Spring 6-2024

The Cascading Effects of Database Design

Liam McCracken
Portland State University

The Cascading Effects of Database Design

by

Liam McCracken

An undergraduate honors thesis submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science

in

University Honors

and

Computer Science

Thesis Advisor

Bruce Irvin

Portland State University

2024

Abstract

This paper details how a relational database influenced the design of the rest of a software project. The software project in question is WonderTix, an open-source ticketing and donation platform for the use of Portland Playhouse under continuous development by teams of Portland State students as their computer science capstone project. The paper, a capstone review thesis, examines WonderTix as an instantiation of the Model-View-Controller design pattern, noting how the model, a relational database based on the SQL standard, influenced the design of the view and controller components. This influence is explored from three angles. First, when the implementation of a new feature clashed with a preexisting database table. Second, the consequences of data in the database being implicit and the effects of making that data explicit. Third, how two database tables were designed in order to influence the design of a future feature.

Keywords: relational databases, SQL, software development, Model-View-Controller, web applications

For my computer science capstone, I worked with a team of students to produce software to support local theater company Portland Playhouse (PPH). PPH is looking for new software to replace their current ticketing and donation management platform. For the past several capstone cycles at Portland State, there have been teams of students developing WonderTix, an open-source web application to fulfill PPH's needs. For this cycle, my team was the latest to work on WonderTix.

Given WonderTix's nature as a continuously developed project, much of the general requirements definition work and broad design had already been completed. Thus, unlike usual capstones, our team had to prioritize features and bugfixes from an already determined, and growing, backlog of issues, and then we had to learn how to integrate our work into a sizable and mildly impenetrable code base, spanning over 350 code and configuration files.

However, my biggest takeaway from the experience was the cascading influence of the database design on the design of the rest of the project. And, more generally, this experience illuminates the importance of the design of the model in instantiations of the Model-View-Controller design pattern.

The Model-View-Controller pattern [1] is used to describe software that is made up of the three parts named in its name. The model is the long-term container of the data associated with the program. The view is how that data is presented to users. The controller manages interactions between the model and the view as well as other programs. The Model-View-Controller pattern is a useful pattern for many types of

software projects like e-commerce platforms and social media sites. It has been in use as a design pattern for web development since at least the early 2000s [2], [3].

As mentioned before, WonderTix is a web application. So, the view is the webpage that users use to interact with WonderTix. The model is the SQL database found on the server that hosts WonderTix. The controller is also found on that server, and it communicates with the model, the view, and a third-party payment processor.

A useful way to think about this type of program is that it is focused on moving and preserving data. WonderTix is a program that collects data that is expected to be accessed again later. For example, a customer purchases a ticket on WonderTix through the view. WonderTix collects the name of the customer, the exact showing they purchased a ticket for, the time of the purchase, the payment processor "payment intent" code (used to track individual orders), and so on. Before the purchase, the controller checked previous ticketing data in the model to see if tickets were still available. After the purchase, when the customer comes to the venue using WonderTix, such as Portland Playhouse, the door list feature of the website would be used to verify the customer had purchased a ticket before they are admitted into the venue. After the showing, site administrators may be curious about the performance of the show, so they would check the door list to see how many people purchased tickets, and how many of them attended. Or they might view one of the sales reports for the last month, which would summarize revenue statistics of ticket purchases during that time. This example should illustrate how one instance of data being entered into the program is then repeatedly be accessed by other parts of the program. Most of WonderTix's functionality follows a similar pattern.

Of course, all programs move data around, but it should be clear why the example above applies to programs employing the Model-View-Controller pattern in this way as opposed to other types of programs like image editors or video games which have different focuses.

WonderTix uses PostgreSQL [4] as its database. PostgreSQL is based on the widely utilized SQL standard, an instantiation of relational data models initially proposed by Edgar F. Codd at IBM [5]. The highest-level data unit in a SQL database is a schema. Within schemas, are tables. Tables are made up of rows and columns, and the intersection of a row and a column is a cell. Columns denote what the data below them represents; a column named “date” implies that the cells below it contains calendar dates. Rows are read as a collection of cells that are meant to be associated with each other. So, returning to the ticket purchase example from earlier, each data point that was mentioned as being collected would all be done in the same row, but in different columns.

A couple of key elements of SQL databases are notable for how their use in the database influenced other parts of WonderTix. First, are “not-null” columns. These are columns where data must be entered in every row instance; they cannot be simply left without a value, or “null”, in some cases like normal columns. Second, are primary-foreign key relations. In a table, there might be a column designated as containing the primary key. The point of a primary key is to distinguish one row from all others so it must be not-null and a unique value. In the case of WonderTix, it is always a generated unique integer. A table may also have any number of foreign key columns. A foreign key is a value

referencing the value of a primary key in another table. Thus, this allows the tables to be "joined" together temporarily when accessing data.

The first project of note is when a capstone partner and I implemented credit card reader support as a payment option for WonderTix. The purpose of this was to open the possibility of in-person ticket purchases at venues using WonderTix. Previously, ticket purchases could only be done by the customer online. Now, a venue employee can perform the order in-person and collect payment.

As we developed the credit card reader support, we had to integrate it with the existing order system that is already a part of WonderTix. Naturally, this includes how orders were stored in the database. In the database, there was already an orders table and an order items table. The columns in that table indicated to us what information was expected be stored. For example, the orders table tracked all general information relating to one order, such as price total and date, while order items tracked the individual tickets purchased in that order. So, we made sure that this information was collected and stored in the database.

Two columns in the orders table are important because their properties were at risk of influencing us in the wrong direction. First was a not-null column that held data referred to as "contact id", a foreign key relation to the contacts table. This meant that every order had to point to an entry on the contacts table, which contains information used for contacting customers, such as their name, email, phone number, and so on. The current method of purchasing tickets through the website collected this information at checkout. We almost, as advised by the database design and by our contact at PPH, added such an

information form to the in-person checkout process as a part of WonderTix's view component.

In the end, we refrained from doing this. Collecting contact information from customers makes sense during an online checkout. There is no rush in completing an order. And, consider that WonderTix allows for refunding tickets and will eventually allow for exchanging tickets, or shows may sometimes have to be cancelled or rescheduled, so customers would need to hear about such occasions. Those possibilities are much easier to handle when there is a direct connection from a ticket purchase to contact information. However, these events where WonderTix would need to easily connect purchases to customers most commonly happen at least hours, if not days, before showings. The card reader purchase method is meant for in-person payments immediately before a show begins. Thus, collecting contact information is less important in these circumstances as the situations in which the information is needed are rarer. Plus, consider any experience when you are going to a scheduled show, whether it is a movie or a play. A goal at the box office is to clear through a line as fast as possible. Collecting a name, address, phone number, and email from every patron would dramatically increase the time needed to get everyone in line into the venue, which is contrary to one of the goals of in-person purchases. For this reason, we opted to remove the not-null restriction from that column of the database so that card reader purchases did not need to supply a contact id. It would have been easy, especially given WonderTix already have an easily reusable contact information form page supported by the view and controller, to not make this decision, and we very will might have if we had been more pressed on time and went along with the

assumptions of the database. And it is also worth pointing out that removing the not-null constraint from that column makes it easier for a future implementer of a new purchasing feature to opt to not supply the contact id value to the purchase table when it should be supplied in actuality. Hopefully, they, like my partner and I, evaluate the necessity of contact information in such an instance.

The other notable column at risk of leading us astray was the payment intent column. Every order through the third-party payment processor used by WonderTix has a unique string of characters used to identify the order called the “payment intent”. This is important to store because it is how WonderTix refers to a specific order when communicating with the payment processor’s API. The problem with the column was not with its existence or constraints, but the pattern of how the controller was using it.

There is another column called “checkout sessions” which stores a unique string that is used to identify a unique checkout session done using the payment processor’s online checkout page. This needs to be supplied with a value for online orders, but it is not needed for in-person orders as that checkout page is not used by the card reader system. However, the current online order system had established a pattern where when an order begins, the checkout session value is saved to the database. Then, when the payment processor informs the WonderTix controller that the order had finished, the payment intent is saved to the database.

This presents a couple of problems. First, it is not intuitive that this is how the controller is using those columns. I only determined this pattern existed through observation and reading specific sections of the code of previous capstone teams.

Second, as the in-person payments did not have a checkout sessions identifier, as the payment processor's checkout page is not used, and due to the nature of working with the payment processor's card reader system, it was best to store the payment intent when the in-person order process began, not when it ended. Thus, just by looking at the database, there would be no straightforward way to verify that the order had reached a completed state. This should already be concerning for any type of purchase, and even more so with the increased volatility of an in-person purchase involving physical hardware.

This led me into my second major WonderTix project: a better system for tracking the payment status of orders. The solution, in theory, ended up being simple. I made explicit in the database a column called "order status". The problems that I just detailed manifested from order status not being explicit in the first place.

This solution had a couple of advantages. First, it puts into the mind of future controller developers that they should be tracking the status of orders when working with the WonderTix order systems. Second, it allows for multiple informative states to exist. The type of the order status column was an enumerator referred to as "state". In line with what already existed as subtext in the WonderTix database were two states, one called "in progress" and the other called "completed". I also added a third state, "failed". Failed means that PPH does not have the money from the purchase. Having studied the payment processor's documentation, not only could orders fail at the time of purchase, but they could also retroactively fail at any time after the purchase for a variety of situations that results in a chargeback occurring. While WonderTix's controller already has a system in place for handling failures at time of purchase, it has no capacity at present to handle

retroactive failures. And while establishing the behavior for the controller to handle the retroactive failures was not a priority for our capstone group at the moment, especially given its complexity, I hope the failed state will spur future controller developers to consider the possibility.

Similarly, I added a status column to the refunds table while I was at it. And, in another attempt at foresight, I also added a column for tracking the origin of orders: whether the orders come from in-person purchases, the standard online checkout, the admin page checkout, and so on. While, at present, neither the controller nor the view uses this status for any functionality, I suspect it will be useful in the future when tracking down why certain orders failed.

The addition of order source, order status, and refund status had me extensively reviewing code, primarily in the controller but occasionally in the view, from the previous teams and updating them to work within the new paradigm. While the changes were ultimately small, the time taken to ensure coverage was not insignificant. This experience shows that not only does the design of the database influence the design of the rest of the project, but that the cost of correcting design choices by the database only grows from its inception. Imagine the time it would have taken a few capstone cycles from now.

The third project I undertook was implementing the controller code for handling the recurring donation system. It is also another instance of the design of the view and controller being driven in-part by the model. Portland Playhouse is a non-profit organization that in-part relies on donations for funding, so WonderTix must support donation management in addition to ticketing. However, I knew with the time left to our

capstone team, I did not have the time to do the implementation. What I did have time for was determining what the steps needed for implementation are and developing what the design should look like. So, I wrote a design document for recurring donations. What I found was that setting up recurring donations in the controller and view were straightforward if you followed our payment processor's, Stripe, API documentation [6], assuming the database was prepared to handle recurring donations. So, most of the document was dedicated to detailing the changes the database needed. In short, the already existing donations table needed to be split into two. One table is the "donations" table, which tracks all individual donation instances, with a foreign key reference to the orders table and, when applicable, a foreign key reference to the second table, the "recurring donations" table, which contains the data pertaining to instances of recurring donations set up by patrons. The recurring donations table contains a contact id foreign key reference, the amount donated every cycle, the frequency of donations, the start date, and then the end date once the patron chooses to end their recurring donation. The actual code, barring unforeseen problems, is primarily the controller moving data around between the payment processor, the view, and the database according to the database tables and what the payment processor API wants.

Across these three projects, a few lessons become clear when working with the model in a Model-View-Controller program. Consider when it is realistic to collect information to store in the model, like in the order source and not-null contact id scenarios. Think carefully about the future scope of your program as early as possible when designing your database schema, as future changes become increasingly costly due

to the required adjustments in the controller and view. Do not leave row information inferred, make it explicit with an additional column. Both of those lessons are shown by the order status situation. The outline for implementing recurring donations gives a clear example of how the design of the model can influence the controller and view. These lessons show the cascading influence even small elements of the model in a Model-View-Controller program can have on the rest of the project.

References

[1] G. Krasner and S. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80,” *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, Sep. 1988.

[2] A. Leff and J. T. Rayfield, “Web-application development using the Model/View/Controller design pattern,” in *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, Sep. 2001, pp. 118–127. doi: 10.1109/EDOC.2001.950428.

[3] A. Syromiatnikov and D. Weyns, “A Journey through the Land of Model-View-Design Patterns,” in *2014 IEEE/IFIP Conference on Software Architecture*, Apr. 2014, pp. 21–30. doi: 10.1109/WICSA.2014.13.

[4] “PostgreSQL.” The PostgreSQL Global Development Group. Accessed: May 24, 2024. [Online]. Available: https://www.postgresql.org/

[5] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[6] “API Reference.” Stripe. Accessed: May 24, 2024. [Online]. Available: https://docs.stripe.com/api