Watchalong

Software Design Specification

5.0

3/27/2024

Group 11
Manan Shukla
Dylan Rambo
Micah Miller

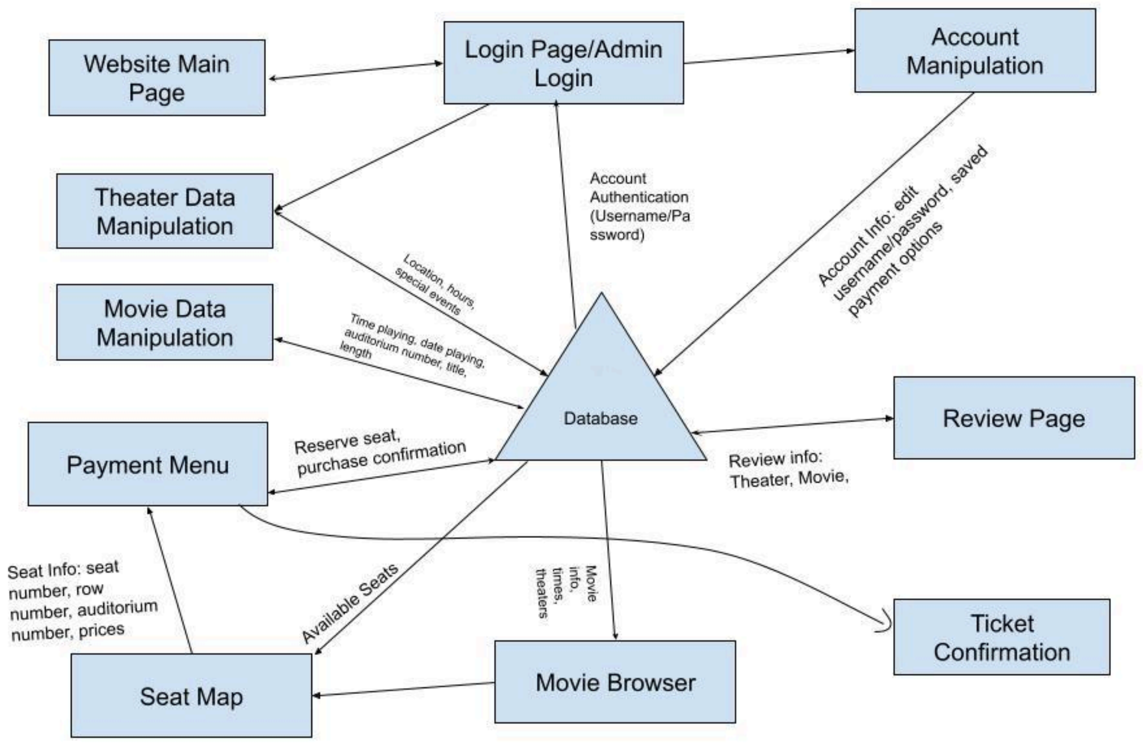
Prepared for
CS 250- Introduction to Software Systems
Instructor: Gus Hanna, Ph.D.
Spring 2024

1. System Description

The entire software system consists of multiple pages for the user to view and use, as well as a database that will be heavily utilized. These pages include a website main page, login page for regular customers and theater admins, account manipulation to edit account information, a movie browser, movie seat map, payment menu, ticket confirmation, review page, and theater/movie data manipulation for admins.

2. Software Architecture Overview

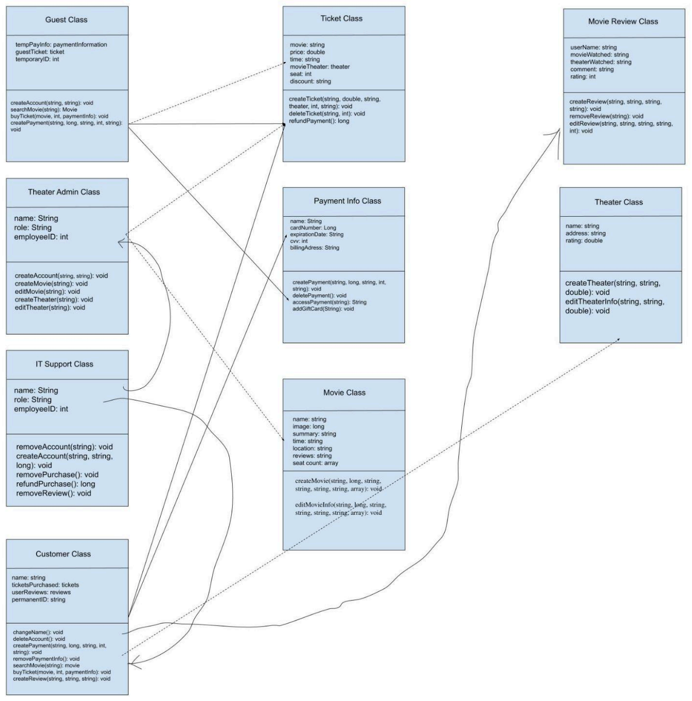
2.1 SWA Diagram


Website Main Page
Login Page/Admin Login
Account Manipulation
Theater Data Manipulation
Account Authentication (Username/Password)
Account Info: edit username/password, saved payment options
Location, hours, special events
Movie Data Manipulation
Time playing, date playing, auditorium number, title, length
Database
Review Page
Payment Menu
Reserve seat, purchase confirmation
Review info: Theater, Movie,
Seat Info: seat number, row number, auditorium number, prices
Available Seats
Movie info, times, theaters
Ticket Confirmation
Seat Map
Movie Browser

2.1.1 Description

The overall design of our software's architecture is heavily dependent on communicating with our database. The software will have 11 components that work together to bring a working theater ticketing system to a customer. The first component is the website's main page. This will mainly be a user interface meant to display to the user what the software actually does. From the main page the user will be able to navigate to the login page or admin login page depending on who is trying to access their login. Once the user enters their username and password the software will bring the associated fields from the database to compare and will authenticate the user if they match. Once successfully logged in, the user will be able to manipulate their existing account information such as their username, password, and saved payment options. This data will be updated in the database by the software. A user may wish to browse movies to watch at a local theater. In this case, the movie browser will take information from the database about available movies to watch, such as a synopsis, showing times, and theater locations. Once an option is clicked on a seat map will be displayed. The seat map will take information from the database about available seats such as seat number, row number, auditorium number, and prices for each seat. That information will be transferred to the payment menu once the user decides to buy a ticket(s). The payment menu will prompt the user for their payment information where the database will send information back to the software to check if the payment has been processed. Then the software will update the available seats in the database. Once the seats have been confirmed to be reserved and paid for, the software will send a confirmation message to the user. Separate from the other pages for purchasing tickets is the review page. Information about past reviews will be brought from the database to the review page. Reviews made by a user will be sent to the database so that it is added to the rest of the reviews. The last two components are the theater and movie manipulation pages for admins. Both will take existing data from the database in case an admin wants to edit existing theater or movie information. New movies or theaters can also be created by admins which will be updated in the database. The information to be transferred between these pages and the database are theater location, theater hours, theater special events, movie times/dates, auditorium number, length of movie, and title.

2.2 UML Class Diagram


Guest Class
tempPayInfo: paymentInformation
guestTicket: ticket
temporaryID: int
createAccount(string, string): void
searchMovie(string): Movie
buyTicket(movie, int, paymentInfo): void
createPayment(string, long, string, int, string): void
Ticket Class
movie: string
price: double
time: string
movieTheater: theater
seat: int
discount: string
createTicket(string, double, string, theater, int, string): void
deleteTicket(string, int): void
refundPayment(): long
Movie Review Class
userName: string
movieWatched: string
theaterWatched: string
comment: string
rating: int
createReview(string, string, string, string): void
removeReview(string): void
editReview(string, string, string, string, int): void
Theater Admin Class
name: String
role: String
employeeID: int
createAccount(string, string): void
createMovie(string): void
editMovie(string): void
createTheater(string): void
editTheater(string): void
Payment Info Class
name: String
cardNumber: Long
expirationDate: String
cvv: int
billingAdress: String
createPayment(string, long, string, int, string): void
deletePayment(): void
accessPayment(string): String
addGiftCard(String): void
Theater Class
name: string
address: string
rating: double
createTheater(string, string, double): void
editTheaterInfo(string, string, double): void
IT Support Class
name: String
role: String
employeeID: int
removeAccount(string): void
createAccount(string, string, long): void
removePurchase(): void
refundPurchase(): long
removeReview(): void
Movie Class
name: string
image: long
summary: string
time: string
location: string
reviews: string
seat count: array
createMovie(string, long, string, string, string, string, array): void
editMovieInfo(string, long, string, string, string, string, array): void
Customer Class
name: string
ticketsPurchased: tickets
userReviews: reviews
permanentID: string
changeName(): void
deleteAccount(): void
createPayment(string, long, string, int, string): void
removePaymentInfo(): void
searchMovie(string): movie
buyTicket(movie, int, paymentInfo): void
createReview(string, string, string): void

2.2.1 Description of Classes/Attributes and Functions

2.2.1.1 Guest Class

This class will allow a user to perform operations on this website without an account

2.2.1.1.1 field variables:

- Holds a variable for tempPayInfo which holds a paymentInformation object. This field variable will be holding an object of the paymentInformation class. It will be deleted once the Guest session is ended along with the rest of the class.
- The guestTicket field variable holds a ticket object. This will be created upon purchase.
- The guestClass will have a temporaryID field variable to be used in the ticketBuying operation.

2.2.1.1.2 operations:

- The createAccount operation will take in strings for name and role. This exists to give the guest an option to create a permanent account.
- The searchMovie function will take a string for the movie name and search the database for the specified movie. Then if found it will return a Movie object.
- The buyTicket operation takes in a movie class, an integer for the userID, and a string for the payment information. This will return a ticket object and an email with ticket information when successful.
- The createPayment operation creates a payment information object. It will take in a string for the name on the card, a long for the card number, a string for the expirationDate, an int for the CVV, and a string for the billingAddress. This will create a payment information object.

2.2.1.1.3 dependencies:

- The guest uses the ticketClass because it will create ticket classes associated with the account
- The guest class uses the payment class to create an object to temporarily store payment info.

2.2.1.2 Theater Admin Class

This class will allow movie theater employees to have an account mainly used to perform operations that update movie information and movie theater information

2.2.1.2.1 field variables:

- The name variable is a string holding the name of the Admin user.
- The role variable is a string that will hold the status of the employee.
- The employee ID variable is an int that will hold a unique number to identify the employee.

2.2.1.2.2 operations:

The class will have several operations.

- The createAccount function will create a theater admin object.
- The createMovie function will take in a string and create a movie object in the database. The editMovie will be able to change the movie info.
- createTheater will take in a string and create a theater object.
- editTheater will update theater info.

2.2.1.2.3 dependencies:

- The theater admin class updates the theater and movie classes.

2.2.1.3 IT Support Class

This class allows IT support staff to fix problems such as adding and removing accounts and refunding purchases.

2.2.1.3.1 field variables:

- The name variable is a string that holds the name of the IT support user.
- The role variable is a string that holds the role of the IT user.
- The employee ID variable is an int that will hold a unique number to identify the IT user.

2.2.1.3.2 operations:

- The removeAccount function will take in a string of the account to remove and will delete the object.
- The createAccount function will take in the necessary info and create an account object.
- the removePurchase function will delete a ticket object.
- The refundPurchase function returns a long of the money being refunded.
- *The removeReview function is a void function that allows IT support to remove a review*

2.2.1.3.3 dependencies:

- This class uses the Theater admin and customer classes because they have user rights to manipulate and create account information as needed.

This class also uses the ticketClass to refund purchases.

2.2.1.4 Payment Info Class

This class will take in the user payment information when they want to buy movie tickets.

2.2.1.4.1 field variables:

- Hold a string variable for name on the card
- Holds a long for the card number
- Holds a string for the expiration date
- Holds an int for CVV.
- Holds a string for a billing address.

2.2.1.4.2 operations:

- The createPayment operation is a constructor for this class that will take in a string, long, string, int, and a string for the field variables.
- deletePayment is an operation that will remove a paymentInfo object and all of its information.
- accessPayment will return a string with the requested information.

- *addPaymentGiftCard will be a void function that will take in a string to add a gift card as a payment method*

2.2.1.4.3 dependencies:

- This class is not really dependent on other classes. It is a core class that other classes rely on.

2.2.1.5 Customer Class

This class allows a general user a more permanent account to use that saves past movie information and allows the user to leave reviews.

2.2.1.5.1 field variables:

- The name variable is a string that holds the name of the customer.
- The tickets purchased variable is of type tickets.
- The user reviews variable is of type reviews.
- The permanent ID variable is a string that will identify the user.

2.2.1.5.2 operations:

- The changeName function will allow the user to update their username.
- The deleteAccount function will allow the user to delete their account.
- The createPayment function takes in the relevant banking info and creates an object.
- removePayment will delete the object.
- The searchMovie function will take in a string of the movie name and return a movie object.
- The buyTicket function takes in the move, cost, and payment info and creates a ticket object.
- The createReview function will take in strings of the review and create an object.

2.2.1.5.3 dependencies:

- This class is dependent on the Ticket class because it creates objects of the ticket class upon purpose.
- This class is dependent on the payment info class because it saves a field variable of the payment info class
- This class is dependent on the movie review class because this class creates objects of the movie review class.

2.2.1.6 Movie Class

This class is how movies will be stored in the database and will hold information about the movie

2.2.1.6.1 field variables:

- Holds a string for the name of the movie
- Holds a long to store the binary for the image of the movie cover
- Holds a string for the summary of the movie

- Holds a string for the time of the movie
- Holds a string for the location of the movie
- Holds a string for the reviews of the movie
- Holds a 2 dimensional array to represent the taken seats and the vacant seats

2.2.1.6.2 operations:

- The createMovie operation takes in a string, long, string, string, string, array to update the mentioned the variables above when creating an object
- The editMovieInfo takes in a string, long, string, string, string, array to update the mentioned variables above.

2.2.1.6.3 dependencies:

- This class is not dependent on other classes but many classes are dependent on this class.

2.2.1.7 Movie Review Class

This class will be associated with movie objects and be created by customer class objects.

2.2.1.7.1 field variables:

- The userName variable is a string that holds the name of the user leaving the review.
- The movieWatched variable is a string that holds the name of the movie being reviewed.
- The theaterWatched variable is a string that holds the theater the user saw the movie at.
- The comment variable is a string holding the review the user is leaving.
- The rating variable is an int holding the number rating.

2.2.1.7.2 operations:

- The createReview function takes in strings and creates an object of the review.
- The removeReview function deletes the instance of the review.
- *editReview is a void method and takes in values for all of the field variables in order to edit an existing review*

2.2.1.7.3 dependencies:

- This class isn’t dependent on any other classes.

2.2.1.8 Ticket Class

This class will be created every time a user purchases a ticket and stores relevant information about the purchase.

2.2.1.8.1 field variables:

- The movie variable is a string holding the name of the movie that the ticket is for.
- The price variable is a double that holds the price of the ticket.
- The time variable is a string variable that holds the time of the movies showing.
- The seat variable is an int holding the seat number.
- The discount variable is a string holding the discount code if used.

2.2.1.8.2 operations:

- The createTicket function takes in the relevant ticket info and creates a ticket object with it.
- The deleteTicket function deletes objects created by the createTicket.
- The refundPayment function returns a long of the ticket price being refunded.

2.2.1.8.3 dependencies:

- This class is not dependent on other classes but other classes are dependent on it.

2.2.1.9 Theater Class

The theater class will be manipulated by movie theater employees and will store relevant information about the theater.

2.2.1.9.1 field variables:

- Holds a string to holds the name of the theater
- Holds a string for the address of the theater
- Holds a double for the rating of the movie

2.2.1.9.2 operations:

- The createTheater operation takes in a string, string, and double to populate the variables mentioned above.
- The editTheaterInfo takes in a string, string, and double to update the variables mentioned above.

2.2.1.9.3 dependencies:

- The class is not dependent on other classes but is a core class that other classes.

3. Development Plan and Timeline

There will be three teams of developers on this project.
Group 1 will be led by Manan and will handle the guest, IT support, and customer classes.
Group 2 will be led by Micah and will handle the ticket, payment info, and movie classes.
Group 3 will be led by Dylan and will handle the theater admin, movie review class, and theater class.
Each group will take 2-3 weeks to complete their classes including the implementation of the attributes and functions.
After this period each group will complete testing on their classes and will have to implement new code if needed.
This period of testing will take 10 weeks.
All three groups will reconvene and test the entire system for a period of 2 weeks.

4. Verification Test Plan

4.1 Introduction

4.1.1 Purpose
The purpose of this test plan is to verify the functionality of all aspects of the software so that the requirements can be fulfilled. Tests will be conducted to target at the unit, functional, and system levels.

4.1.2 Scope
The test plan covers the entire architecture of the system, including the website main page, login page, account manipulation, theater data manipulation, movie data manipulation, review page, movie browser, search bar, seat map, and payment menu.

4.2 Test Objectives:
The goal of the testing process is to catch any immediate problems as well as identify areas that may pose future problems. At the most fundamental level, the unit tests seek to ensure each section of the code performs its intended task which is separate from the overall functionality of the program that the user experiences. For example, one of the unit tests, tests whether or not the theater admin class properly updates the theater objects name. This type of testing can be done in the software and does not need to be done via the actual software. The next stage of testing, functional testing, will seek to ensure that the program functions well from the level of using the software as a user would without looking at the code. This is a type of black box testing. Testing should be performed by looking at all the functions a user can perform and attempting to use them as intended and even as not intended to see how the software performs. Examples include things like creating a user account and then attempting to make updates to user info such as the username and password. Success would be seen in-website. The final stage of testing would be system testing which would seek to test whether all functions of the system work in unison. This includes things such as testing whether the bank api for payments works and whether the system can handle a large amount of traffic. This stage of testing might need to be performed at both the user and coding level. For example high amounts of traffic could be simulated using test objects whereas testing whether the software works on different platforms would be performed from the UI level. At each level of testing, attempts should be made to break the software or system to ensure things will run smoothly. For example testing weird inputs to see whether the software allows it should be done. If done improperly, this could slip by and not cause any compiling errors but it could break the functionality of the system. This is why both unit and functional testing need to be done.

4.3 Test Environment
4.3.1 Hardware:
These tests will take place on mobile devices, laptops, and desktops. However among these devices of various different types will be tested that operate off different operating systems. This will be addressed in more detail in section 4.3.2. Laptops and desktops will both produce relatively the same results so main testing will be performed on a desktop computer. However extra cautionary testing will be performed on the laptop but it will be minimized to reduce time and costs.

4.3.2 Software:

Among mobile devices the main operating systems will be tested. So the website will be tested using the native browser of IOS devices, Android devices, and HarmonyOS that runs on Huawei devices. Among computers the website will be accessed on Microsoft Windows, Linux, and MacOS. Since browsers among these systems will produce relatively the same results we just need to make sure that the pages display properly among the three operating systems but main functionality testing will occur in Windows. Among browsers the main ones to be tested will be Google Chrome, Microsoft Edge, Microsoft Explorer, Firefox, Safari and Android browser. Tests will be performed within each of these browsers. One important note is that while tests will be performed on the devices, operating systems, and browsers listed above; the most in-depth and comprehensive testing will be performed on a desktop computer running a windows operating system and utilizing the google chrome browser.

4.4 Test Scenarios:

Our testing will need to be very extensive, however some of the specific testing scenarios will be described here.

Our first test case example is testing the payment menu. This is a unit test that tests whether payment info can only enter correctly such as the CCV only being an int. This can be done with a test object.

The next thing we will test is whether the createTheater function can update the database properly. This can also be done as a unit test with an object.

The edit movie test will see if the editMovieInfo function updates the movie object's field variables. This is a unit test and can be done with a theater admin object. A simple print output test with expected versus actual results could work for all the unit tests.

The final unit test example is to test whether the buyTicket Function works. This can be done within the guest and customer class and checking to see if a ticket object is created.

The changePasswordTest will be a functional test to see if from a user's perspective their password changes in the login info section. This can be done by acting as a user and seeing if the password shows up correctly.

The removePurchase_dataBase_update will be another functional test to see if an IT support user can delete a payment and the corresponding payment object gets deleted.

The movieSearchTest will be another functional test to see whether the search function works properly. This can be done by attempting every scenario a user might run into. This includes the tester trying stuff like entering a word, entering a phrase, and entering a genre. Any errors may require unit testing to resolve.
The next test is the seatAvailabilityUpdate test. This is a unit test to test the whole system functionality. When a user buys a ticket, the system needs to almost instantly update the database

to show that seat as taken so that if another user is trying to buy that ticket it will show up as taken. Huge issues would arise if a ticket that no longer is available is able to be purchased.

The accountInfoUpdate test will be a system test to ensure that when information is updated in one section of the system, it is concurrently updated in other areas. For example if a user leaves a review, and then updates their username, the old review should show as left under the updated username and not the old one.

Another test example is the ChangeofOS test. This is a third system test that will ensure that the software can be used equally well on any operating system. Further testing beyond these specifics will be necessary.

We will need to test that redundancies in the database do not occur so the system will not permit two of the exact same theater to be created. Testing will involve ensuring the system compares theater addresses to make sure duplicates do not exist
We will need to test that a valid email address is required for the user to make an account. An invalid email address should give the user some sort of message to inform them their email is invalid.
Guest class users should not be able to leave a review on any movie. The system must ensure this by checking and making sure that the ID type is not of the guest type before allowing them to leave a review.

A test scenario for account creation is to verify that a user should only be able to create an admin account as long as they have a valid admin ID. This test would be a unit test to make sure that only a valid ID already in the database can be used to create an admin account.

Testing should be done to ensure that the deleteAccount method wipes the corresponding account data from the database. Holding on to unnecessary information creates security risks for the user and company. This can be done by deleting a test account object and then attempting to access that information and hopefully not finding it.

When a user selects to leave a review the system will check if the user has a purchase for a ticket of that movie in their ticket history. If they do not they will be notified that they have not watched this movie so they cannot leave a review

Removing a review from the review page also needs to be tested. This would be a unit test to verify that the data from the review was deleted from the database. This would protect the user from having a permanent review on the website.

4.5 Testing Approach:

Much of the content held and displayed on this website is user generated. The accounts, the movies, and reviews are some of the core data on this system which is all heavily reliant on user input. With such being the case much of the testing will have to be done manually.
Some pre-programmed inputs can be developed and then executed simultaneously to check for edge cases where there are two conflicting inputs such as for buying the same seat. This can also be utilized to load test the website and ensure it operates smoothly with moderate levels of usage.
Testers will have to manually check much of the functionality such as systems for creating movie listings, theaters, leaving reviews, and editing accounts and purchase information.

4.6 Test Schedule:
Unit Tests will take approximately 1 week to complete. Bug fixes will be applied in the midst of testing.
Functional Tests will be performed afterwards and will take 4 weeks to complete. This ensures cohesion between the software modules. The first week will be used to identify errors within the program. The second week will be used to resolve issues found within the code. Then the last 2 weeks will be spent confirming the proper functionality of the code and fixing any smaller issues found during the testing. System Tests will be performed afterwards for the next 5 weeks. The first 2 weeks will be spent on administrative side testing. The next 2 weeks will be spent on client side testing. The last week will be spent ensuring systemwide communication operates properly.

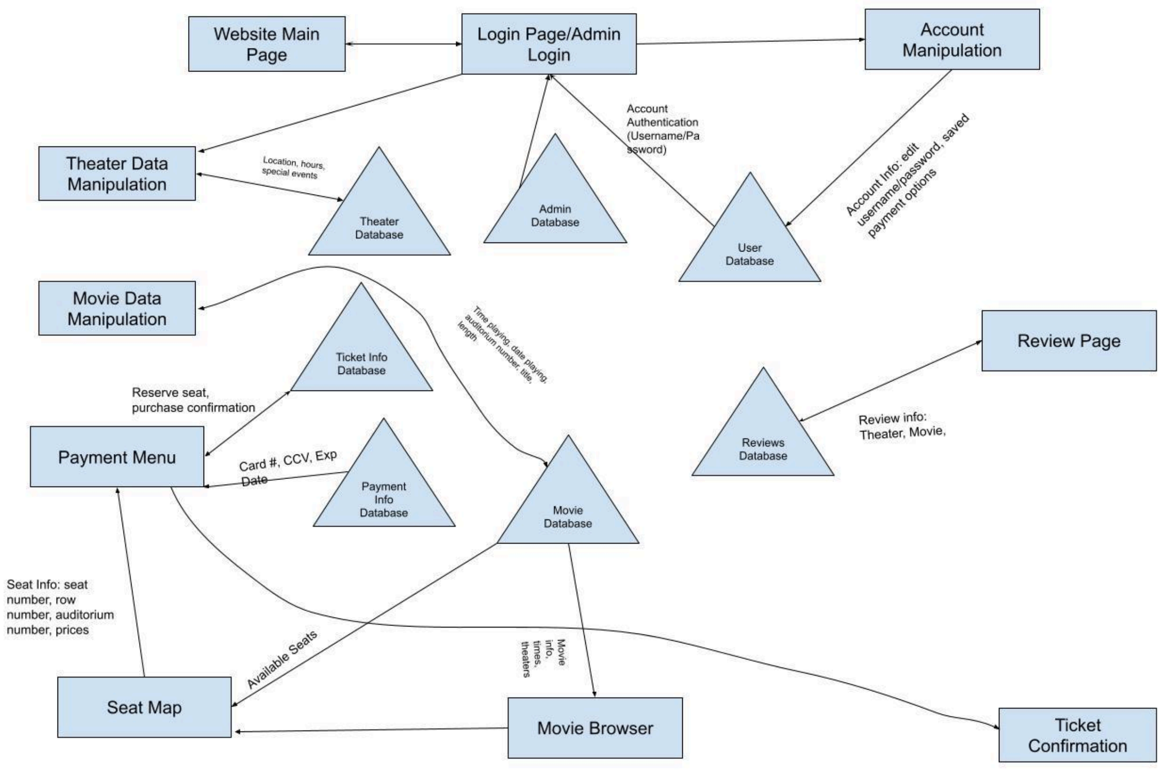
4.7 Risks and Mitigation:
One risk is receiving invalid payment information.
To solve this problem simple misinputs that do not meet the required amount of characters can be caught by the system. However invalid cards that are of proper length will need to be checked through a bank before a bank API before the transaction is finalized.
Another risk is with identical account creation. If hypothetically two users were to create an account with the same name and password how would the system differentiate the two at login time. The solution to this issue is having the userID that is associated with every user account to be required to be entered at login. This way everyone has a unique identifier and no information could completely overlap.
Another risk is when people purchase the same seat. If two users bought the same seat at about the exact same time there would be issues. This issue can be solved by having a condition that if two users bought a seat at the same time and neither of them was warned that the other already bought the seat then while the payment for the seat is pending the system will look at the userID for both customers and whoever's userID has the lower value will get the seat and the other will have their transaction canceled and be notified that the seat was taken.
One more risk is with payment security. There is no simple solution to this problem. The only proper way to manage the risks is to keep payment information in a smaller database that is much more secure and has much higher restricted access. Along with that it will have to be maintained with up to date security patches.

4.8 Conclusion:

After the steps depicted in the verification test plan are completed the website will be in a state where there is a high confidence level for the system to execute as intended. Any major issues should not arise. However with the increase in usage, less severe issues may be found that can be resolved with the help of the IT class and later fully patched by the development team.

5. Data Management Strategy

5.1 Updated SWA Diagram

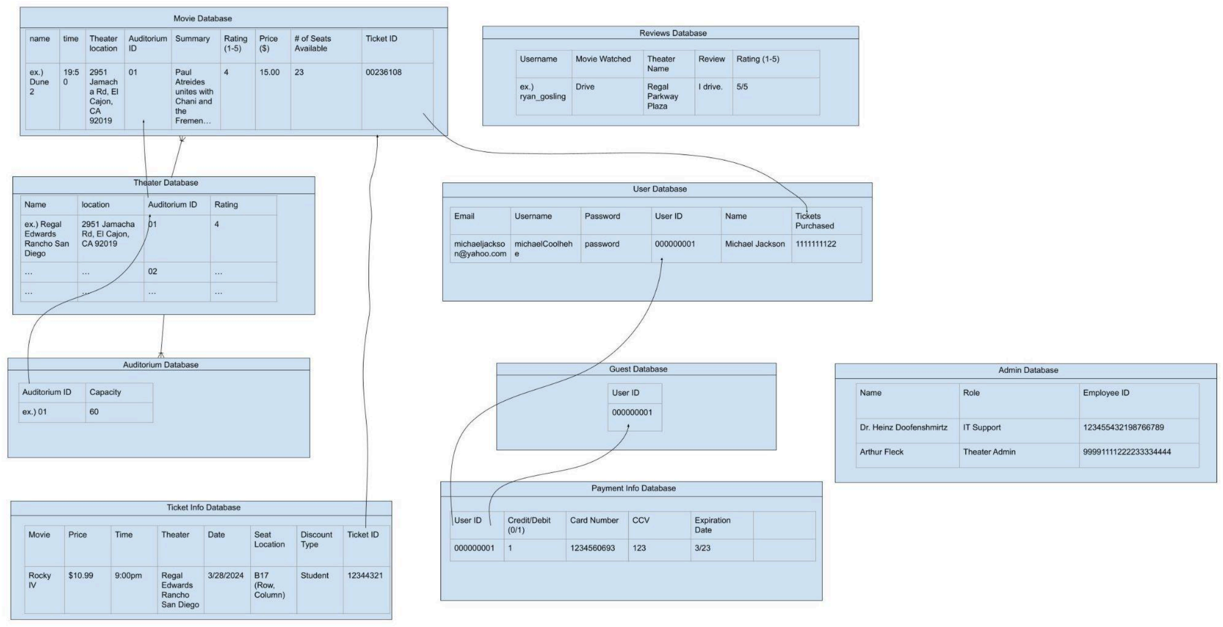


Website Main Page
Login Page/Admin Login
Account Manipulation
Theater Data Manipulation
Location, hours, special events
Theater Database
Admin Database
Account Authentication (Username/Password)
User Database
Account Info: edit username/password, saved payment options
Movie Data Manipulation
Ticket Info Database
Time playing, date playing, auditorium number, title, length
Review Page
Reserve seat, purchase confirmation
Payment Menu
Card #, CCV, Exp Date
Payment Info Database
Reviews Database
Review info: Theater, Movie,
Movie Database
Seat Info: seat number, row number, auditorium number, prices
Available Seats
Movie info, times, theaters
Seat Map
Movie Browser
Ticket Confirmation

5.1.1 Description

The updated SWA diagram integrates our data management strategy of using multiple databases for each of our specific parts of the program. Instead of one generic database, there are 7 databases containing information of which each part of the system uses. These include the theater, admin, user, review, movie, payment info, and ticket info database. Specific data transfers between each page and each database will be more straightforward with the new architecture design.

5.2 Data Management Diagram



Movie Database

name	time	Theater location	Auditorium ID	Summary	Rating (1-5)	Price ($)	# of Seats Available	Ticket ID
ex.) Dune 2	19:50	2951 Jamacha Rd, El Cajon, CA 92019	01	Paul Atreides unites with Chani and the Fremen…	4	15.00	23	00236108

Reviews Database

Username	Movie Watched	Theater Name	Review	Rating (1-5)
ex.) ryan_gosling	Drive	Regal Parkway Plaza	I drive.	5/5

Theater Database

Name	location	Auditorium ID	Rating
ex.) Regal Edwards Rancho San Diego	2951 Jamacha Rd, El Cajon, CA 92019	01	4
…	…	02	…
…	…	…	…

User Database

Email	Username	Password	User ID	Name	Tickets Purchased
michaeljackson@yahoo.com	michaelCoolhehe	password	000000001	Michael Jackson	1111111122

Auditorium Database

Auditorium ID	Capacity
ex.) 01	60

Guest Database

User ID
000000001

Admin Database

Name	Role	Employee ID
Dr. Heinz Doofenshmirtz	IT Support	123455432198766789
Arthur Fleck	Theater Admin	99991111222233334444

Payment Info Database

User ID	Credit/Debit (0/1)	Card Number	CCV	Expiration Date	
000000001	1	1234560693	123	3/23	

Ticket Info Database

Movie	Price	Time	Theater	Date	Seat Location	Discount Type	Ticket ID
Rocky IV	$10.99	9:00pm	Regal Edwards Rancho San Diego	3/28/2024	B17 (Row, Column)	Student	12344321

5.2.1 Description

The above diagram is a visual representation of our data management strategy. Our data will be separated into 9 different databases which are all SQL. Every database except for the review and admin databases have connections between other fields in the other databases. For example, the Ticket ID field in the ticket info database has a direct connection to the movie database where individual movie tickets will have a unique ID based on the data in the ticket info database. Another example of the connections between databases is between the user, guest, and payment info databases. Every user will have a unique user ID and every guest will also have a unique user ID. These will be able to connect to the payment info database where a customer's payment information will be associated with each individual user and guest. The difference between the user ID field for the user database and the user ID for the guest database is that the guest ID will be permanently deleted after a transaction is completed. There is also a connection between the movie database's ticketID field and the user database's tickets purchased field in order to store the tickets that the user has purchased in the past. These 9 databases will help store our data in a straightforward and optimized manner.

5.3 Explanation of Strategy

The database management strategy that we chose was SQL. The reason for this choice was because it was most optimized for the manner we wish to store our data. Data in SQL is stored in a format that shows the field variables of a class in different columns and each row represents a new object of that class. This is optimal in our case because our data follows a rigid structure that can be stored in this format to allow for ease of understanding the data that is being stored. Since the structure is very rigid and doesn't vary between different objects of the same class we can clearly show the correlation between the data and how they interact with each other. SQL also has many tools to prevent or also rectify corruption issues. This is good because the data being stored is meant to be stored for long term usage. For example purchase data needs to be stored so that users have the ability to write reviews later on. If that information was lost then users would lose access to some of the privileges they get through the purchase of a movie. SQL is also optimal for accessing, removing, appending, and other data altering commands.This is useful especially for the admin classes that would need to often manipulate theater data, movie data, and user data. Most data stored in our database is not static and will likely need to be changed frequently. SQL is also reliable when multiple users access the databases so it makes it useful for our application where thousands of people may be looking at different movies, writing reviews, purchasing seats, or editing account information at the same time. SQL also supports security and has functions built in just for that. This will be needed especially for specific databases that store payment information.

5.4 Database Organization

The databases were divided into 9 groups. These were representative of the main classes that many objects would be created from. These databases were the movie database, theater database, auditorium database, ticket info database, review database, user database, guest database, payment info database, and admin database. It is necessary to divide the databases as such because the objects from these classes all require varying levels of security. So it makes sense that depending on whatever data is accessed most frequently versus whatever data is most private the access to the database is limited and filtered accordingly. For example most people will be accessing movie info, theater info, auditorium info,and review info so those databases are more open for users since that information is open to the public anyways and does not need to be secured. The only thing to monitor is that hackers do not get in to modify the information into something that gives incorrect information. Next is the guest database. We need to temporarily create an object of the guest class for a singular user session for a customer that does not want to create an account. At the end of the session the object will be wiped. This database is unique because of how often objects are created and wiped. It will almost always be performing an operation on the contents within its database. User info database and ticket info database need to be more secure because those databases hold more personal information that only really needs to be available to the singular user. So access to this database is much more filtered. The admin database is a step up in security just so that admin credentials and employee information cannot be taken and used to interfere with the website. Lastly is the payment info database. This database needs to be one of the most secure databases. The information being held here needs to be secured in such a manner that only the sole user associated with that payment information can access it. Also this database will not be altered as much or accessed that frequently so the operations in this database will be much more limited. In summary, the main points for determining the division of the databases was how frequently accessed the database would be this is correlated with the level of security necessary with the database. Obviously every database would be made with the intent of being secured but some databases need to be more secure than others which is why special precautions such as limited access will be put into place which obviously does not make sense to keep on every database as it will hinder system processes. Lastly is dynamic allocation of memory. Some databases such as the guest, user, and movie database will be expanding rapidly and so there should be tools put in place to help allow the databases to expand rapidly as needed. Obviously admin database and theater databases are things that expand much less frequently so it would not make sense to allocate as much storage to it as once. This is vital when considering cost efficiency.

5.5 Data Organization

The data in each database is split up based on each data type contained within the database. Each row of data in each database corresponds to a singular collection of data for that associated object. For example, each row in the user database is split up based on each individual user and their unique user ID. Each column is split based on the different data types associated with each individual object. Our data is organized into rigid partitions in order to be distinguishable from one another. These partitions are also important in making connections between databases and the data within them.

5.6 Possible Alternatives

We could have chosen to use a form of non-SQL such as a key-value database. If we chose this route we would likely have structured the database with far less subdivisions because each data input is identifiable by its key value. Another non-SQL option could've been some sort of tree.

5.7 Discussion

Choosing SQL with the table structure allows for very neat data organization with relationships between the different tables/databases. This will likely sacrifice speed of searching and potential updating it as well. Choosing non-SQl, especially a tree, would make searching very efficient, and also adding and removing stuff as well. A key-value database would require less subdivisions of the data but would be more difficult to interpret.