Kilimanjaro Trekker System Software Design 2.0

Prepared by: Ella Suchikul, Sawyer Blakenship, Isabelle Bernal

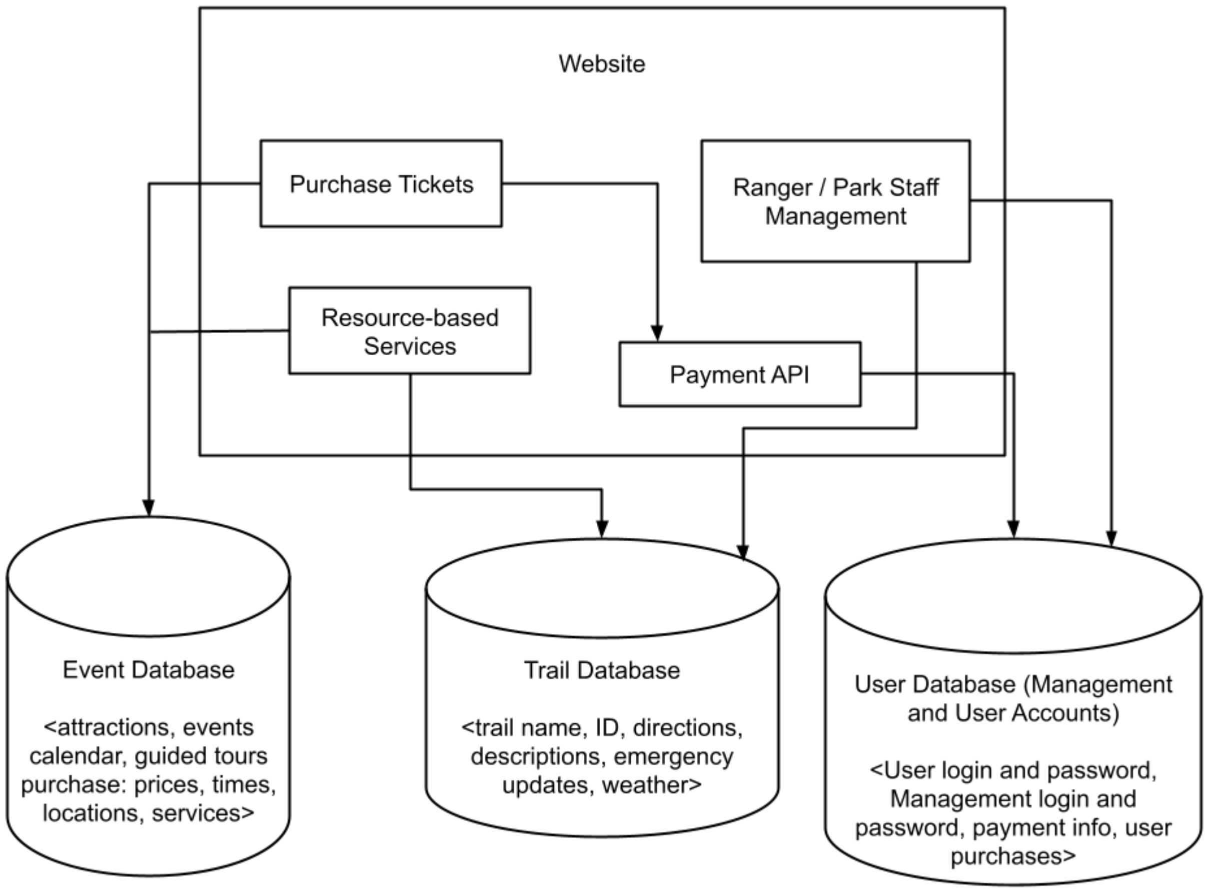
April 13, 2023

1. System Description

The Kilimanjaro Trekker Tracking System serves as an online support system for the Tanzanian tourist business so that the Kilimanjaro National Park is easier to navigate. The Kilimanjaro Trekker Tracking System is made available online through a website and through an interactive display at the park. The Kilimanjaro Trekker Tracking system maintains information about the twelve different trails, provides information on significant events in the area, helps people plan trips to the park, offers guided hikes through the trails, and gives updates in regard to evacuation and safety protocols. This website is essential because it allows not only the park rangers, but the tourists to have quick and easy access to anything they need to know about the park.

2. Software Architecture Diagram

Software Architecture Diagram: Kilimanjaro Trekker System


Website
Purchase Tickets
Ranger / Park Staff Management
Resource-based Services
Payment API
Event Database
<attractions, events calendar, guided tours purchase: prices, times, locations, services>
Trail Database
<trail name, ID, directions, descriptions, emergency updates, weather>
User Database (Management and User Accounts)
<User login and password, Management login and password, payment info, user purchases>

Software Architecture Diagram Description: The software architecture diagram covers all the major components of the Kilimanjaro Trekker Tracking System. The system is a website that can be accessed and read by any user, and edited by staff only. The major components of the software are resource-based services, purchasing tickets, and ranger/park staff management. The databases involved are events, trails, and users.

The resource-based services component entails an event database and a trail database. This is due to the resources and services offered by the software to support the users. The event database contains attractions, events, and information on guided tour purchases (prices, times, and services). The trail database contains the trail name, ID, directions, information on each trail (length, elevation, estimated timing, etc.), trail weather, and emergency updates.

The payment API component inherits the purchase tickets component and entails a user database. This component allows for a smooth payment process for the users and allows the Tanzanian tourist business to collect the money easily and efficiently. The purchase ticket component goes straight to the payment API component if the user wants to pay right away without looking at the event database. The user database then holds payment information, stores the purchase, and provides account management.

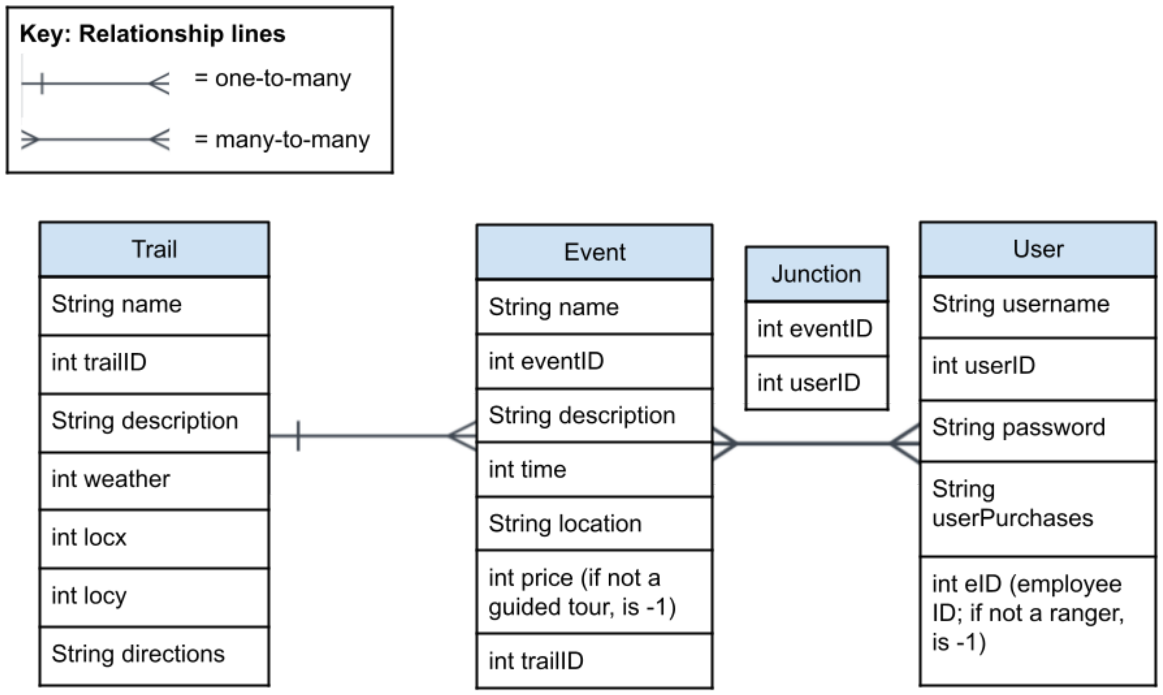
The purchase ticket component entails the event database and payment API component. This component supports the user with convenient trip information and a convenient payment process. The payment API component inherits the purchase ticket component and allows for all the information in regards to payment to be stored in the userdata base. The trip database contains all the information in regards to attraction, pricings, event calendar, and guided tours.

The ranger/park staff management component entails a user database and a trail database. The component allows for the management of the national park to gain full access to user and trail information. The user database holds management login information, payment information on clients, and client purchase history. The trail database gives management full access to updates and view trails, emergencies, weather, map, and directions.

This Software Architecture Diagram has been modified since our initial design. We changed the plan a trip database to just be an events database since guided tours, park organized activities, and reservations fall under the category of events. We also changed the data that the trail database will store and removed the trail map from the database since that is more of a display rather than data that needs to be stored. We did this as part of our data management strategy to keep it more concise and organized.

3. Data Management Strategy

SQL Diagram:



SQL Diagram Description & Design Decisions: We chose SQL as our data management strategy because there are strong relationships between the data in our table and because it's very structured. The trail and event databases have a one-to-many relationship. There are many events to each individual trail, such as specific guided tour groups and park organized events. The event database then stores which trail or location the specific event takes place through the trail id attribute, this approach easily stores relationships. The event and user databases have a many-to-many relationship. This is slightly harder to handle in SQL, but through a simple junction table, we are able to relate users and events. There are many events for many users to participate in. Many-to-many is a good representation of this relationship since the cardinality can be one-to-one, one-to-many, or many-to-many. Multiple users can sign up for the same array of events, one user can sign up for just one event, one user can sign up for many events, and so on. The user database then stores what the users purchase history. User and trail as classes in the program have a relationship in the sense of users having access to view trails or rangers having edit-access. However, we do not need to store that information since it is just available on the system to view. Both trails and users do not store any data relating to each other, rather a relationship through events. The user does not specifically have a trail or purchase a trail, which is why we did not draw a cardinality line between them.

The trail table represents the trail database which stores information about the various trails at the park, the trail's attributes, and provides the rangers/staff after giving appropriate id with access to updating anything in regards to the trail. Thus, we created the fields such as name, ID, description weather, locx, locy, and directions which allows the program to store the necessary information that either allows the staff to update the trail efficiently or allows the user to view the trail. We created the ID field which has a data type of an integer to store all the different trail's IDs. This is necessary for the program to identify if someone part of the staff is trying to gain access to the trails or a tourist, thus giving the correct features to the correct user. In the event that the ID is wrong or not found then it would not give the user access to anything due to the precaution of keeping the rebels out of the system. Also, the ID allows the events to relate themselves to the trail by storing the ID in their trail ID. We created the field name to store all the different trail names. This makes it easy for the user to know which trail's data they are accessing. We created the field description which has a data type of a string to store information on each trail (length, elevation, estimated timing, etc.). This data is very helpful to tourists and trekkers when deciding what trail they want to explore. We created the field weather which has a data type of an integer to store the precise temperature while visiting the trail to convenience the tourists while they visit. The weather field is an important attribute of the trail that can either be updated/viewed by the staff or viewed by the tourists. This is connected to a weather application that gives real-time updates. We then created the fields locx and locy which both have a data type of an integer so that the database has the exact coordinates of the trail and stores the information of where the trail starts and ends. These two fields are also a convenience to the park rangers since they are conveniently able to update the coordinates just in case they need to modify the trail for any reason. We added the field directions which stores the information in regards to where the trail is located inside the park and how to get there. This field which has a data type of a string, stores the directions of the trail so that the user can read directions on how to get to the specific trail they are looking for. Lastly, we created the name field which has a data type of a string, to store the name of each of the trails for the users to view as well to keep track of which trail is which.

The user table represents the user database which entails information on the different users: rangers and tourists. Thus, we have the fields username, userID, password, userPurchases and eID which allows the program to store the necessary information needed and indicate who the specific user is. The userID has a data type of an integer that simply stores IDs so the junction table can relate users with events for when users have signed up for events. While this may seem redundant with the eID, the eID must be consistent with the actual employee IDs, and is not given to tourists, so although having both of them requires more space, we put both to add functionality. We created the eID field which has a data type of an integer, to store the staff's employee IDs. This is important as all rangers must login to the system with their employee IDs, which also prevents rebels from accessing the site since there is a different employee ID associated with each user. This field stores -1 if the user is not a ranger, which is all other users (tourists and trekkers). We also created the two fields username and password in which username has the data type of a string and password has a data type of an integer that stores the username and passwords of all users in the database. This is crucial for account logins to check that no other user is trying to gain access to the system through an account that is not theirs. The system will compare the data entered to the data stored in the user database and return -1 if there is a username or password that is not in the system. This can be used as a precaution to the rebels and the park wanting to make sure that the rebels stay out of the system, or just a user accidentally entering the wrong information. Lastly, we created the field userPurchases which has a data type of a string to store the purchases the user has made, whether that is a guided tour or an event. This information is helpful to store as an order confirmation that the user made the purchase for when they get to the park.

The event table represents the event database which entails information regarding specific events that are occurring in the park. Thus, we created the fields name, eventID, description, time, location, and price which will allow the program to determine specific information for each event that is occurring and give an accurate description of each one. The name field has a data type of string that stores the names of all the events, available for users to view. The eventID field which has a data type of an integer stores an ID that can also be put into the junction table to relate users to events. Events can be classified as a reservation, park-organized events, or guided tours. The description field has a data type of a string which is very important because this stores the information about what the event is about, which is easy for the users to access. After reading the information about the event, the user can decide to purchase or attend if they please. The time field has a data type of an integer that stores the time the event will be taking place. We also created a location field, that has a data type of a string that stores the location and address of the event in the park so that the user has no issues with finding where the event is. Lastly, we created the price field. The price field is a little more complex in that it has a data type of an integer that stores the data of what the price is for each specific event but it will store -1 if it is not a guided tour. This is because guided tours need to be purchased, whereas other park-organized events are free. If it is free, the user just needs to purchase a ticket to reserve their spot, but will not need to pay anything.

The junction table is used to store all the relationships between users and events. Every user can sign up for as many events as they like, and multiple users can sign up for the same event, so to relate them, the easiest SQL approach is to add a junction table. This table simply contains a userID and an eventID. For each event that a user signs up for, both the eventID and userID are added to the table. This way, when someone wants to find the users attending an event, they can look up the eventID, and find all userIDs related to that event.

We chose to have 4 databases to store our 3 main portions of data: trails, users, and events. We split up the data based on the category we saw best fit for each. Any information on the trails such as name, ID, description, weather, location, directions, and updates will be stored in the trails database. Any information on the users, whether it be a ranger/park staff or a tourist/trekker, will be stored in the users database. Some examples of user data are name, password, ID, account info, payment info, and type of user. Any information on events, such as guided tours or events in the area, are stored in the events database. Some examples of events data are time, date, location, description, and price. Also the relationships between users and events are stored in the junction table. These four databases make it clean, concise, and it is clear where to find certain pieces of data. If we make too many databases, it may be confusing and harder to find information that we are looking for, which is why we think that only 4 databases are necessary.

Alternatives & Tradeoffs:

Some possible alternatives we could have used for organization of data:

There are a few possible alternatives we could have used for the organization of our data. One alternative is using one database for tourists and one database for rangers. This would probably end up using slightly less space for large databases, because tourists wouldn't need to have an employee ID of -1. This would result in slightly less space being used, but would also make it more complex. We would have to have more relationships. Another alternative could be using separate databases for events and guided tours. If we separated events and guided tours into two databases, we could have slightly different attributes that are more specific to each type. We could save space by removing price from the events because only guided tours can be purchased. However, this would make things more complex. All in all, these data organization alternatives have their pros and cons. When deciding how to go about this, it really depends on what the client wants, what storage we are limited to, and what type of data we are storing.

Some possible alternatives we could have used for technology: SQL v. NoSQL

We could have implemented NoSQL in a couple of ways. If we used Document NoSQL, we could have an easier to edit database. A document approach to database design would allow rangers to be able to edit databases without developer help because the documents are flexible. This would be more user friendly because this way rangers would be able to access the system without having to understand any of the complex code. We could have also implemented a Graph NoSQL. With a graph, we could use more dynamic storage techniques For example, we could have each trail point to a list of events, so it's easier to associate them. Having pointers could make associations easier to implement.

NoSQL has advantages and disadvantages for our data. Here are some advantages. NoSQL is used for real time applications. While a small difference in latency isn't too important for our application, quick updates can be good, for having users be able to quickly notice changes. Currently, we just have a string for userPurchases. SQL requires us to allocate a certain amount of space for each string and it must be the same for all users. If we do a low amount to save space, we might run out of space if a user buys a lot of things. If we do a large amount, many users with 1 or 2 purchases will have tons of allocated space that is not used. NoSQL's dynamic memory allocating properties could be useful, as it would allow each user to have unlimited purchases, and simultaneously conserve space for users who have few purchases. Also, NoSQL makes it very easy for rangers to update data, which they will need to do often.

While NoSQL has some advantages, it also isn't the best fit for our data. NoSQL is useful for large databases but we only have 4 tables and there will end up not being a lot of data stored most likely. NoSQL is helpful for object oriented programming-esque storage, as it allows one to use inheritance, encapsulation, and other more abstract storage options. However, none of these are useful for our simple database. Also NoSQL is good for dynamic storage options like arraylists. This could be useful, but doesn't help for this application (besides maybe it could be used instead of the junction table). NoSQL databases are quicker at querying large databases, but our database will most likely not end up storing a lot of data. NoSQL can withstand partial outages. This isn't important for such a small system. Also, NoSQL is less able to perform dynamic operations. We need to do financial transactions and SQL is more secure for this. SQL's structure makes it efficient, less complex, and consistent. SQL is better at queries.

If we had a complex database where every database was related, especially with many-to-many relationships, it would be very difficult to relate the databases through simple relationship IDs, which is the approach used in SQL. Using a NoSQL approach such as a graph could make this easier to implement. However, for a simple set of databases without complex relationships, SQL is simpler and works very well.