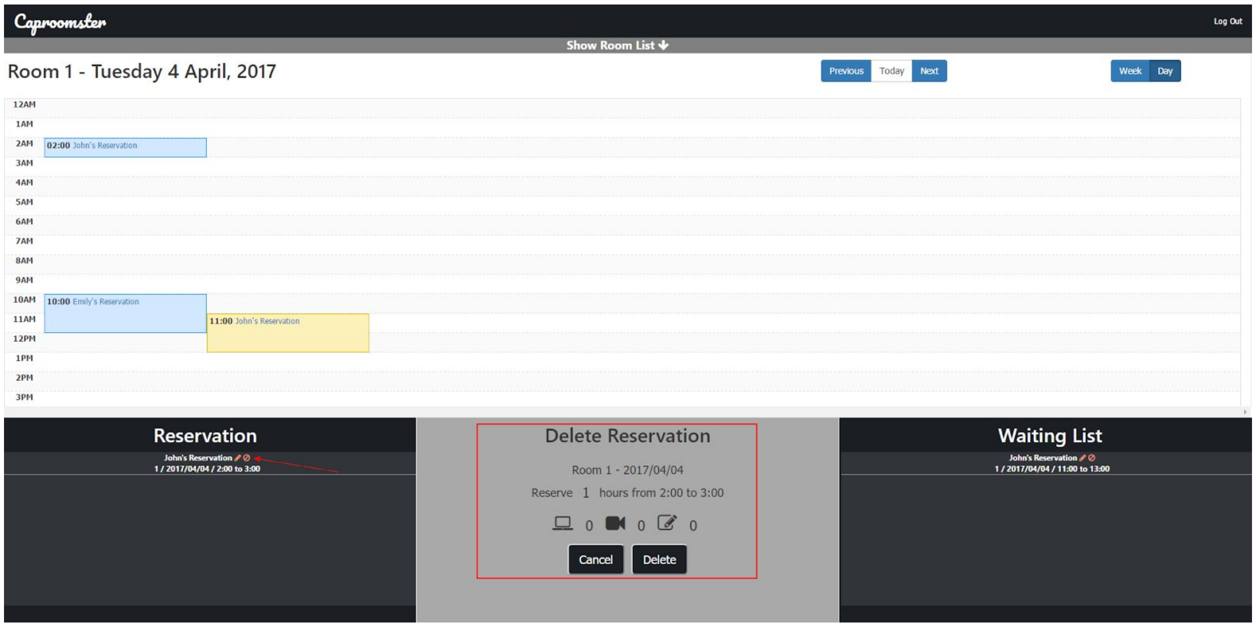
5) Dashboard - make reservation:

- Users can select a timeslot under daily calendar view to start making reservation
- Users have options to enter the length of reservation from 1 to 3 hours.
- Users have options make reservations for repeated weeks from 0 to 2. 0 means only making one reservation on selected date.
- Users have options to require projector, laptop, and whiteboard.
- Users can create the reservation but clicking the confirm button, or cancel this operation by clicking the cancel button
- If the reservation is valid, after clicking confirm button will show you a success message.
- If the reservation is invalid, after clicking the confirm button will show you an error message



6) Dashboard - modifying reservation:

- Users can start modifying any of his/her reservations or waiting reservations by clicking the pen icon next to each reservation.
- Users can modify the date to a desire future date.
- Users can modify the length of the reservation.
- Users can modify the numbers of required equipments.
- Users can modify the reservation by clicking the modify button, or cancel this operation by clicking the cancel button
- If the reservation is valid, after clicking modify button will show you a success message.
- If the reservation is invalid, after clicking the modify button will show you an error message



7) Dashboard - deleting reservation:

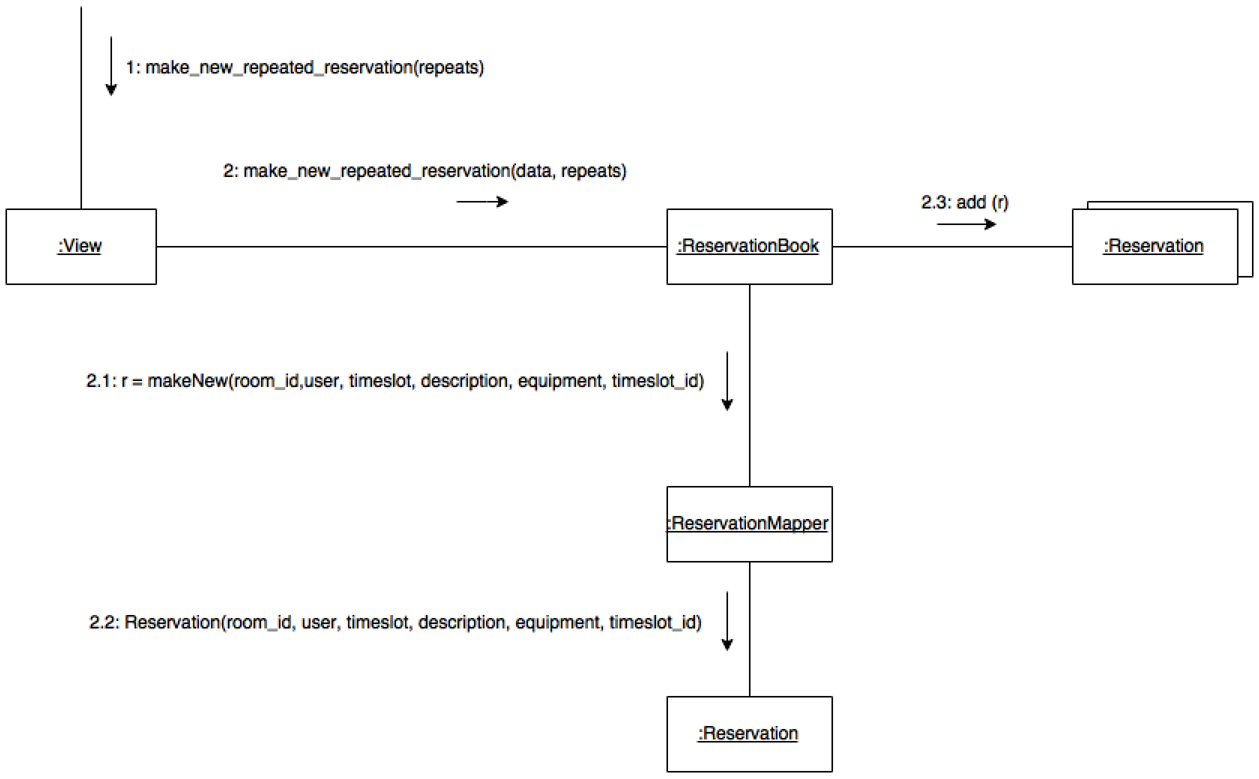
- Users can start deleting any of his/her reservations or waiting reservations by clicking the cross icon next to each reservation.
- Users can delete the reservation by clicking the delete button, or cancel this operation by clicking the cancel button
- If the reservation is valid, after clicking delete button will show you a success message.
- If the reservation is invalid, after clicking the delete button will show you an error message

Use case realizations

Note: Only core components are shown in the object diagram in order to reduce complexity so it is easier to understand.

Contract CO1: Make New Reservation
Operation: make_new_reservation(data)
Cross References: Use Case Reserve Room
Preconditions:
- A ReservationBook session is underway

Postconditions:
- Reservation instance r was created
- Attributes of r were initialized
- r is added to ReservationBook



1: make_new_repeated_reservation(repeats)
2: make_new_repeated_reservation(data, repeats)
:View
:ReservationBook
2.3: add (r)
:Reservation
2.1: r = makeNew(room_id,user, timeslot, description, equipment, timeslot_id)
:ReservationMapper
2.2: Reservation(room_id, user, timeslot, description, equipment, timeslot_id)
:Reservation

Figure 7. Make New Reservation Communication Diagram

This communication diagram represents the make new reservation operation and operation contract, which was built from the specified system operations and the system sequence diagram (ssd) pertaining to the reserve room use case. For instance, the ssd for the make new reservation use case specified that the system would need a make new reservation operation, and therefore it was added to the system operations and then further detailed in its operation contract by specifying its pre and post conditions. The above diagram satisfies these conditions since it shows how the operation creates a Reservation instance and initializes it, and also shows that it is added to the reservationBook. The process of creating sequence diagrams from use cases, then listing the identified system operations, and then creating operation contracts for each of these operations was the process used to produce each of the following communication diagrams.

When the user calls the make_new_repeated_reservation method, The View object then calls make_new_repeated_reservation in the ReservationBook class, which passes a message to the ReservationMapper in order to create and save the reservation in the database. From there a message is passed to Reservation to create an object of its type, which is captured and then added by ReservationBook to its own reservation collection.

Contract CO2: View Schedule
Operation: viewSchedule()
Cross References: Use Case View Schedule
Preconditions:

- User is logged in.
- A ReservationBook session is underway.

Postconditions:

- A schedule is created from information contained in the ReservationBook

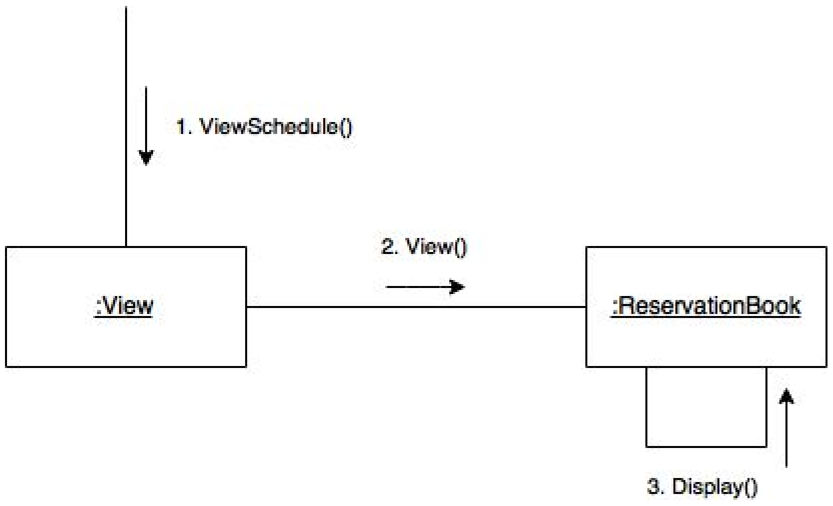


Figure 8. View Schedule Communication Diagram

If a user calls the viewSchedule() it is passed to an instance of the View class. Then the message from the ~~Registry~~ View is passed to ReservationBook. From there, ReservationBook calls the display() method from itself and displays the schedule.

Contract CO3: Modify Reservation
Operation: modify_reservation(data, reservationId)
Cross References: Use Case Modify Reservation
Preconditions:
- User is viewing their reservations
- The user has reservations

Postconditions:
- Reservation entry attributes are updated

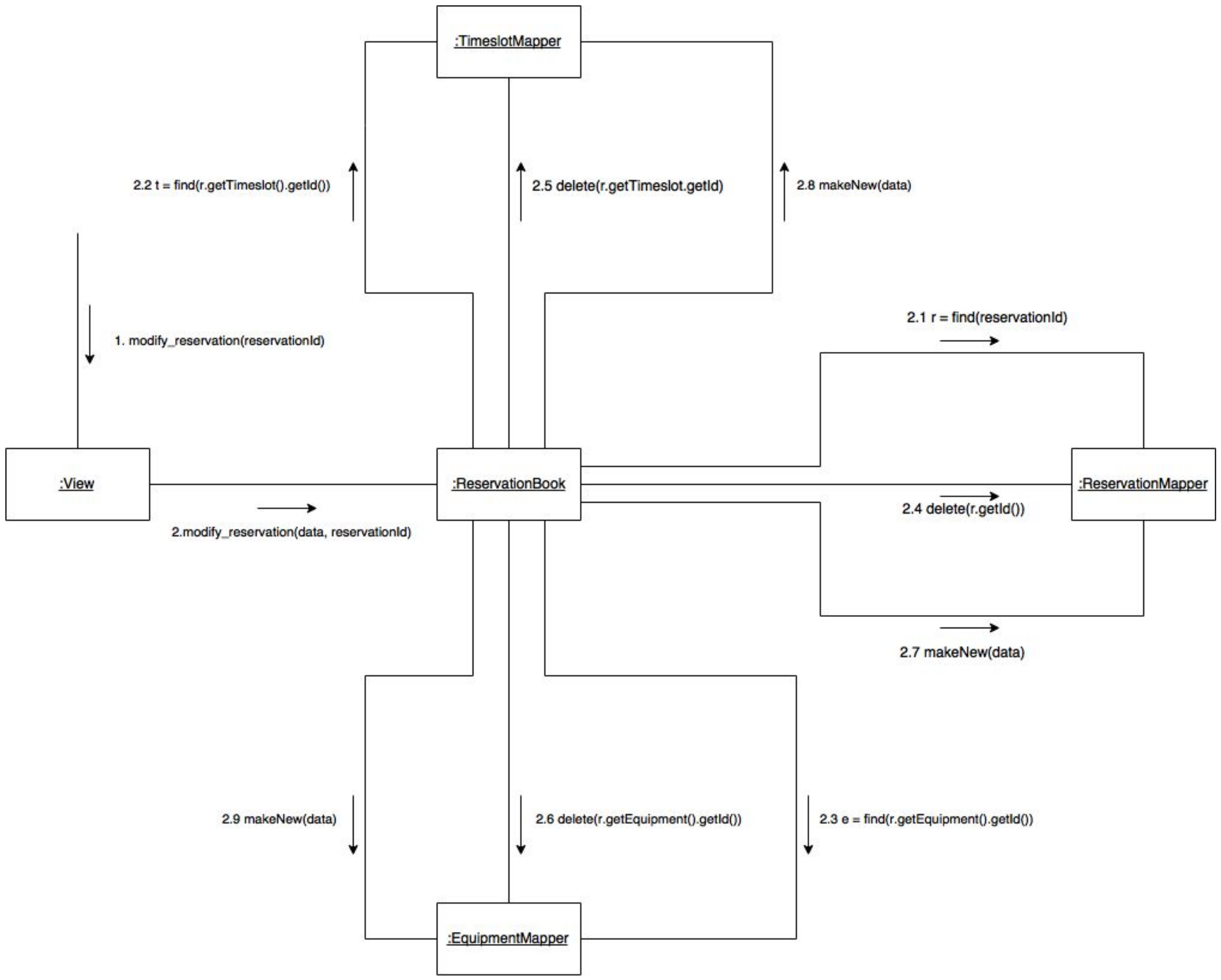

:TimeslotMapper
2.2 t = find(r.getTimeslot().getId())
2.5 delete(r.getTimeslot.getId)
2.8 makeNew(data)
1. modify_reservation(reservationId)
2.1 r = find(reservationId)
:View
:ReservationBook
:ReservationMapper
2.modify_reservation(data, reservationId)
2.4 delete(r.getId())
2.7 makeNew(data)
2.9 makeNew(data)
2.6 delete(r.getEquipment().getId())
2.3 e = find(r.getEquipment().getId())
:EquipmentMapper

Figure 9. Modify Reservation Communication Diagram

This communication diagram represents the modify reservation operation. When the user modifies the time for one of his reservations a message is sent to the view with the reservation's id. The view then passes this message to the reservation book by calling modify_reservation. This then finds the targeted reservation object from the ReservationMapper. Once the reservation object is found, the associated Timeslot and Equipment object are also found with their respective mapper. All these objects (Reservation, Timeslot and Equipment) are then deleted via their respective mappers. Once this is done, new Reservation, Timeslot and Equipment object are created with the new data that the user has requested.

Contract CO4: Cancel Reservation
Operation: delete_reservation(reservationId)
Cross References: Use Case Modify Reservation
Preconditions:

- User is viewing their reservations
- The user has reservations

Postconditions:

- Reservation entry removed from ReservationBook

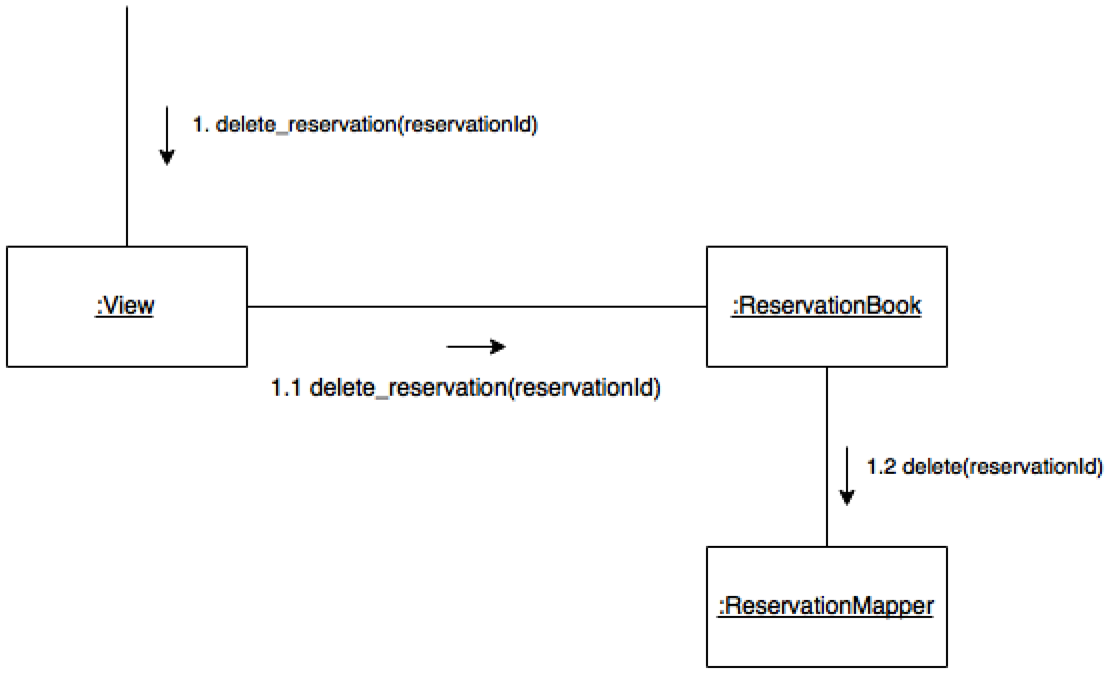

1. delete_reservation(reservationId)
:View
:ReservationBook
1.1 delete_reservation(reservationId)
1.2 delete(reservationId)
:ReservationMapper

Figure 10. Cancel Reservation Communication Diagram

If a user wants to cancel a reservation, delete_reservation(reservationId) method is called. After that, a message from View is passed to ReservationBook, and from there a message is passed to ReservationMapper with the reservationID in order to delete the specified reservation from the database.

~~**Contract CO5**: Add to Waiting List~~
~~Operation: commit_new_waiting(room, user, time, description, equipement, message)~~
~~Cross References: Use Case Add to Waiting List~~
~~Preconditions:~~

- ~~The user has initiated a Reservation Session~~
- ~~User is logged in~~
- ~~The user has tried to make a new reservation, but the room is unavailable~~

~~Postconditions:~~

- ~~A WaitingList Waiting instance w was created and placed into the appropriate waiting list depending on the user's capstone status~~
- ~~Attributes of w were initialized~~
- ~~Instance w was added to user's myWaitingList~~

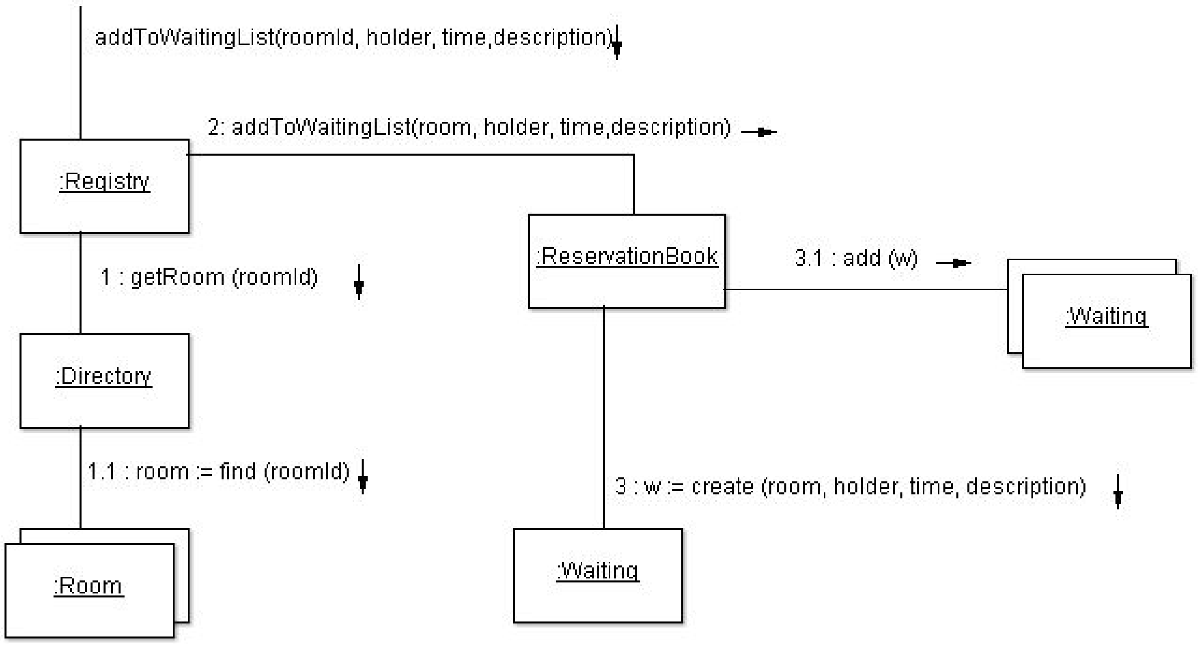
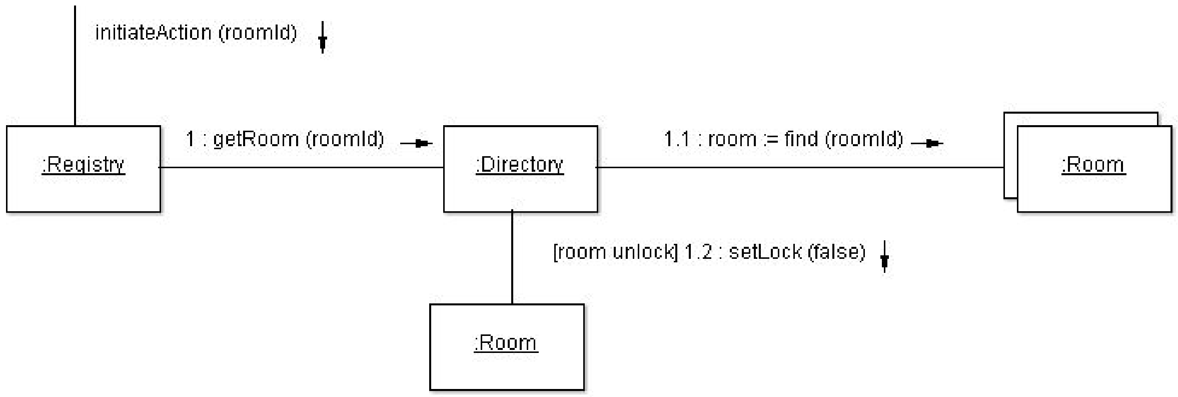

addToWaitingList(roomId, holder, time,description)
2: addToWaitingList(room, holder, time,description)
:Registry
1 : getRoom (roomId)
:Directory
1.1 : room := find (roomId)
:Room
:ReservationBook
3.1 : add (w)
:Waiting
3 : w := create (room, holder, time, description)
:Waiting

Figure 11. Add to Waiting List Communication Diagram

~~If a user tries to makeNewReservation() and the time slots are taken addToWaitingList() method is called. After it is called, it is passed to an instance of the Registry class, which passes a message to the directory to find the specified room in its Room collection. Because the room is not available at that time, a message from Registry is passed to ReservationBook, and from there a message is passed to Waiting to create an object of its type, which is captured and then added by ReservationBook to its own Waiting collection.~~

~~When the find(room id) message is passed from the Directory object, the find method of the room mapper is called, and then the RoomIdMap is checked to see if the map exists in memory, if it does not then the mapper asks the Table Data Gateway (TDG) to retrieve it from the database, it then adds it to the identity map to show that it has been created in memory. If it already exists in memory then it is simply retrieved.~~

~~**Contract CO6:** Initiate Action~~ ~~Operation: initiateAction(roomId)~~ ~~Cross References: Use Case Reserve Room, Add to Waiting List, Modify Reservation, Cancel Reservation~~ ~~Preconditions:~~ - ~~The user has initiated a Reservation Session~~ ~~Postconditions:~~ - ~~The room's lock is set to true~~



~~Figure 12. Initiate Action Communication Diagram~~

~~This communication diagram represents the initiate action operation. When the user initiates an action such as reserving a room, adding him/herself to a waiting list, and modifying or canceling a reservation, an initiate action message is passed to the registry along with the id of the room to which the operation is related. Once the registry gets this message, it sends a get room message to the directory, which then sends a find message to its collection of rooms, once it finds the room, it "locks" the room's lock. When the find room message is passed, the room mapper checks if the room exists in memory with the identity map, if it does it retrieves it, if it does not the mapper asks the tdg to retrieve it from the database. When the room's lock is "locked" the room's mapper registers the object as "dirty" in the unit of work since its state has been changed.~~

~~**Contract CO7:** End Action~~
~~Operation: endAction(roomId)~~
~~Cross References: Use Case Reserve Room, Add to Waiting List, Modify Reservation, Cancel Reservation~~
~~Preconditions:~~
- ~~The initiateAction(roomId) operation has been called~~

~~Postconditions:~~
- ~~The room's lock is set to false~~

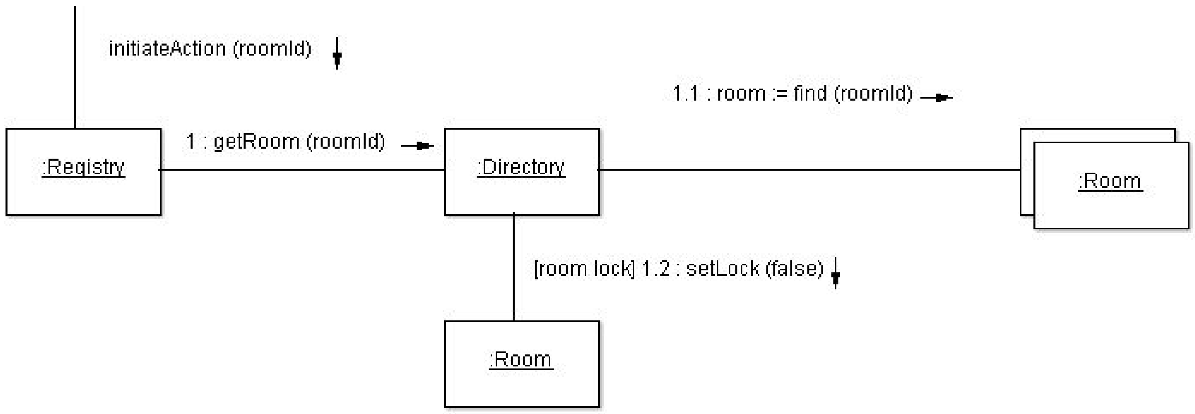


Figure 13. End Action Communication Diagram

~~This communication diagram represents the end action operation. When the user ends an action such as reserving a room, adding him/herself to a waiting list, and modifying or canceling a reservation, an end action message is passed to the registry along with the id of the room to which the operation was related. Once the registry gets this message, it sends a get room message to the directory, which in turn passes a find room message to its collection of rooms. Once the room is found an unlock message is passed to it and its lock is "unlocked". Similarly to the initiate action operation diagram, when the find room message is passed, the room mapper checks if the room exists in memory with the identity map, if it does, it retrieves it, if it does not the mapper asks the tdg to retrieve it from the database. When the room's lock is "unlocked" the room's mapper registers the object as "dirty" in the unit of work since its state has been changed.~~

~~Contract CO8:~~ ~~Update Waiting~~
~~Operation: updateWaiting(roomId)~~
~~Cross References: Modify Reservation, Cancel Reservation~~
~~Preconditions:~~
- ~~The modifyReservation(reservationId) or cancelReservation(reservationId) operation has been called~~

~~Postconditions:~~
- ~~If room is available at Waiting w's timeslot, an instance of Reservation r is created~~
- ~~Reservation r's attributes are initialized with the w's attributes~~
- ~~Reservation r is added to reservationBook and w is removed from reservationBook~~

This is an internal implementation detail, and thus it is irrelevant.

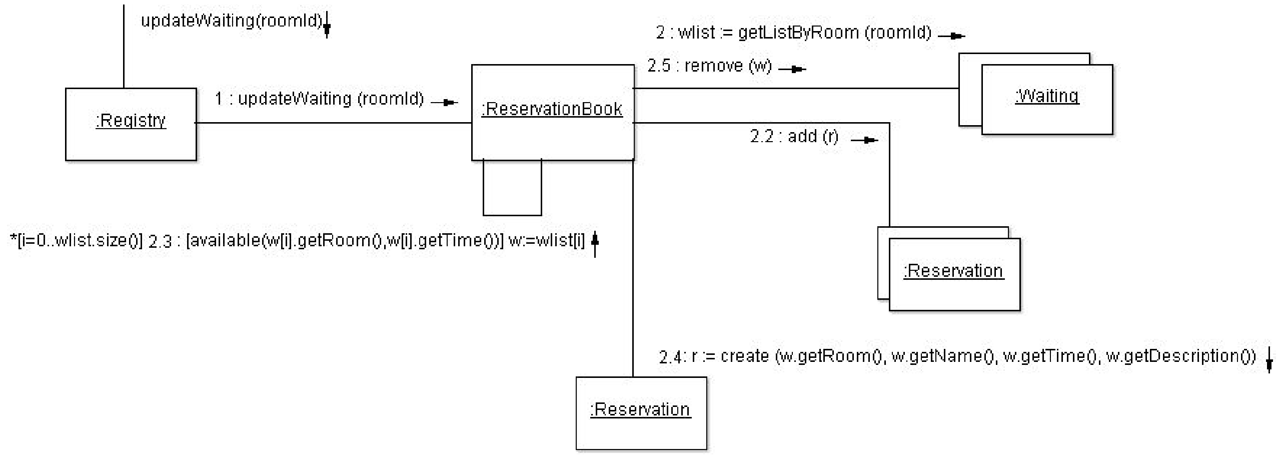

updateWaiting(roomId)
:Registry
1 : updateWaiting (roomId)
:ReservationBook
2 : wlist := getListByRoom (roomId)
2.5 : remove (w)
:Waiting
2.2 : add (r)
:Reservation
*[i=0..wlist.size()] 2.3 : [available(w[i].getRoom(),w[i].getTime())] w:=wlist[i]
2.4: r := create (w.getRoom(), w.getName(), w.getTime(), w.getDescription())
:Reservation

Figure 14. Update Waiting Communication Diagram

~~This communication diagram represents the update waiting list operation. Whenever the user modifies or cancels a reservation an update waiting list message is sent to the registry. The registry then passes an update waiting list message to the reservation book with the modified or canceled room's id and time. The reservation book then passes a message to its queue of users waiting for a room, which finds if an instance in the waiting list can be made into a reservation because of the previous reservation modification or cancellation. If such an instance in the waiting list exists it is removed and then the reservation book creates a reservation with the information provided by it and stores it in its reservation collection. The process of finding waiting list instances to be made into reservations is done until none of them can be converted. Furthermore, when an instance in the waiting list is being retrieved, a message is sent to the waiting list mapper, which then asks its identity map if the object already exists in memory, if it does it is retrieved, if not the mapper asks the tdg to retrieve it from the database. When a waiting list instance is removed, its mapper registers it as~~ deleted in the unit of work.

Testing

As the system requires validation that it satisfies the requirements that were given, both unit tests, as well as system tests were performed.

Unit tests - Code Coverage

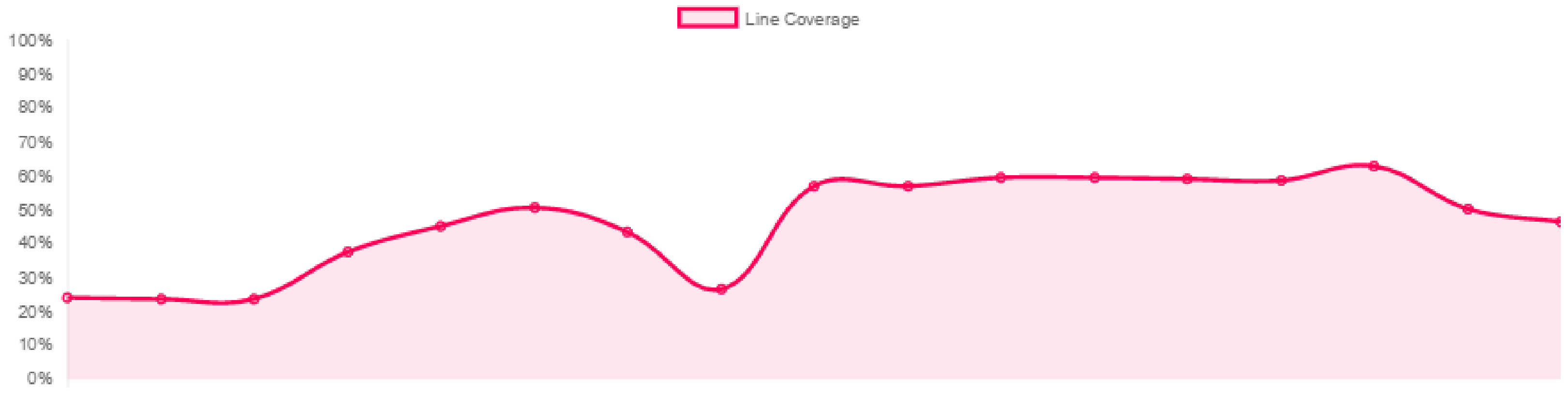


Figure - The Progression of statement test coverage since the start of test coverage monitoring

Unfortunately, as the project was not tested at all prior to the start of our development, the test coverage was at **0%** initially. As development of new requirements progressed, the test coverage steadily increased. Finally, at present time, the statement coverage of the system stands at approximately **46%**.

System Tests - Verifying requirements are met

To ensure that requirements are met with the development of our system, black-box testing of our application was performed. Expected Interactions with the server were written as scripted tests, to be performed against our system. Each use case would thus have The outcome of this effort was a guarantee that all code contributions satisfy initially obtained requirements. Shown below are the system tests that were used:

System Test 1	
Test Name	test_login
Description	Test is used to determine if the system behaves correctly when an authorized user logs in.
Test Data	Username: 'Jake' Password: 'pass'
Expected Outcome	User is successfully logged in.

System Test 2	
Test Name	test_logout
Description	Test is used to determine if the system behaves correctly when an authorized user logs out
Test Data	Username: 'Jake' Password: 'pass'
Expected Outcome	User is successfully logged out.

System Test 3	
Test Name	make_new_reservation_times
Description	Test is used to determine if the system behaves correctly when an authorized user makes a new reservation.
Test Data	Username: 'Jake' Password: 'pass' Reservation: 2000/03/19 1 PM - 2 PM
Expected Outcome	Reservation is successfully created

System Test 4	
Test Name	make_new_reservation_times_more_than_3_hours_long
Description	Test is used to determine if the system behaves correctly when an authorized user makes a new reservation that is longer than allowed time.
Test Data	Username: 'Jake' Password: 'pass' Reservation: 2000/03/19 1 PM - 11 PM
Expected Outcome	System rejects reservation request.

System Test 5	
Test Name	make_new_reservation_date_more_than_3_elems
Description	Test is used to determine if the system behaves correctly when an authorized user makes a new repeat reservation that for more than 3 weeks
Test Data	Username: 'Jake' Password: 'pass' Reservation: 2000/03/19 1 PM - 11 PM, 5 Repeats
Expected Outcome	System rejects reservation request.

System Test 6	
Test Name	get_reservations_by_user_with_login
Description	Test is used to determine if the system behaves correctly when an authorized user attempts to view their current reservations
Test Data	Username: 'Jake' Password: 'pass'
Expected Outcome	System shows the current user's reservations.

System Test 7	
Test Name	delete_reservation
Description	Test is used to determine if the system behaves correctly when an authorized user attempts to cancel their existing reservation
Test Data	Username: ‘Jake’ Password: ‘pass’ Reservation: An existing reservation tied to the user.
Expected Outcome	System cancels the given reservation.

Appendix A. REST API Documentation

General Information

Most resources require a user that is **logged in** in order to be accessed. If an unauthenticated user attempts to access one of these resources without being logged in, the response will be:

401 - UNAUTHORIZED

```
{
  "unauthorized": "Not logged in. You must login."
}
```

Resource - `/login`

GET

Checks if the current user is logged in or not.

Possible Responses

200 - OK

```
{
    "success": {
        "username": "iscapstone"
    }
}
```

401 - UNAUTHORIZED

```
{
  "unauthorized": "Not logged in. You must login."
}
```

POST

Login attempt. Returns a session token.

Expected Request Payload

John is a regular student. All users have been pre-generated.

```
{
  "username":"John",
  "password":"pass"
}
```

Success Response

200 - OK

```
{
    "data": {
        "capstone": "True",
        "username": "Mary"
    },
    "login success": "Successfully logged in"
}
```

Resource - `/logout`

GET

Allows the current user to log out.

Success Response

200 - OK

```
{
  "logout success": "Successfully logged out."
}
```

Resource - `/rooms/all`

GET

Retrieves all CAPSTONE rooms in the system.

Success Response

200 - OK

```
{
    "rooms": [
       1,
       2,
       3,
       4,
       5
    ]
}
```

Resource - reservations/create

POST

Creates a new reservation for a specific timeslot, user and room.
If there is a time conflict, you will be put on the waitlist.
If there is not enough equipment for you reservation, you will be put on the waitlist.
If your are trying to reserve for more than 3 hours in a given week, this call with abort. No reservations or waiting lists will be created/altered.

Expected Request Payload

```
{
      "roomId":"1",
      "username": "Mary",
    "timeslot":  {
       "startTime": "7",
       "endTime": "8",
       "date": "2020/03/19"
    },
    "equipment":  {
       "laptop": 1,
       "projector": 1,
       "board": 1
    },
      "description": "cool meeting"
}
```

Success Response

200 - OK

```
{
    "makeNewReservation": "successfully created reservation",
    "reservation": "83e51d8f-d1d7-4a9b-aa06-ae29db717c46"
}
```

Resource - `reservations/repeat/:repeats`

POST

Creates a new reservation for a specific timeslot, user and room, every week, for `repeats` weeks in the future.
`repeats` needs to be an integer between 0 and 2.
If there is a time conflict, you will be put on the waitlist.
If there is not enough equipment for your reservation, you will be put on the waitlist.
If you are trying to reserve for more than 3 hours in a given week, this call with abort. Your previously registered reservations/waitings won't be reverted.

Expected Request Payload

```
{
      "roomId":"1",
      "username": "Mary",
    "timeslot":  {
       "startTime": "7",
       "endTime": "8",
       "date": "2020/03/19"
    },
    "equipment":  {
       "laptop": 1,
       "projector": 1,
       "board": 1
    },
      "description": "cool meeting"
}
```

Success Response

200 - OK

```
{
    "reservations": [
         {
                 "description": "cool meeting",
                 "equipment": {
                         "equipmentId": "eb14b4d7-799f-4550-b2b1-9b4a897e34eb",
                         "laptops": 1,
                         "projectors": "1",
                         "whiteboards": 1
                 },
                 "reservationId": "f8d8b8eb-8da0-4f94-b31a-9cc062170ad6",
                 "room": {
                         "roomId": 1
                 },
                 "timeslot": {
                         "date": "2020/03/19",
                         "endTime": 8,
                         "startTime": 7,
                         "timeId": "96285b1a-311d-4769-95b5-b24719223d27",
                         "userId": "Mary"
                 },
                 "user": {
                         "isCapstone": true,
                         "username": "Mary"
                 }
         }
    ],
    "success": "You have successfully repeated your reservations. Results are shown below.",
    "waitings": [
         {
                 "description": "cool meeting",
                 "equipment": {
                         "equipmentId": "84248089-54d5-41eb-8255-d98ca6e01364",
                         "laptops": 1,
                         "projectors": "1",
                         "whiteboards": 1
                 },
                 "room": {
                         "roomId": 1
                 },
                 "timeslot": {
                         "date": "2020/03/26",
                         "endTime": 8,
                         "startTime": 7,
                         "timeId": "1ae98160-da92-451d-8e5d-554eb271c6ad",
                         "userId": "Mary"
                 },
                 "user": {
                         "isCapstone": true,
                         "username": "Mary"
                 },
                 "waitingId": "62fc2ad1-2278-4e24-9a9f-cf32387dea3d"
         }
    ]
}
```

Resource - `reservations/modify/:reservationId`

PUT

Modifies the existing reservation with `reservationId` and replaces it with the new reservation in the payload.

If your new reservation makes you exceed your allowed 3 hours per week of reservations, the creation of the new reservation with abort and the old reservation will be placed back.

Hitting this endpoint cause an update of all the waiting list reservations to successful reservations if possible.

Expected Payload

```
{
      "roomId":"1",
      "username": "Mary",
    "timeslot":  {
       "startTime": "4",
       "endTime": "5",
       "date": "2020/03/19"
    },
    "equipment":  {
       "laptop": 1,
       "projector": 1,
       "board": 1
    },
      "description": "cool meeting"
}
```

Success Response

200 - OK

```
{
    "makeNewReservation": "successfully created reservation",
    "reservation": "83e51d8f-d1d7-4a9b-aa06-ae29db717c46"
}
```

Resource - `reservations/user/:userId`

GET

Gets all the reservations of a specific user

Success Response

200 - OK

```
{
    "reservations": [
        {
            "description": "cool meeting",
            "equipment": {
                "equipmentId": "c98a52ea-a1f0-4452-b77b-a172e5de12ec",
                "laptops": 1,
                "projectors": "1",
                "whiteboards": 1
            },
            "reservationId": "83e51d8f-d1d7-4a9b-aa06-ae29db717c46",
            "room": {
                "roomId": 1
            },
            "timeslot": {
                "date": "2020/03/19",
                "endTime": 8,
                "startTime": 7,
                "timeId": "18fc4a23-83f8-4b13-a71b-862de4e003a2",
                "userId": "Mary"
            },
            "user": {
                "isCapstone": true,
                "username": "Mary"
            }
        }
    ],
    "username": "Mary",
    "waitings": [
        {
            "description": "cool meeting",
            "equipment": {
                "equipmentId": "f51e1be0-8f32-409b-9030-85ab772a9e71",
                "laptops": 1,
                "projectors": "1",
                "whiteboards": 1
            },
            "room": {
                "roomId": 1
                },
                "timeslot": {
                        "date": "2020/03/19",
                        "endTime": 8,
                        "startTime": 7,
                        "timeId": "034df89d-e3b3-4dfb-b1b8-8341dfd3b8bd",
                        "userId": "Mary"
                },
                "user": {
                        "isCapstone": true,
                        "username": "Mary"
                },
                "waitingId": "9091b42d-9f8b-464f-936f-16db3f01fa5e"
        }
    ]
}
```

Resource - `reservations/room/:roomId`

GET

Gets all the reservations within a particular room.

Success Response

200 - OK

```
{
    "reservations": [
        {
                "description": "cool meeting",
                "equipment": {
                        "equipmentId": "c98a52ea-a1f0-4452-b77b-a172e5de12ec",
                        "laptops": 1,
                        "projectors": "1",
                        "whiteboards": 1
                },
                "reservationId": "83e51d8f-d1d7-4a9b-aa06-ae29db717c46",
                "room": {
                        "roomId": 1
                },
                "timeslot": {
                        "date": "2020/03/19",
                        "endTime": 8,
                        "startTime": 7,
                        "timeId": "18fc4a23-83f8-4b13-a71b-862de4e003a2",
                        "userId": "Mary"
                },
                "user": {
                        "isCapstone": true,
                        "username": "Mary"
                }
        }
    ],
    "roomId": "1",
    "waitings": [
        {
                "description": "cool meeting",
                "equipment": {
                        "equipmentId": "f51e1be0-8f32-409b-9030-85ab772a9e71",
                        "laptops": 1,
                        "projectors": "1",
                        "whiteboards": 1
                },
                "room": {
                        "roomId": 1
                },
                "timeslot": {
                        "date": "2020/03/19",
                        "endTime": 8,
                        "startTime": 7,
                        "timeId": "034df89d-e3b3-4dfb-b1b8-8341dfd3b8bd",
                        "userId": "Mary"
                },
                "user": {
                        "isCapstone": true,
                        "username": "Mary"
                },
                "waitingId": "9091b42d-9f8b-464f-936f-16db3f01fa5e"
        }
    ]
}
```

Resource - `reservations/all`

GET

Gets all the reservations in the system

Success Response

200 - OK

```
{
    "reservations": [
         {
                 "description": "cool meeting",
                 "equipment": {
                         "equipmentId": "c98a52ea-a1f0-4452-b77b-a172e5de12ec",
                         "laptops": 1,
                         "projectors": "1",
                         "whiteboards": 1
                 },
                 "reservationId": "83e51d8f-d1d7-4a9b-aa06-ae29db717c46",
                 "room": {
                         "roomId": 1
                 },
                 "timeslot": {
                         "date": "2020/03/19",
                         "endTime": 8,
                         "startTime": 7,
                         "timeId": "18fc4a23-83f8-4b13-a71b-862de4e003a2",
                         "userId": "Mary"
                 },
                 "user": {
                         "isCapstone": true,
                         "username": "Mary"
                 }
         }
    ],
    "waitings": [
         {
                 "description": "cool meeting",
                 "equipment": {
                         "equipmentId": "f51e1be0-8f32-409b-9030-85ab772a9e71",
                                "laptops": 1,
                                "projectors": "1",
                                "whiteboards": 1
                        },
                        "room": {
                                "roomId": 1
                        },
                        "timeslot": {
                                "date": "2020/03/19",
                                "endTime": 8,
                                "startTime": 7,
                                "timeId": "034df89d-e3b3-4dfb-b1b8-8341dfd3b8bd",
                                "userId": "Mary"
                        },
                        "user": {
                                "isCapstone": true,
                                "username": "Mary"
                        },
                        "waitingId": "9091b42d-9f8b-464f-936f-16db3f01fa5e"
                }
        ]
}
```

Resource - `reservations/:reservationId`

DELETE

Deletes a reservation or waiting based on the specified ID. Returns `404` if no reservation or waiting is found with that ID.
Hitting this endpoint causes an update of all the waiting list reservations to successful reservations if possible.

Success Response

If deleting a reservation: 200 - OK

```
{
    "reservationId": "b29615b7-2895-4929-955a-ea0491e283de",
    "success": "reservation successfully deleted"
}
```

If deleting a waiting: 200 - OK

```
{
    "success": "reservation on waiting list successfully deleted",
    "waitingId": "3b0c25cf-999e-4638-b27b-d372607cfe96"
}
```