

Concordia University

Department of Computer Science

and Software Engineering

Software Architecture and Design I

SOEN 344 S --- 2017

Chronos

Software Architecture Documentation

|  |  |
| --- | --- |
| **Team members** | |
| **Name** | **Student ID** |
| Nicholas Burdet | 29613773 |
| Georges Mathieu | 26863477 |
| Olivier Cameron-Chevrier | 27228805 |
| Stefano Pace | 27454716 |
| Julian Ippolito | 27419112 |
| Adam Trudeau-Arcaro | 27459157 |
| Joey Tedeschi | 27513062 |

**Document history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| February 21st, 2017 | 0.1 | Adding Appendix to include the Modification Log | Stefano Pace |
| February 22nd, 2017 | 0.2 | Preliminary Editing of Sections 1, 2.1, 3 | Stefano Pace |
| March 16th, 2017 | 0.3 | Editing of Section 2 SDs | Adam Arcaro |
| March 16th, 2017 | 0.4 | Editing of Section 2 SDs | Joey Tedeschi |
| March 16th, 2017 | 0.5 | Editing of Section 2 SDs | George Mathieu |
| March 16th, 2017 | 0.6 | Editing of Section 2 SDs | Nicholas Burdet |
| March 16th, 2017 | 0.7 | Editing Appendix – Modifications Log | Julian Ippolito |
| March 16th, 2017 | 0.8 | Editing of Section 4 | Adam Arcaro |
| March 17th, 2017 | 0.9 | Editing of Section 2 SDs | George Mathieu |

\*\*\*\*\*\*\*Please Note:

Please note, there was an issue regarding committing changes for multiple members to GitHub. Using the GitHub desktop application, changes were overridden when (after committing to their own branch beforehand for testing), Stefano as team lead decided to merge to the develop Branch once testing was committed and verified by him. Thus, most of the commits now fall under his name. This issue was only noticed near the end of the project timeline. However, this is just to note that the number of commits found in the project history don’t accurately reflect the total commits per team member. For more detailed breakdown of the tasks assigned and completed by each member of the team, please consult the issues tab in GitHub. Thank you for your understanding.

Table of Contents

[1. Introduction 1](#_Toc477512148)

[1.1 Purpose 1](#_Toc477512149)

[1.2 Scope 1](#_Toc477512150)

[1.3 Glossary of Terms 1](#_Toc477512151)

[2. Logical View 2](#_Toc477512152)

[2.1 Layers 3](#_Toc477512153)

[2.1.1 Data Layer 3](#_Toc477512154)

[2.1.2 Domain Layer 3](#_Toc477512155)

[2.1.3 Presentation Layer 3](#_Toc477512156)

[2.2 Class Diagram 4](#_Toc477512157)

[2.3 Interaction Diagrams 6](#_Toc477512158)

[2.3.1 View Calendar 6](#_Toc477512159)

[2.3.2 Request Reservation 7](#_Toc477512160)

[2.3.3 Modify Reservation 9](#_Toc477512161)

[2.3.4 View Reservation 11](#_Toc477512162)

[2.3.5 View Reservation List 11](#_Toc477512163)

[2.3.6 Cancel Reservation 12](#_Toc477512164)

[2.3.7 Referenced Sequence Diagrams 12](#_Toc477512165)

[3. Use Case View 16](#_Toc477512166)

[4. Data Model 17](#_Toc477512167)

[5. Installation Manual 18](#_Toc477512168)

[5.1 System Requirements 18](#_Toc477512169)

[5.2 Downloading Chronos 19](#_Toc477512170)

[5.3 Setting up the Database 19](#_Toc477512171)

[5.4 Install Chronos 20](#_Toc477512172)

[6. Modification Log 23](#_Toc477512173)

[6.1 Styling in the Calendar 23](#_Toc477512174)

[6.2 Reservation Before Current Time Functionality 24](#_Toc477512175)

[6.3 Deleting Individual Reservations 26](#_Toc477512176)

[6.4 Overlapping Reservations in Different Rooms 27](#_Toc477512177)

[6.5 Waitlist Functionality 28](#_Toc477512178)

[6.5.1 Waitlist position not an attribute stored in the Database 28](#_Toc477512179)

[6.5.2 Waitlisted Reservation counted in Check for Maximum allowed per Week 28](#_Toc477512180)

[6.5.3 Waitlisted Reservation Deletion when reaching Maximum for the week 29](#_Toc477512181)

[6.5.4 Waitlisted Reservation Deletion for overlapping Rooms 29](#_Toc477512182)

[6.5.5 Waitlist position updates upon Cancelation of Reservation 30](#_Toc477512183)

[6.6 Modification Functionality 30](#_Toc477512184)

[6.7 My Reservations View Functionality 31](#_Toc477512185)

[6.8 Mutual Exclusion and Room Locking 32](#_Toc477512186)

[6.9 Closing the Window logs out the User 34](#_Toc477512187)

[6.10 Bad code smells 34](#_Toc477512188)

**List of Figures**

[Figure 1: Layered Architecture Diagram: 2](#_Toc477467308)

[Figure 2: Main Class Diagram 4](#_Toc477467309)

[Figure 3: Full Implementation of Data Package 5](#_Toc477467310)

[Figure 4:Main Class Diagram 5](file:///C:\Users\space\Documents\University\Winter%202017\SOEN%20344\Project\Tyrell-Corp\Documentation\SAD\SAD%20-%20Draft%20-%20V0.5.docx#_Toc477467311)

[Figure 5: Sequence diagram for View Calendar’s viewCalendar() system operation 6](#_Toc477467312)

[Figure 6: Sequence diagram for showRequestForm() system operation 7](#_Toc477467313)

[Figure 7: Sequence diagram for requestReservation() system operation 8](#_Toc477467314)

[Figure 8: Sequence diagram for getModificationForm() system operation 9](#_Toc477467315)

[Figure 9: Sequence diagram for modifyReservation() system operation 10](#_Toc477467316)

[Figure 10: Sequence diagram for View Reservation’s viewReservation() system operation 11](#_Toc477467317)

[Figure 11: Sequence diagram for viewReservationList() system operation 11](#_Toc477467318)

[Figure 12: Sequence diagram for Cancel Reservation’s cancelReservation() system operation 12](#_Toc477467319)

[Figure 13: Referenced Sequence Diagram Find Active Reservations 12](#_Toc477467320)

[Figure 14: Referenced Sequence Diagram Find Reservation 13](#_Toc477467321)

[Figure 15: Referenced Sequence Diagram Find Time Slot Reservations 13](#_Toc477467322)

[Figure 16: Referenced Sequence Diagram Find User Reservations 14](#_Toc477467323)

[Figure 17: Referenced Sequence Diagram Complete Work 15](#_Toc477467324)

[Figure 18: Use Case Diagram 16](#_Toc477467325)

[Figure 19. Entity-relationship diagram for the system database 17](#_Toc477467326)

[Figure 20: PHP test file output 19](#_Toc477467327)

# Introduction

In software development, an ordered process is required in engineering a software system. The main stages of this process include requirements engineering, design, implementation, testing, maintenance and version control. The Software Architecture Document (SAD) is dedicated to the second stage of the software development process: design.

In this SAD document, we will provide the various UML diagrams needed to structure the implementation of the software project.

## Purpose

The SAD is a documentation artifact that describes the high-level architecture and low-level design of a software system. It is used to illustrate the system with UML diagrams providing information on its architecture and specific scenarios. In this SAD, it contains architectural views, use cases, class diagram and entity relation diagram.

This document is intended to be used by the team responsible for implementing the system. This development team will produce the code repository respecting the architecture provided in the SAD.

## Scope

The SAD for Chronos, an online university conference room reservation system, is applied to the implementation of its code. It is affected by the Software Requirements Specification (SRS) document made at the previous stage of development: requirements engineering. The architecture for Chronos’ code is constructed following the requirements stated in its SRS.

## Glossary of Terms

|  |  |
| --- | --- |
| Term | Definition |
| Active Reservation | The earliest reservation stored for a room at a specific time slot. This reservation currently holds the room and is not part of the waiting list |
| Available | Room that can be reserved by a user |
| Reservation | Time slot which has been booked by a user |
| Room | The entity the user wants to reserve |
| SAD | Document containing artifacts related to the architecture of the software |
| Time Slot | One-hour long time interval starting on the hour |
| User | Student or staff member who is registered with the engineering faculty |
| Waiting list | A series of reservations that represents the order in which users will take over a canceled reservation for a specific room and time |

# Logical View

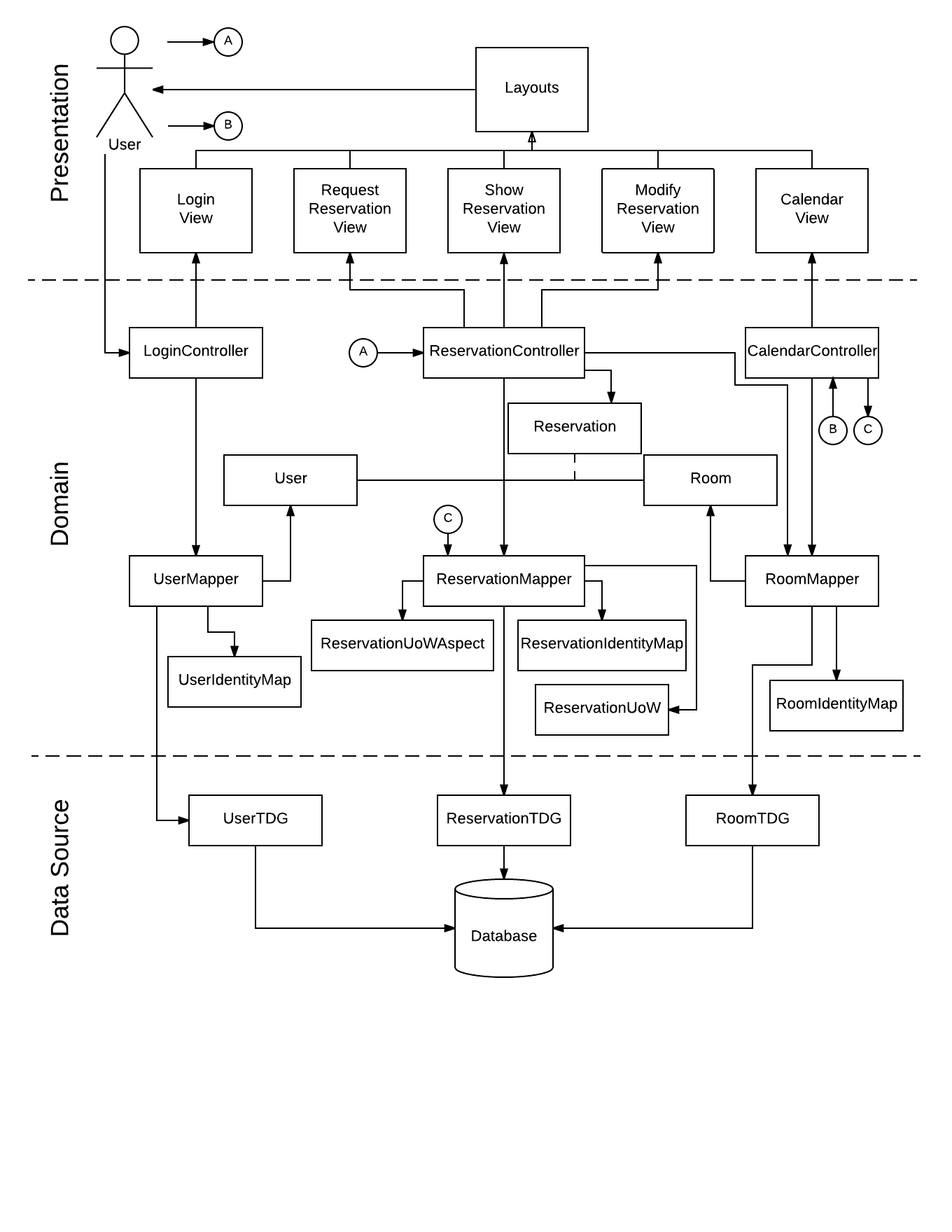


Figure : Layered Architecture Diagram:

## Layers

### Data Layer

The data layer is the lowest layer of the system; it contains the Table Data Gateways (TDG) that interact with the database to persist objects of their related classes. The data layer can only be accessed by a mapper interacting with its respective TDG.

### Domain Layer

The domain layer is the middle layer of the system; it contains the main objects of the system, their controllers, and their respective mappers. The domain layer also contains the Identity Maps used to synchronize the use of objects throughout the system, as well as the Units of Work used to group change to any object and the Aspects.

### Presentation Layer

The presentation layer is the outermost layer of the system. This outermost layer is the layer the user interacts with. It contains all the views generated by controllers in the domain level and based on the user’s input.

## Class Diagram

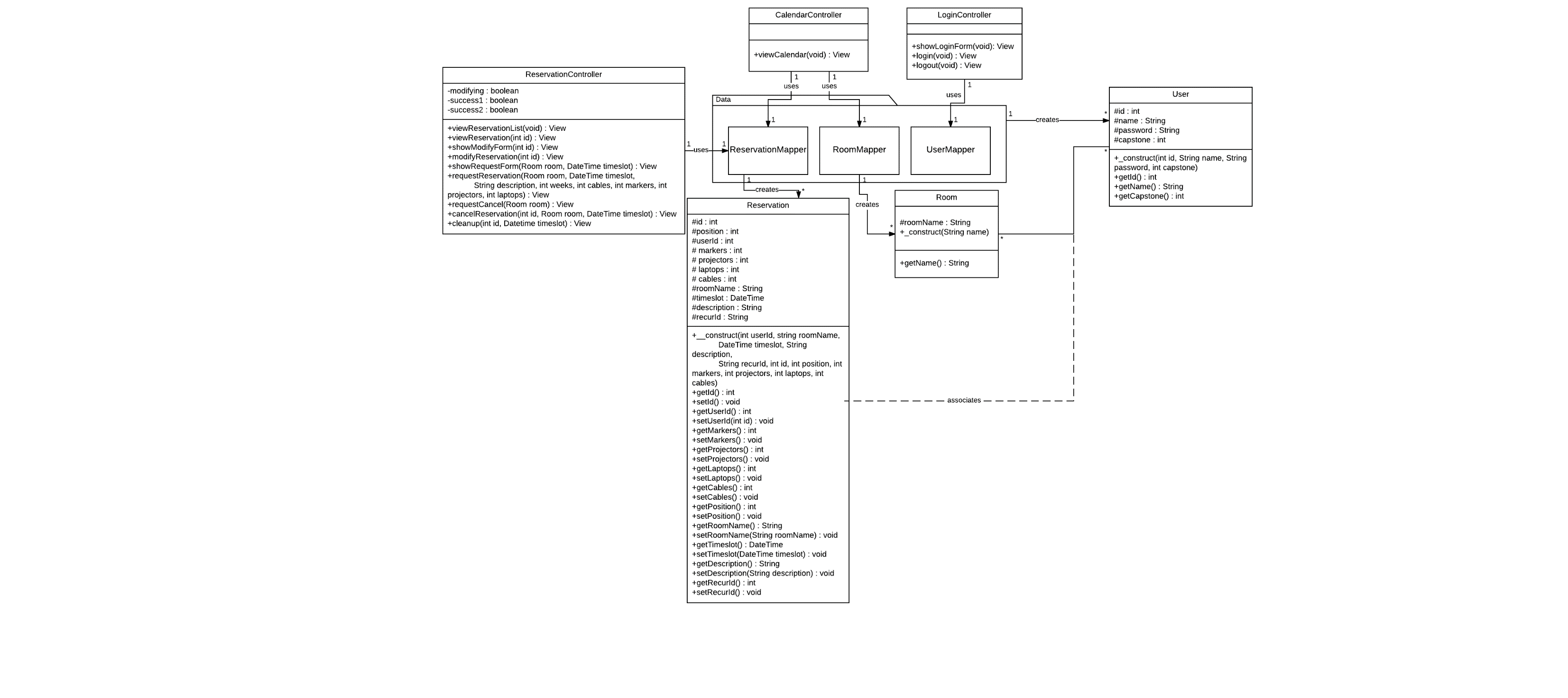


Figure : Main Class Diagram

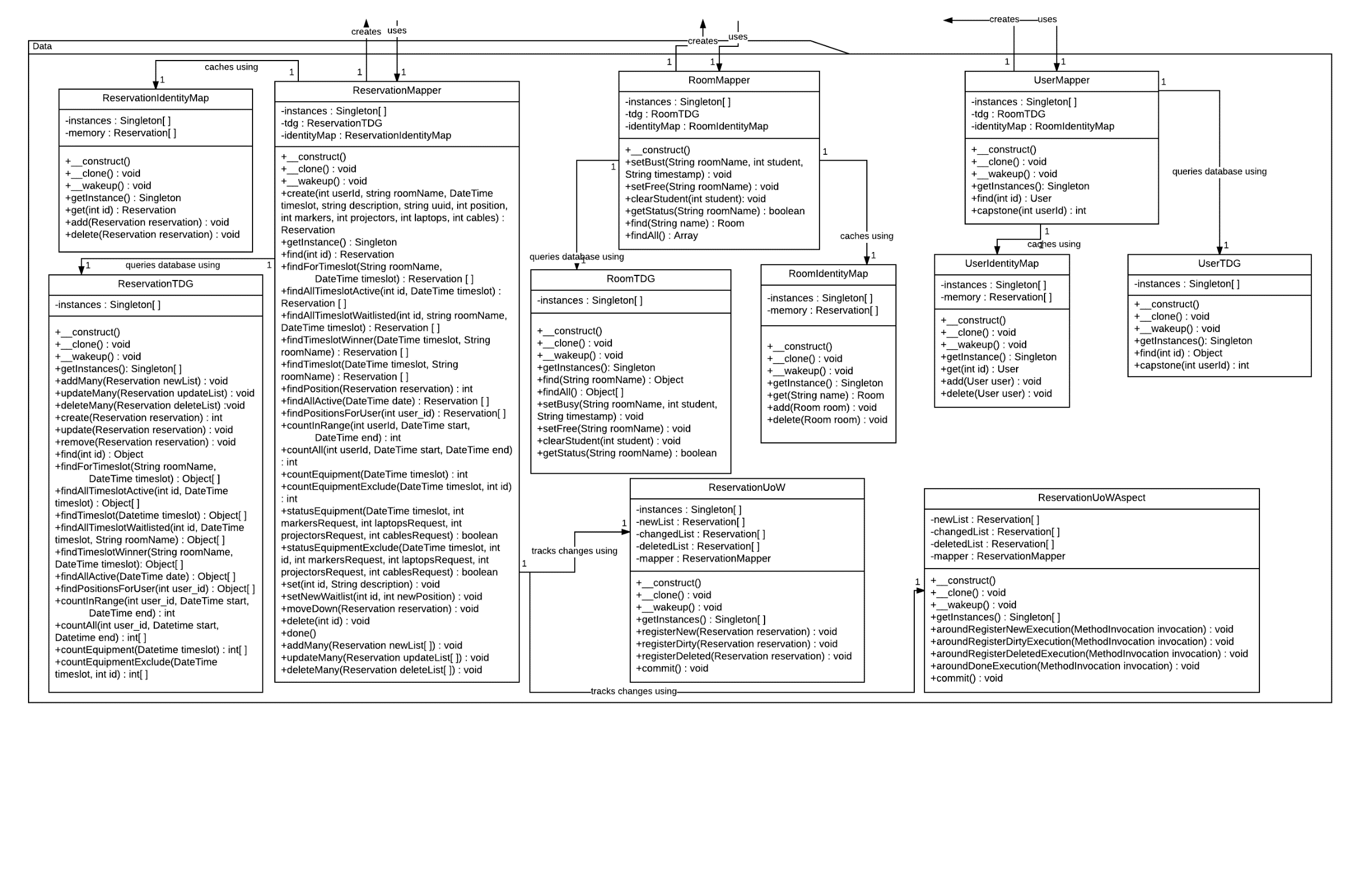


Figure : Full Implementation of Data Package

Figure 4:Main Class Diagram

## Interaction Diagrams

See section 2.3.7 for all referenced sequence diagrams

### View Calendar

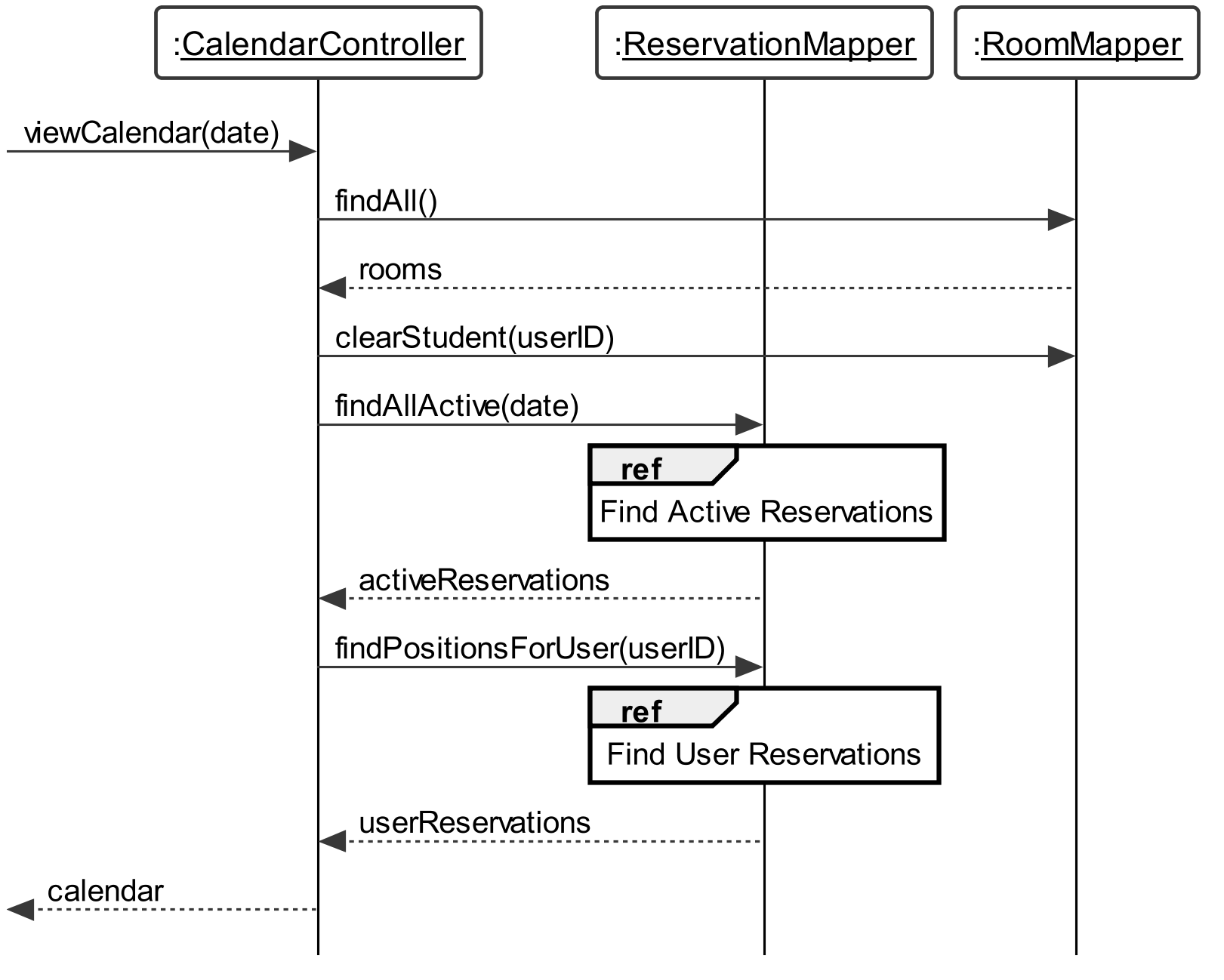


Figure : Sequence diagram for View Calendar’s viewCalendar() system operation

### Request Reservation

#### showRequestForm

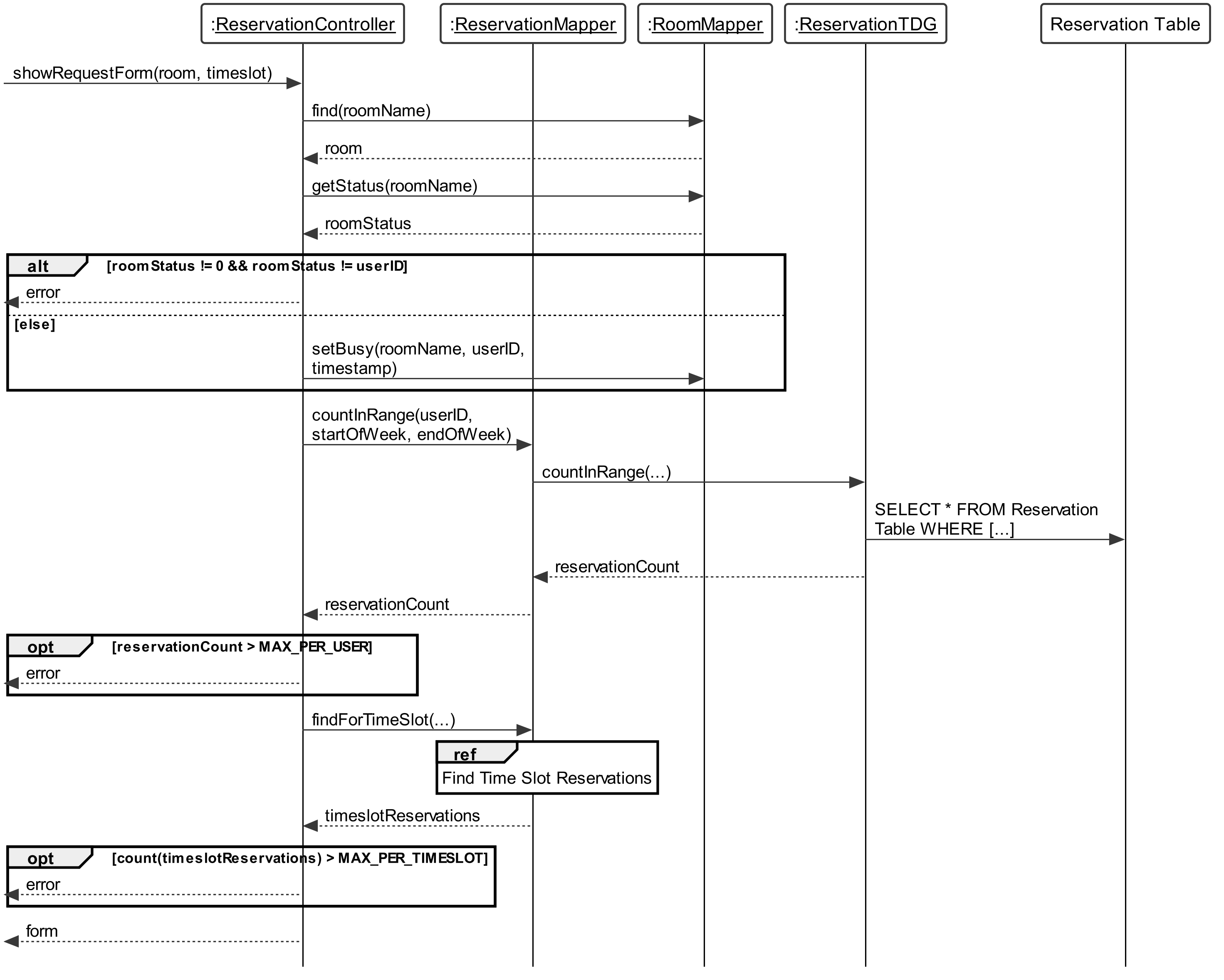


Figure : Sequence diagram for showRequestForm() system operation

#### requestReservation

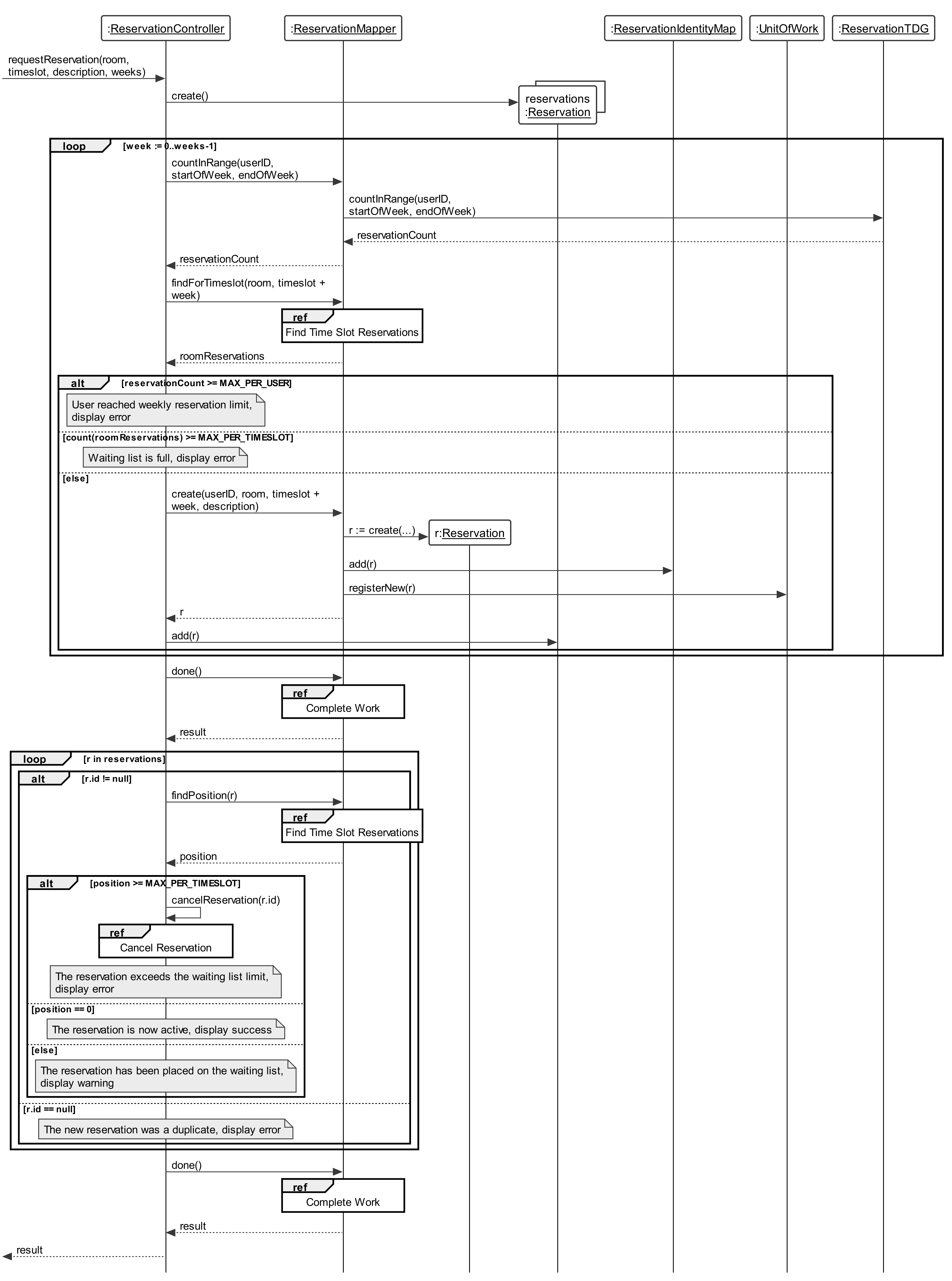


Figure : Sequence diagram for requestReservation() system operation

### Modify Reservation

#### showModifyForm

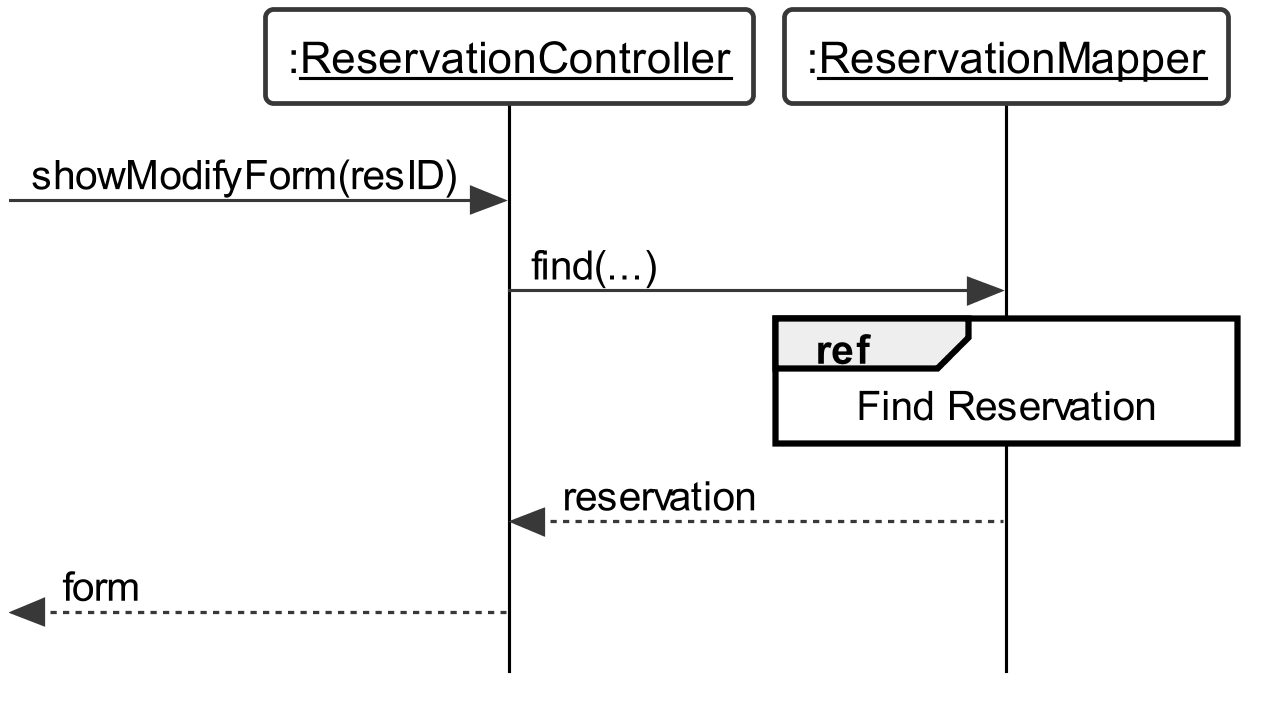


Figure : Sequence diagram for getModificationForm() system operation

#### modifyReservation

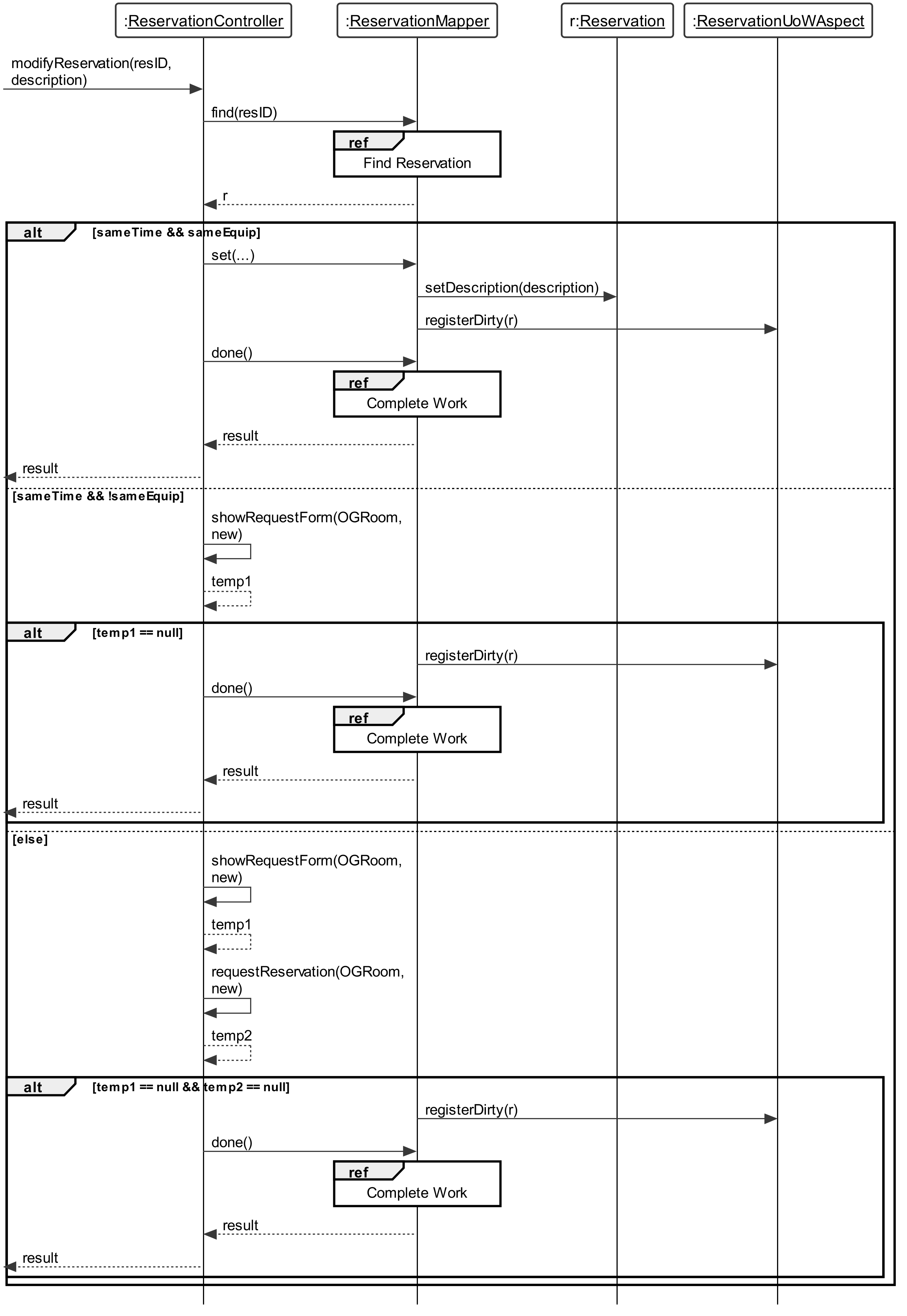


Figure : Sequence diagram for modifyReservation() system operation

### View Reservation

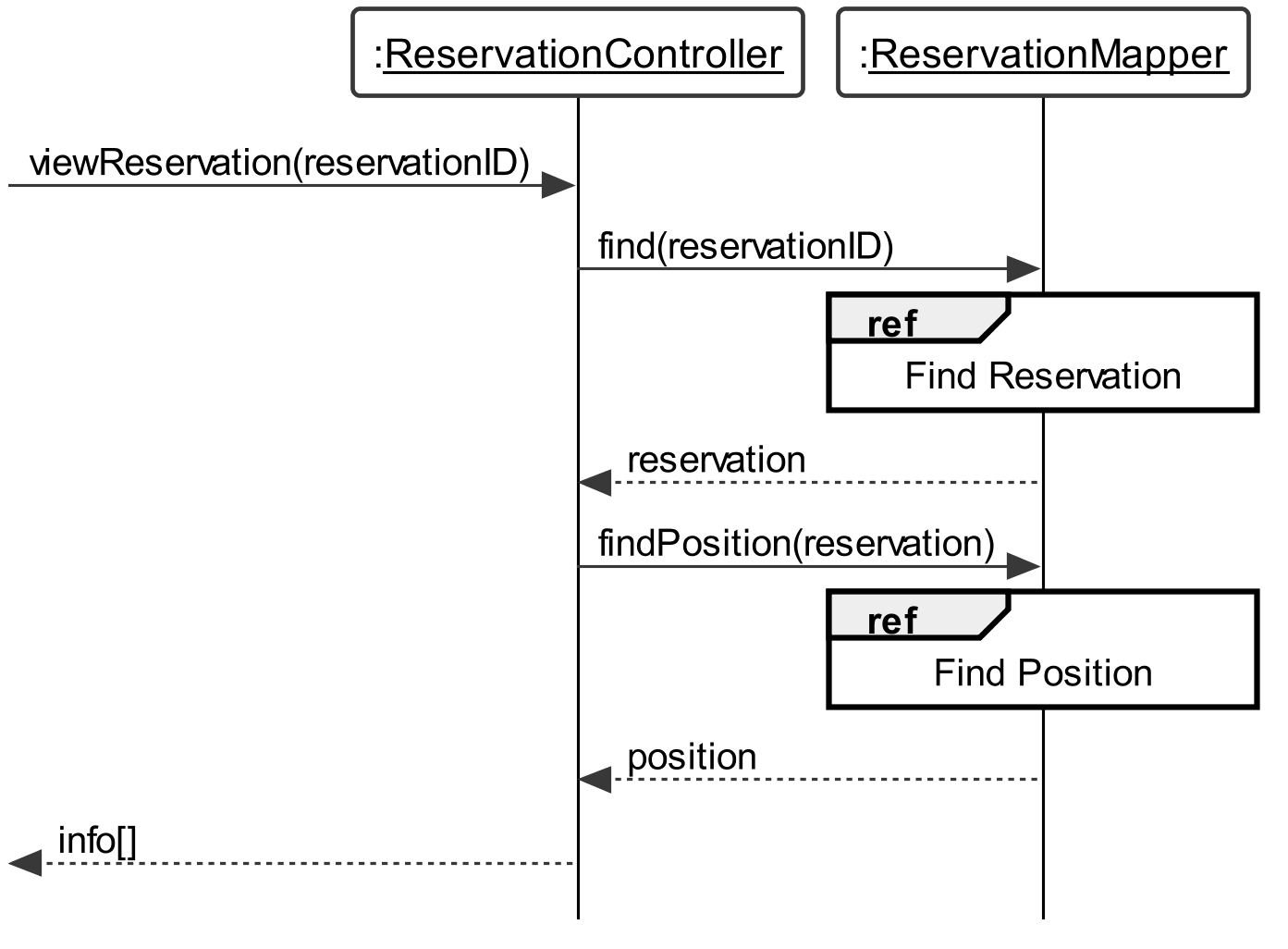


Figure : Sequence diagram for View Reservation’s viewReservation() system operation

### View Reservation List

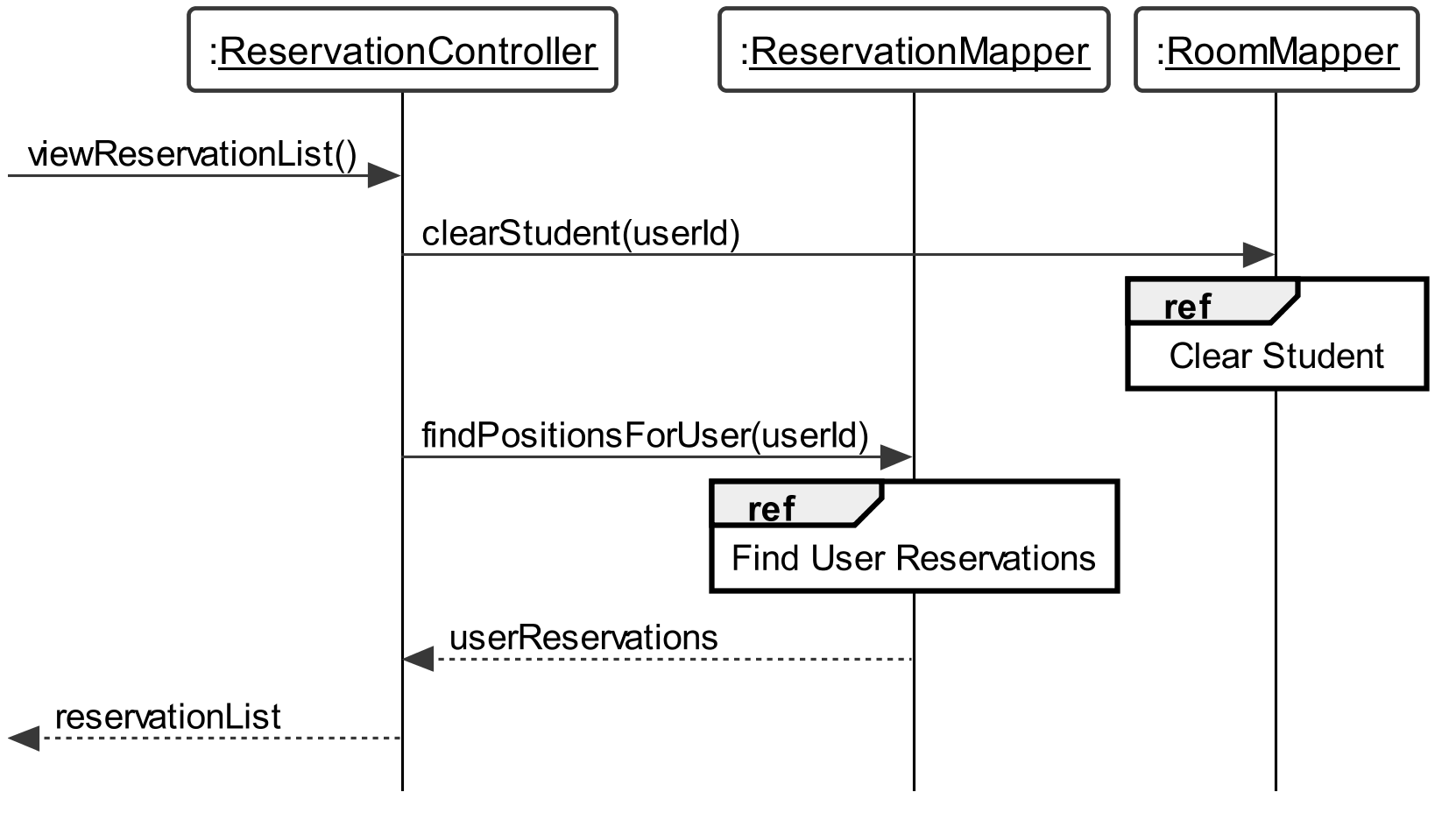


Figure : Sequence diagram for viewReservationList() system operation

### Cancel Reservation

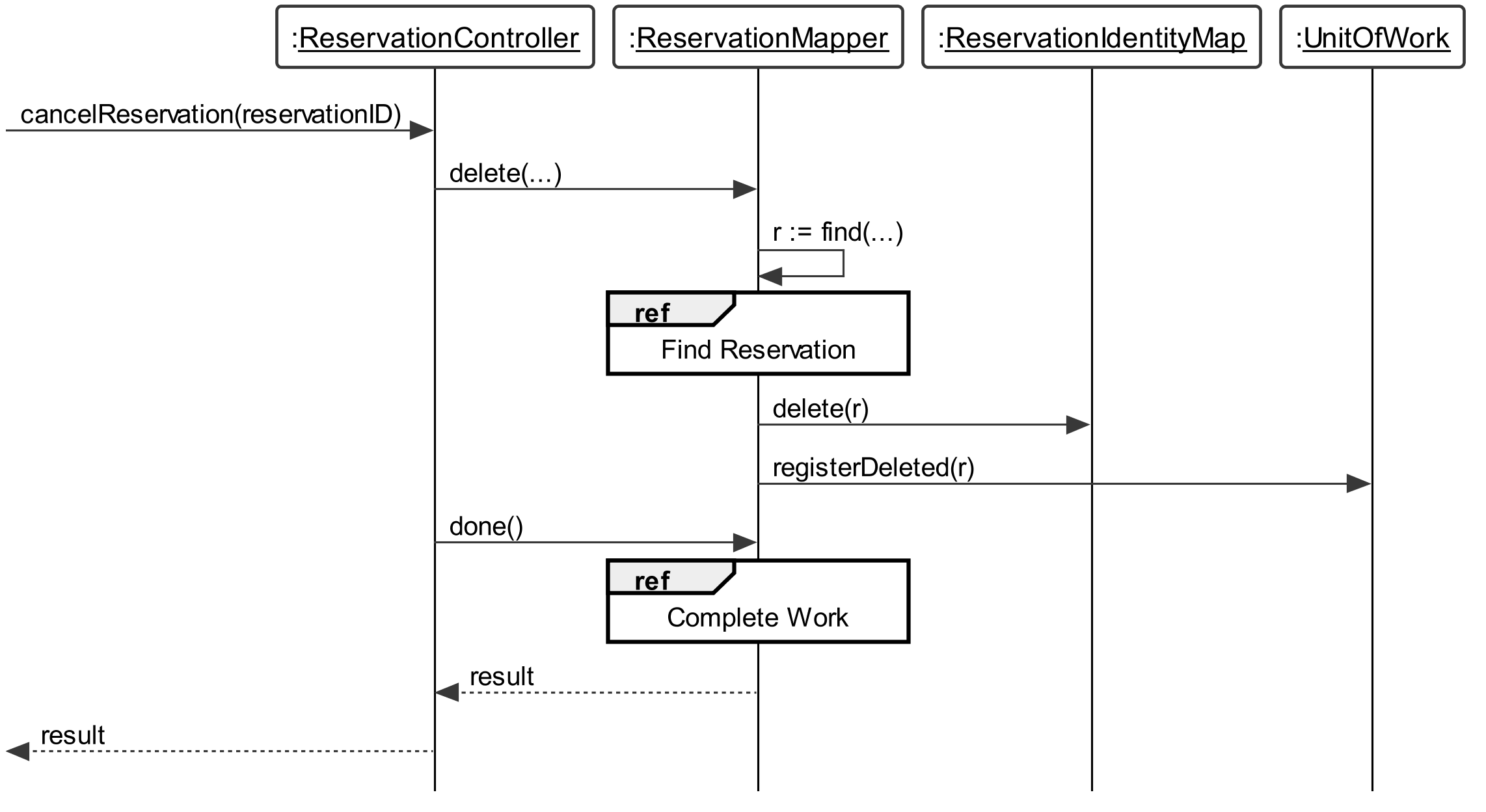


Figure : Sequence diagram for Cancel Reservation’s cancelReservation() system operation

### Referenced Sequence Diagrams

#### Find Active Reservations

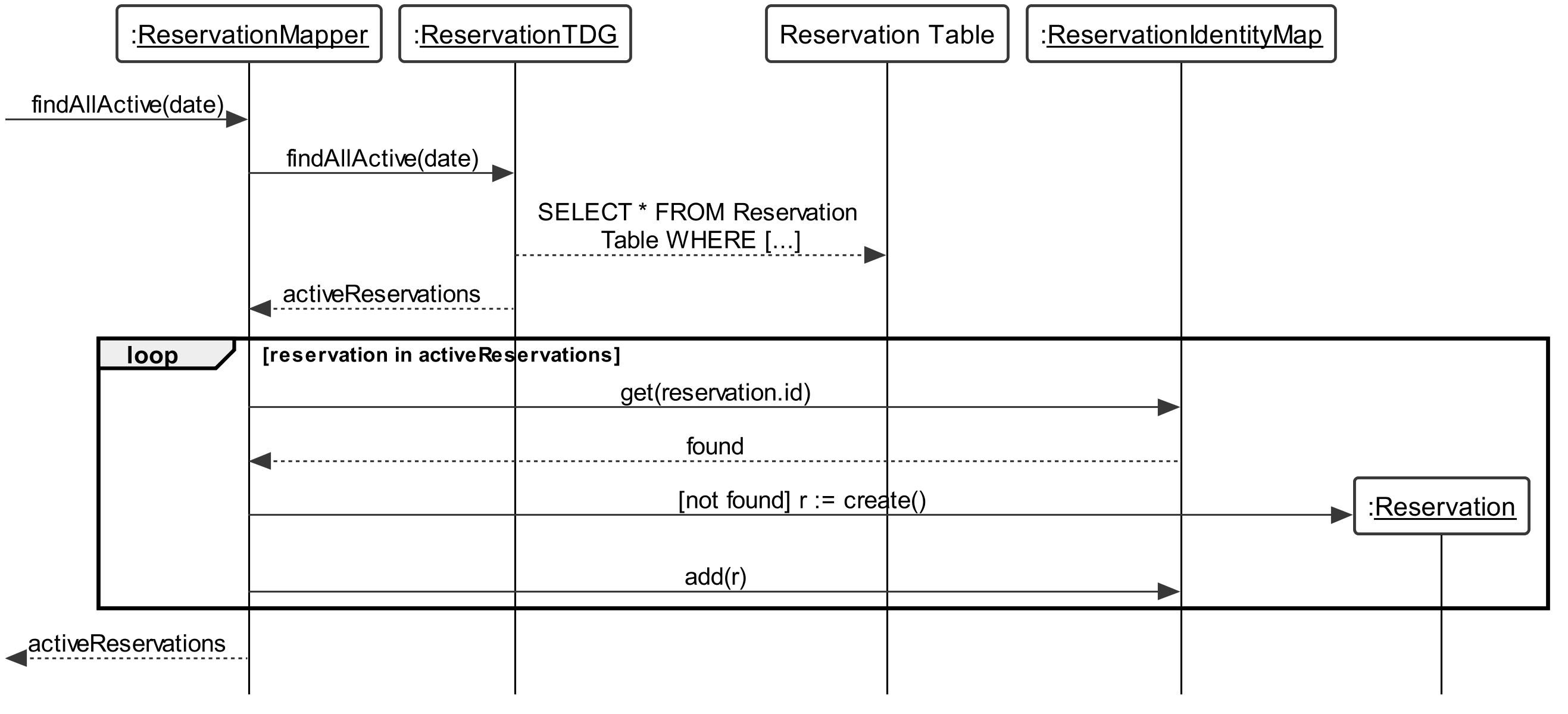


Figure : Referenced Sequence Diagram Find Active Reservations

#### Find Reservation

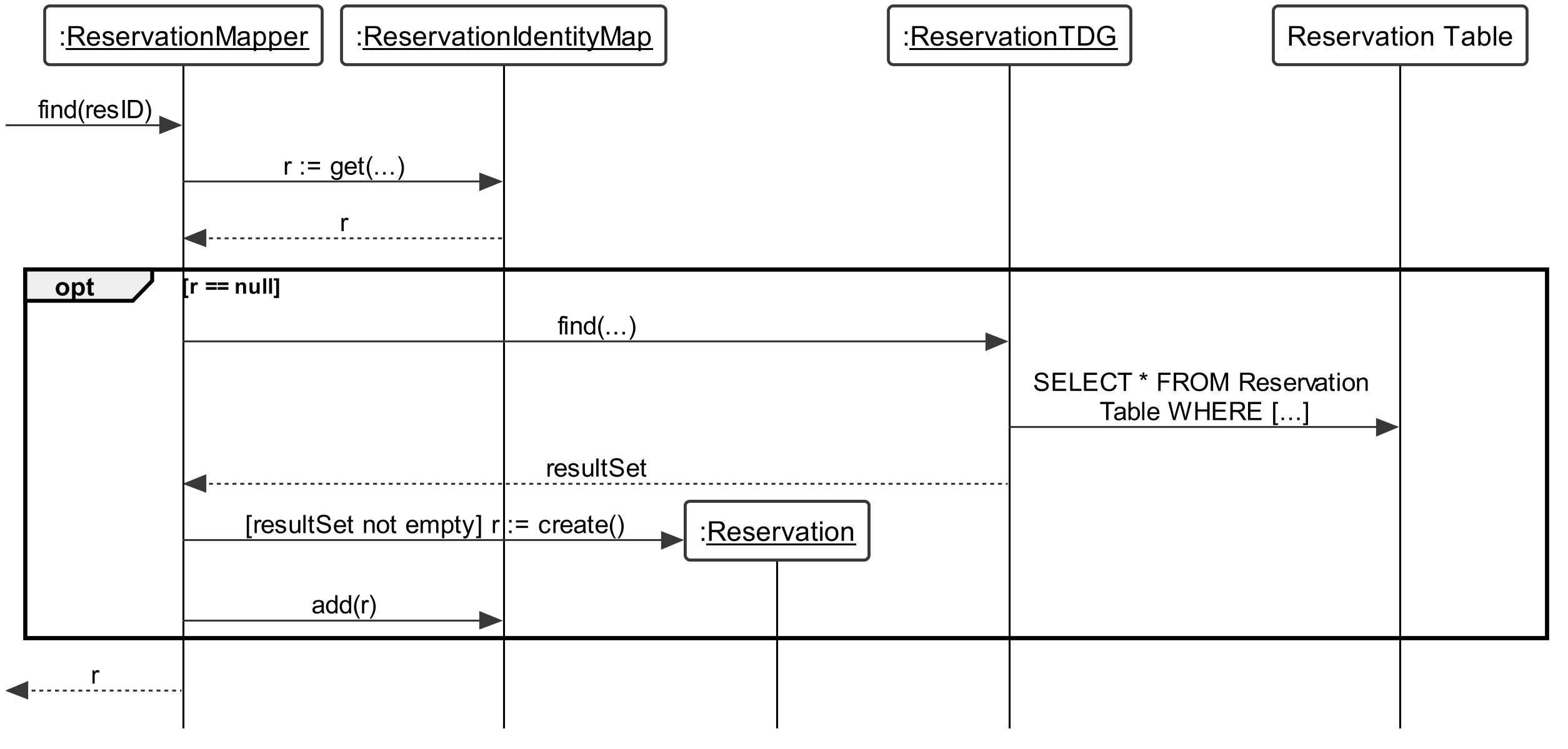


Figure : Referenced Sequence Diagram Find Reservation

#### Find Time Slot Reservations

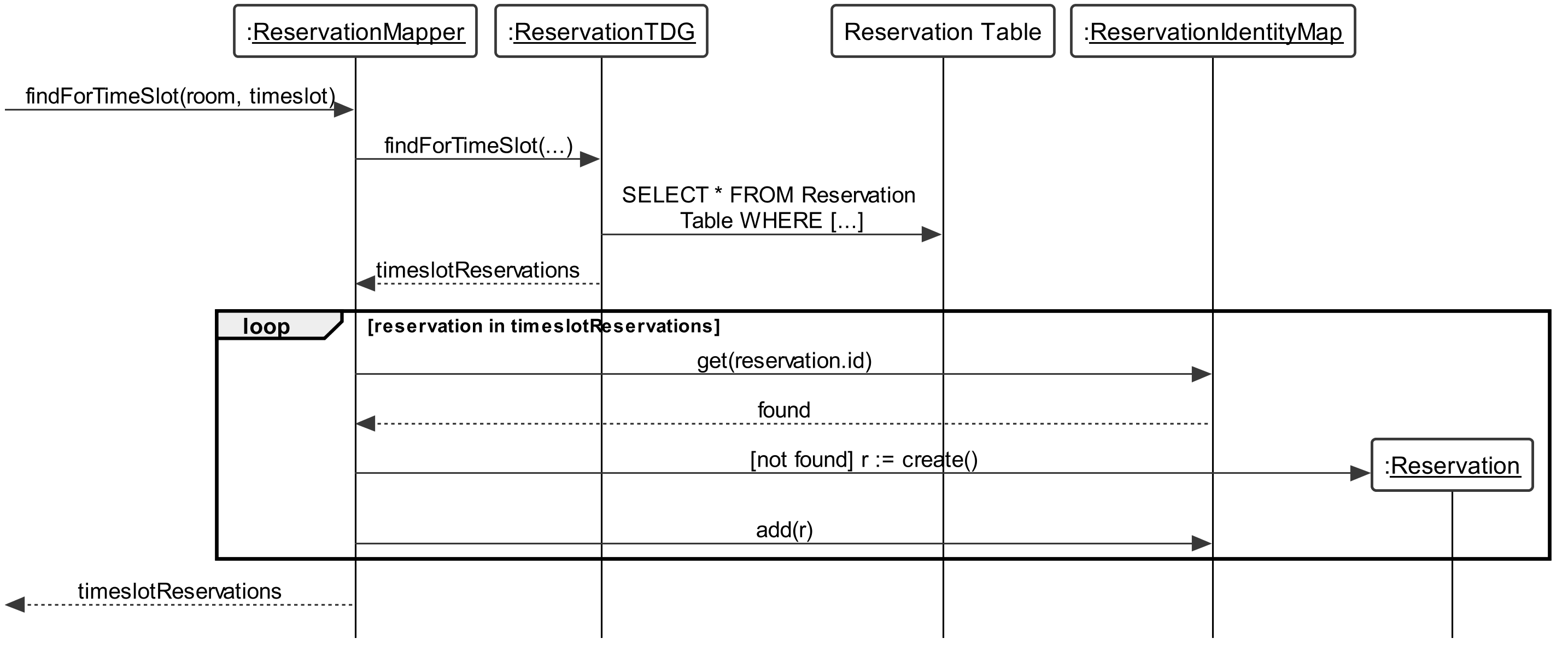


Figure : Referenced Sequence Diagram Find Time Slot Reservations

#### Find User Reservations

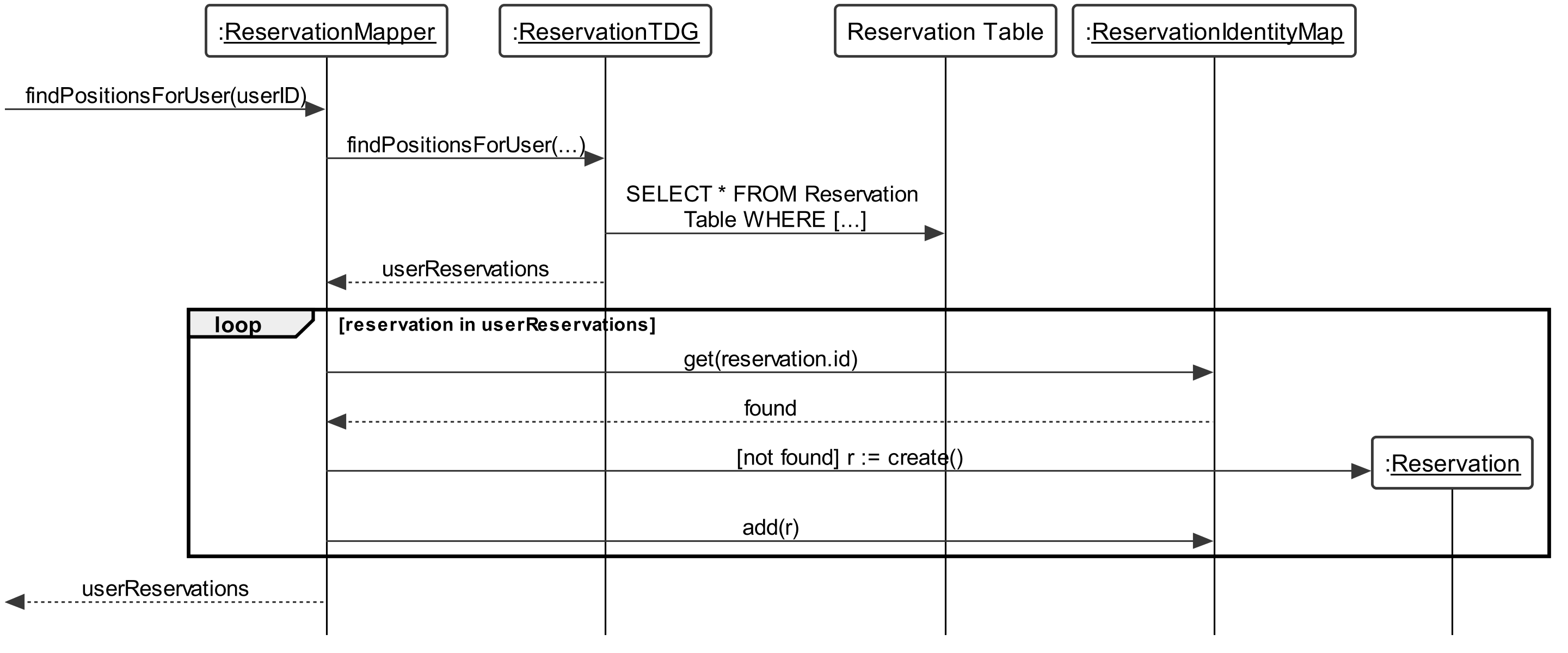


Figure : Referenced Sequence Diagram Find User Reservations

#### Complete Work

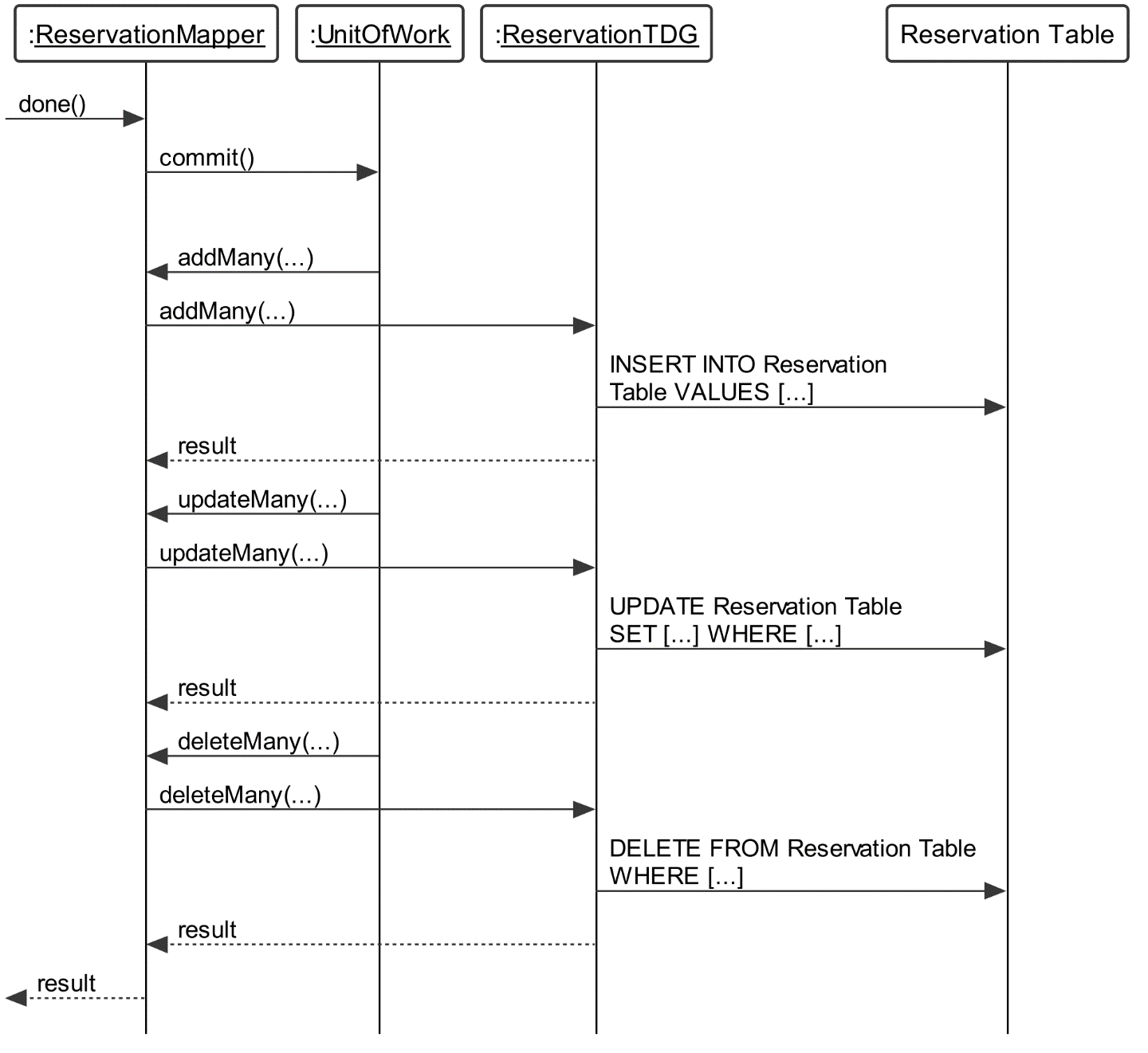


Figure : Referenced Sequence Diagram Complete Work

# Use Case View

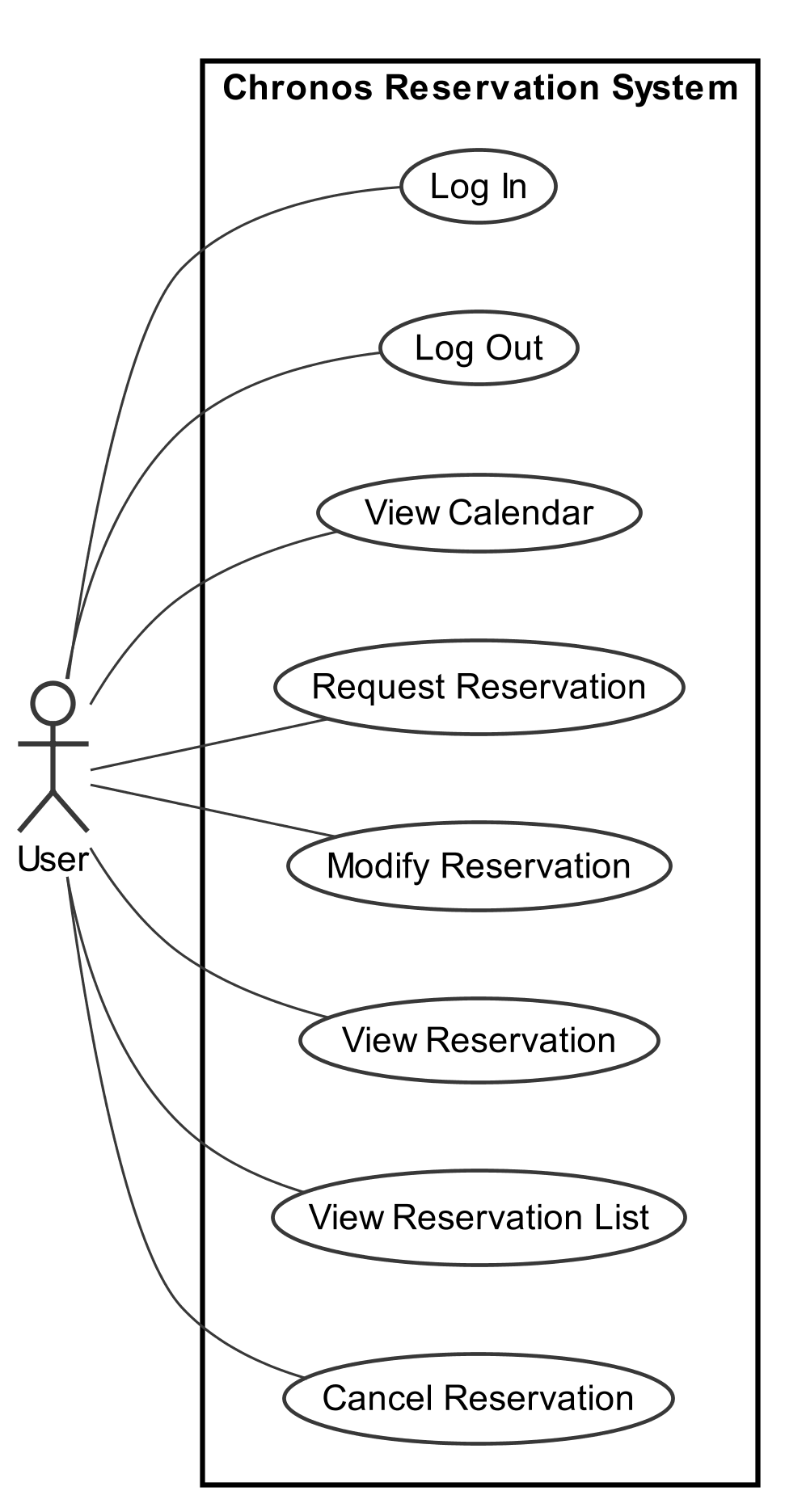


Figure : Use Case Diagram

# Data Model

For persisting data across system requests, a data model is used inside of a relational database management system. This database is accessed using the Table Data Gateway classes within the system. There are three tables in the database, corresponding to the three domain classes that require persistence in the system: Room, User and Reservation.

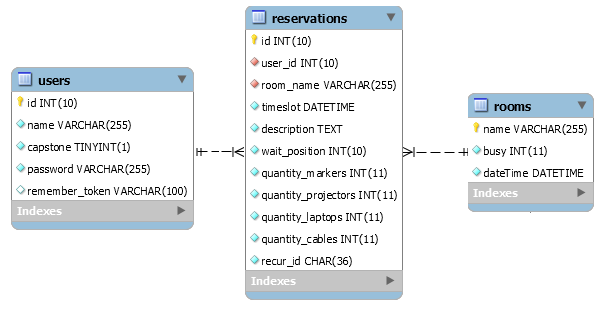


Figure . Entity-relationship diagram for the system database

The Room object maps to the *rooms* table, using the room name as its primary key. It contains a busy column to show which user is currently occupying the room as well as the timestamp (dateTime column) the original entered the room. The User object maps to the *users* table, where the user id is used as the primary key. The table also holds a name and a password field, both strings, for storing the respective attributes of the User domain class. It also contains a column named capstone to differentiate between regular students and capstone students.

Then, to represent the relationship between a Room and a User as shown in the class diagram, we use a *reservations* table to map the Reservation class along with foreign key columns to both rooms’ and users’ primary key columns. That way, we can represent the many-to-many relationship between *rooms* and *users* using a row in the *reservations* table. This table holds additional columns to represent the timeslot of type DATETIME, a reservation’s description of type TEXT, and a recur\_id to hold the 36-character long UUID string for linking recurring reservations together. A reservation is unique across a specific user\_id, room\_name and description, however, it is given an additional primary key as an automatically-incrementing integer id to represent insertion order. There is also a wait\_position to determine the waiting list position of a user. Lastly, quantity\_markers, quantity\_projectors, quantity\_laptops, quantity\_cables are used to store the amount of equipment associated to each reservation.

# Installation Manual

Installing Chronos on a server is a straightforward procedure and only requires about 1 to 2 hours of time.

## System Requirements

To install Chronos, you will need an Internet-connected server running a PHP-capable web server and database software. The recommended tested requirements are listed below.

* Apache 2.4 or nginx 1.6
  + Configured to serve files from a “**public**” directory under the installation directory
* PHP 7.0
  + Required extensions: OpenSSL, PDO, Mbstring, Tokenizer
* MySQL 5.6

For this installation manual, an Ubuntu Linux 15.10 installation will be assumed. We will also assume the following paths:

* Installation path is **/var/www/chronos**
* Document root of web server **/var/www/chronos/public**
* URL of website is **http://chronos.chrs.pw**
* Web server user is **www-data**
* MySQL database address is **localhost**

1. Please run the following commands to check your environment:

dev:~$ **php --version**

PHP 7.0.5-2+deb.sury.org~wily+1 (cli) ( NTS )

Copyright (c) 1997-2016 The PHP Group

Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies

with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2016, by Zend Technologies

dev:~$ **mysql -uroot -p -e "SELECT VERSION();"**

Enter password:

+-------------------------+

| VERSION() |

+-------------------------+

| 5.6.28-0ubuntu0.15.10.1 |

+-------------------------+

1. Ensure that your web server and PHP are enabled on your server by creating a sample “index.php” file in the document root and load the website in your web browser.

dev:~$ **cd /var/www/chronos/public**

dev:/var/www/chronos/public$ **echo "<?php phpinfo(); ?>" > index.php**

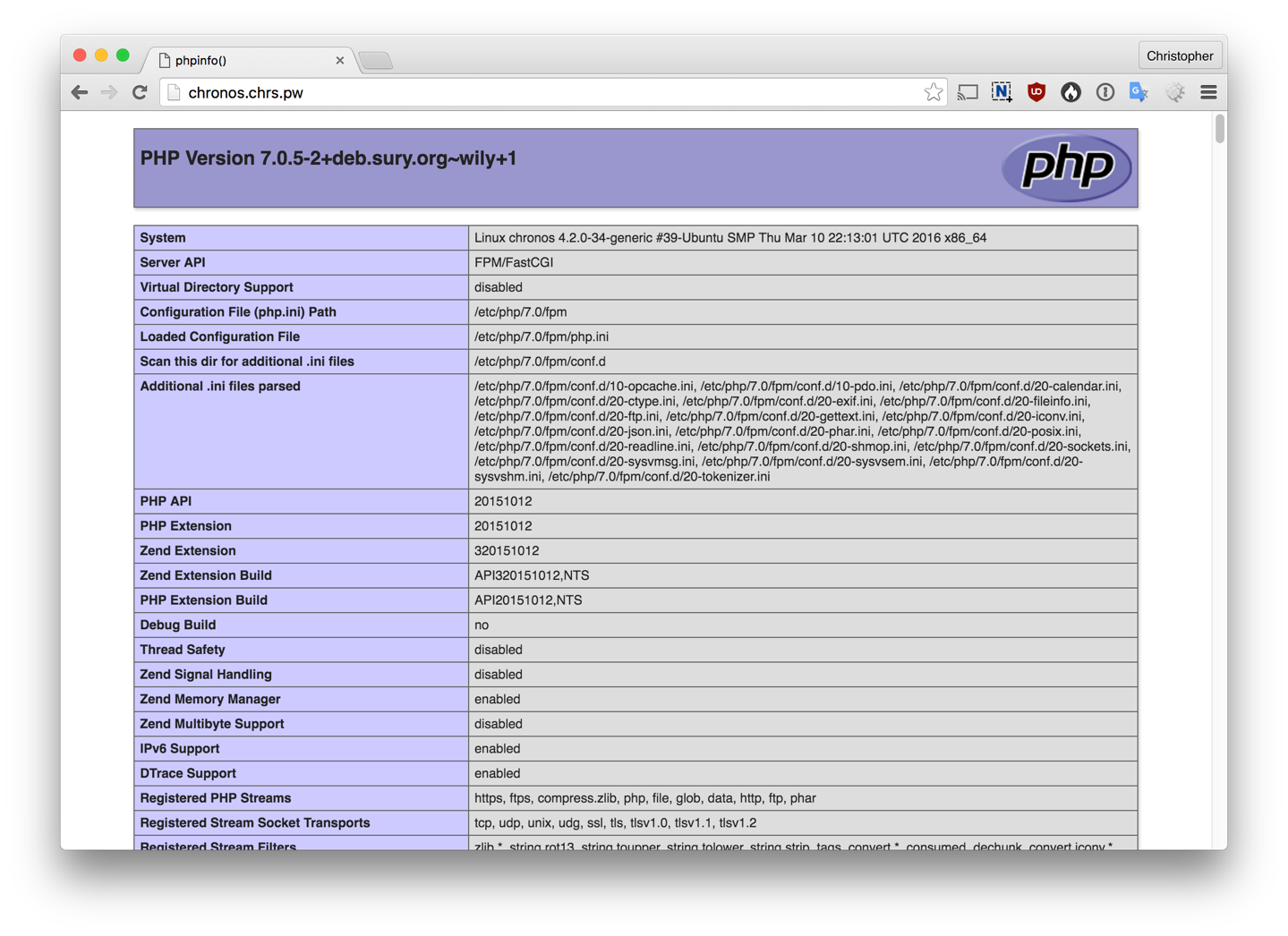


Figure : PHP test file output

## Downloading Chronos

Obtain a Chronos release by downloading it from the official repository, and extract it into your installation directory.

1. Change your directory to the installation path:

dev:~$ **cd /var/www/chronos**

1. Download the Chronos release:

dev:/var/www/chronos$ **wget \**

[**https://github.com/Shmeve/soen343-emu/releases/download/v0.0.1/chronos.tar.gz**](https://github.com/Shmeve/soen343-emu/releases/download/v0.0.1/chronos.tar.gz)

1. Extract the Chronos system files to the installation path:

dev:/var/www/chronos$ **tar --strip-components=1 -zxvf chronos.tar.gz**

## Setting up the Database

After extracting the Chronos system files, it is necessary to create a database and a user for Chronos.

1. Log into your MySQL database as an administrative user, and issue the following commands:

dev:~$ **mysql -uroot -p**

Enter password:

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 3

Server version: 5.6.28-0ubuntu0.15.10.1 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective

owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> **CREATE DATABASE chronos;**

Query OK, 1 row affected (0.00 sec)

mysql> **CREATE USER 'chronos'@'localhost' IDENTIFIED BY 'password';**

Query OK, 0 rows affected (0.00 sec)

mysql> **GRANT ALL PRIVILEGES ON chronos.\* TO 'chronos'@'localhost';**

Query OK, 0 rows affected (0.00 sec)

mysql> **FLUSH PRIVILEGES;**

Query OK, 0 rows affected (0.00 sec)

This has created a database “**chronos**” and a user “**chronos**” with password “**password**” with privileges on that database, ready for use with Chronos.

Now, you must configure Chronos with this database information.

1. First, copy the example “.env.example” configuration file:

dev:/var/www/chronos$ **cp .env.example .env**

1. Edit the “.env” file and change the DB variables to match the appropriate values:

DB\_HOST=localhost

DB\_DATABASE=**chronos**

DB\_USERNAME=**chronos**

DB\_PASSWORD=**chronos**

## Install Chronos

Most of the Chronos installation is done via Composer, PHP’s package manager. An install script is included to accelerate the process, which will create all necessary database tables and install all dependencies.

1. Execute install.sh:

dev@:/var/www/chronos$ **./install.sh**

All settings correct for using Composer

Downloading 1.2.0...

Composer successfully installed to: /var/www/chronos/composer.phar

Use it: php composer.phar

Application is now down.

Loading composer repositories with package information

Installing dependencies from lock file

- Installing symfony/finder (v3.1.7)

Loading from cache

[…]

Generating autoload files

> php artisan clear-compiled

> php artisan optimize

Generating optimized class loader

Compiling common classes

Migration table created successfully.

Migrated: 2016\_11\_16\_000000\_create\_users\_table

Migrated: 2016\_11\_17\_045934\_create\_rooms\_table

Migrated: 2016\_11\_17\_051557\_create\_reservations\_table

Generating optimized class loader

Compiling common classes

Application cache cleared!

Application is now live.

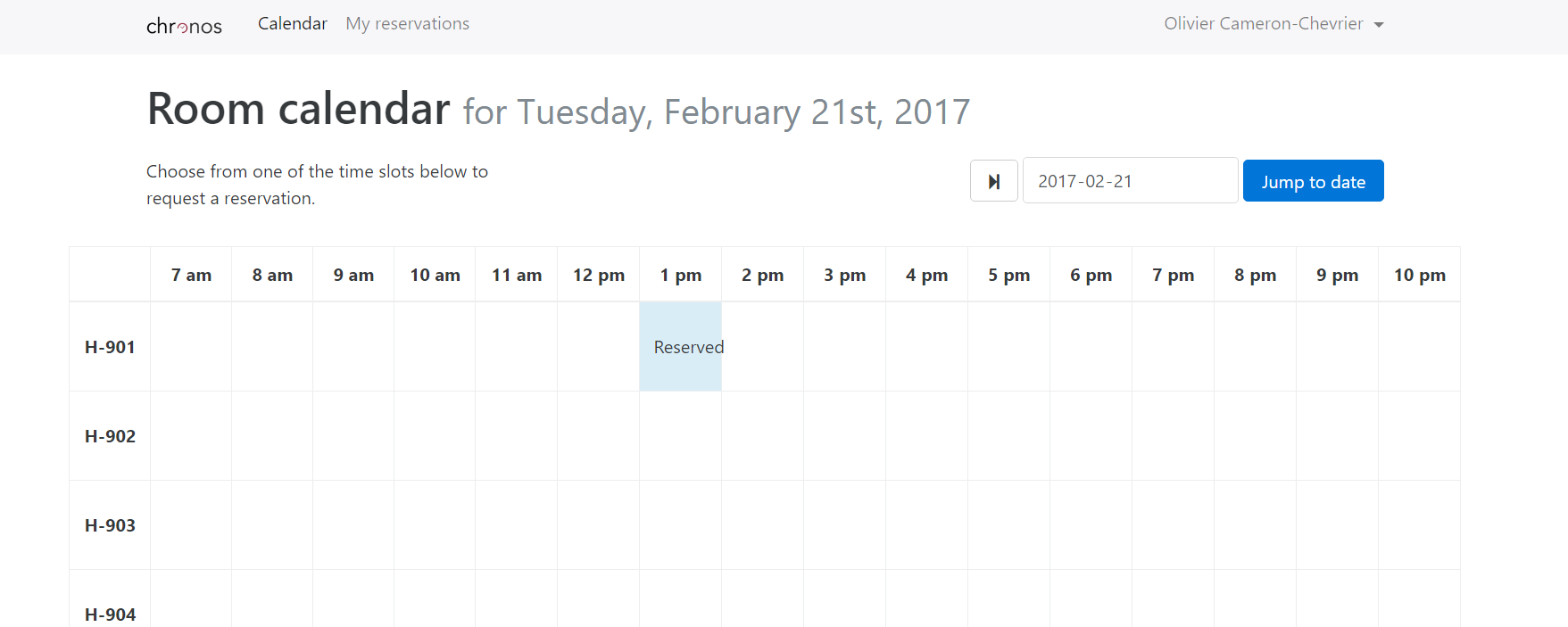
**Appendix – Modifications Log**

# Modification Log

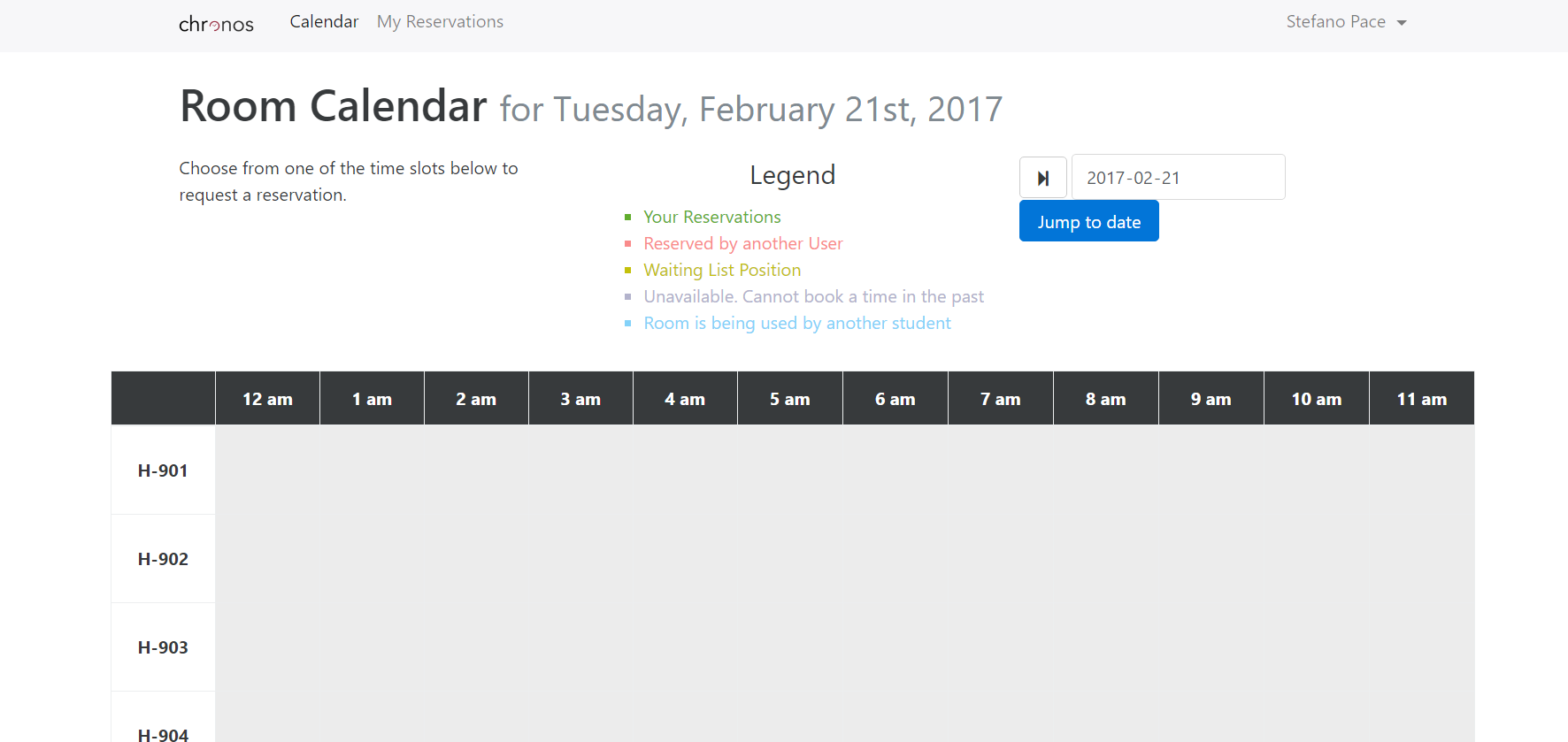
After reading both the SRS and SAD documentation, the team created a testing plan to ensure that all functionalities were working as stated by the previous team. After all testing was completed, the team noticed many inconsistencies as well as bugs which would have to be addressed before the newly requested functionality could be incorporated. The sections below will detail how these problems were solved.

## Styling in the Calendar

The team noticed that there was an issue in the how the Calendar presented information to the users. Depending on the size of your screen, when displaying the words “Reserved, Taken, Waiting #” to indicate the respective scenarios, sometimes the words would overlap with the boxes assigned. See below as an example:



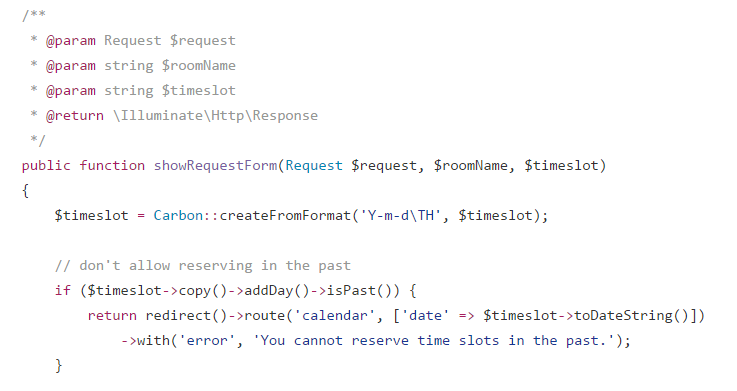
Thus, we decided that we would replace the wording in the boxes with a color legend to avoid the problem with the word overlapping. This is the improved result:



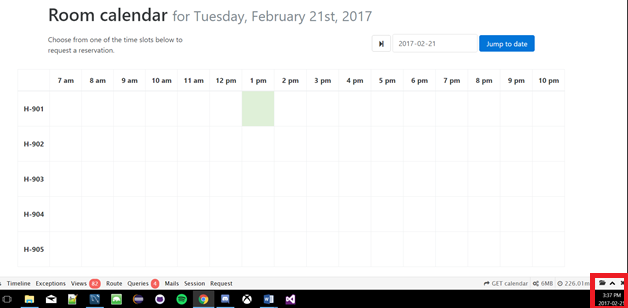
## Reservation Before Current Time Functionality

During the inspection of the code, it was noticed that a function had been put in place to verify if a user attempts to create a Reservation on a timeslot that has passed. When this check returned true, the user would receive a message informing them that the system does not allow this functionality. Upon testing this, the team came to the realization that the code did not in fact work as intended. See below as an example.

\*\*This can be found in the original Repo (<https://github.com/Shmeve/soen343-emu>), within the method “showRequestForm” beginning on line 114 of the ReservationController.php file.



This was the output following a test:

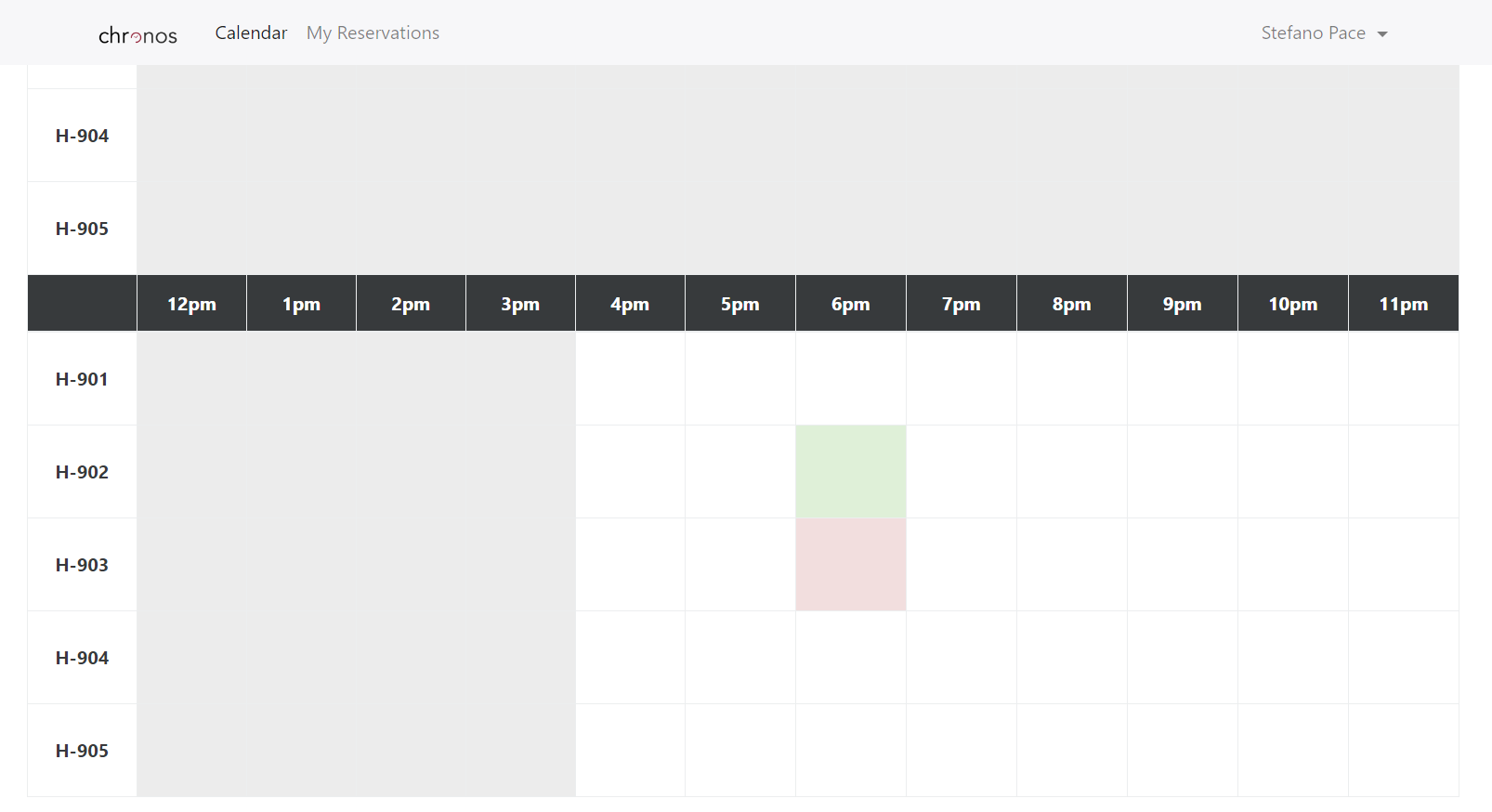


\*\*\*\*Please note the time at the bottom right corner of the photo ( 3:37PM)

Thus, the team refactored the Calendar functions to encompass the following logic:

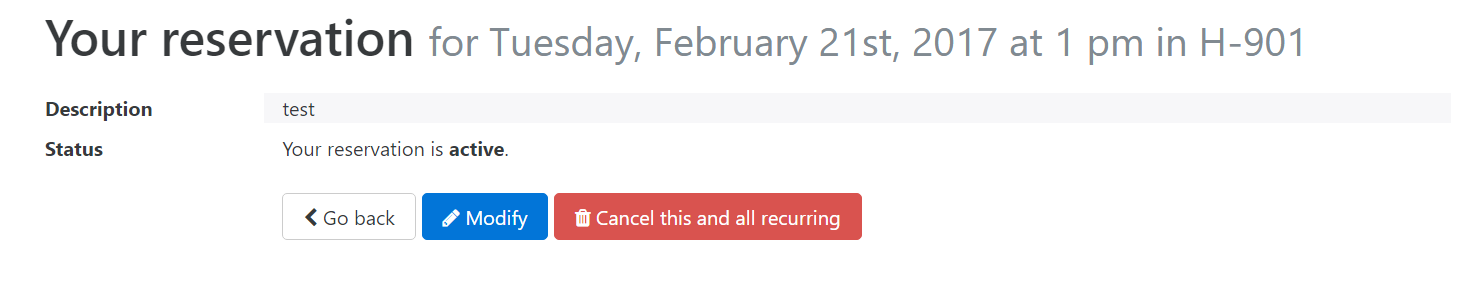
* If the user is trying to make a Reservation today, then only populate clickable timeslots on future times. (I.E if it is 3:25pm, only make 4pm and later clickable).
* Disable the user’s ability to view Reservation before the current time (so they cannot be modified)
* If the user is trying to make a Reservation in the future (I.E not today), then populate all clickable timeslots normally.

See the example below: (As the Legend previously indicated, grey signifies an unavailable time in the past, green signifies your active reservation, and red signifies a reservation by someone else).

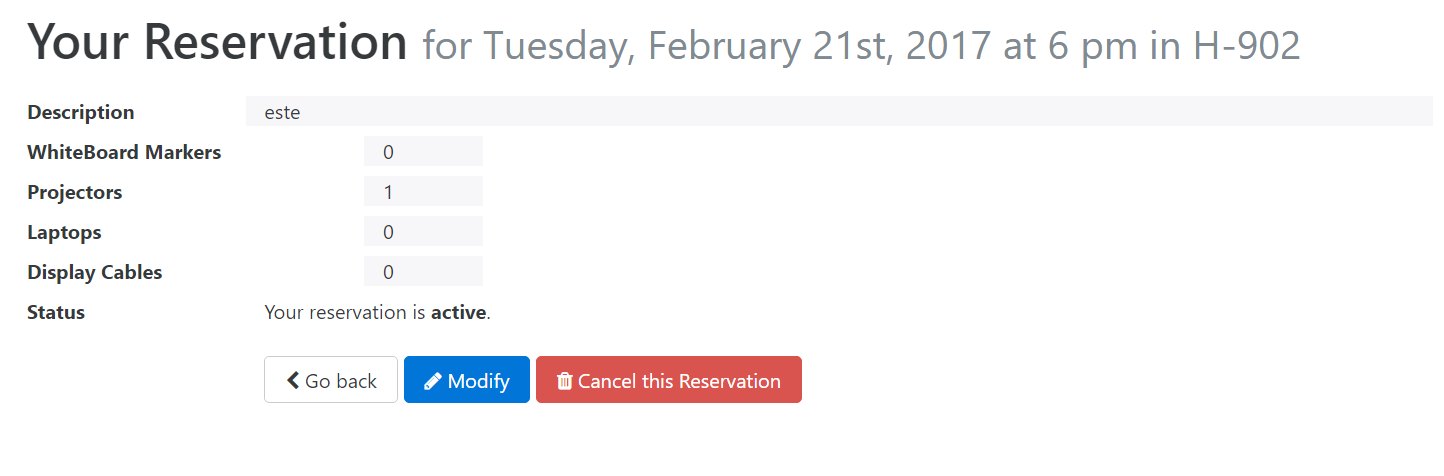


## Deleting Individual Reservations

The original code for the deletion of reservations previously grouped all “Recurring” reservations together using a generated UUID. When the deletion method would be invoked, instead of deleting the individual reservation, the system would delete the currently selected one as well as all the ones associated through the recursion in the future. Although warned a button labelled “Cancel this and all recurring”, this proved to be highly impractical as the system should allow individual deletions. Please see the example below:

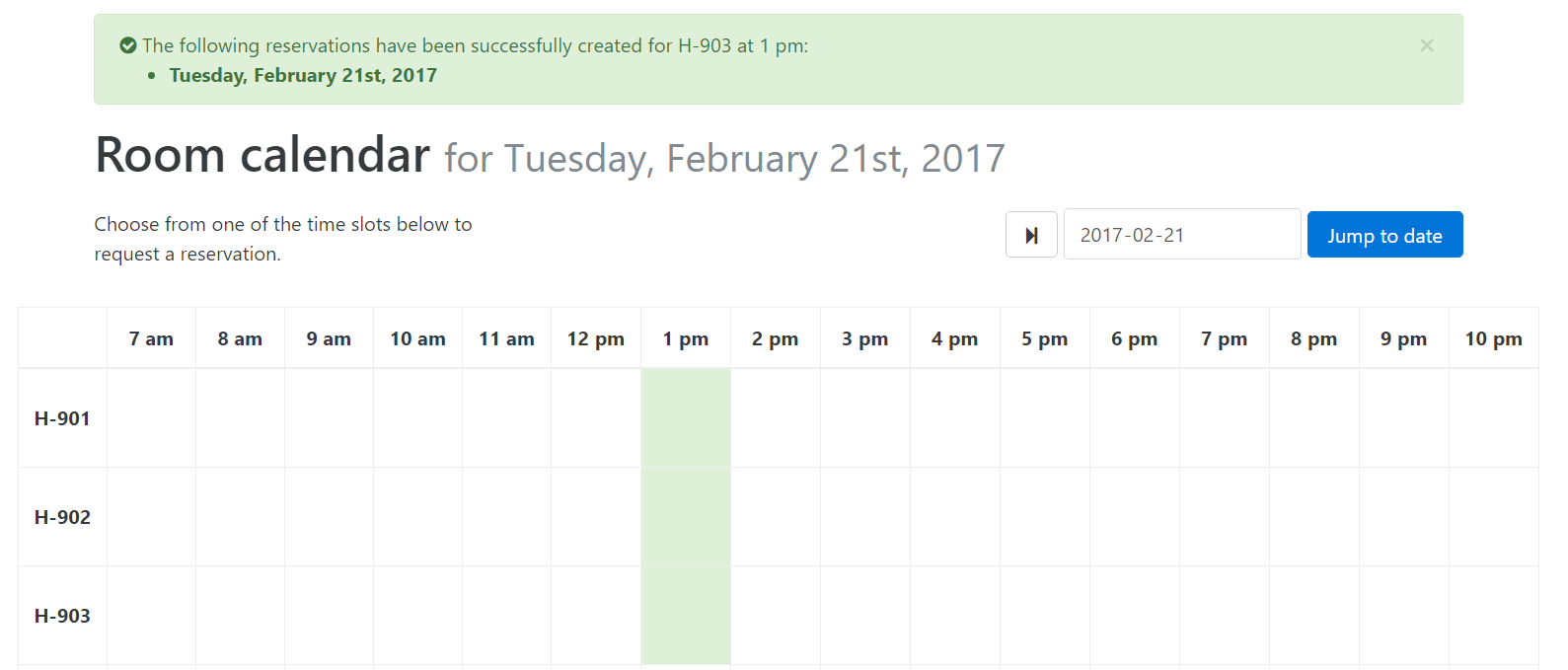


Instead of using the generated UUID as the criteria for deletion from the database, the team implemented the primary keys as the criteria for deletion. Thus, the individual reservations were enabled:

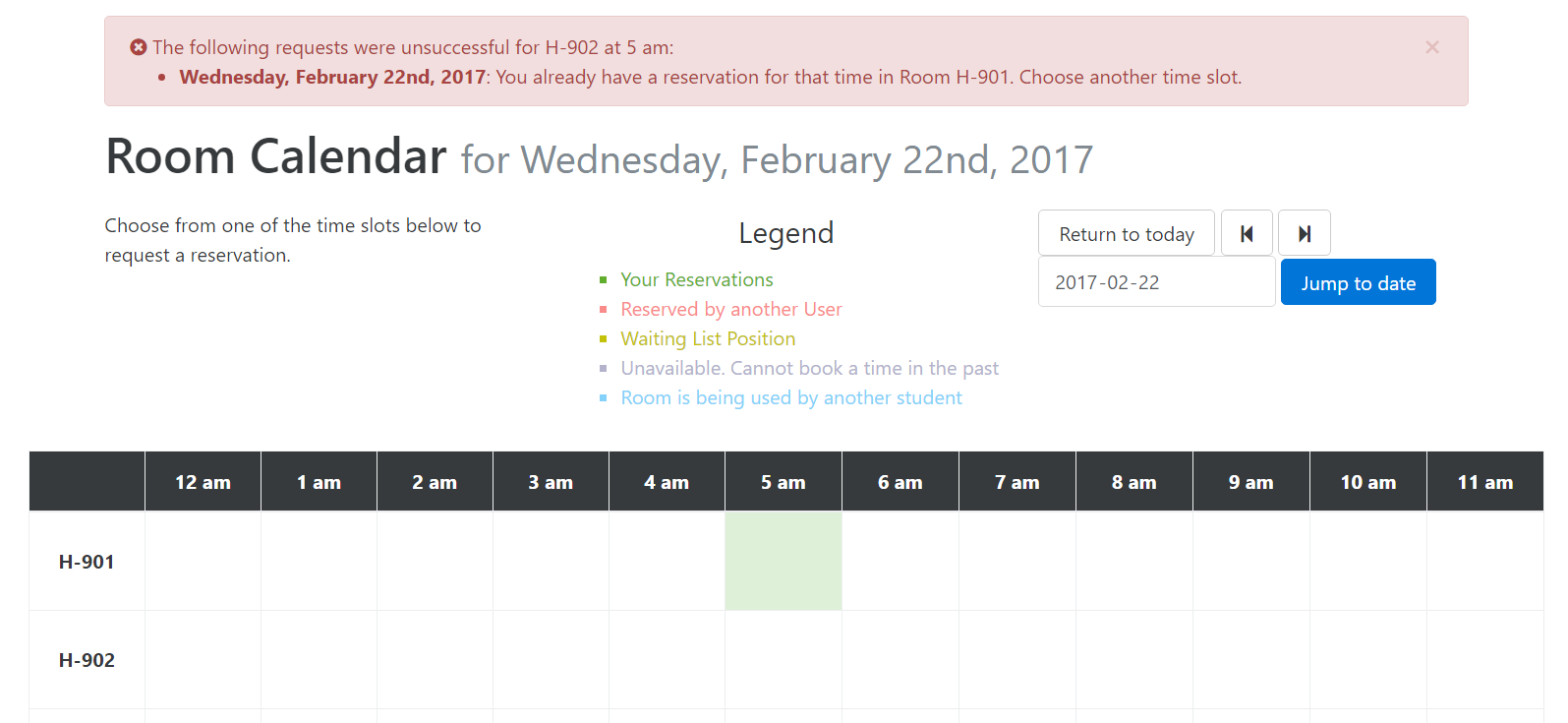


## Overlapping Reservations in Different Rooms

While creating test reservations for the system, the team noticed that users could have active overlapping reservations. This means that a single user can have reservations for a certain Timeslot in 1 or more rooms. See below:



This is a problem, because one user can control all the rooms for a certain timeslot. Therefore, the team incorporated a check to prevent this scenario. This is the result:



## Waitlist Functionality

The Waitlist Functionality was the component in the system that had the most problems. Because these problems are numerous, this section will be split into various subsections to provide details in a more organized fashion.

### Waitlist position not an attribute stored in the Database

The position was something which was dynamically created upon each request to view either the calendar or the details of an existing reservation. This was inefficient in many ways, as the waitlist position of any user for a reservation in the system could not be checked through the database. As administrators, the team would have no way of knowing who was on the list and in what order. Therefore, if reports of errors would arise, there would be very limited ways to troubleshoot. Not to mention that it would be impossible to sign in to people’s account to obtain such information because all the passwords were protected using hash functions (and it would be unprofessional as well as unethical to log into someone else’s account).

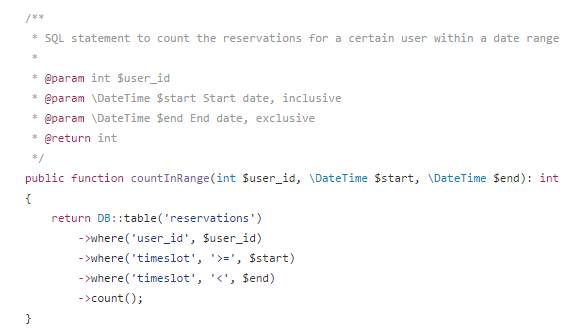
This would pose a problem when implementing the new functionality of Capstone Priority and Equipment. When necessary we would need to shift people down or up in the list. If this had to be dynamically calculated on the fly (instead of being calculated once and just having the position return for any subcomponent which required it), a large burden would be put on the server from all the database pulls that would be required, especially as the user base would increase.

Thus, the team add a column in the database which signifies the position they are in the waitlist. A “0” signifies they are the active reservation, where as any number above is their waiting position. The maximum waiting position implemented through the system is 3.

### Waitlisted Reservation counted in Check for Maximum allowed per Week

As stated in the original 343 requirements, “A user may create only up to some maximum

allowable number of reservations per week.” As such the previous team implemented a check to ensure that if a user requested to make a reservation, that they had not surpassed this limit for the given week. However, this check did not work properly as it included Waitlisted Reservation in the count. This is a problem because if all the reservations a user creates in a week are on the waitlist and they reach the maximum, then they technically do not have any active reservations and therefore should still be able to a make more requests.



To fix this issue, the team simply modified the check for the maximum allowed per week to only count those reservations which were labelled as active by adding an extra where condition. This way a user can technically have infinitely many reservations on waitlists. This leads to the next point.

### Waitlisted Reservation Deletion when reaching Maximum for the week

Now that waitlisted reservations are not counted in the maximum for the week, when the user reaches the maximum for the week, all waitlisted reservations for that same week must be deleted. This never had to be previously implemented because a user would never have could have infinitely many reservations on waitlists. The team created a method to handle such events.

### Waitlisted Reservation Deletion for overlapping Rooms

As stated in the original 343 requirements, “A user who wants to reserve a time slot for a room that is already booked at that time can be placed on a waiting list and be able to obtain the room upon cancellation of the current reservation. Upon obtaining such reservation, the user is removed from any and all other waiting lists on any other room that has been reserved over the same time slot.”

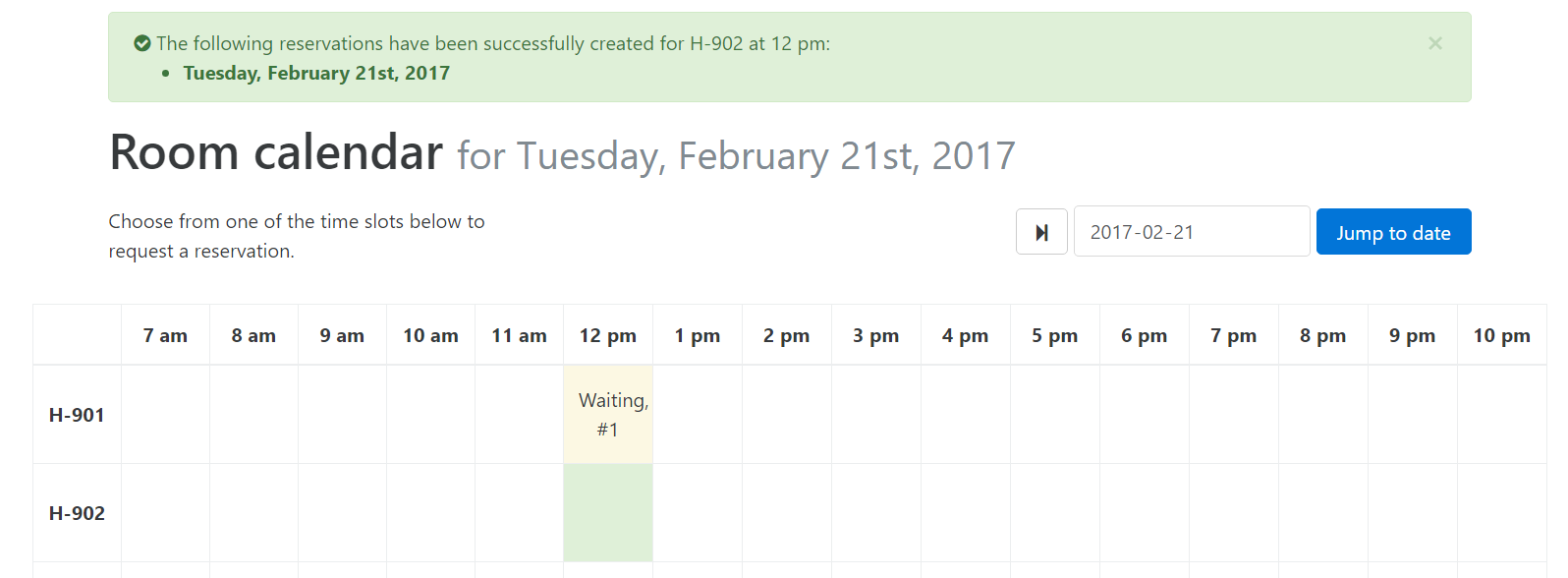
For further details on what the previous team said about the Waiting List, please consult their SRS document, specifically section 1.2.3, labelled “Waiting List”. Here is the excerpt to be discussed by this appendix:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*The application does so by removing the former user from the reservation slot, adding the new user to the slot and removing this new user from all waiting lists for any other rooms reserved over the same time slot. This means that if a user needs a room for a given time slot and all of them are currently occupied, the user can reserve multiple rooms for the same time and will be placed on waiting lists for all of them. The user will only be taken off these waiting lists if they obtain one of the reservations, cancel a particular reservation or if the time reserved for has passed.*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This feature was tested by the team but the results could not be duplicated. See below:



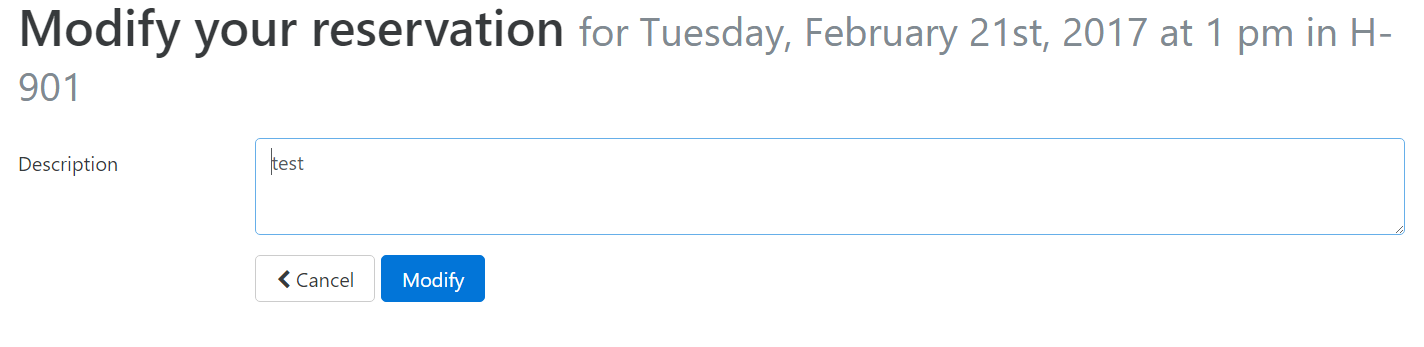
Thus, modifications were made to the method by the team and now all overlapping waiting list reservations are deleted.

### Waitlist position updates upon Cancelation of Reservation

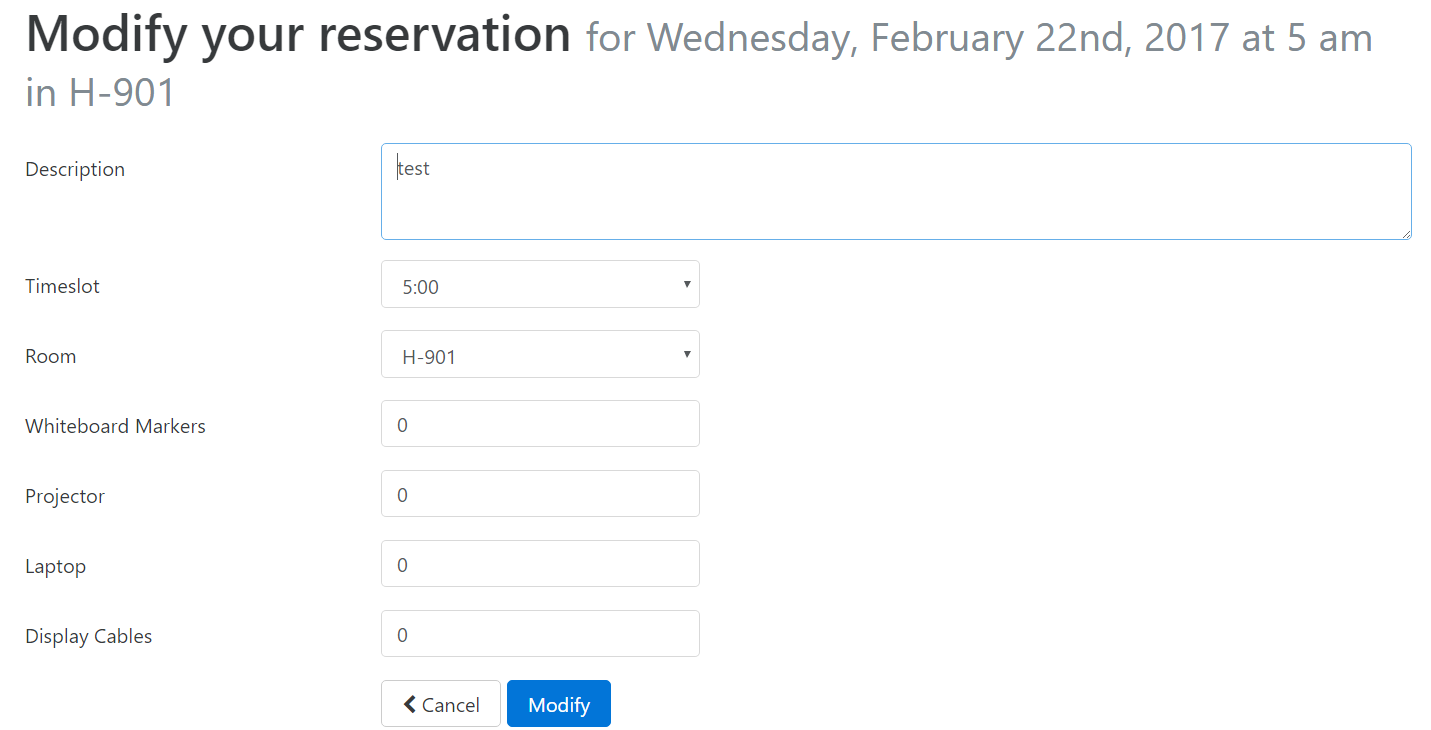
As waitlist positions were not previously stored in the database, there was no need to update them upon the cancelation of a reservation (as they were dynamically calculated when navigating to the page). As they are now stored in the database, a function was implemented by the team to update the database (I.E move all relevant users down in the list) when either the active user or waitlisted user cancels their reservation.

## Modification Functionality

During testing, it was discovered that when attempting to modify any reservation (active or waitlisted) the users were only able to modify the description associated. Although this technically abides by the original specifications of modification (as no specification for what type of modification was included), this does not accurately represent a real-world scenario. A user should be able to change both the room and timeslot that the Reservation takes place in. See the example below:

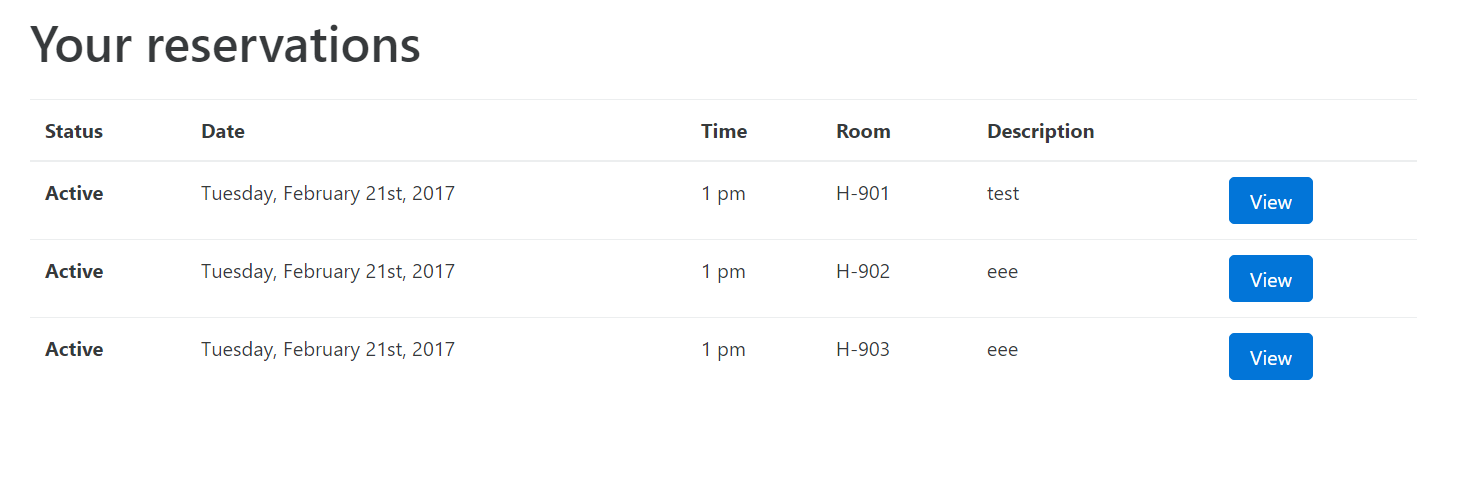


After modifying the form and respective functions, this was the result:

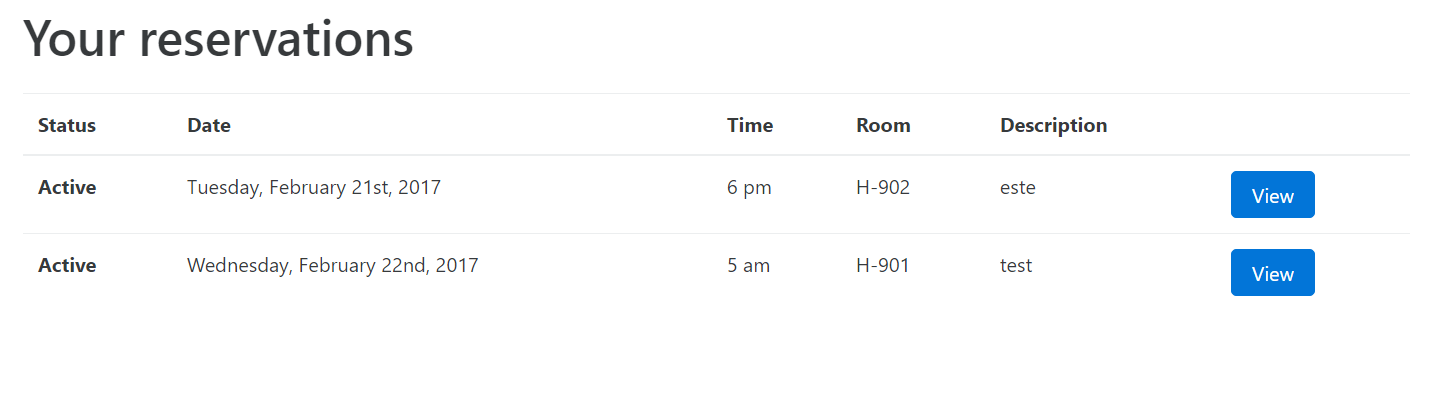


## My Reservations View Functionality

In the previous iteration of the system, when the user opted to view their list of reservations, they could view reservations that had previously passed. Through this menu, they could then modify the reservation and change details. Although this proved to be useless to the user as the resulting reservations would not be active, it increased the amount of connections/requests the server would need to handle.



By removing the option to view reservations that already passed, the team eliminated any possibility of these extra connections/requests taking place, therefore reducing the load on the server.



## Mutual Exclusion and Room Locking

For future details on what the previous team said about Mutual Exclusion and Room locking, please consult their SRS document, specifically section 1.2.2, labelled “Mutual Exclusion”. Here is the excerpt to be discussed by this appendix:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*This requirement has been modified from its original specification for reasons of technical feasibility. Due to the complications of modern web server technologies, there is no straightforward way to implement mutual exclusion for page load connections. A user could select a time slot (thereby “locking” it), and simply close their browser window. As server connections are closed immediately after page load, the server is stuck with a locked time slot with no knowledge of the client having abandoned it. This would require implementing expirations for the locks, which is outside of the scope of this application. Additionally, simply halting a user’s incoming connection to view a presently locked room, without implementing a technology such as socket-driven communication, would cause the user’s browser to hang indefinitely and provide a negative experience.*

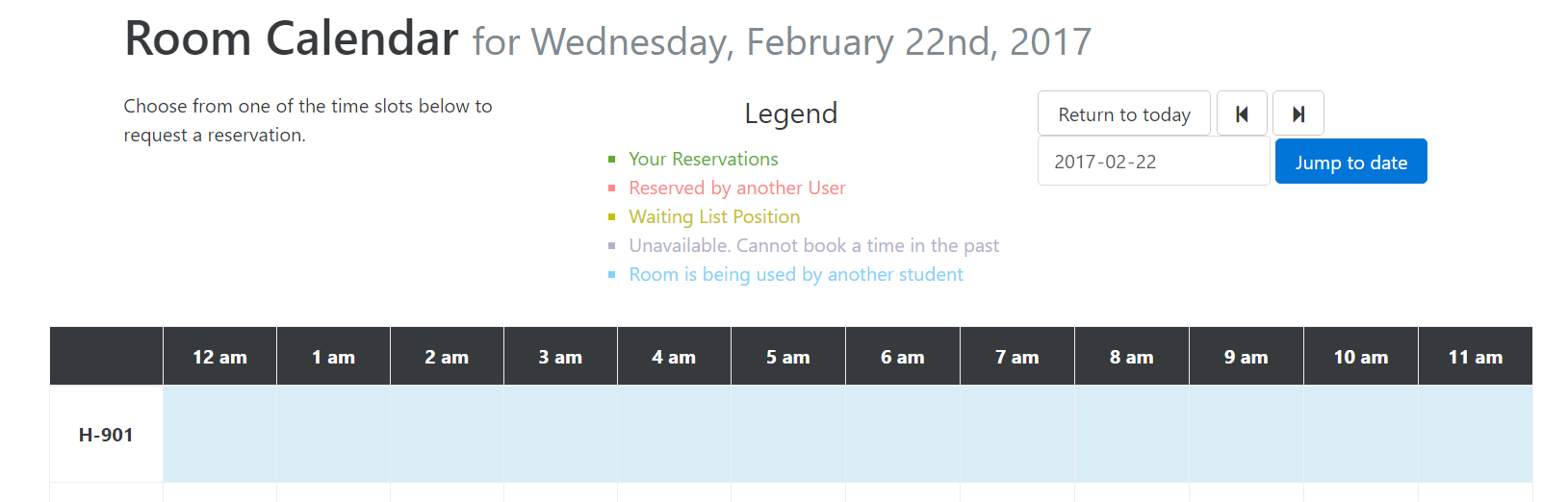
*The requirement will instead be partially satisfied by gracefully displaying an error message telling the user that the room was already reserved in the case of an attempted reservation of an already booked room, instead of locking a certain room from all readers.*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

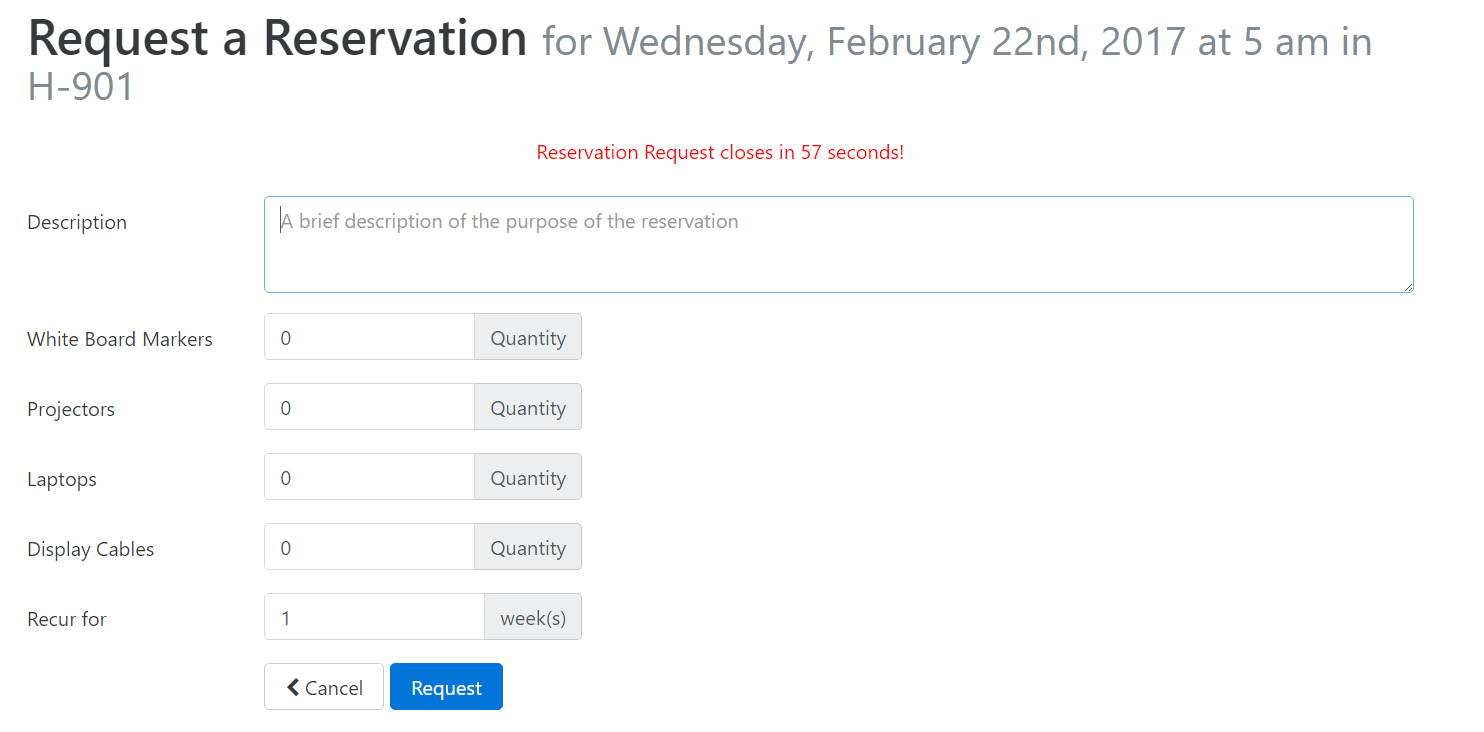
Although much of what is said holds true and is to be addressed specifically in the new requirements, the 343 requirements clearly stated that “For any given room, the operations to create, to modify and to cancel a reservation are all \write" operations, whereas view is a read operation. Writers operate in self-exclusion, i.e. only one writer can be active at a time. If a client wishes to write and the resource is not available (another client is writing on the resource), then the client must wait, until the resource becomes available. Writers and readers operate in mutual exclusion. The system may allow multiple readers at a time. The system must provide safety, liveness and fairness. “

The previous iteration of the system attempted to handle this by implementing a race condition (I.E whichever user would get the reservation connection to the server first would win, and the others would obtain an error.) However, this is inefficient because not only does it burden the server by handling more requests than necessary, but it is also based on varying connection speeds and random prioritization by the server’s request queue.

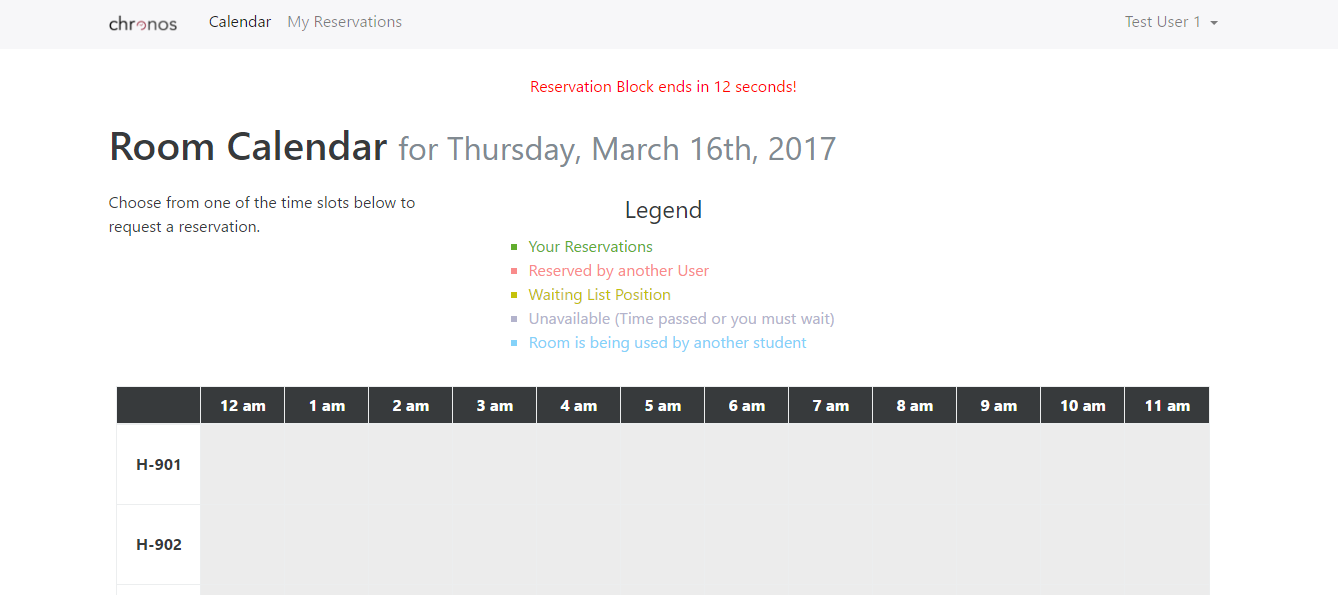
Thus, the team implemented a lock on any room that a user is currently in. This means that writers now operate in self-exclusion and only one user can attempt to write to a timeslot in a room at given time. See the results:



As well, to prevent a user from infinitely keeping a room locked, a 60 second timer was implemented which, at completion, will unlock the room regardless of the status of the user’s reservation attempt. See the result below:



Lastly, a timer was implemented to prevent the user from re-entering the room an unlimited amount of times



It is understandable that the team deemed a browser crash out of scope, but these 2 features were essential some form of guarantee towards safety, liveness, and fairness.

## Closing the Window logs out the User

For future details on what the previous team said about Logging out the user when Closing the Windows, please consult their SRS document, specifically section 2.2, labelled “Product Functions”. Here is the excerpt to be discussed by this appendix:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*After completing every task, the user has the possibility to log out or to close the window which in this case will automatically log the user out.*

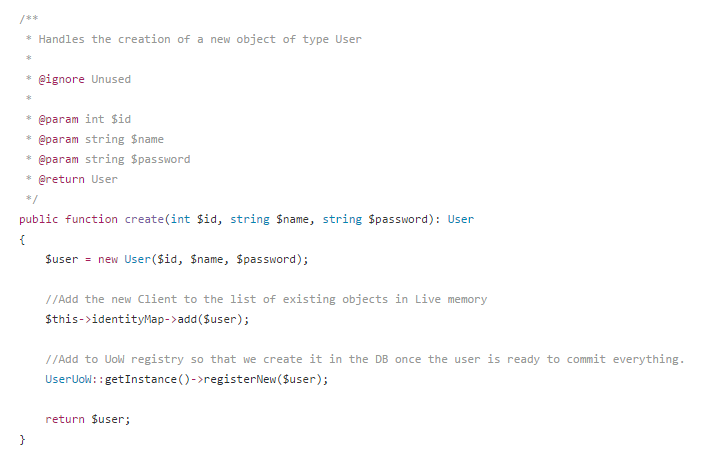
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

After attempting to replicate the result specified, the team was unsuccessful. Upon further investigation, it was determined the function implemented in the previous iteration did not work as intended, so it was scoped out.

The team believed that it was advantageous for the user to be automatically signed in after navigating back to the webpage, just like many popular websites offer (I.E Google, Yahoo, GitHub and Reddit).

## Bad code smells

Upon verifying the code in the previous team’s repo, the team noticed that many of the functions written were not actually used. Thus, the team removed all functions that were commented as “@unused” to eliminate all bad code smells and to improve overall organization. An example is presented below:



The following is a list of all methods that were deleted from the corresponding files because they were unused:

1. UserMapper.php
   1. Create
   2. Set
   3. Delete
   4. Done
   5. addMany
   6. updateMany
   7. deleteMany
2. UserUoW.php (Entire File)
3. UserTDG.php
   1. addMany
   2. updateMany
   3. deleteMany
   4. create
   5. update
   6. remove
4. User.php
   1. setName
   2. getPassword