CMP-5012B
SOFTWARE ENGINEERING GROUP PROJECT STAGE 2

Group 25

Event Booking System

Group members:

Martin Siddons – 100225776
Chris Sutton – 006214363
Lena Almatrodi – 100255110
Aurelie Sing Fat – 100249917

Date: 09 June 2020

Table of Contents

1. Stage 1 Feedback Address (S2.1 in mark sheet) 4

1.1. Table of Contents for Feedback 4

1.2. Revised Feature Matrix 7

1.3. Revised Textual Use Case 9

1.4. Revised Natural Language Modelling 11

1.4.1. Events Coordinator (EC) 11

1.4.2. Agent 13

1.4.3. Customer 14

2. Stage 2 Additional Requirements and Updates (S2.2. in mark sheet) 15

2.1. Revised MoSCoW 15

2.2. Functional requirements 16

2.2.1. Complex Event Handling 16

2.3. Non-functional requirements 17

2.3.1. Encryption System 17

2.3.2. UI Redesign 18

2.4. Object Oriented Analysis (OOA) 19

2.5. Object Oriented Design (OOD) 19

2.5.1. Model class diagram (final iteration) 19

2.5.2. MVP diagram (final iteration) 21

2.6. Implementation 22

2.6.1. Details 22

2.6.2. Front end (UI/View) state diagram (final iteration) 22

2.6.3. Front end design (final iteration) 23

2.6.4. Sequence diagrams (final iteration) 23

2.6.5. Application specific code 24

2.6.6. Testing 24

2.6.6.1. Unit tests (dynamic white box) 24

2.6.6.2. User test (dynamic black box) 24

4. References 26

5. Glossary 27

APPENDIX 28

A1. Application Screenshots to show UI changes 28

A2. Event View as Agent 34

A3. Folder Organisation 35

A4. White Box Testing Code 36

A4.1. DatabaseManager Test Setup.......... 36
A4.2. DatabaseManager unit tests 36
A4.3. DatabaseManager Test Cleanup.......... 40
A4.4. User Test Setup.......... 40
A4.5. User Unit Tests.......... 41
A4.6. User Test Clean-up.......... 41
A5. Code Snippets 42
A5.1. DatabaseManager.java : deleteBooking() 42
A5.2. DatabaseManager.java : deleteEvent() 42
A5.3. BookingViewPresenter : verifySecretWord().......... 43
A5.4. BookingViewPresenter : SaveBookingButtonPressed() snippet.......... 44

1. Stage 1 Feedback Address (S2.1 in mark sheet)

1.1. Table of Contents for Feedback

Feedback label	Feedback statement	Feedback address	Update (with hyperlinks)
F1.1	*You should add your own system to the feature matrix for the next Stage 2 so you can see how it compares to the five similar systems.*	Feedback noted, feature matrix has now been updated with our system.	1.2. Revised Feature Matrix
F2.1	*It is better to use a table style format for use cases with success scenarios (rather than 'happy path') with clearly numbered steps and alternative scenarios (instead of 'unhappy path') also with corresponding numbered steps that relate to the success scenario.*	Table style format for the use cases has now been added to this report.	1.3. Revised Textual Use Case
F2.2	*Please put Personas in an appendix if there are many!*	Feedback noted, however there are no further personas to add to this report.	N/A
F3.2	*Although the analysis from use cases is thorough, the NLM approach is not clearly illustrated. Using for example coloured text to emphasise re-occurring entities which are then classified as potential classes, attributes and relationships is a clearer way of illustrating how these are derived and captured as OOA model in 3.2 (Initial class diagram).*	NLM approach is updated and all classes, attributes, and relationships have been color-coded in this report.	1.4. Revised Natural Language Modelling

F4.1	*Event uses an array of Booking(s) which are passed via a mutator. This is not a composite relationship as Event is not instantiating any Booking(s). The relationship is aggregate instead (white diamond). The same in the addBooking method where the booking b is passed as a parameter.*	This has been understood and amended on diagrams, along with adding the new class and methods.	**Error! Reference source not found.** 2.5.2. MVP diagram (final iteration)
F5.1	*The buttons at the bottom right of various windows that constitute actions (e.g. Save, Add, etc.) are too small and using italic text for such buttons is not a good idea. They should be larger than all the rest and preferably have larger or boldface fonts so they are clearly visible (Currently they are not).*	Feedback noted, all buttons constituting actions have been changed and added to this report. In addition, many other fields have had their fonts increased or elements repositioned to look clearer on modern monitors.	2.3.2. UI Redesign Appendix: A1. Application Screenshots to show UI changes
F5.2	*Make sure that if someone has no write rights that they cannot alter the information even if this has no effect in the database. Reason is that once they removed information they'll have to go out and re-enter if the information was important. As such the text fields should be frozen.*	A check is now performed when the Event view is loaded so that if the currently logged in account is not an Event Coordinator, the fields will show as labels instead.	Appendix: A2. Event View as Agent Also shown in Presentation.
F5.3	*Make sure that entered passwords are masked.*	Masks are now applied to appropriate fields.	Appendix A1: Figure 1
F5.4	*Use password hashes (encryption) when passwords are transmitted to the database.*	Encryption class has been created for generating password salt and hashes. Various changes have been made to the model and presenter to deal with the addition of encryption.	2.3.1. Encryption System
F5.5	*Put each of M, V and P in a separate folder for clarity.*	Feedback noted. Each of M, V and P are now in separate folders.	Appendix: A3. Folder Organisation

F5.6	*Enhance the 'Create New Event' feature to handle an event with a complex schedule. For example, an event is scheduled 7pm on Monday, 6pm on Tuesday, 3pm on Wednesday, etc. : can you handle such schedule?*	Although progress was made in designing this feature, implementation proved too difficult in the time remaining and so the feature was ultimately not integrated into the project.	2.2.1. Complex Event Handling
F5.7	*No testing reported.*	Dynamic White box testing implemented from the start of phase two and expanded on throughout development.	2.6.6. Testing Appendix: A4. White Box Testing Also featured in presentation.
F6.1	*Put screenshots in appendix.*	Agreed, all relevant screenshots of the application can now be found in the appendix	Appendix (multiple sections).
F7.1	*Do not overuse fade-out and fade-in and if you consider to use them, make sure they are timed correctly. Currently it feels like I missing out on information due to fade-outs happening too soon.*	Noted, presentation was edited differently to accommodate this.	Shown in Presentation.
F7.2	*No unit test demo-ed as requested in guidelines document.*	Added unit tests to program and shown them in presentation.	2.6.6. Testing Appendix: A4. White Box Testing Also featured in Presentation.

Table 1. Stage 1 feedback address.

1.2. Revised Feature Matrix

Feature	WeezEvent	TicketSource	Eventbrite	RSVPify	BookingLive	Our System (EBS)
Create / Define Venue	✓	✓	✓	✓	✓	✓
Create Event at venue	✓	✓	✓	✓	✓	✓
Pick date and time for event	✓	✓	✓	✓	✓	✓
Ability to add different ticket types for different seats/ areas	✓	✓	✓	✓	✓ (as upsell)	✓
Ability to add different tickets for different ages	X	X	✓	✓	✓	✓
Ability to add a visual for the event	✓	X	✓	✓	✓	X
Can add YouTube video to event	X	X	✓	X	✓	X
Ability to see revenue taken from tickets so far	✓	X	✓	✓	✓	✓
Allows widget integration into eXisting site	✓	✓	✓	✓ (when not taking payment)	✓ (iframe)	X
Allows sales / advertising via Facebook integration	✓	✓	✓	✓	X	X
Gives subsite / minisite to advertise and allow bookings	✓	X	✓	✓	✓	X
Ability to customise the design of the ticket	✓	X	X	✓	X	X
Generate and print tickets for physical sale	✓	✓	X	X	X	X
System holds account details of attendees	✓	✓	✓	✓	✓	✓
System sends confirmation to users when ticket is booked	✓	✓ (mailchimp)	✓	✓	✓	✓
Email to event organiser when ticket booked / quota reached / sold out	✓	?	?	?	?	X
Allows creation of discount codes	✓	X	X	✓	X	X
Send invites by email	✓	X	✓	✓	X	X
Allows reports to be made	✓	✓	✓	✓	✓	✓
Integration with Stripe / Paypal	X	✓ (Stripe)	✓ (Paypal)	✓ (Stripe)	✓ (Stripe)	X
Allows payment with Debit / Credit Card	✓	✓	✓	✓	✓	X
Configurable booking fee	✓	✓	✓	✓	X	X

Allows booking co-ordinator (BCO) to see attendees	✓	✓	✓	✓	✓	✓
Allows BCO to view graphs on event attendance etc.	✓	✓	✓	✓	X (table only)	✓
Allows BCO to add attendees to system	?	?	✓	✓	✓	✓
Provides on-the-door ticket validation	X	✓	X	✓ (QR)	X	X
Seating plan management	X	✓	X	✓	X	X
Togglable ability to allow customers make donation to charity	X	✓	✓	✓	X (could be added as upsell)	X
Ability to check details are correct before publication	✓	X	✓	✓	✓	✓
Can set a maximum number of tickets to sell	✓	✓	✓	✓	✓	✓
Allows additional costs to be added to ticket (food/drink)	X	✓	✓	✓	✓	✓
Ability to set a time and date to stop selling tickets	X	✓	✓	X	✓	X
Booking interface for customers to book their own tickets	✓	X	✓	✓	✓	X
Links event venue to Google Maps location	X	X	✓	X	X	X
Event designer system for minisite	X	X	✓	X	X	X
Can limit the number of ticket sales per transaction	X (default only)	X (default limit)	✓	X	X (10 by default)	X
Can set different entry times per ticket	X	X	✓	X	X	X
Minisite traffic analytics	X	X	✓	X	✓	X
Allows a set refund policy	X	X	✓	X	X	X
Can set up surveys to go out to attendees	X	X	✓	X	X	X
Customisable URL for minisite	X	X	X	✓	X	X
Specific settings for dinner can be toggled on	X	X	X	✓	X	X
Ability to ask guests custom questions (hotel, travel plans)	X	X	X	✓	X	X
Ability to send a .ics calendar invite to attendees	X	X	X	✓	X	X

1.3. Revised Textual Use Case

Title: Agent makes a new booking.

Summary: Customer contacts Agent with inquiry about an event. Agent takes Customer details and saves them to the database. Agent then makes a booking with the requested details.

Actors: Customer, Agent

Stakeholders: Customer, Agent, Events Coordinator

Preconditions: Event Coordinator and Agent have accounts in the database. Event Coordinator has created a valid event. Agent has logged into the system.

Triggers: Agent receives a phone call/in-person request from the customer.

Version: 2.0

Main Success Scenarios:

Actions of Actors	Actions of System
1. Customer inquiries about making a booking on a certain event	
2. Agent navigates to event list page.	
3. Agent double clicks on the requested event.	4. The system presents the event details screen.
5. Agent navigates to bookings tab.	
6. Agent clicks on 'Make new Booking' button.	7. The system presents the new booking screen.
8. Agent requests customer username, password, and address.	
9. Customer provides details.	
10. Agent enters details into text fields and clicks 'Add new Customer' button.	11. System saves new customer to database.
12. Agent selects new customer.	
13. Agent types in requested number of tickets of the requested type.	
14. Agent types in requested number of sundries of the requested type.	
15. Agent clicks on 'Save Booking' button.	16. System presents a success message.
	17. System saves new booking to database.
	18. System generates confirmation email and sends it to customer.

Alternative Scenarios:

A1. Event not located in database.

A1 scenario starts at point 3 of the Main Success Scenario.

Action of Actors	Action of System
3. Agent double clicks on the requested event.	4. Agent double clicks on the requested event.
	5. Error message presented.
	6. Failure logged.

A2. Incorrect Booking page for 'New Booking'.

A2 scenario starts at point 6 of the Main Success Scenario.

Action of Actors	Action of System
6. Agent clicks on 'Make new Booking' button.	7. System fails to correctly present empty booking page template.
	8. Error message presented.
	9. Failure logged.

A3. Unable to save new customer to database.

A3 scenario starts at point 10 of the Main Success Scenario.

Action of Actors	Action of System
10. Agent enters details into text fields and clicks 'Add new Customer' button.	11. System fails to save new customer to database.
	12. Error message presented.
	13. Failure logged.

A4. Error in saving new booking.

A4 scenario starts at point 15 of the Main Success Scenario.

Action of Actors	Action of System
15.Agent clicks on 'Save Booking' button.	16. System fails to save new booking to database.
	17. Error message presented.
	18.Failure logged.

A5. No confirmation email sent to customer.

A5 scenario starts at point 17 of the Main Success Scenario.

Action of Actors	Action of System
	17. System saves new booking to database.
	18.System fails to generate confirmation email and send it to customer. 19. Error message presented.
	20.Failure logged.

1.4. Revised Natural Language Modelling

1.4.1. Events Coordinator (EC)

Log into the system:

1. The Events Coordinator types their details into the account login screen
2. The system processes their login
3. The system returns the Events Coordinator 'dashboard' upon successful login

Add new event:

1. From the 'dashboard', the Events Coordinator clicks on 'Add new event' button.
2. The system presents the new event screen, providing options for adding start time/end time, date, ticket types, prices and sundries.
3. The Events Coordinator enters event details and clicks on 'Publish event'.

Design ticket:

1. From the 'dashboard', the Events Coordinator clicks on 'Edit event' button.
2. The system presents the event edit screen.
3. The Events Coordinator clicks on 'Upload ticket design' button.
4. The system allows them to select an image file from their computer, and uploads the image, attaching it to the ticket.
5. The Events Coordinator clicks on 'Apply'.

View revenue:

1. From the 'dashboard', the Events Coordinator navigates to the events list screen.
2. The system presents each event's total revenue (a function of ticket sales).

Notify customers of event changes:

1. From the 'dashboard', the Events Coordinator navigates to the events list screen.
2. The Events Coordinator clicks on the event they wish to edit.
3. The system presents the edit event screen.
4. The Events Coordinator makes any edits they wish.
5. The system generates an email to notify any customers who have tickets to that event.
6. The system sends the emails.

Create discount codes:

1. From the 'dashboard', the Events Coordinator navigates to the events list screen.
2. The Events Coordinator clicks on the event they wish to create a discount code for.
3. The system presents the edit event screen.
4. The Events Coordinator types the required discount code in the 'Discount code' field and chooses how much (in percent) a customer will save.
5. The Events Coordinator clicks on 'Apply'.

Generate reports:

1. From the 'dashboard', the Events Coordinator navigates to the Reports page.
2. The Events Coordinator selects the scope of the report from several drop-down boxes and clicks 'Generate'.
3. The system generates the report according to the specified scope and presents it as a PDF.

Customise the look and feel of the customer-facing pages:

1. From the 'Edit Event' screen Events Coordinator clicks 'Customise look'
2. The Events Coordinator specifies an image and colour scheme for the corresponding event.
3. The Events Coordinator clicks 'Accept', system changes the image and colour scheme for the corresponding ticket sales page.

Add sundry items to an existing event:

1. From the 'Edit Event' screen, the Events Coordinator clicks the 'Add new items' button from the Sundry Items section
2. The Events Coordinator adds the details of the new items including name, description and price and sales limit
3. The Events Coordinator clicks accept, system adds tax and additional (pre-set) fees to price and adds it to the event.

Add different types of ticket to an existing event:

1. From the 'Edit Event' screen, the Events Coordinator clicks the 'Add new ticket' button in the tickets box of the system
2. The Events Coordinator enters the new ticket details including name, price, ticket sales limit
3. The Events Coordinator confirms details and the system adds the ticket to the event for sale and updates the maximum number of tickets to be sold to include the new tickets

Change the maximum number of tickets to be sold:

1. From the 'Edit Event' screen, the Events Coordinator clicks the 'Total Max tickets' field and enters the limit number, which cannot be lower than the amount of tickets sold so far
2. System changes the maximum number of tickets for sale and tickets remaining based off of this number

To generate and print tickets from the system:

1. From 'Dashboard', the Events Coordinator clicks the events button
2. The Events Coordinator finds and selects the event they wish to print tickets from
3. The Events Coordinator clicks the 'print tickets' button
4. The Events Coordinator enters how many tickets to print – system limits to remaining tickets available
5. System builds and sends request to specified printer
6. System deducts number of printed tickets from remaining ticket count

1.4.2. Agent

Log into the system:

1. The agent types their details into the account login screen
2. The system processes their login
3. The system returns the agent 'dashboard' upon successful login

Make a booking:

1. Customer calls agent to make a booking or for enquiry.
2. From the 'dashboard', the agent accesses details of event.
3. System shows event description start time/end time, date, ticket types, prices and sundries available.
4. Agent asks for customer details.
5. Agent makes a booking on the event.
6. System records customer details for the booking.
7. Agent asks customer for payment details.
8. System processes payment.
9. System generates receipt and emails ticket to customer.

Edit a booking:

1. From the 'dashboard', agent accesses the customer panel to view details of customer's booking.
2. Agent makes changes to customer's booking.
3. System records the changes made.
4. System processes additional payment if required.
5. System generates new receipt and emails edited ticket to customer.

Send emails to advertise an event:

1. From the 'dashboard', Agent clicks 'Events' button, system displays list of events.
2. Agent selects the event they wish to email about to view event's details.
3. System displays event information page.
4. Agent clicks 'advertise event' button.
5. The system accesses the list of previous customers that are happy to be emailed
6. System sends emails to advertise upcoming event.

1.4.3. Customer

View customer's information in the system:

1. The customer phones the box office to speak to an agent
2. From the agent 'dashboard', the agent clicks the 'find customer' button and requests the customers' name and postcode
3. The agent enters the above details into the search box on the 'find customer' screen
4. The system displays the customer's account information including full name, address, email and purchase history on the 'customer details' screen

Create a new account:

1. The customer phones the box office to speak to an agent
2. From the agent 'dashboard', the agent clicks the 'Add new user' button, the system displays the corresponding page
3. The agent asks the customer for their details including name, address and email for contact
4. The agent confirms with the customer that they are happy to receive emails from them
5. The agent confirms the above fields, the system adds a record for the new customer

Make a booking:

1. From the agent 'customer details' screen, the agent clicks the 'new booking' button
2. The customer confirms which event they would like to make a booking, the agent clicks the booking as listed on the given screen
3. The customer gives the number of tickets and any extras they would like to order
4. The agent checks the tickets are bookable, confirms the customer's request and submits the details to the system
5. The system creates a new booking for that customer on the selected event and removes the number of tickets sold from the total available
6. The system generates a ticket and receipt which is emailed to the customer

View a previous booking:

1. From the agent 'customer details' screen, the agent finds the booking the customer has requested and clicks it
2. The agent relays the requested information to the customer

View the frequently asked questions (FAQs):

1. The customer phones the box office to speak to an agent
2. The customer relays the question to the agent
3. The agent gives the customer the answer to their question

2. Stage 2 Additional Requirements and Updates (S2.2. in mark sheet)

2.1. Revised MoSCoW

Feature	Status	Added
Must Have		
Ability for EC to create, edit, and delete an event.	Specification met	Stage 1
System to notify customers by email when a booking has been made.	Specification met	Stage 1
Agent must have ability to make, edit and delete bookings for customers.	Specification met	Stage 1
Should Have		
Ability for EC and Agent to view all events and see them in detail.	Specification met	Stage 1
Ability for EC to view how much total revenue has been generated by bookings.	Specification met	Stage 1
Ability for EC to view how much revenue has been generated per event.	Specification met	Stage 1
Ability for EC to generate reports of various criteria and print them in PDF form.	Specification not met	Stage 1
Ability for EC to add sundry items to an event which can be added to a booking.	Specification met	Stage 1
Ability for EC to specify several different types of ticket for an event.	Specification met	Stage 1
Ability for EC to specify the maximum number of tickets which can be sold per event.	Specification met	Stage 1
The system should have the ability to generate and print/email tickets.	Specification not met	Stage 1
Each booking should be able to contain several tickets.	Specification met	Stage 1
System should retain customer data for subsequent bookings.	Specification met	Stage 1
Some level of security and account management.	Specification met	Stage 1
Different levels of access for EC and Agent.	Specification met	Stage 1
EC/Agent should be able to verify the identity of the Customer.	Specification met	Stage 2
Events should have the ability to handle a complex schedule.	Specification not met	Stage 2
Could Have		
Ability for EC to design a custom ticket template to be printed/emailed to customer.	Specification not met	Stage 1
Notify customers automatically by email if an event they have booked has been edited.	Specification not met	Stage 1
Ability for EC to create discount codes which are automatically applied to a booking.	Specification not met	Stage 1
The system could generate and distribute marketing emails to customers held within the database.	Specification not met	Stage 1
The system could handle payment processing through some third party API.	Specification not met	Stage 1
The system should make it easy for the user to see what elements are interactable and be easy to navigate.	Specification met	Stage 2

Feature	Status	Added
Won't Have		
Online integration with event websites.	Specification not met	Stage 1
The customer will not be able to make their own bookings online.	Specification not met	Stage 1
The customer will not be able to view their own bookings or booking history.	Specification not met	Stage 1
Any kind of registration page—not necessary in stage 1.	Specification not met	Stage 1
Google Maps integration when showing event locations.	Specification not met	Stage 1
Credit card processing/data handling.	Specification not met	Stage 1

2.2. Functional requirements

2.2.1. Complex Event Handling

"Events should have the ability to handle a complex schedule". This feature was added as a direct result of feedback received from Stage 1 (F5.6). We first discussed how we could go about adding this functionality and came up with a couple different ways to do it.

The first idea was to have a new attribute in the event table to record whether the event was repeatable weekly and if so, have a script on the database that would run every week and duplicate those events for the following two or three weeks. We did not like this approach, however, as it took processing away from our application itself, and could easily mess up (duplicating duplicated events every week).

Another idea we explored was to have the start and end dates and times be set in an ArrayList which could be added to either manually or automatically for repeating events. This would require a new table in the database and additional code in DatabaseManager to handle loading from the new table. The Event class would need to be adjusted, as would the unit tests. EventListView would need to be adjusted to show multiple dates and times for an event, and there would need to be changes to EventView and additional code written in EventViewPresenter to allow the user to easily add new dates to an event. This concept proved cumbersome in execution, however, as it was difficult to show multiple dates on an event in the EventListView, so with time running out we had to go back and think of another implementation.

The final idea we decided on was to have the ability to easily duplicate an event while providing a different start and end date. This would be in the form of a button on EventView which would simply duplicate the current entry without a date and time and allow the user to input a new date and time and save it. It was an improvement on what we started with, but we felt it did not quite solve the issue we felt was highlighted in F5.6 and so we shelved the development and stashed these changes for after other features were completed. Unfortunately, however, these other features took longer to implement than we expected and so this feature was not realised by the due date.

2.3. Non-functional requirements

2.3.1. Encryption System

"Some level of security and account management". Although this feature was added to the MoSCoW analysis in Stage 1, it was not fully realised in the application until Stage 2. In Stage 1, we added the ability for the user to add and delete customers, as well as change their passwords should the customer wish to do so. However, these passwords (and the passwords for staff members) were held in plaintext in the database. We recognised that this was a big security issue at the time and as it was also picked up in our Stage 1 feedback (F5.4) we knew we needed to spend a large part of Stage 2 on this.

The implementation of this system proved more challenging than we had first expected, as everything built in Stage 1 relied on the password being transmitted and stored as a String, but password hashes are byte arrays. In addition, we also needed to deal with password salting, as salting and hashing passwords (with multiple passes) is the industry standard these days. To do this, we created an Encryption class to generate salts and hashes. We then modified all User classes (including EventCoordinator, Agent and Customer classes) to handle passwords as byte arrays and hold the password salt. We had to change the design of the database to handle passwords as BLOBs and add a new field for salt BLOBs. We then and modified the method for adding new customers so it took the customer object and a plaintext password as a String, called on the new Encryption methods and stored this information in the database along with the other customer information. We re-wrote the code for authentication to pull the salt and hash from the database, then generate a hash based off of the password entered by the user in the Login form and the salt from the database and compare the two hash arrays – if the arrays match then the password is the same and the user is authenticated as before.

Following this implementation, we realised that all the unit tests regarding adding customers to the database, and anything dealing with users or authentication which were written early on were now broken so these had to be rewritten and rerun to ensure they still performed correctly. These modified unit tests can be found in appendix A4.5. User Unit Tests.

There was some temporary code added to generate salts and hashes for existing users in the database and test the hash matched a hash generated from their previous plaintext password. However, this code was removed from the application before the update was pushed and so it unfortunately cannot be included here.

"EC/Agent should be able to verify the identity of the Customer". This is a new feature for Stage 2 which came about early on while thinking of aspects of security for the above feature. We understood that we wanted a way for staff to verify the identity of Customers as impersonation would be possible under the Stage 1 system which could be another security concern.

Once the above encryption feature was complete, we began work on modifying the booking view/edit screen to accommodate this new feature. With this feature, we realised that the phrase 'password' did not make sense to use regarding customers as they were no longer logging in to the system themselves but through a proxy. In line with what some other businesses do in this situation, we renamed usages of 'Password' to 'Secret Word' where customers are concerned, which gives a distinction between something that should be hidden from view (password) and something that the user will need to see in order to ensure they have the correct spelling. We decided that this should mean that any entering of a Secret Word should not be masked, but that we could keep secret words as hashes in the database as the functionality was already set up for it and it would be the safer option.

For implementation, the mini form in the middle of the screen, shown in Figure 6, takes a plaintext string and upon clicking the 'Verify' button, creates a hash from the chosen customer's salt and compares this hash to the hash stored on that customer object. If the two hashes match, a success message is printed to an empty label under the verify button. The code for this can be found in appendix A5.3. BookingViewPresenter : verifySecretWord().

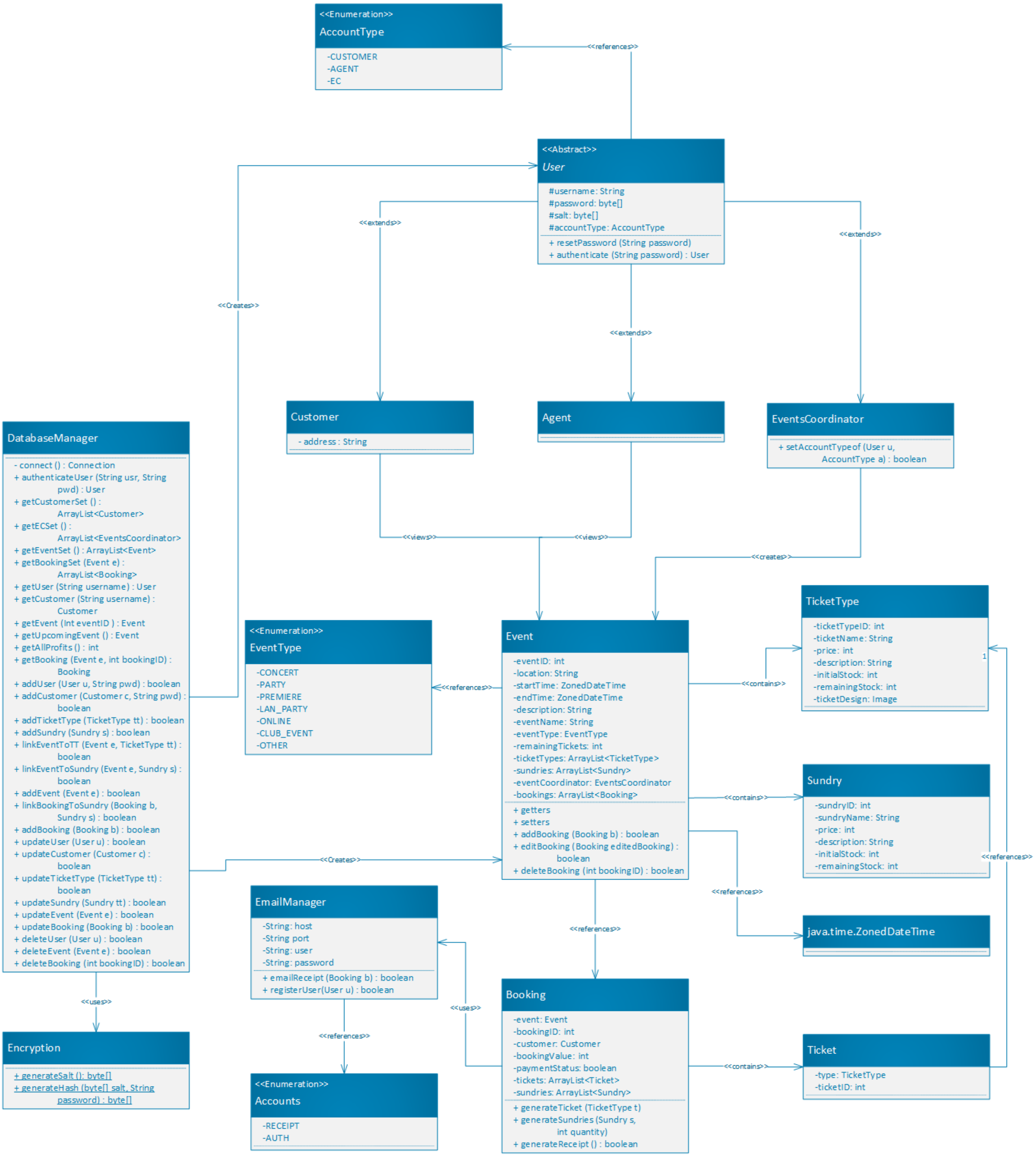
A small piece of code was also added to the Save Booking button to ensure that the form could not be saved unless the customer was verified first. This code can be found in appendix A5.4. BookingViewPresenter : SaveBookingButtonPressed() snippet.

2.3.2. UI Redesign

"The system should make it easy for the user to see what elements are interactable and be easy to navigate". Following on from the design in Stage 1, we decided that this should be a feature we should draw more notice to in Stage 2, especially as it was picked up in feedback F5.1 that certain elements were not well designed. We decided that to start, we would make changes that would fulfil the requirements of the feedback, mainly by making the buttons stand out more. Second to this, we wanted to completely overhaul the UI from the ground up to make it look much less like a basic JavaFX window and more like a modern app. To this end we researched different modern UI style guides and settled on Material Design. This is the style that Google has used for the past few years and can be seen in their apps and web applications and is a style which many other developers have begun to use in recent years. Ultimately however, we did not have the time to incorporate this into the project but some reference material can be found in the links in 4. References.

The additional changes that were made were to increase the size of the buttons and make them bold, add colour to certain buttons and increase the font size of the UI elements in general to make them easier to read on modern high-resolution displays. This can be seen in appendix A1. Application Screenshots to show UI changes.

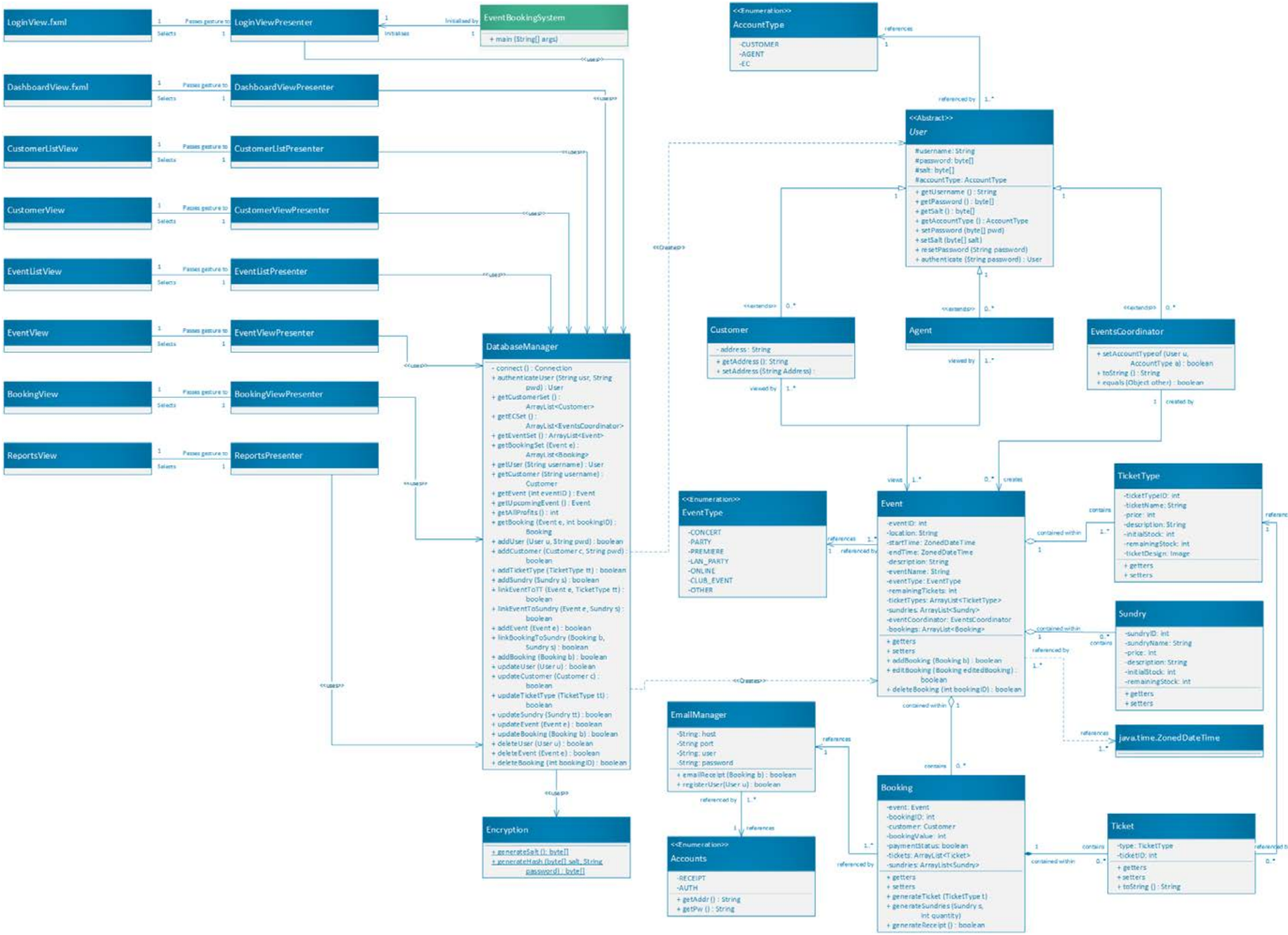
2.4. Object Oriented Analysis (OOA)


<<Enumeration>>
AccountType
-CUSTOMER
-AGENT
-EC
<<references>>
<<Abstract>>
User
#username: String
#password: byte[]
#salt: byte[]
#accountType: AccountType
+ resetPassword (String password)
+ authenticate (String password) : User
<<extends>>
<<extends>>
<<extends>>
<<Creates>>
Customer
- address : String
Agent
EventsCoordinator
+ setAccountTypeof (User u, AccountType a) : boolean
<<views>>
<<views>>
<<creates>>
DatabaseManager
- connect () : Connection
+ authenticateUser (String usr, String pwd) : User
+ getCustomerSet () : ArrayList<Customer>
+ getECSet () : ArrayList<EventsCoordinator>
+ getEventSet () : ArrayList<Event>
+ getBookingSet (Event e) : ArrayList<Booking>
+ getUser (String username) : User
+ getCustomer (String username) : Customer
+ getEvent (Int eventID) : Event
+ getUpcomingEvent () : Event
+ getAllProfits () : int
+ getBooking (Event e, int bookingID) : Booking
+ addUser (User u, String pwd) : boolean
+ addCustomer (Customer c, String pwd) : boolean
+ addTicketType (TicketType tt) : boolean
+ addSundry (Sundry s) : boolean
+ linkEventToTT (Event e, TicketType tt) : boolean
+ linkEventToSundry (Event e, Sundry s) : boolean
+ addEvent (Event e) : boolean
+ linkBookingToSundry (Booking b, Sundry s) : boolean
+ addBooking (Booking b) : boolean
+ updateUser (User u) : boolean
+ updateCustomer (Customer c) : boolean
+ updateTicketType (TicketType tt) : boolean
+ updateSundry (Sundry tt) : boolean
+ updateEvent (Event e) : boolean
+ updateBooking (Booking b) : boolean
+ deleteUser (User u) : boolean
+ deleteEvent (Event e) : boolean
+ deleteBooking (int bookingID) : boolean
<<Enumeration>>
EventType
-CONCERT
-PARTY
-PREMIERE
-LAN_PARTY
-ONLINE
-CLUB_EVENT
-OTHER
<<references>>
Event
-eventID: int
-location: String
-startTime: ZonedDateTime
-endTime: ZonedDateTime
-description: String
-eventName: String
-eventType: EventType
-remainingTickets: int
-ticketTypes: ArrayList<TicketType>
-sundries: ArrayList<Sundry>
-eventCoordinator: EventsCoordinator
-bookings: ArrayList<Booking>
+ getters
+ setters
+ addBooking (Booking b) : boolean
+ editBooking (Booking editedBooking) : boolean
+ deleteBooking (int bookingID) : boolean
<<contains>>
TicketType
-ticketTypeID: int
-ticketName: String
-price: int
-description: String
-initialStock: int
-remainingStock: int
-ticketDesign: Image
1
<<contains>>
Sundry
-sundryID: int
-sundryName: String
-price: int
-description: String
-initialStock: int
-remainingStock: int
<<references>>
<<references>>
java.time.ZonedDateTime
<<Creates>>
EmailManager
-String: host
-String port
-String: user
-String: password
+ emailReceipt (Booking b) : boolean
+ registerUser(User u) : boolean
<<uses>>
<<references>>
<<references>>
<<uses>>
Encryption
+ generateSalt (): byte[]
+ generateHash (byte[] salt, String password) : byte[]
<<Enumeration>>
Accounts
-RECEIPT
-AUTH
Booking
-event: Event
-bookingID: int
-customer: Customer
-bookingValue: int
-paymentStatus: boolean
-tickets: ArrayList<Ticket>
-sundries: ArrayList<Sundry>
+ generateTicket (TicketType t)
+ generateSundries (Sundry s, int quantity)
+ generateReceipt () : boolean
<<contains>>
Ticket
-type: TicketType
-ticketID: int

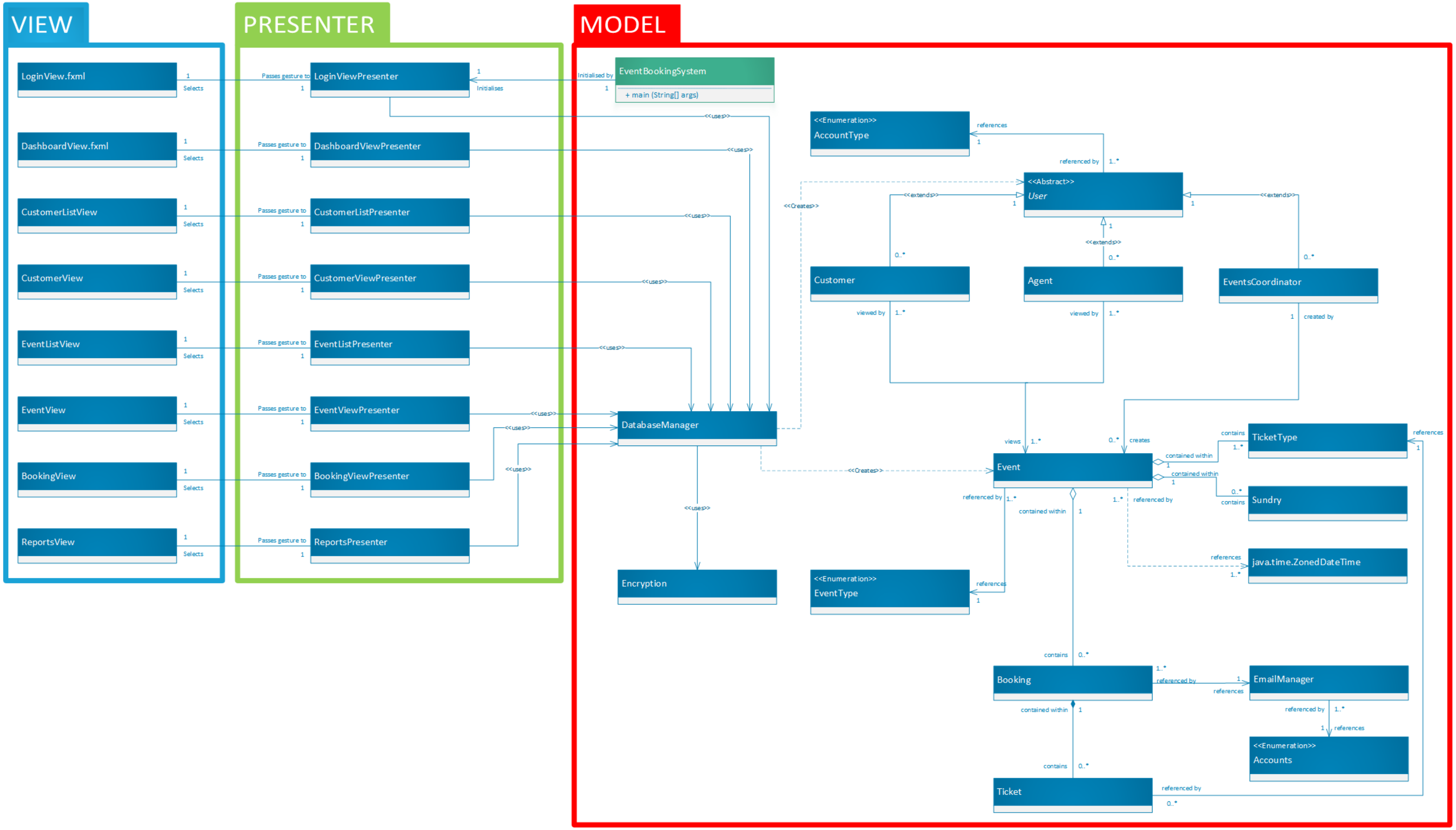
2.5. Object Oriented Design (OOD)

2.5.1. Model class diagram (final iteration)

(Diagram shown on following page)


LoginView.fxml
1
Passes gesture to
Selects
1
LoginViewPresenter
1
Initialises
Initialised by
1
EventBookingSystem
+ main (String[] args)
<<uses>>
DashboardView.fxml
1
Passes gesture to
Selects
1
DashboardViewPresenter
<<uses>>
CustomerListView
1
Passes gesture to
Selects
1
CustomerListPresenter
<<uses>>
CustomerView
1
Passes gesture to
Selects
1
CustomerViewPresenter
<<uses>>
EventListView
1
Passes gesture to
Selects
1
EventListPresenter
<<uses>>
EventView
1
Passes gesture to
Selects
1
EventViewPresenter
<<uses>>
BookingView
1
Passes gesture to
Selects
1
BookingViewPresenter
<<uses>>
ReportsView
1
Passes gesture to
Selects
1
ReportsPresenter
<<uses>>
DatabaseManager
- connect () : Connection
+ authenticateUser (String usr, String pwd) : User
+ getCustomerSet () : ArrayList<Customer>
+ getECSet () : ArrayList<EventsCoordinator>
+ getEventSet () : ArrayList<Event>
+ getBookingSet (Event e) : ArrayList<Booking>
+ getUser (String username) : User
+ getCustomer (String username) : Customer
+ getEvent (int eventID) : Event
+ getUpcomingEvent () : Event
+ getAllProfits () : int
+ getBooking (Event e, int bookingID) : Booking
+ addUser (User u, String pwd) : boolean
+ addCustomer (Customer c, String pwd) : boolean
+ addTicketType (TicketType tt) : boolean
+ addSundry (Sundry s) : boolean
+ linkEventToTT (Event e, TicketType tt) : boolean
+ linkEventToSundry (Event e, Sundry s) : boolean
+ addEvent (Event e) : boolean
+ linkBookingToSundry (Booking b, Sundry s) : boolean
+ addBooking (Booking b) : boolean
+ updateUser (User u) : boolean
+ updateCustomer (Customer c) : boolean
+ updateTicketType (TicketType tt) : boolean
+ updateSundry (Sundry tt) : boolean
+ updateEvent (Event e) : boolean
+ updateBooking (Booking b) : boolean
+ deleteUser (User u) : boolean
+ deleteEvent (Event e) : boolean
+ deleteBooking (int bookingID) : boolean
<<uses>>
Encryption
+ generateSalt (): byte[]
+ generateHash (byte[] salt, String password) : byte[]
<<Creates>>
<<Creates>>
<<Enumeration>>
AccountType
-CUSTOMER
-AGENT
-EC
references
1
referenced by
1..*
<<Abstract>>
User
#username: String
#password: byte[]
#salt: byte[]
#accountType: AccountType
+ getUsername () : String
+ getPassword () : byte[]
+ getSalt () : byte[]
+ getAccountType () : AccountType
+ setPassword (byte[] pwd)
+ setSalt (byte[] salt)
+ resetPassword (String password)
+ authenticate (String password) : User
1
1
1
<<extends>>
0..*
<<extends>>
0..*
<<extends>>
0..*
Customer
- address : String
+ getAddress (): String
+ setAddress (String Address) :
viewed by
1..*
Agent
viewed by
1..*
EventsCoordinator
+ setAccountTypeof (User u, AccountType a) : boolean
+ toString () : String
+ equals (Object other) : boolean
1
created by
views
1..*
0..*
creates
<<Enumeration>>
EventType
-CONCERT
-PARTY
-PREMIERE
-LAN_PARTY
-ONLINE
-CLUB_EVENT
-OTHER
references
1..*
1
referenced by
Event
-eventID: int
-location: String
-startTime: ZonedDateTime
-endTime: ZonedDateTime
-description: String
-eventName: String
-eventType: EventType
-remainingTickets: int
-ticketTypes: ArrayList<TicketType>
-sundries: ArrayList<Sundry>
-eventCoordinator: EventsCoordinator
-bookings: ArrayList<Booking>
+ getters
+ setters
+ addBooking (Booking b) : boolean
+ editBooking (Booking editedBooking) : boolean
+ deleteBooking (int bookingID) : boolean
contains
contained within
1
1..*
TicketType
-ticketTypeID: int
-ticketName: String
-price: int
-description: String
-initialStock: int
-remainingStock: int
-ticketDesign: Image
+ getters
+ setters
references
1
contained within
1
0..*
contains
Sundry
-sundryID: int
-sundryName: String
-price: int
-description: String
-initialStock: int
-remainingStock: int
+ getters
+ setters
referenced by
1..*
references
1..*
java.time.ZonedDateTime
contained within
1
contains
0..*
EmailManager
-String: host
-String port
-String: user
-String: password
+ emailReceipt (Booking b) : boolean
+ registerUser(User u) : boolean
references
1
referenced by
1..*
1
references
<<Enumeration>>
Accounts
-RECEIPT
-AUTH
+ getAddr () : String
+ getPw () : String
1..*
referenced by
Booking
-event: Event
-bookingID: int
-customer: Customer
-bookingValue: int
-paymentStatus: boolean
-tickets: ArrayList<Ticket>
-sundries: ArrayList<Sundry>
+ getters
+ setters
+ generateTicket (TicketType t)
+ generateSundries (Sundry s, int quantity)
+ generateReceipt () : boolean
1
contains
contained within
0..*
Ticket
-type: TicketType
-ticketID: int
+ getters
+ setters
+ toString () : String
referenced by
0..*

2.5.2. MVP diagram (final iteration)



2.6. Implementation

2.6.1. Details

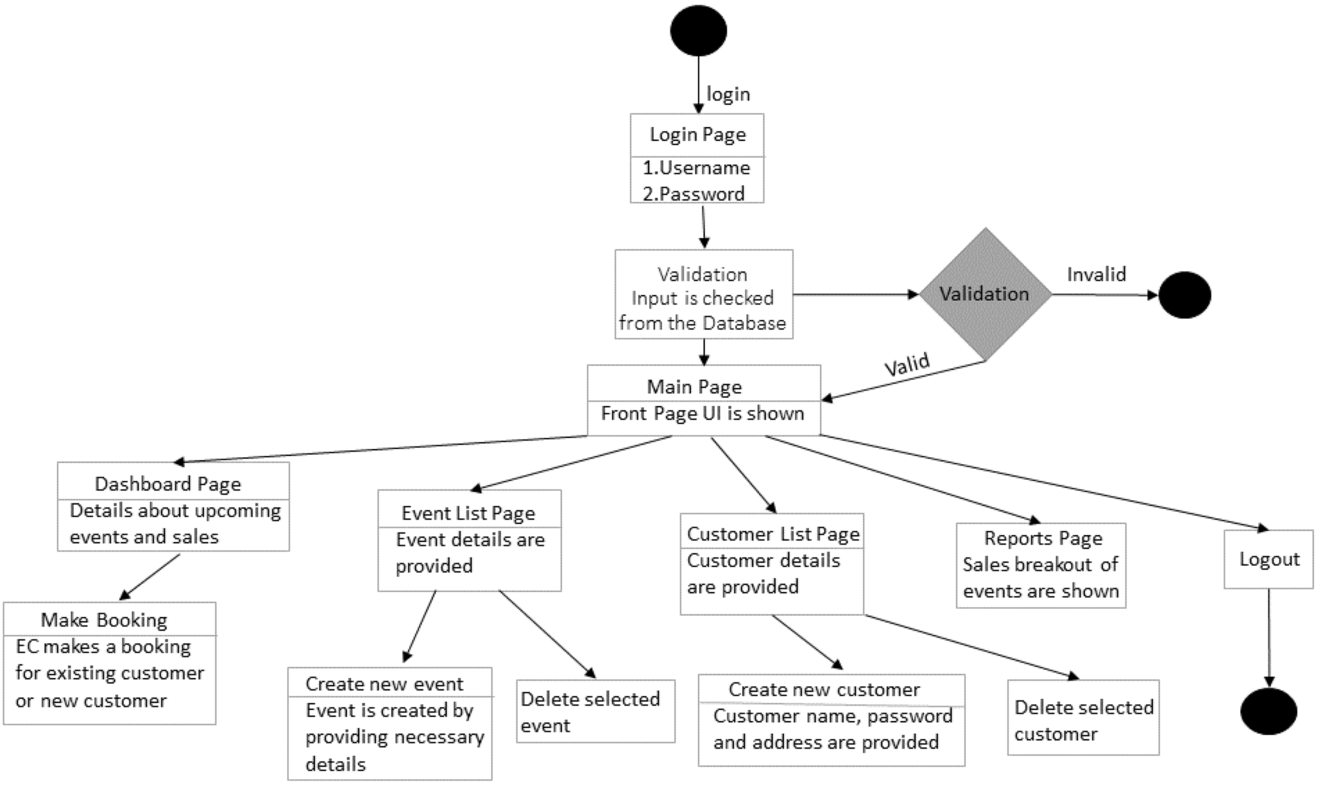
For stage two, we have stuck with using Java and JavaFX and writing in IntelliJ. This is because we felt the setup used in stage one could give us all the functionality we need, in an environment that we are already familiar with. Porting the codebase to Python did not make sense to us after the revision to the stage two requirements.

We are not using a framework or package manager for stage two as we feel it is not necessary for us to achieve what we need to do. Our DBMS is SQLite integrated into Java using JDBC (sqlite-jdbc-3.30.1). This allows us to manage the database itself from within IntelliJ as well as have Java understand the SQL statements we wrote. For testing, we have used JUnit. This was an easy decision for us due to how well it is integrated into IntelliJ and the Java ecosystem.

Regarding external APIs and Libraries, we use the Javax mail and activation libraries to handle connection to our external email account and the sending of emails, all via our Email package.

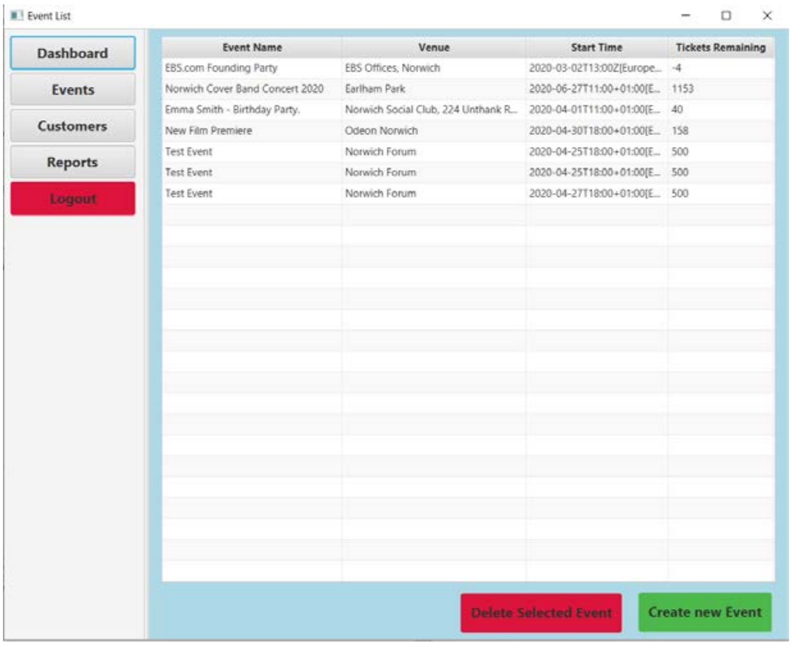
2.6.2. Front end (UI/View) state diagram (final iteration)

State diagram showing actions of the Event Coordinator.



2.6.3. Front end design (final iteration)

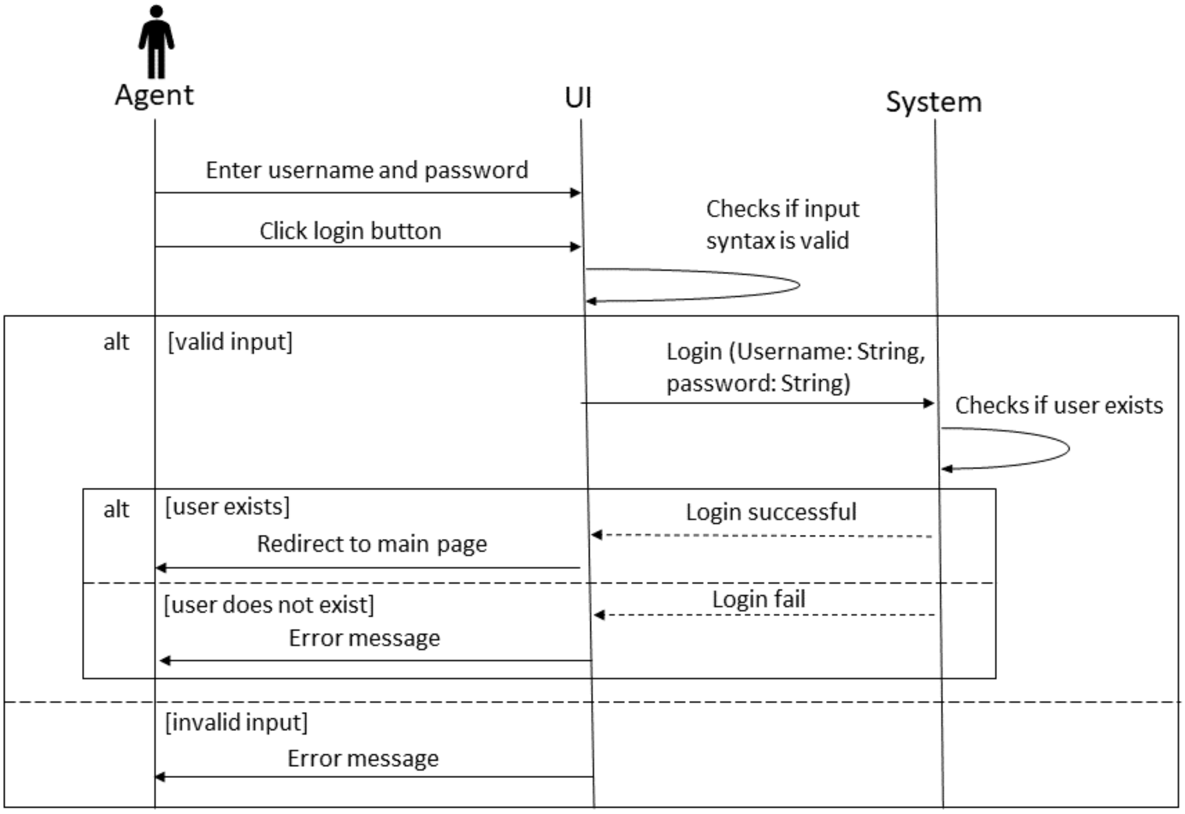
From the main page when we double click on Events, we have the Event List Page as shown below. The details of the events are shown; Event Name, Venue, Start Time and Tickets remaining. The Event Coordinator can choose to “Create new Event” and “Delete Selected Event”



Event Name	Venue	Start Time	Tickets Remaining
EBS.com Founding Party	EBS Offices, Norwich	2020-03-02T13:00Z[Europe...	-4
Norwich Cover Band Concert 2020	Earlham Park	2020-06-27T11:00+01:00[E...	1153
Emma Smith - Birthday Party.	Norwich Social Club, 224 Unthank R...	2020-04-01T11:00+01:00[E...	40
New Film Premiere	Odeon Norwich	2020-04-30T18:00+01:00[E...	158
Test Event	Norwich Forum	2020-04-25T18:00+01:00[E...	500
Test Event	Norwich Forum	2020-04-25T18:00+01:00[E...	500
Test Event	Norwich Forum	2020-04-27T18:00+01:00[E...	500

2.6.4. Sequence diagrams (final iteration)

Sequence diagram showing the login procedure of the Agent.



2.6.5. Application specific code

The following are links to specific code in the appendix referenced elsewhere in this report.
A5.1. DatabaseManager.java : deleteBooking()
A5.2. DatabaseManager.java : deleteEvent()
A5.3. BookingViewPresenter : verifySecretWord()
A5.4. BookingViewPresenter : SaveBookingButtonPressed() snippet

2.6.6. Testing

In stage one, we did not have much opportunity for testing due to a tight schedule and some sub-par organisation on our part. This was remedied in stage two, in which we undertook extensive white box testing from day one, and plans were drawn up for black box user testing also.

2.6.6.1. Unit tests (dynamic white box)

JUnit was used to construct unit tests for all relevant model classes. Each method within a class was tested; dummy data was provided, and the expected result was asserted by the test. If the result was not as expected, the test would fail. By limiting the scope of the test to a single method, bugs could be quickly identified and rectified. Constructing unit tests early and running them often throughout development reduces the overall cost incurred by fixing bugs further down the line.

Certain classes were unsuitable, as they only contained simple getter/setter methods; testing these is essentially the same as testing the JVM, thus not very useful.

A more detailed description of the individual unit tests can be found in appendix A4. White Box Testing .

2.6.6.2. User test (dynamic black box)

Ideally any software product must be tested by those intended to use it. Unfortunately, global circumstances prevented us from implementing the kind of user tests we otherwise would have. However, plans were made to outline the method we would have used.

Testers would have been selected from our personal acquaintances. Perhaps counter-intuitively, we would have selected those who had limited technical aptitude, as these are the kind of testers who are likely to home in on concepts or UI elements which developers take for granted. A list of actions would have been provided for them to accomplish using the software, during which they would have been observed (Moderated Usability Test). Finally, a simple questionnaire would have been provided, asking the tester about their experience. This feedback would then have been considered in the next iteration of the development process, especially if certain suggestions were frequently made.

Questionnaire draft: All statements would be answered on a scale from 1 to 7, with 1 being Strongly Disagree, and 7 being Strongly Agree.

- It was easy to use the system.
- The information (on screen messages) provided with the system were clear.
- When I made a mistake with the system, I could recover quickly and easily.
- I liked the aesthetics of the system.
- The organisation of information on the screens was clear.
- The provided tasks were clear and not confusing.
- The system has all the functionality and capability I expect it to have.

- Further suggestions or comments:

This method of usability testing would have worked well for us, given the timeframe and access to participants, however it is not without its faults. Moderated Usability Testing is intuitive, and allows the developer to get immediate feedback on why a tester performed a specific action, or their attitude towards certain features. The observer can also help to clear confusion on the tester's part and understand their cognitive processes. It is difficult, however, not to influence the tester with instructions, and the observer can impede the natural understanding which the user would have come to on their own.

Questionnaires allow you to quantify data which is otherwise subjective and fuzzy. This gives you a solid foundation on which to build future development. However, it is extremely difficult as a developer to ask the right questions, and not be influenced by your own bias.

4. References

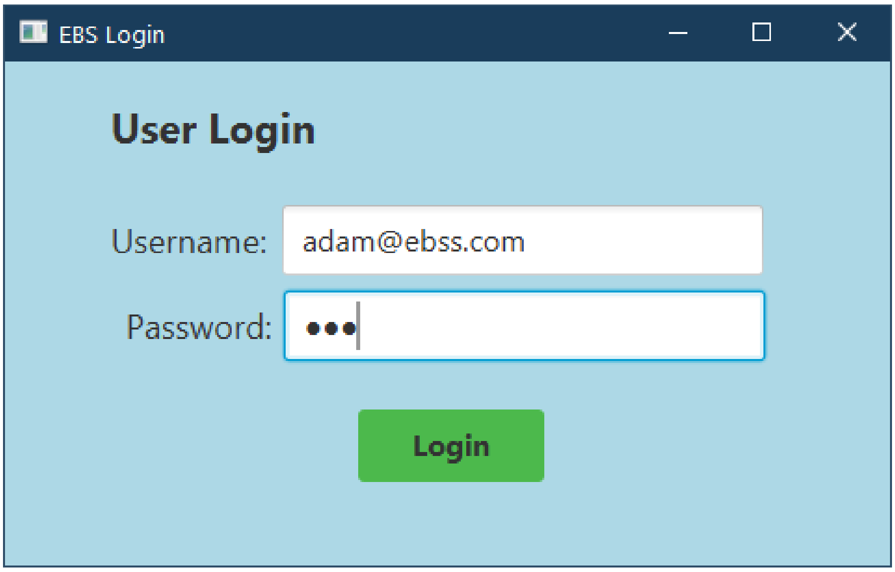
Material Design:
Material Design Components: https://material.io/components
Material for Java: http://www.jfoenix.com/
UI Design Example: https://www.youtube.com/watch?v=4vTc6UZcIH4
Accessibility ideas: https://gov.uk/ & https://www.youtube.com/watch?v=JHaLzm-FGsc

5. Glossary

- Event Coordinator (aka. “EC”) – User responsible for adding, editing, and deleting Events from the System. User in charge of the organisation of an event.
- Customer – User who contacts an Agent to make a booking for an Event.
- Agent – User responsible for liaising with Customers and creating Bookings.
- Staff – Users holding an EC or Agent account who interact directly with the system.
- Event – The collection of details Concert, Party, Premiere etc.
- Booking (Verb) – The process of booking an event, undertaken by an Agent - confirmation is received after request is accepted by the System.
- Booking (Noun) - The output of the booking process, stored within the system’s database.
- Venue – Location booked for an event.
- Account Login – Validation process for a given email and password.
- Database – Records all information relating to Events, Users and Bookings.
- Ticket – Physical or digital information given to a Customer which allows access to an Event.
- TicketType (TT)– Description of a type of Ticket attached to an event e.g. VIP, Standard.
- Sundry – An additional item attached to an Event, available for purchase by a Customer when making a Booking.

APPENDIX

A1. Application Screenshots to show UI changes

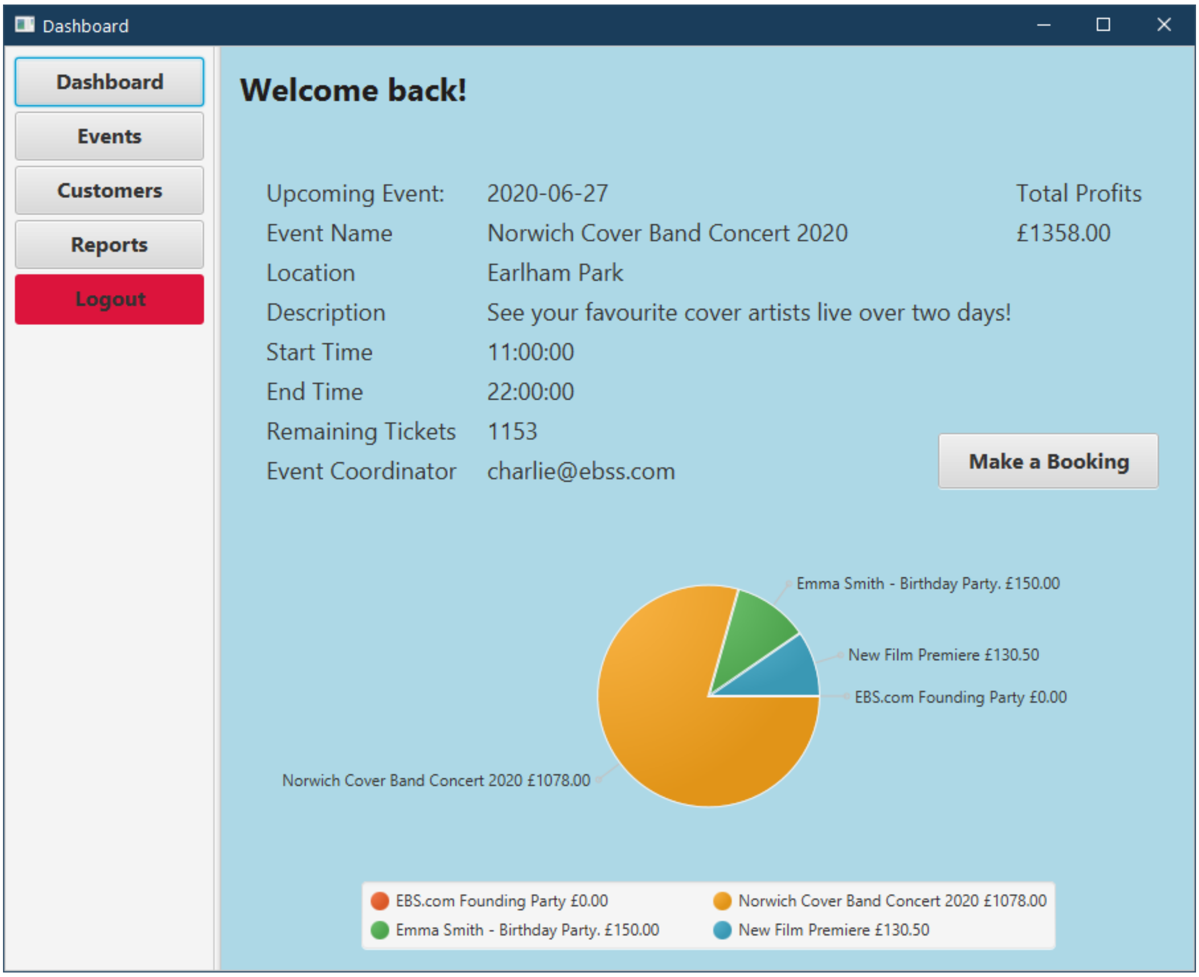


User Login

Username: adam@ebss.com

Password: •••

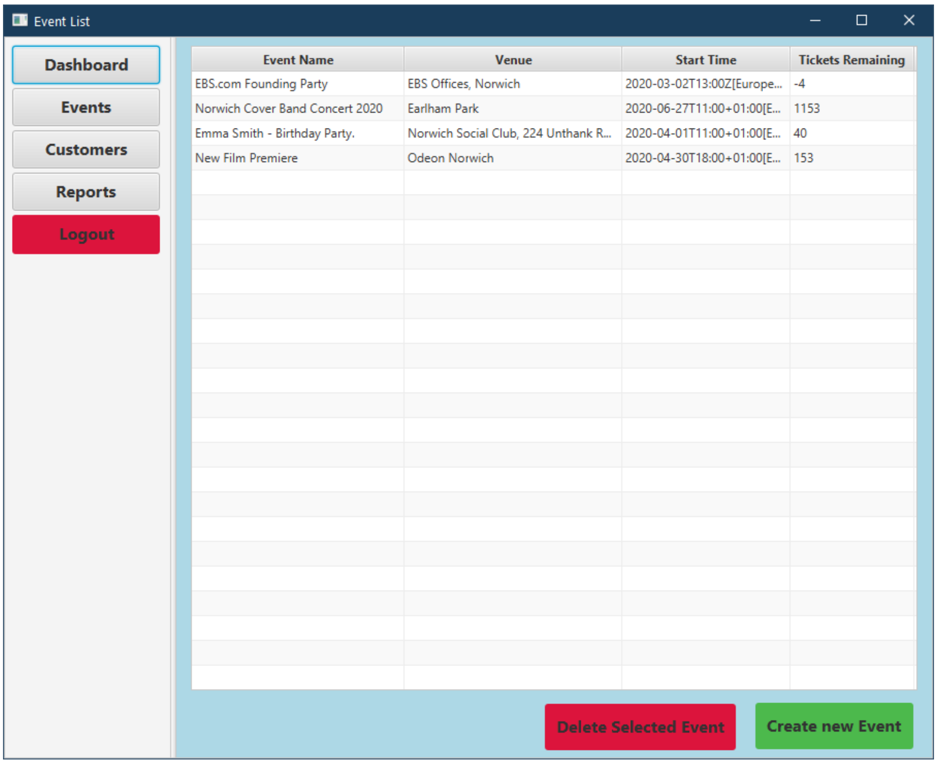
Figure 1 - Login Screen showing larger fonts, new Login button design and password masking.


Dashboard
Dashboard
Events
Customers
Reports
Logout
Make a Booking
Emma Smith - Birthday Party. £150.00
New Film Premiere £130.50
EBS.com Founding Party £0.00
Norwich Cover Band Concert 2020 £1078.00
● EBS.com Founding Party £0.00
● Norwich Cover Band Concert 2020 £1078.00
● Emma Smith - Birthday Party. £150.00
● New Film Premiere £130.50

Welcome back!

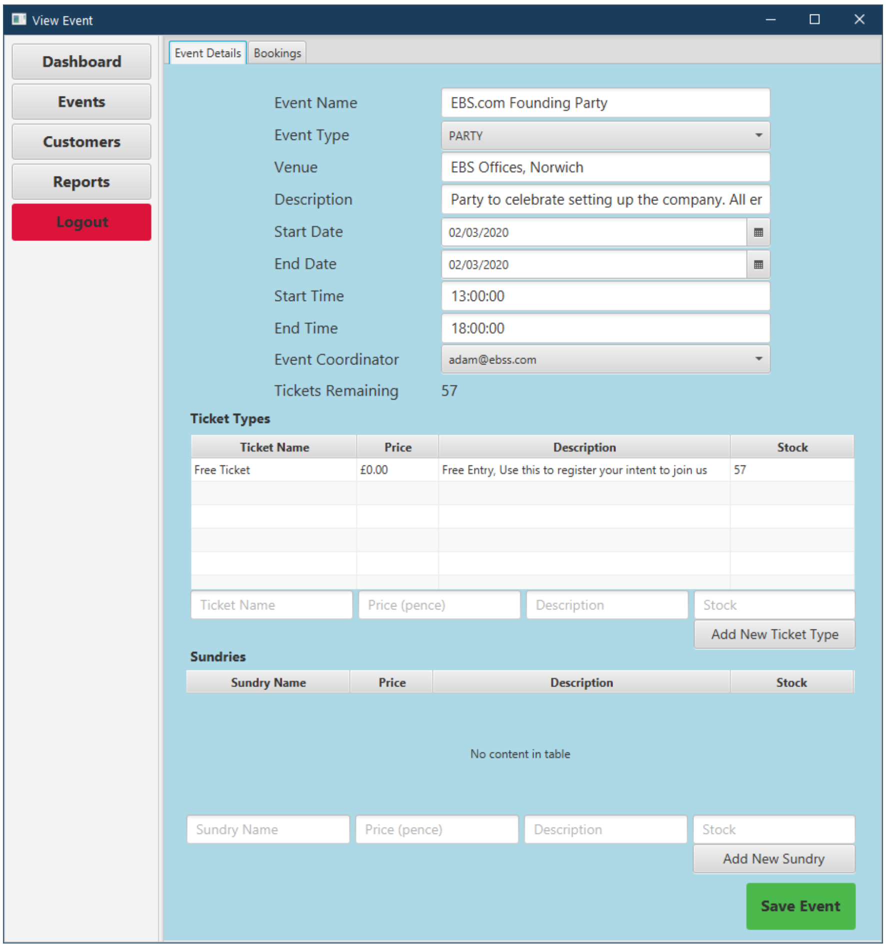
Upcoming Event:	2020-06-27	Total Profits
Event Name	Norwich Cover Band Concert 2020	£1358.00
Location	Earlham Park	
Description	See your favourite cover artists live over two days!	
Start Time	11:00:00	
End Time	22:00:00	
Remaining Tickets	1153	
Event Coordinator	charlie@ebss.com	

Figure 2 - Dashboard when logged in as an Event Coordinator. Note the larger font and use of colour in the left bar.



Event Name	Venue	Start Time	Tickets Remaining
EBS.com Founding Party	EBS Offices, Norwich	2020-03-02T13:00Z[Europe...	-4
Norwich Cover Band Concert 2020	Earlham Park	2020-06-27T11:00+01:00[E...	1153
Emma Smith - Birthday Party.	Norwich Social Club, 224 Unthank R...	2020-04-01T11:00+01:00[E...	40
New Film Premiere	Odeon Norwich	2020-04-30T18:00+01:00[E...	153

Figure 3 - Events List screen showing larger, colour-coded buttons.



Event Name	EBS.com Founding Party
Event Type	PARTY
Venue	EBS Offices, Norwich
Description	Party to celebrate setting up the company. All er
Start Date	02/03/2020
End Date	02/03/2020
Start Time	13:00:00
End Time	18:00:00
Event Coordinator	adam@ebss.com
Tickets Remaining	57

Ticket Types

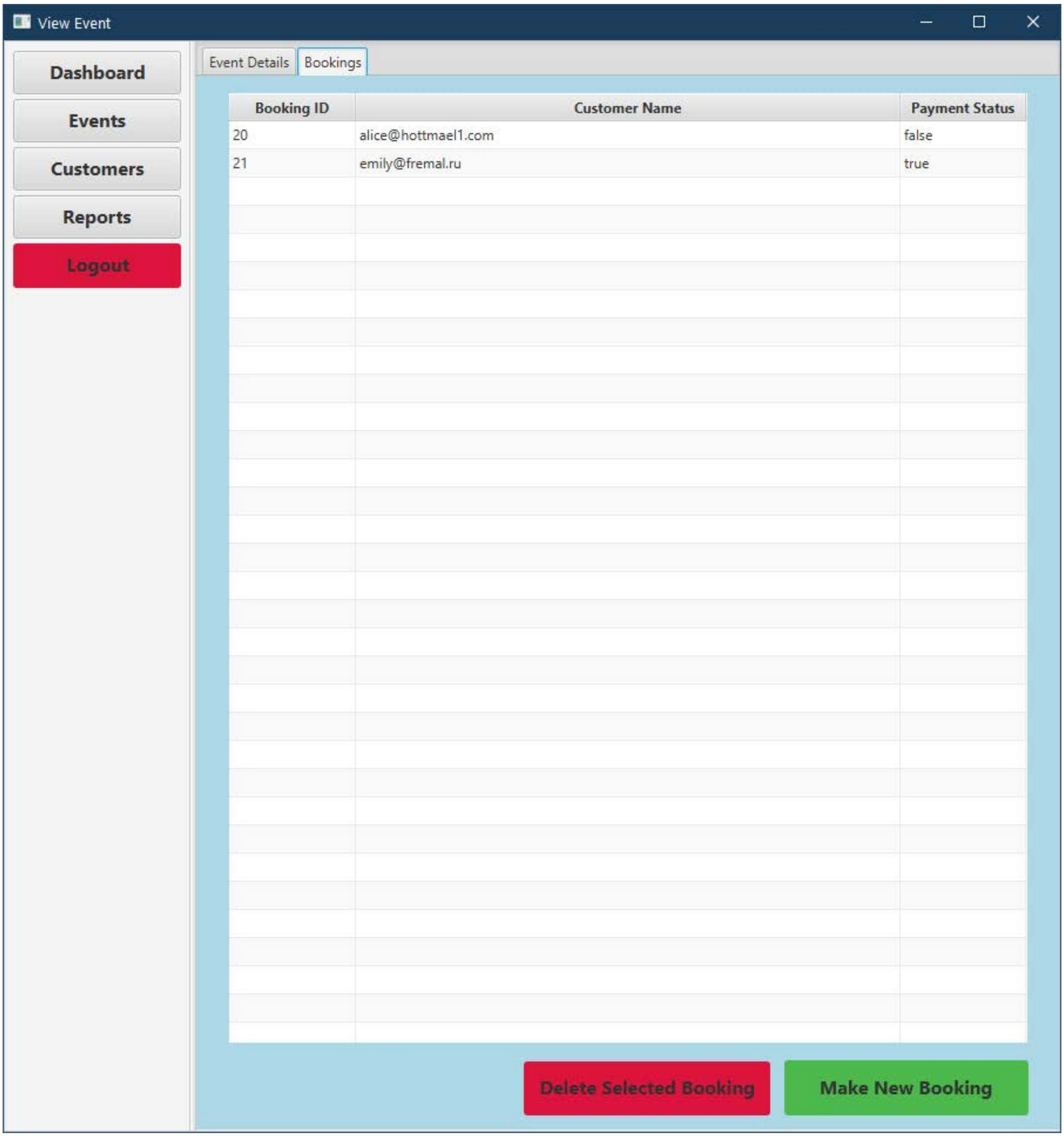
Ticket Name	Price	Description	Stock
Free Ticket	£0.00	Free Entry, Use this to register your intent to join us	57

Sundries

Sundry Name	Price	Description	Stock

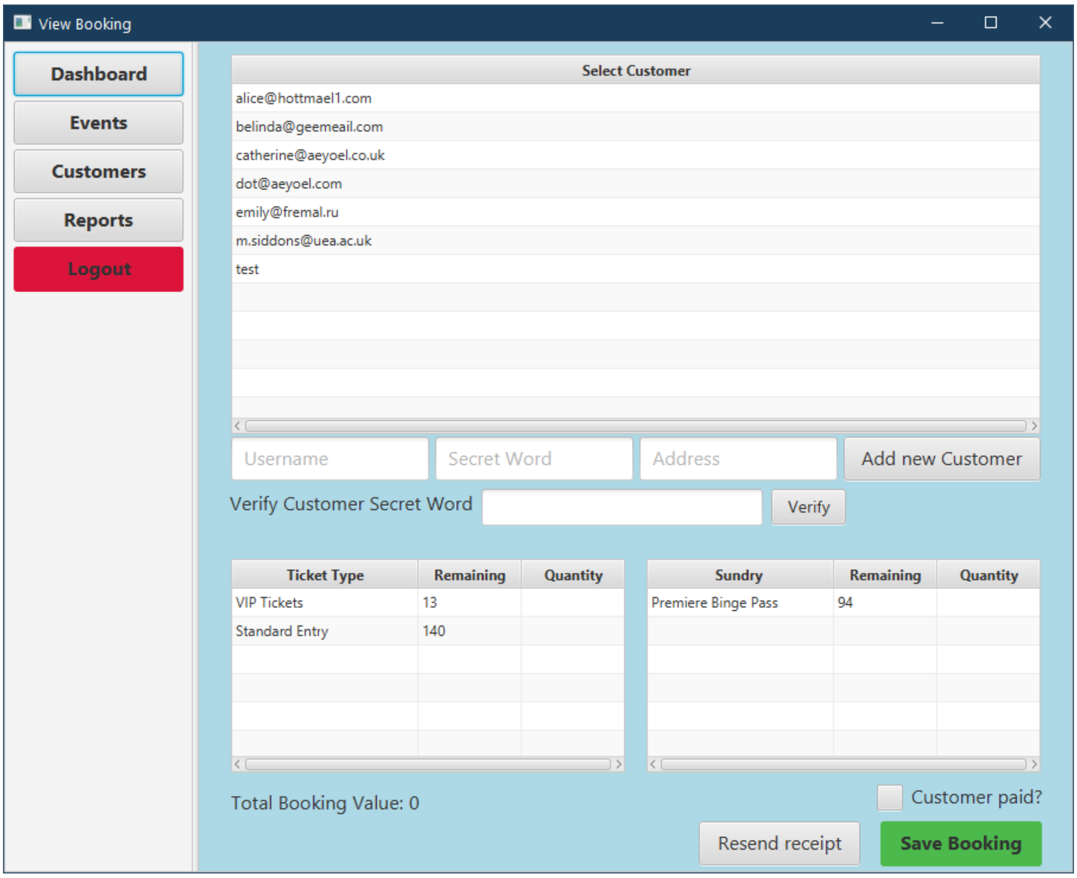
No content in table

Figure 4 - Edit Event screen. The fonts and buttons are larger overall, and the save button is now colour-coded. The Add New Event screen is the same as this.



Booking ID	Customer Name	Payment Status
20	alice@hottmael1.com	false
21	emily@fremal.ru	true

Figure 5 - Bookings List. Again, showing larger, colour-coded buttons.



Select Customer
alice@hottmael1.com
belinda@geemeail.com
catherine@aeyoel.co.uk
dot@aeyoel.com
emily@fremal.ru
m.siddons@uea.ac.uk
test

Verify Customer Secret Word

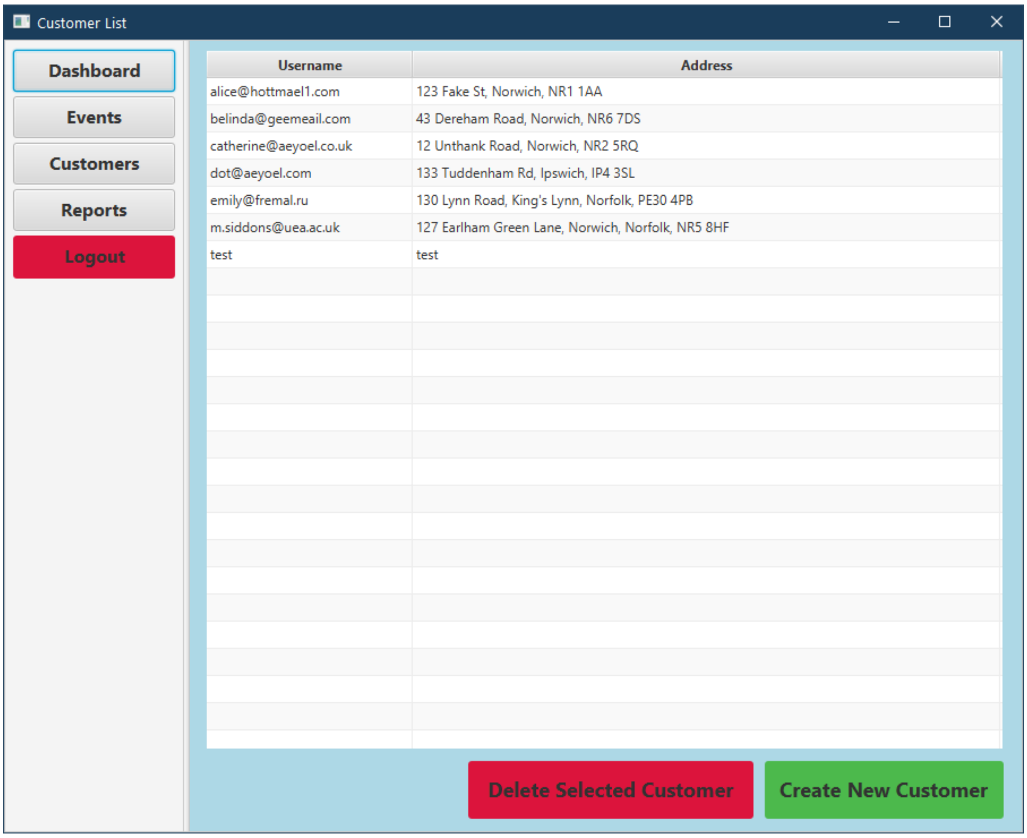
Ticket Type	Remaining	Quantity
VIP Tickets	13	
Standard Entry	140	

Sundry	Remaining	Quantity
Premiere Binge Pass	94	

Total Booking Value: 0

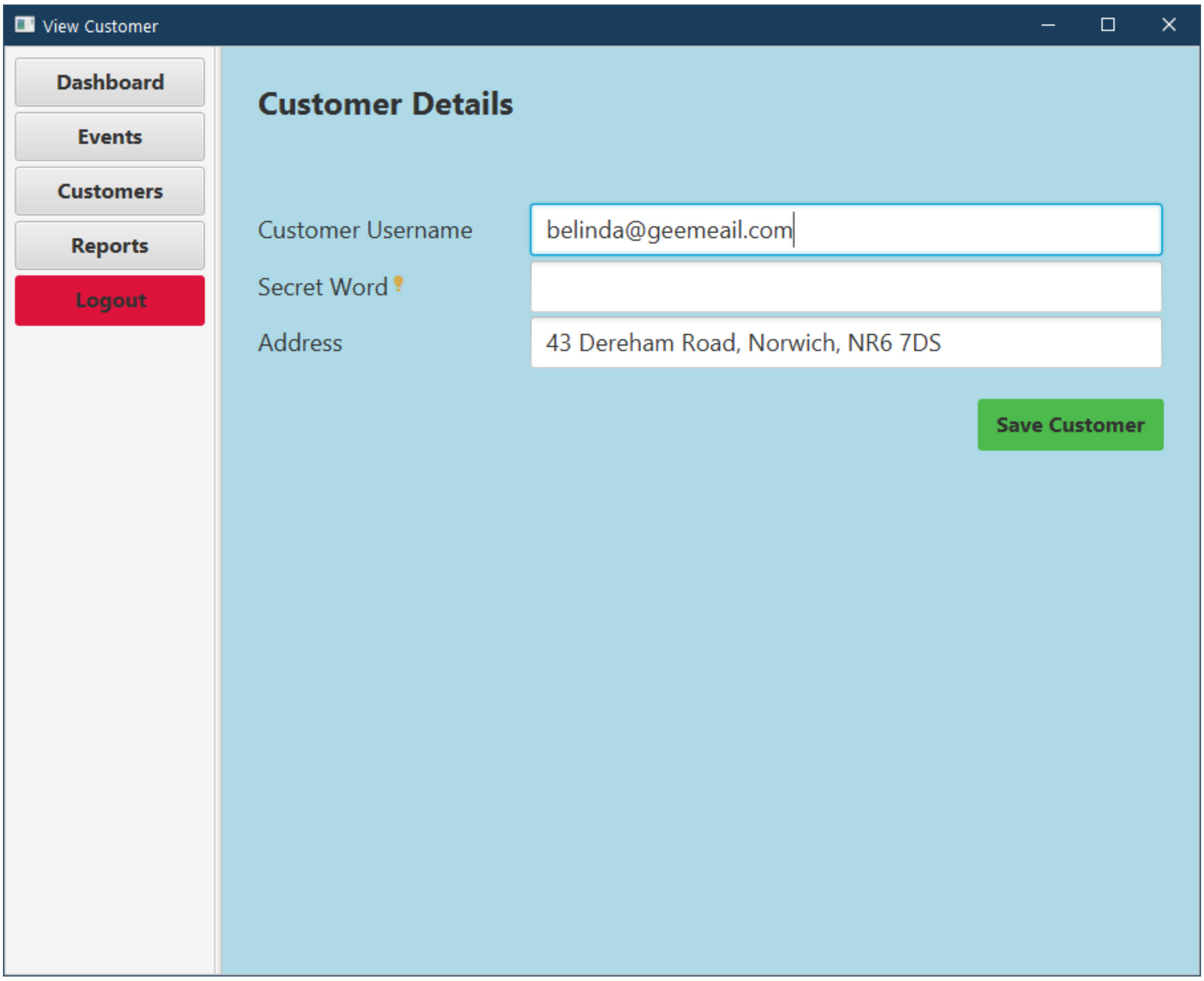
□ Customer paid?

Figure 6 - New Booking screen. All elements have been increased in size where possible and spacing adjusted. This also shows the new Customer Verification feature.



Username	Address
alice@hottmael1.com	123 Fake St, Norwich, NR1 1AA
belinda@geemeail.com	43 Dereham Road, Norwich, NR6 7DS
catherine@aeyoel.co.uk	12 Unthank Road, Norwich, NR2 5RQ
dot@aeyoel.com	133 Tuddenham Rd, Ipswich, IP4 3SL
emily@fremal.ru	130 Lynn Road, King's Lynn, Norfolk, PE30 4PB
m.siddons@uea.ac.uk	127 Earlham Green Lane, Norwich, Norfolk, NR5 8HF
test	test

Figure 7 - Customer List screen. Buttons changed here again to match the rest of the application.



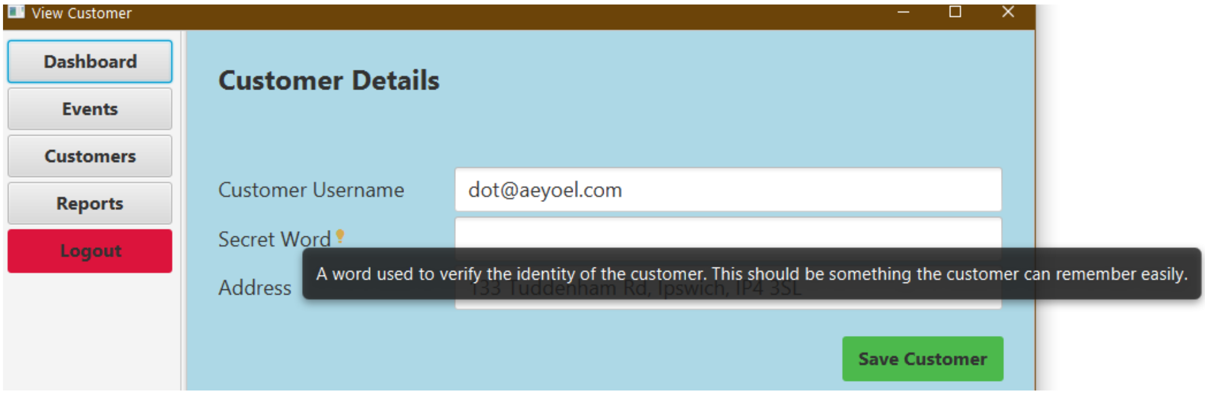
Customer Details

Customer Username: belinda@geemeail.com

Secret Word:

Address: 43 Dereham Road, Norwich, NR6 7DS

Figure 8 - Customer Edit screen. Elements repositioned and increased in size, and title added. Mouseover added to "secret word" label with additional information. The Add New Customer screen is the same but with empty fields.



Customer Details

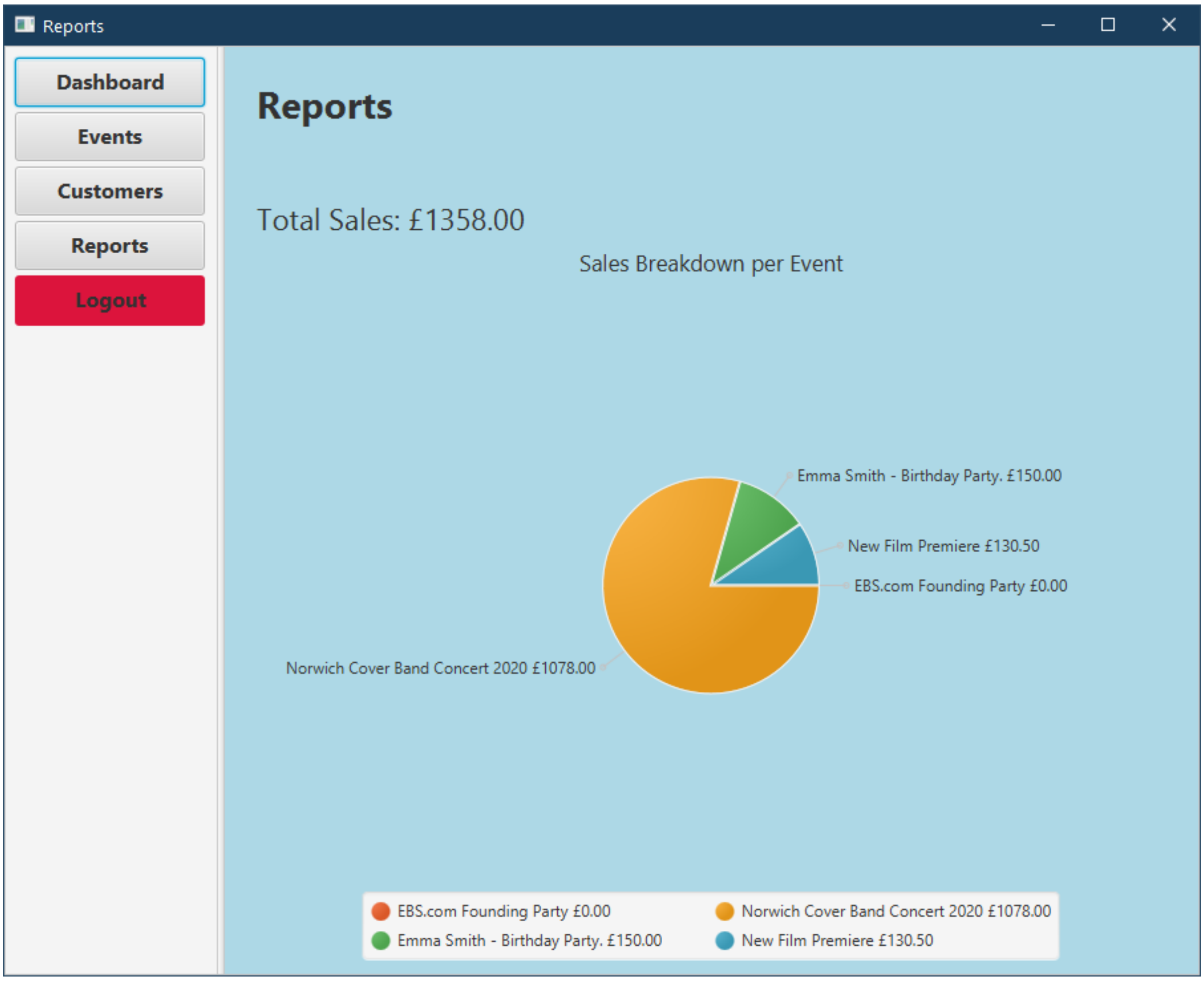
Customer Username: dot@aeyoel.com

Secret Word:

Address: 133 Tuddenham Rd, Ipswich, IP4 3SL

A word used to verify the identity of the customer. This should be something the customer can remember easily.

Figure 9 - Customer Add/Edit Mouseover. The mouseover popup that appears explains what Secret Word means.


Reports
Dashboard
Events
Customers
Reports
Logout
Sales Breakdown per Event
Emma Smith - Birthday Party. £150.00
New Film Premiere £130.50
EBS.com Founding Party £0.00
Norwich Cover Band Concert 2020 £1078.00
● EBS.com Founding Party £0.00
● Norwich Cover Band Concert 2020 £1078.00
● Emma Smith - Birthday Party. £150.00
● New Film Premiere £130.50

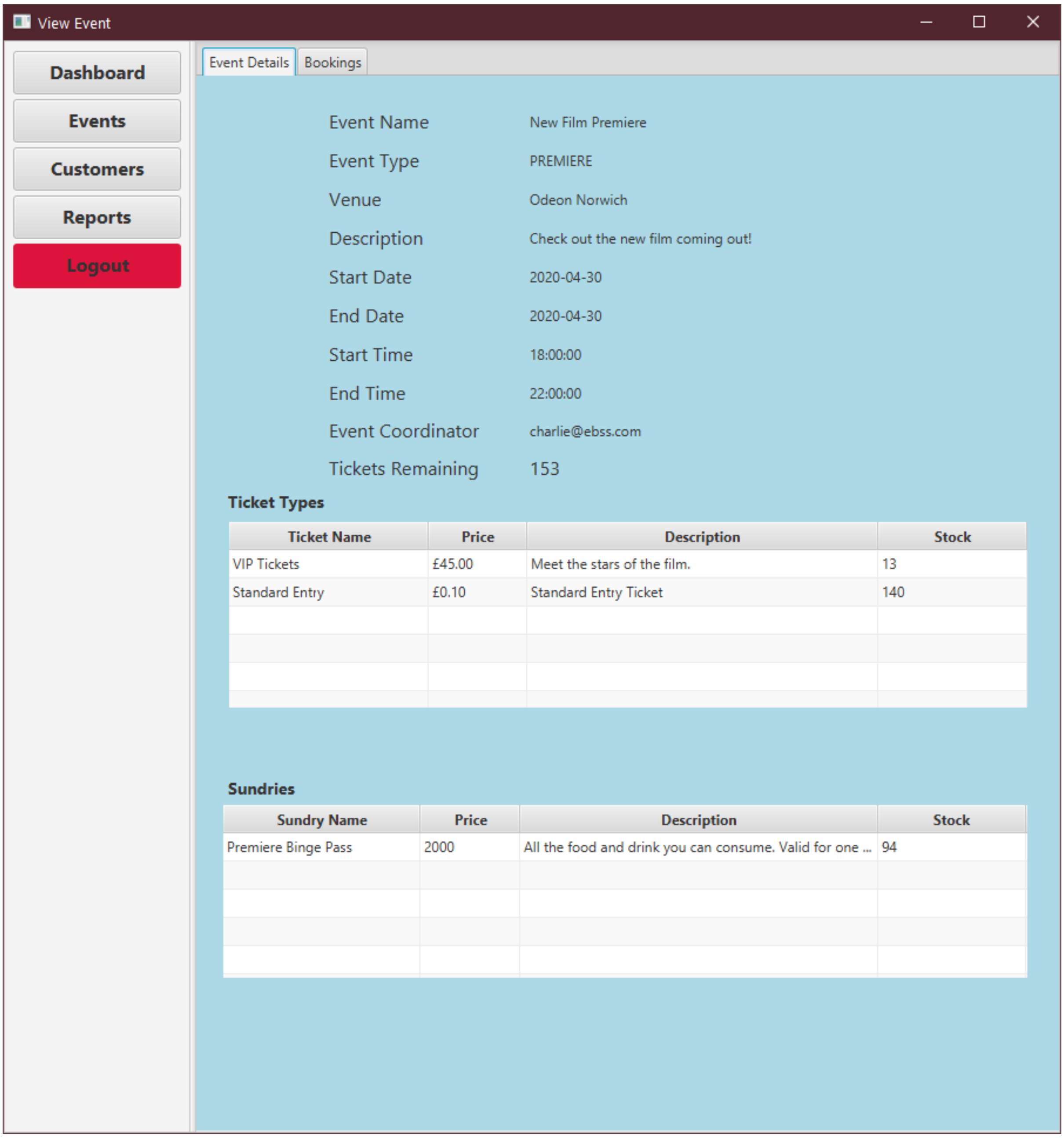
Reports

Total Sales: £1358.00

Figure 10 - Reports screen. This has not changed for Stage 2 but is featured here for completeness.

A2. Event View as Agent

The Event view screen changes when logged in as an agent to ensure that the fields cannot be edited and the buttons to submit changes or delete the event are removed. Compare Figure 4 to the screenshot below.



Event Name New Film Premiere

Event Type PREMIERE

Venue Odeon Norwich

Description Check out the new film coming out!

Start Date 2020-04-30

End Date 2020-04-30

Start Time 18:00:00

End Time 22:00:00

Event Coordinator charlie@ebss.com

Tickets Remaining 153

Ticket Types

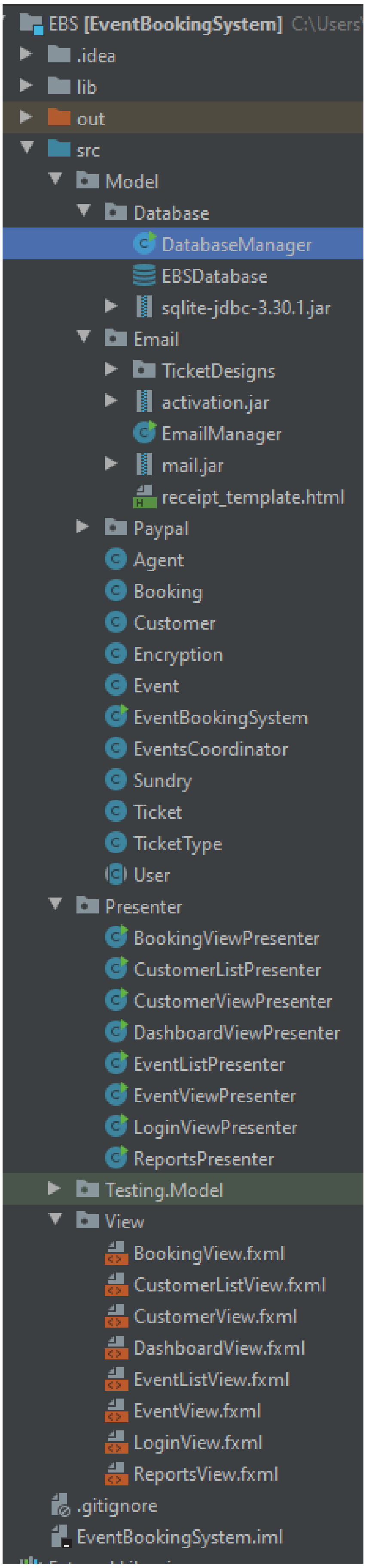
Ticket Name	Price	Description	Stock
VIP Tickets	£45.00	Meet the stars of the film.	13
Standard Entry	£0.10	Standard Entry Ticket	140

Sundries

Sundry Name	Price	Description	Stock
Premiere Binge Pass	2000	All the food and drink you can consume. Valid for one ...	94

Figure 11 – View Event screen when logged in as an Agent

A3. Folder Organisation



```
EBS [EventBookingSystem] C:\Users\
  .idea
  lib
  out
  src
    Model
      Database
        DatabaseManager
        EBSDatabase
        sqlite-jdbc-3.30.1.jar
      Email
        TicketDesigns
        activation.jar
        EmailManager
        mail.jar
        receipt_template.html
      Paypal
      Agent
      Booking
      Customer
      Encryption
      Event
      EventBookingSystem
      EventsCoordinator
      Sundry
      Ticket
      TicketType
      User
    Presenter
      BookingViewPresenter
      CustomerListPresenter
      CustomerViewPresenter
      DashboardViewPresenter
      EventListPresenter
      EventViewPresenter
      LoginViewPresenter
      ReportsPresenter
    Testing.Model
    View
      BookingView.fxml
      CustomerListView.fxml
      CustomerView.fxml
      DashboardView.fxml
      EventListView.fxml
      EventView.fxml
      LoginView.fxml
      ReportsView.fxml
  .gitignore
  EventBookingSystem.iml
```

As noted on the feedback (F5.5), the application code for Stage 1 was not organised very well which made it hard to find the files you wanted. The project has now been refactored into Model, View, Presenter and Testing folders, with specific files for Database and Email found in the Model folder. The new layout is shown in detail on the left.

This makes it far easier to find the files needed within the project, especially between those in the Presenter and the Model as they all have the same icon.

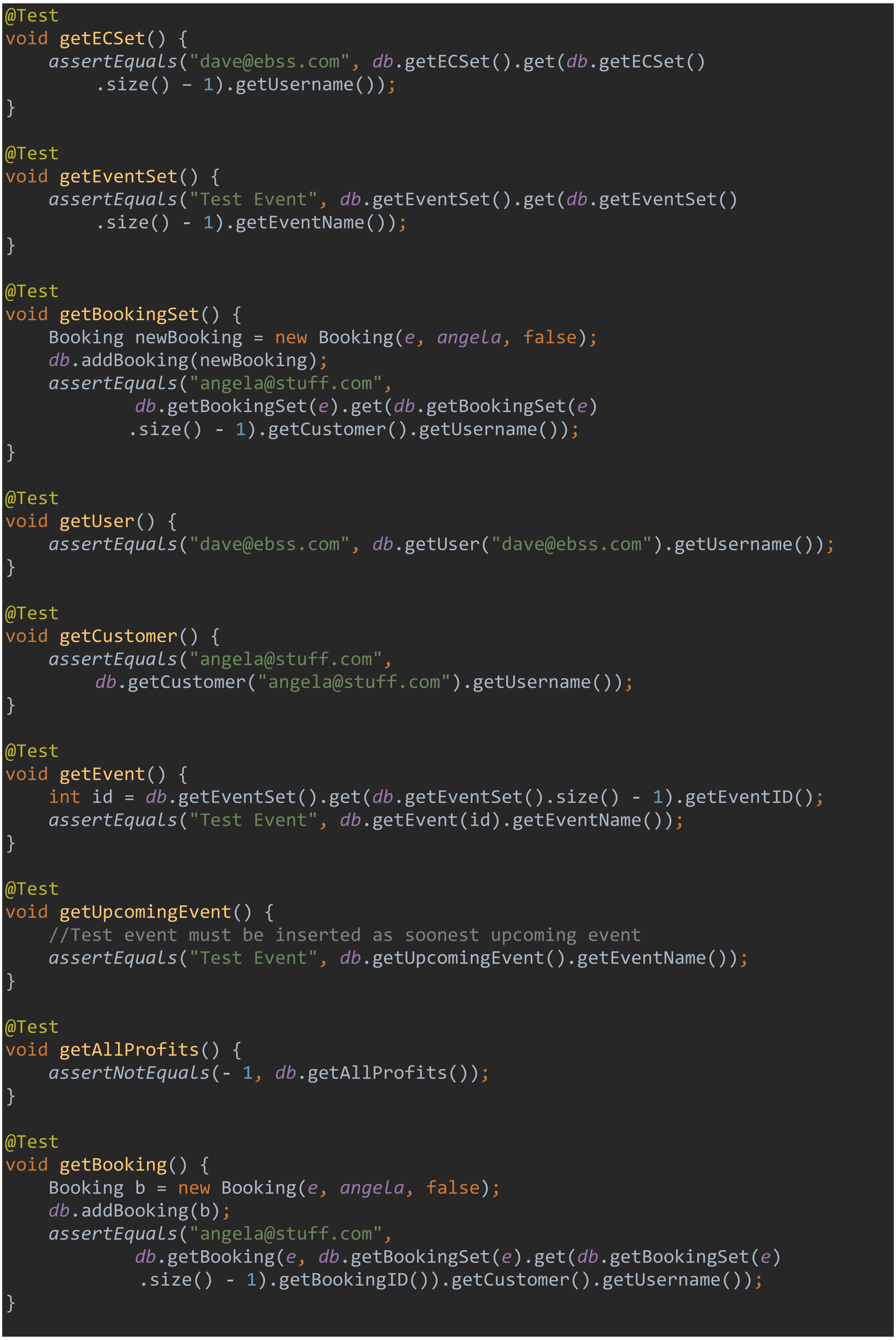
A4. White Box Testing Code

A4.1. DatabaseManager Test Setup

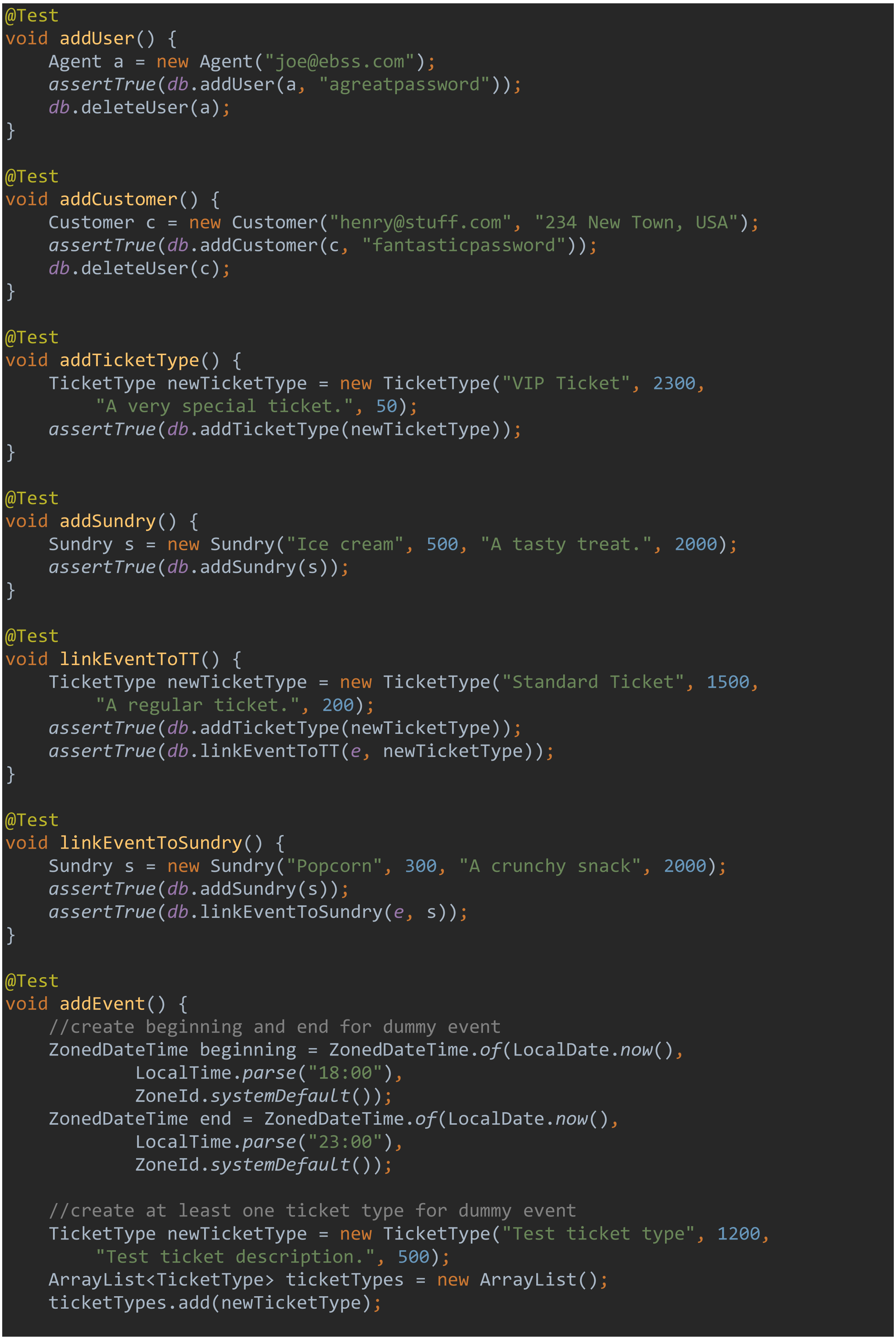
```
class DatabaseManagerTest {

    private static DatabaseManager   db    = new DatabaseManager();
    private static EventsCoordinator dave = new
        EventsCoordinator("dave@ebss.com");
    private static Agent             rachel = new Agent("rachel@ebss.com");
    private static Customer          angela = new Customer("angela@stuff.com",
        "123 Fake St.");
    private static Event             e;
    private static Booking           newBooking;

    @BeforeEach
    void beforeEach() {
        db.addUser(dave, "password");
        db.addUser(rachel, "thisisapassword");
        db.addCustomer(angela, "supersecurepassword");

        //create beginning and end for dummy event
        ZonedDateTime beginning = ZonedDateTime.of(LocalDate.now(),
                LocalTime.parse("18:00"),
                ZoneId.systemDefault());
        ZonedDateTime end = ZonedDateTime.of(LocalDate.now(),
                LocalTime.parse("23:00"),
                ZoneId.systemDefault());

        //create at least one ticket type for dummy event
        TicketType newTicketType = new TicketType("Test ticket type", 1200,
                 "Test ticket description.", 500);
        ArrayList<TicketType> ticketTypes = new ArrayList();
        ticketTypes.add(newTicketType);

        //create an optional sundry for dummy event
        Sundry newSundry = new Sundry("Test sundry", 500,
                 "Test sundry description", 500);
        ArrayList<Sundry> sundries = new ArrayList();
        sundries.add(newSundry);

        e = new Event("Norwich Forum", beginning, end, "A test event.",
                 "Test Event", Event.EventType.OTHER,
                ticketTypes, sundries, dave);
        db.addEvent(e);
    }
```

A4.2. DatabaseManager unit tests

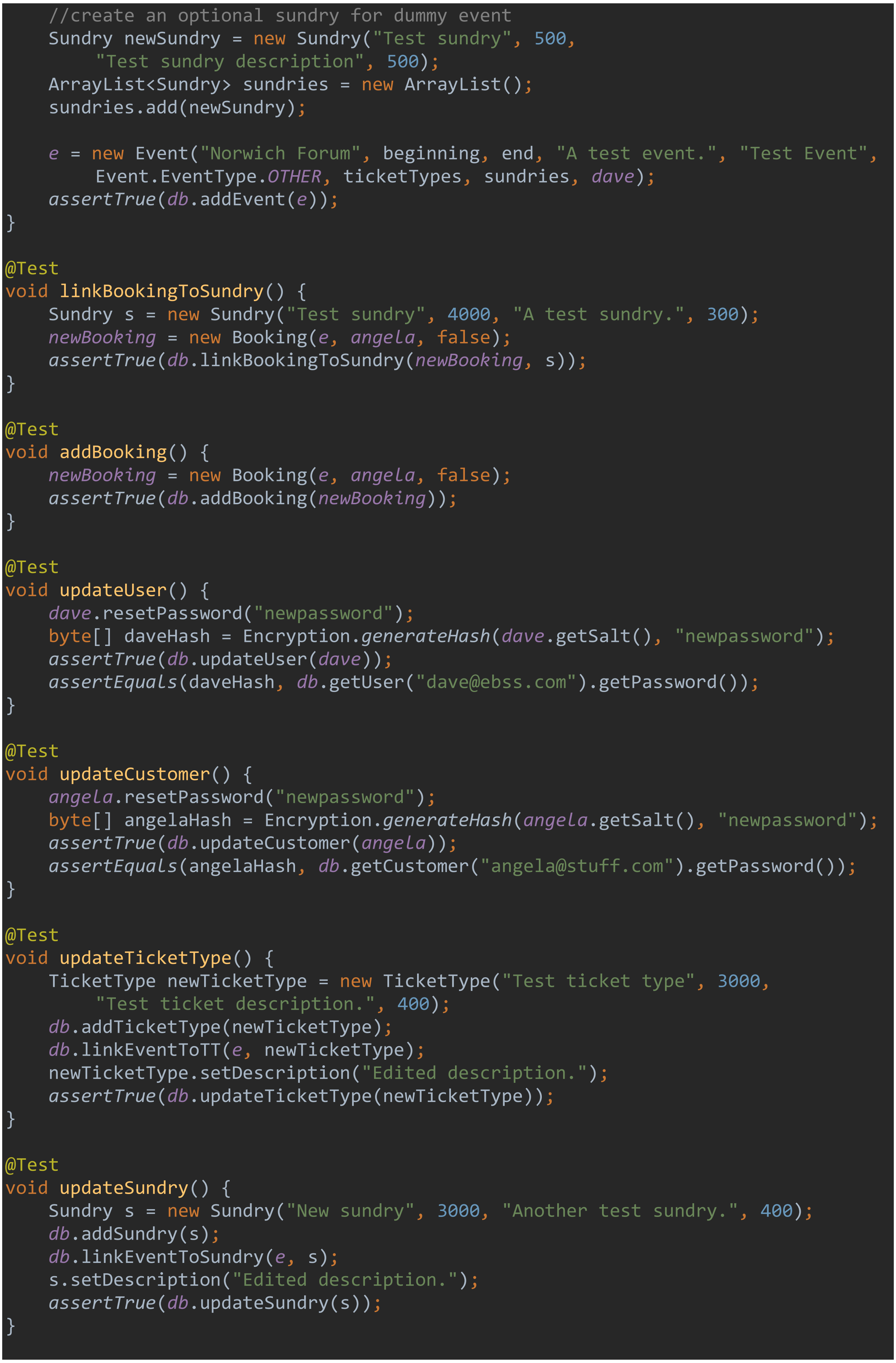
```
@Test
void authenticateUser() {
    assertNotNull(db.getUser("dave@ebss.com"));
}

@Test
void getCustomerSet() {
    assertEquals("angela@stuff.com",
        db.getCustomerSet().get(db.getCustomerSet().size() - 1).getUsername());
}
```



```
@Test
void getECSet() {
    assertEquals("dave@ebss.com", db.getECSet().get(db.getECSet()
        .size() - 1).getUsername());
}

@Test
void getEventSet() {
    assertEquals("Test Event", db.getEventSet().get(db.getEventSet()
        .size() - 1).getEventName());
}

@Test
void getBookingSet() {
    Booking newBooking = new Booking(e, angela, false);
    db.addBooking(newBooking);
    assertEquals("angela@stuff.com",
            db.getBookingSet(e).get(db.getBookingSet(e)
           .size() - 1).getCustomer().getUsername());
}

@Test
void getUser() {
    assertEquals("dave@ebss.com", db.getUser("dave@ebss.com").getUsername());
}

@Test
void getCustomer() {
    assertEquals("angela@stuff.com",
        db.getCustomer("angela@stuff.com").getUsername());
}

@Test
void getEvent() {
    int id = db.getEventSet().get(db.getEventSet().size() - 1).getEventID();
    assertEquals("Test Event", db.getEvent(id).getEventName());
}

@Test
void getUpcomingEvent() {
    //Test event must be inserted as soonest upcoming event
    assertEquals("Test Event", db.getUpcomingEvent().getEventName());
}

@Test
void getAllProfits() {
    assertNotEquals(- 1, db.getAllProfits());
}

@Test
void getBooking() {
    Booking b = new Booking(e, angela, false);
    db.addBooking(b);
    assertEquals("angela@stuff.com",
            db.getBooking(e, db.getBookingSet(e).get(db.getBookingSet(e)
            .size() - 1).getBookingID()).getCustomer().getUsername());
}
```



```
@Test
void addUser() {
    Agent a = new Agent("joe@ebss.com");
    assertTrue(db.addUser(a, "agreatpassword"));
    db.deleteUser(a);
}

@Test
void addCustomer() {
    Customer c = new Customer("henry@stuff.com", "234 New Town, USA");
    assertTrue(db.addCustomer(c, "fantasticpassword"));
    db.deleteUser(c);
}

@Test
void addTicketType() {
    TicketType newTicketType = new TicketType("VIP Ticket", 2300,
        "A very special ticket.", 50);
    assertTrue(db.addTicketType(newTicketType));
}

@Test
void addSundry() {
    Sundry s = new Sundry("Ice cream", 500, "A tasty treat.", 2000);
    assertTrue(db.addSundry(s));
}

@Test
void linkEventToTT() {
    TicketType newTicketType = new TicketType("Standard Ticket", 1500,
        "A regular ticket.", 200);
    assertTrue(db.addTicketType(newTicketType));
    assertTrue(db.linkEventToTT(e, newTicketType));
}

@Test
void linkEventToSundry() {
    Sundry s = new Sundry("Popcorn", 300, "A crunchy snack", 2000);
    assertTrue(db.addSundry(s));
    assertTrue(db.linkEventToSundry(e, s));
}

@Test
void addEvent() {
    //create beginning and end for dummy event
    ZonedDateTime beginning = ZonedDateTime.of(LocalDate.now(),
            LocalTime.parse("18:00"),
            ZoneId.systemDefault());
    ZonedDateTime end = ZonedDateTime.of(LocalDate.now(),
            LocalTime.parse("23:00"),
            ZoneId.systemDefault());

    //create at least one ticket type for dummy event
    TicketType newTicketType = new TicketType("Test ticket type", 1200,
        "Test ticket description.", 500);
    ArrayList<TicketType> ticketTypes = new ArrayList();
    ticketTypes.add(newTicketType);
```



```
    //create an optional sundry for dummy event
    Sundry newSundry = new Sundry("Test sundry", 500,
        "Test sundry description", 500);
    ArrayList<Sundry> sundries = new ArrayList();
    sundries.add(newSundry);

    e = new Event("Norwich Forum", beginning, end, "A test event.", "Test Event",
        Event.EventType.OTHER, ticketTypes, sundries, dave);
    assertTrue(db.addEvent(e));
}

@Test
void linkBookingToSundry() {
    Sundry s = new Sundry("Test sundry", 4000, "A test sundry.", 300);
    newBooking = new Booking(e, angela, false);
    assertTrue(db.linkBookingToSundry(newBooking, s));
}

@Test
void addBooking() {
    newBooking = new Booking(e, angela, false);
    assertTrue(db.addBooking(newBooking));
}

@Test
void updateUser() {
    dave.resetPassword("newpassword");
    byte[] daveHash = Encryption.generateHash(dave.getSalt(), "newpassword");
    assertTrue(db.updateUser(dave));
    assertEquals(daveHash, db.getUser("dave@ebss.com").getPassword());
}

@Test
void updateCustomer() {
    angela.resetPassword("newpassword");
    byte[] angelaHash = Encryption.generateHash(angela.getSalt(), "newpassword");
    assertTrue(db.updateCustomer(angela));
    assertEquals(angelaHash, db.getCustomer("angela@stuff.com").getPassword());
}

@Test
void updateTicketType() {
    TicketType newTicketType = new TicketType("Test ticket type", 3000,
        "Test ticket description.", 400);
    db.addTicketType(newTicketType);
    db.linkEventToTT(e, newTicketType);
    newTicketType.setDescription("Edited description.");
    assertTrue(db.updateTicketType(newTicketType));
}

@Test
void updateSundry() {
    Sundry s = new Sundry("New sundry", 3000, "Another test sundry.", 400);
    db.addSundry(s);
    db.linkEventToSundry(e, s);
    s.setDescription("Edited description.");
    assertTrue(db.updateSundry(s));
}
```

```
@Test
void updateEvent() {
    e.setDescription("Edited description.");
    assertTrue(db.updateEvent(e));
}

@Test
void updateBooking() {
    Booking b = new Booking(e, angela, false);
    db.addBooking(b);
    assertTrue(db.updateBooking(b));
}

@Test
void deleteUser() {
    assertTrue(db.deleteUser(rachel));
}

@Test
void deleteBooking() {
    Booking b = new Booking(e, angela, false);
    db.addBooking(b);
    assertTrue(db.deleteBooking(db.getBookingSet(e).get(db.getBookingSet(e)
        .size() - 1).getBookingID()));
}

@Test
void deleteEvent() {
    assertTrue(db.deleteEvent(e));
}
```

A4.3. DatabaseManager Test Cleanup

```
@AfterEach
void afterEach() {
    db.deleteUser(dave);
    db.deleteUser(rachel);
    db.deleteUser(angela);
    db.deleteEvent(e);
}
```

A4.4. User Test Setup

```
class UserTest {
    private static DatabaseManager   db     = new DatabaseManager();
    private static EventsCoordinator dave   = new
        EventsCoordinator("dave@ebss.com");
    private static Agent             rachel = new Agent("rachel@ebss.com");
    private static Customer          angela = new Customer("angela@stuff.com",
         "123 Fake St.");

    @BeforeAll
    static void beforeAll() {
        db.addUser(dave, "password");
        db.addUser(rachel, "thisisapassword");
        db.addCustomer(angela, "supersecurepassword");
    }
```

A4.5. User Unit Tests

```java
@Test
void resetPassword() {
    dave.resetPassword("password123");
    byte[] davePwd = Encryption.generateHash(dave.salt, "password123");
    assertEquals(davePwd, dave.getPassword());

    rachel.resetPassword("password123");
    byte[] rachelPwd = Encryption.generateHash(rachel.salt, "password123");
    assertEquals(rachelPwd, rachel.getPassword());

    angela.resetPassword("password123");
    byte[] angelaPwd = Encryption.generateHash(angela.salt, "password123");
    assertEquals(angelaPwd, angela.getPassword());
}

@Test
void authenticate() {
    User u = User.authenticate("dave@ebss.com", "password123");
    assertNotNull(u);
    assertEquals("dave@ebss.com", u.getUsername());
}
```

A4.6. User Test Clean-up

```java
@AfterAll
static void afterAll() {
    db.deleteUser(dave);
    db.deleteUser(rachel);
    db.deleteUser(angela);
}
```

A5. Code Snippets

A5.1. DatabaseManager.java : deleteBooking()

```
public boolean deleteBooking(int bookingID) {
    // Delete from booking, booking_sundry and ticket in one transaction
    try(Connection conn = this.connect()){
        conn.setAutoCommit(false);

        String sql = "DELETE FROM booking WHERE bookingID = ?";
        PreparedStatement prep = conn.prepareStatement(sql);
        prep.setInt(1, bookingID);
        prep.executeUpdate();

        sql = "DELETE FROM booking_sundry WHERE bookingID = ?";
        prep = conn.prepareStatement(sql);
        prep.setInt(1, bookingID);
        prep.executeUpdate();

        sql = "DELETE FROM ticket WHERE bookingID = ?";
        prep = conn.prepareStatement(sql);
        prep.setInt(1, bookingID);
        prep.executeUpdate();

        conn.commit();
        conn.setAutoCommit(true);
        return true;
    }
    catch(SQLException ex){
        System.out.println("Error: " + ex.getMessage());
    }
    return false;
}
```

A5.2. DatabaseManager.java : deleteEvent()

```
public boolean deleteEvent(Event e) {
    // First get all the bookings for this event and delete them.
    ArrayList<Booking> bookings = new ArrayList<>(getBookingSet(e));
    for (Booking b : bookings){
        deleteBooking(b.getBookingID());
    }

    // Next get lists of sundryIDs and tickettypeIDs related to this Event.
    ArrayList<Sundry> sundries = e.getSundries();
    ArrayList<TicketType> ticketTypes = e.getTicketTypes();

    // Now we can delete event_sundry, sundry, event_tickettype, tickettype
    // and event entries.
    try (Connection conn = this.connect()){
        conn.setAutoCommit(false);

        String sql = "DELETE FROM event_sundry WHERE eventID = ?";
        PreparedStatement prep = conn.prepareStatement(sql);
        prep.setInt(1, e.getEventID());
        prep.executeUpdate();

        for (Sundry s : sundries) {
            sql  = "DELETE FROM sundry WHERE sundryID = ?";
            prep = conn.prepareStatement(sql);
            prep.setInt(1, s.getSundryID());
            prep.executeUpdate();
        }

        sql = "DELETE FROM event_tickettype WHERE eventID = ?";
        prep = conn.prepareStatement(sql);
        prep.setInt(1, e.getEventID());
        prep.executeUpdate();

        for (TicketType tt : ticketTypes) {
            sql  = "DELETE FROM tickettype WHERE ticketTypeID = ?";
            prep = conn.prepareStatement(sql);
            prep.setInt(1, tt.getTicketTypeID());
            prep.executeUpdate();
        }

        sql = "DELETE FROM event WHERE eventID = ?";
        prep = conn.prepareStatement(sql);
        prep.setInt(1, e.getEventID());
        prep.executeUpdate();

        conn.commit();
        conn.setAutoCommit(true);
        return true;
    }
    catch (SQLException ex) {
        System.out.println("Error: " + ex.getMessage());
    }
    return false;
}
```

A5.3. BookingViewPresenter : verifySecretWord()

```
public void verifySecretWord(){
    Customer c = customerTable.getSelectionModel().getSelectedItem();
    if (c != null){ ;
        byte[] providedPwd = Encryption.generateHash(c.getSalt(),
                 secretWord.getText());
        boolean match = true;
        // compare each byte of the hash of the provided password with the
        password held in the DB.
        for (int i = 0; i < 16; i++) {
            if (providedPwd[i] != c.getPassword()[i]){
                match = false;
            }
        }
        if (match){
            verifyStatus.setText("Secret Word Verified");
            verified = true;
        }
        else{
            verifyStatus.setText("Secret Word not verified");
            verified = false;
        }
    }
    else{
        verifyStatus.setText("Please select a Customer");
    }
}
```

A5.4. BookingViewPresenter : SaveBookingButtonPressed() snippet

```
else if (!verified) {
    successMessage.setText("Please verify customer Secret Word.");
}
```