

**Lightweight and Performance Driven Ticketing System**

**Project Report**

Gabe Harman

Department of Computer Science and Information Systems

University of North Georgia

[Gtharm7223@ung.edu](mailto:Gtharm7223@ung.edu)

James Farlow

Department of Computer Science and Information Systems

University of North Georgia

[Jrfarl0545@ung.edu](mailto:Jrfarl0545@ung.edu)

**Abstract**

This project was an exercise in software development, specifically web development and database management. We researched various ticket management solutions and tried to implement a solution to unsolved issues we found. We tried to create a ticket management software that was lightweight, simple to use, and easily customizable. We used a webserver connected to a database in order to implement this and were able to achieve most of our goals.

**1 Introduction**

Our project is a web API based ticketing system. The problem we aim to solve is the need for a simple and easy-to-use ticketing system. We want to provide only the necessary features in a way that is readable and understandable. We aim to make a ticketing system that is lightweight and easy to use.

Ticketing Systems in General can be used for multiple purposes. The most common use is support tickets, but they can also be used for change requests and resource allocation. One article detailing their use for E-Resource allocation mentions “Every stage of the electronic resource life cycle, from trial, to acquisition, to activation, to maintenance and troubleshooting, to renewal or cancellation, requires a complex series of nonlinear tasks, typically involving handoffs between multiple library staff members and follow-ups with external players” [2]. This non-linear multi-person process can be tracked using tickets.

Through our research we found the most example of ticketing systems built for support purposes, such as Service Now, OSTicket and AskTech to name a few. Our ticketing system aims to be more generalized and contain features that can be applied in multiple use cases with a high level of control available to the end users. If an organization wants to use it for help desk purposes, they should be able to, but our system should also be usable for the lesser filled niches while still providing compatibility with other third-party software.

**2 Related Work**

**2.1 TaDaa: real time Ticket Assignment Deep learning Auto Advisor for customer support, help desk, and issue ticketing systems**

This paper focuses on using deep learning AI to efficiently assign tickets to the team or person that will best handle it. Their AI, TaDaa, “leverages the latest Transformers models and machine learning techniques quickly assign issues within an organization, like customer support, help desk and alike issue ticketing systems” [1]. This paper is useful to us in two main areas. The above quote shows how reading papers about Ticketing Systems in general can identify was that they are used. Knowing how ticketing systems are used helps us develop ours to fit those use cases. The other biggest way this paper helps us is identifying what a ticket needs to create an efficient system. “The entire machine learning system only uses three columns: ‘group’, ‘resolver’, and ‘description’. Where ‘group’ is the team from which the ticket is resolved by, ‘resolver’ is the specific person who solved the ticket, and ‘description’ is the text of the ticket” [1]. These three columns can be translated into the parts a ticket needs. If an efficient system can be constructed using that, we don’t need to overcomplicate our system.

**2.2 Managing Electronic Resource Workflows Using Ticketing System Software.**

This paper is about how libraries have used ticketing systems to solve their unique set of needs. It discusses why ticketing systems may not be the best solution for every library, namely “unless a library is able to piggy-back on the campus CRM system, has the technical staff to develop an open source solution, or chooses to go with a free option, there will be a financial cost to implement a ticketing system” [2] The financial costs being the biggest potential downside the paper speaks about. The paper finds that ticketing systems greatly benefit libraries that made use of them, “Libraries that have implemented a ticketing system to track end user issues and internal library workflows have seen increased collaboration and improvements in the efficiency, effectiveness, and morale of librarians and staff.” [2] This paper is very useful to us in finding additional use cases outside the most common - help desk tickets. The paper also details several specific features that are useful in the use case it covers such as priority level. The paper does have a glaring issue of not mentioning most of the ticketing software the libraries covered use. It mentions a software called LibAnswers, but does say that not all the libraries use that software.

**2.3 The New AskTech: Implementing a Ticketing System Platform for Technical Services Resource Troubleshooting.**

This article is about the specific case of the Duke University Libraries Technical Services department changing from one ticketing system to another. They discuss the benefits of the new system, such as “the centralization of tickets, more efficient workflows, the ability to see patterns among tickets such as noting problems by a particular vendor, and the ability to track workloads and make necessary staffing changes.” [3] This paper mainly covers their original issues and how they chose a new ticketing system. This is useful to us as it shows what businesses may be looking for in a ticketing system. The paper contains useful points such as “it [is] important that the system [has] the ability to report on relevant statistics” [3]. We can implement features like that into our ticketing system to create a better product. The paper only briefly covers how the department faired after fully switching to their new system, so we can only really look at their critiques of their old system.

**2.4 Impact of Critical and Auto Ticket: Analysis for Management and Workers Productivity in**

**using a Ticketing System**

This article focuses on the use cases of the Service Now ticketing system that is primarily used by larger enterprises and help desks. Service Now has more advanced features such as “Knowledge Management, Dashboards, and Task boards that can help the team to find references in fixing escalated reports and monitoring the open ticket count” [4]. Like our planned system, Service Now has a priority option that is required to categorize tickets. They categorize their tickets as low, mid, high, urgent, or critical. Each category has a predefined action that may alert management that can be configured by an administrator. Service Now takes an interesting approach by also categorizing tickets by reviewing what servers and services are being affected. By doing so, systems in production are automatically given a critical status and are escalated automatically. An interesting aspect of Service Now is their use of auto ticket. A report and ticket are automatically generated if predetermined conditions are met such as a database being unresponsive. By doing so, situations can be handled quickly. The report does not mention any security vulnerabilities that come from the data or connections that are required to keep the system operational. Having all systems interconnected could pose a security risk depending on how data is handled.

**2.5 Categorization and Visualization of Issue Tickets to Support Understanding of Implemented Features in Software Development Projects.**

This paper revolves around the specific use case of software development and the onboarding of new personnel and systems. The structure of their tickets is remarkably similar to the idea that we had of categorizing by team, priority, and category. The group in this article chose to categorize by “the title, description, comments, issue date, due date, assignee, etc” [5]. They however decide to take the categorization process a couple of steps further by implementing what they call “ticket vectorization” which they classify as any relevant data inserted by a developer running through an NLP (natural language processing) filter, keyword search, and weight the use of each keyword in the ticket. Another major difference is their focus of visualizing data. After categorization, their program creates a heatmap which prioritizes automatically the ticket required lifetime. Their study, however, did not use the full potential of categorization such as permission restrictions and correct assignment of team members. The study in this case was a “perfect scenario” that all team members must be working on the same project as one team. In no circumstances did they study multiple teams handling multiple projects and categorization of multiple projects. For larger firms and groups multiple teams will be working on multiple projects simultaneously.

**3 Dataset and Features**

With our Database, we decided to make a modular system so that an organization using our system could customize as needed. For example, The “Status\_Code” field in the “Tickets” table points to a table named “Status”. In that table, the organization could define statuses as needed for their implementation. The field defaults to a “Status\_Code” of 0 which is intended to represent a new ticket before anything is done to it, but a variety of codes could be added. Another example of modularity is the two tables used to represent groups. “Groups” could be used to represent roles that a user plays such as Customer or IT Support. They could also be used to represent organizational groups such as a dedicated group for Facilities and Operations.

We used several tables in our database to fit the needs of the project, with the “Tickets” table being accessed and modified the most. It represents a ticket as it is submitted by a user and responded to. The title and description columns are used to give human readable descriptions of the reason for the ticket to be submitted. The “Ticket\_ID” field is used for labeling the ticket internally within the software as well as tying other tables to the ticket table. Tickets have a “Status\_Code”, which points to a reference table that holds the human readable status names. Tickets also have id fields for the users tickets are created by and assigned to. These point to the “Users” table. There is an id field for an assigned group as well which narrows down the users that can be assigned a ticket, since as the number of users grows finding a user out of every user can become cumbersome. The “Metadata” field is used for storing temporary data that gets overwritten such as the last user that edited a ticket. “Date\_Created” is auto filled at creation time with the current time. It is displayed to users in case that is relevant information. Tickets also have a “Priority\_Level” which points to a reference table for priorities.

To hold data on the people using our software we use a table called “Users”. The “User\_ID” field is for identifying individual users and connecting other tables back to the user table. The first and last name fields are there to provide more human readable identification, though they are not unique fields and cannot be used internally to identify a user. The “Username” field is the user's logon. The “Password” field is the encrypted version of their password. “Session\_IP” is used to track what computer a user logged on from for security reasons such as preventing session hijacking. The “Email” field is included for human use when it’s relevant. The table also has a “User\_Status” field. This field is only set to 1 to mean a user is active or 0 to mean they are inactive.

The “Status” table is a reference table for ticket statuses. Since statuses are user definable, it’s important to have a reference table where the name is tied to an id code. This helps with organizing tickets by status, such as providing the number of unassigned open tickets.

The “Priority” table is very similar to the status table. It is a reference table with an ID field and a human readable name field. It is used when setting a priority for a ticket.

The “Comments” table represents additional information added to tickets after they are submitted. “Comment\_ID” tracks the comment itself. The "Parent\_Ticket\_ID” points to which ticket on the "Tickets” table the comment was made on. The “Parent\_Comment\_ID” is currently unused as we make comments linearly, but it gives the option to make comments as a tree instead. The “Comment\_Text” is what information the user added. “Date\_Created” is filled with the current time when the comment is submitted. It is extra information to help users. The “Created\_By\_ID” points to the “Users” table to show who created the comment.

Groups are split into multiple tables. The groups themselves are stored in the “Group\_Reference” table. This table has an id field for pointing other tables to a specific group as well as a name field to provide human readable identification.

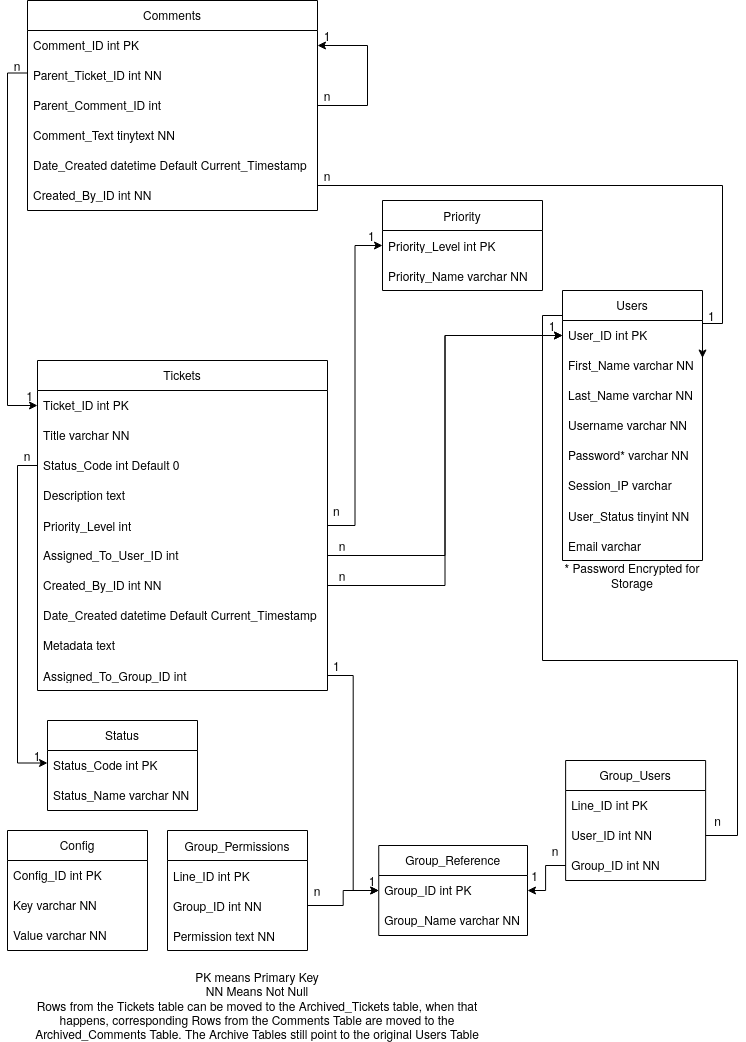
To store what users are in groups, we use the “Group\_Users” table. The “Line\_ID” field is for internal use as the table’s primary key. The “Group ID” points to a group in the Group Reference Table. The “User\_ID” points to a user on the “Users” table. This is a separate table because one user can be in multiple groups and a group can hold multiple users.

Since permissions are tied to groups and a group can hold multiple permissions, that needed a table as well. The “Group\_Permissions” table uses a “Line\_ID” for its primary key then has a pair of a “Group\_ID” which points to the group reference table as well as a permission held by that group.

The “Config” table holds options that users can set. The “Config\_ID” field is an internal primary key. “Key” has the name of the config. “Value” has the value assigned to that config.

The main configuration is designed around the modular thought as well by using a key-value-pair type system. This allows for large changes to the overall system from one central location. Items such as logos, titles, and other values that are global will be stored in that location. Some other variables are designed directly in the program that may only be renamed. A great example of that design choice would be the default status on tickets being 0 resulting in a waiting ticket. Users extending our code have the option to implement automated scripts to set other values than default as needed.

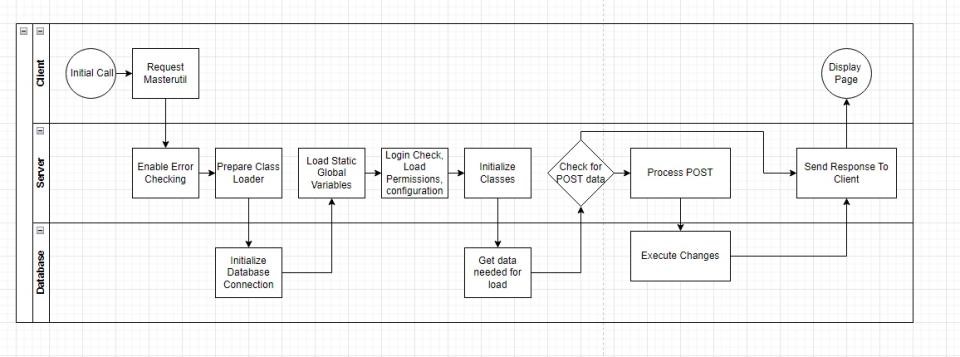
We also tried to take user friendliness into account. Most of the primary keys such as “Ticket\_ID” and “User\_ID” are Auto Incrementing fields. These are fields that exist for reference, and as such should not be updated or set manually. On the “Status” table however, the primary key is not an auto incrementing field. This is intentional. The idea behind this is that status codes likely have meaning behind them. For example, in our default data set, we create a status code of –1 to represent an error. An important note for this table is that there must be at least one row that all tickets can share. If there is not at least one status an exception will be thrown. There are measures in place to prevent an administrator from deleting all status codes.



*Figure 3.1: Diagram of the Relational Database Model used*

**4 Methods and/or Methodology**

Our application utilizes a client-server architecture. The decision was made to use apache2 being it is almost universally available on Linux operating systems and its ability to interface with php, our preferred language that we based the entire project around. The server distribution that we are running for testing and development is Ubuntu 22.04.1. The apache2 and php versions respectively are version 2.4.52 and version 8.0.24. At the time of writing this report, php is natively able to handle the MySQL connection using a module called mysqli, however the module used in this case is on the list to be depreciated within the next few versions. If that happens, the move to a PDO extension would be necessary but should not be difficult due to all calls being made through custom database class.



*Figure 4.1: Interaction diagram between the client, database, and server*

The database will get its own section in this documentation due to how data-driven our application is. We elected to make a key-value system that stores configuration values. The configuration is loaded each time there is a page load. Additionally, there is a call to the database that grabs the user object on every page load due to a concern about data becoming inaccurate if stored in a cookie or session variable for example. With these two things in mind, the database is regularly used on each page start up. Caching is possible but was providing inaccurate results if tickets or certain user attributes were updated. All insert statements other than one or two are performed over parameterized queries severely limiting the risk of sql injection. Other measures have been put in place to prevent sql injection on queries that cannot be parameterized.

To secure data being sent between the client and the server, we elected to use Let's Encrypt ssl certificates through a program called certbot. Certbot is a free-to use program that verifies and implements SSL certificates automatically. There is some user intervention needed depending on how apache vhosts are implemented, server hostnames, and configuration of the web server. Certbot will auto renew any expiring certs every 60 days through a cron job it places on the Linux system. If this program is used on windows, it is highly recommended to still use certbot with IIS. The auto-renew functionality is then run through created events instead of CRON jobs.

The correct fields and foreign relations they contain are shown in Figure 3.1. There is a sql file that will replicate our installation exactly. The administrator user should not in any case be deleted from the system. This is the global user that bypasses all permissions. If this user is deleted from the system and a new administrator is not created, any permissions not granted to a group would then be inaccessible without direct database access.

Permissions are declared in the “MasterUtil” file in the home directory. This file is called for all pages no matter if they are public or private and contains data vital in keeping the site running. It was decided to assign permissions to groups only and let users inherit their group permissions. Users are allowed to be in multiple groups, so we did not see any risk in implementing this system. It was our belief that this would cut down on inconsistent user experiences being users could be “singled out”. In our case, any users with the same groups will have the exact same experience.

For the front-end we elected to use Bootstrap 5 and jQuery 3.6.1 and associated modules. Most pages implement bootstrap cards and gridding to keep the application responsive. On some pages, jQuery is implemented to both make asynchronous ajax calls and enable the ability to use jQuery datatables. The jQuery datatables plugin applies functionality like pagination and searching easily while keeping the syntax the same. If the datatables plugin were ever depreciated or removed from the project, no serious impact would be noticeable except for the search, pagination, and layout functions.

No load balancers were implemented in this project. It is not suggested to try and implement load balancing due to the data-driven nature. Unless there is code put in place to prevent race conditions, the program would be susceptible and in turn, could be inaccurate. The initial design idea was for the program to be lightweight and performant on even ARM processors.

All the PHP classes are public with private attributes. For PHP this means that any data that needs to be retrieved or updated must go through a function call. PHP will throw fatal errors if data is being accessed directly. Any class that has variables to be updated and retrieved regularly will have both getter and setter functions that allow all attributes to be accessed in that class. Due to the classes only being used on a server-side programming language, the risk was deemed acceptable.

Class fields that have arrays are handled via JSON which is available on almost all modern programming languages. All classes directly correspond to the database fields. For example, if the Users mysql table had a column named ‘User\_ID’, the corresponding attribute would be $user->User\_ID. All class functions are currently dynamic and are accessed by the arrow operator (->).

**5 Experiments, Results, and Discussion**

Our test data was made by the team to test the core functionality of our system. The test data is designed to test that database integrity is properly implemented. Our database implements several foreign keys, and the testing data can be used to ensure those relations are working as intended. The test data tries things that will not work such as purposefully attempting to enter duplicate values into unique fields and trying to enter a value that does not exist as a primary key into a foreign key field. The test data also tries things that are intended to work, such as duplicate values in fields that are not unique. Currently our test data is designed to insert rows in all tables and set up somewhat of a “demo mode.” From there the system can again be reset and configured to the consumer’s liking.

After the ability to add tickets through the web page was implemented most of our test tickets were created that way. This is for a few reasons. Some fields, such as the metadata field, are unwieldy to set through a SQL script. It also doubles on testing the protections in place to make sure users do not cause SQL errors, such as checking all nullable fields are set.

This testing brought multiple errors to light. For example, we had protections on the login page to make sure users enter a username and password. We initially forgot to include the same protections on the register screen. Other examples of errors caught in testing were forgetting to update the PHP to reflect changes made to the database or using “SELECT \*” statements resulting in the wrong number of fields being returned.

Most testing wound up being done incrementally as features were implemented. This meant at points development was slowed as bug fixes were made. Overall, I think this was the right decision as making too many changes before testing would have made it difficult to find what caused the errors.

The metrics we used to evaluate our project were the time it takes to complete a task and the number of steps it takes to complete a task. The app is very responsive, and we were able to write the SQL calls in ways that are simple and effective. One example is the code to archive a ticket. While initially it was written to retrieve the ticket data, then insert the ticket into the archive table, by using a Select Into statement instead, we were able to make the function much more concise and efficient.

**6 Conclusion and Future Work**

The final product has some quirks and areas where we could have done better, but overall, it came together very well. If we were to do this again there are a few things we could do better to have a smoother development experience. One example is to be on the same page in terms of design. When I (Jim) was designing the database, I followed practices that involved splitting up tables and relying heavily on joins for the sake of data integrity. Gabe preferred a design that was less reliant on joins for the sake of performance. Communicating this more efficiently would have resulted in a more cohesive app and design.

Something we should have focused more on was the API. The intent and basis for this project was the API, since that was what would make the project most customizable. We wound up saving the API for towards the end of the project while we focused on the functionality. It may have gone more smoothly if the API were developed alongside functionality, i.e., when a feature is implemented then it is also added to the API.

There are several ways that our app could be improved and added to. The current view that lists tickets only allows for simple viewing and sorting. It lists all the tickets in the tickets table. When a ticket is archived, it is copied to a different table and removed from the original table, which means it is only accessible directly from the database. The list tickets view could include the option to switch between archived and active tickets. If that were done archived tickets would need to be viewable on the audit screen. An archived ticket shouldn’t be changed so it would need different options from an active ticket. One option is to provide a button to “reopen” the ticket. This wouldn’t unarchive the ticket though. Instead, it would create a new ticket using the title and description of the archived ticket. This is to preserve the data integrity of both the tickets and archived tickets tables as the primary key must not be reused.

Another feature to implement in the future would be additional default values. We left a lot to be customized by our app’s users, but that could result in some problems. For Example, we only include one default priority. We were able to include several rows of other defaults in the status table, and with some thought more universal priorities could be added in much the same way. More defaults like this would be desirable for our goals surrounding ease of use.

Smaller things to implement in the future could include a unified design. Since parts of the site’s visuals were designed by Gabe and parts by Jim, there are some parts that should look the same or similar but don’t. For example, the view tickets page had a different aesthetic from other paginated parts of the site before reworking it, and other areas such as the login and register pages could benefit from the same. We also could have done more testing to ensure the design is mobile friendly. Everything works on mobile and is scalable, however button and font sizes would need to be tweaked to have a mobile experience on par with the desktop experience.

**7 Contributions**

In order to split the work, we each focused on what we knew most with some overlap.

Jim focused on the database and SQL. He did the initial database design and created the tables. He had fewer contributions on the web server, doing smaller features such as working on pagination and implementing the archival feature.

Gabe focused on the web server. He set up the server we used as our test environment and created most of the pages and functionality. On the database side most of his contributions were adding a row where needed or error fixing and creating the database class.

**References**

[1] Leon Feng, Jnana Senapati, and Bill Liu. TaDaa: real time Ticket Assignment Deep learning Auto Advisor for customer support, help desk, and issue ticketing systems. Retrieved October 2, 2022 from <https://arxiv.org/ftp/arxiv/papers/2207/2207.11187.pdf>

[2] Kelly Smith. 2016. Managing Electronic Resource Workflows Using Ticketing System Software. *Serials Review* 42, 1 (March 2016), 59–64. Retrieved October 2, 2022 from <https://eds.p.ebscohost.com/eds/pdfviewer/pdfviewer?vid=1&sid=aedd20cd-4e12-47ab-a021-830249fc1136%40redis>

[3] Dennis Christman. 2018. The new AskTech: Implementing a ticketing system platform for Technical Services Resource Troubleshooting. *Serials Review* 44, 3 (2018), 193–196. Retrieved October 2, 2022 from <https://eds.s.ebscohost.com/eds/pdfviewer/pdfviewer?vid=1&sid=2a19529a-1edc-4a65-aed7-e5cbad2a96be%40redis>

[4] Kent Aglibar and Nelson Rodelas. 2022. Impact of Critical and Auto Ticket: Analysis for Management and Workers Productivity in using a Ticketing System. (January 2022). Retrieved October 2, 2022 from <https://www.stepacademic.net/ijcsr/article/view/293/119>

[5] Ryo Ishizuka, Hironori Washizaki, Yoshaki Fukazawa, Saori Ouji, Shinobu Saito, and Yukako Iimura. 2022. Categorization and Visualization of Issue Tickets to Support Understanding of Implemented Features in Software Development Projects. (March 2022). Retrieved October 2, 2022 from <https://www.proquest.com/docview/2649015598/fulltextPDF/474F253A9C454D6BPQ/1?accountid=>