Solving Pay Stub Compliance Issues Through Asynchronous Generation and Persisting To S3

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Preethi Chidambaram

Spring, 2022

Technical Project Team Members

Rosanne Vrugtman, Briana Morrison, Department of Computer Science

Abstract

As a payroll and benefits management platform, Gusto currently does not keep track of pay stub version history because they are generated on demand. To address this compliance issue, I created a backend model to store pay stubs and persist them to S3, Amazon's storage service. I structured the design of this model to be similar to pre-existing models containing pdf attachments. I also modified the on-demand pdf generation behavior to be asynchronous to improve overall performance. Storing the pay stub pdfs in S3 proved to be a simple, cost-efficient solution to tracking version history. Throughout the refactoring process, I found that preserving aspects of the existing pay stub behavior was important for maintaining a satisfactory customer experience. In terms of future work, a backfill should be run so that all existing pay stubs are regenerated and stored in S3 to ensure that the new behavior is consistent with the old records. While the backend work for this project was completed, the frontend system for displaying version history to the user is still being developed and needs to be adequately tested before being shipped to production.

1. Introduction

In Gusto's payroll system, pay stubs are always generated on demand using the most up to date data stored in payrolls from the associated pay period. This is acceptable for most cases; however, it gets tricky if any data is modified after the original payroll was processed. For example, when a reversal payroll is run, the system generates a pay stub which may not be an accurate representation of what the employee was actually paid at the time of running the payroll. Another possible case may occur if an employee logs paid time off (PTO) after the payroll is processed and the system assumes it should deduct the amount of PTO from the employee's regular hours worked. This automatic update is generally an acceptable solution but is not accurate for all cases.

Gusto was considered to be non-compliant due to its failure to maintain an accurate paper trail of employee pay when a payroll was originally run. If a customer were to specifically request a pay stub from the time of running the payroll, Gusto is required to provide it. Since the system could not support this behavior, Gusto relied on customer support lines to retrieve the correct information and generate the pay stub file manually for the customer.

2. Background

Some terminology that will be heavily used involves the software that was used for the project. The project added a feature to Gusto's codebase, which uses the programming language Ruby and web application framework Ruby on Rails. Additionally, several payroll processing terms will be used to explain different components of the project. Payrolls refer to the collection of data representing the pay earned by a group of employees during a pay period. A pay stub is a summary of all the earnings for a given employee during a pay period. Payrolls represent the actual money movement, while pay stubs act as a receipt for the payroll. Payrolls can exist in states such as unprocessed, where the data is in a draft state, or processed, where the data has been submitted and the money movement process has started. Ideally, payroll data should be immutable once it's processed, but this is not always the case.

Reversal payrolls are generated when a customer wants to undo a payroll after it has been processed and the money movement has been completed. The reversal payroll is essentially a payroll with negative amounts that reverses the original payroll. Retroactive PTO (paid time off) refers to a specific scenario in which an employee submits a time off request for a pay period after the payroll for that period has been processed. Both of these processes change the contents of the pay stub and overwrites the older version, which essentially causes the system to lose this older data.

3. Related Works

Previous works of literature on digital payroll systems outline the basic components for a successful system. Mahajan, et al (2015) posit that a typical computerized payroll system contains three main layers: presentation, business, and data access [1]. The presentation layer refers to the front-end web page the user interacts with. The business layer defines how the backend payroll calculations and modifications are made, while the data access layer defines how the data is stored. As opposed to Gusto's Ruby on Rails-based payroll system, the system proposed by Mahajan, et al (2015) utilizes C# for the business layer and SQL for the database. Pay stub version history is specifically addressed in this system since Mahajan, et al proposed that individual pay slips should be printed for users to make sure they are receiving updated information. This system was proposed as an alternative to manual payrolls, so it is one of the seminal works that laid the foundation for other automated payroll systems. However, Mahajan, et al. fails to address how their system handles certain features such as PTO and reversal payrolls.

While the problem of pay stub versioning is addressed, the authors' solution of keeping a physical paper trail is not feasible in modern day systems. Another system that helped inform my project is the UKG Pro human resources platform. UKG Pro allows employers to set up automated payroll processing for their company and specifically addresses how to configure retroactive pay. Helton (2021) explains that retroactive pay is an important part of every payroll system because it allows employers to make up for compensation shortfalls in a previous pay period.

4. Process Design

The design of this project began by first compiling the backend and frontend requirements so that a project timeline could be constructed. The following subsections go into more detail about how the project was divided.

4.1 System Requirements and Overview

When a user views pay stubs from previous pay periods, they should have the option to see pay stub versions if the pay stub data was changed from the time the payroll was run. As for the backend requirements, we aimed to store the pay stubs in Amazon Simple Storage Service, or S3, as they were generated. The main parts to this project involved linking a backend Ruby model to S3, creating a background job to generate pay stubs asynchronously, and updating the user interface to display these changes.

4.2 Persisting Pay Stubs

A model in a web framework represents the data that is being transferred between the user and the server. Creating a new model involves two main steps: running the necessary migrations, and writing the code to configure the model. The migration feature allows developers to easily modify an application's database schema without having to drop and recreate tables. Active Record is the Ruby on Rails interface that binds these tables to the Ruby program code that manipulates them. Gusto already had a Paystub model that kept track of several data points for every pay stub in the system, however it did not store the actual pay stub pdf file. The pdf file was generated upon user request by aggregating the Paystub model's data into a clean, readable format. I ran a migration that added a new table to store these pdfs and then I created the model, PaystubFile, which was linked to a specific Paystub instance. The PaystubFile model also interacted with S3 through a Ruby gem called Paperclip. Paperclip adds a "has_attachment" field to the model so that it can automatically upload and retrieve files from S3. Paperclip also generated a url that stores the PaystubFile's location in S3.

4.3. Asynchronous File Generation

After configuring the PaystubFile model, I started integrating this model into the pdf generation process. The original on-demand generation behavior needed to be changed because it was viewed as a process that blocked other action items. My mentor and I decided that the asynchronous PaystubFile generation process would take place after an employer runs payroll. I utilized events and event consumers to structure the asynchronous pdf generation process. Events in programming refer to a set of actions or occurrences which gets notified to the system so that the system can react accordingly. Gusto's system already implemented the PayrollFinishedProcessing event, which was emitted every time a payroll was run and finished processing. I created an event consumer, GeneratePaystub, that listened for the emission of this event and then called a worker class. This worker class interacted with Sidekiq, a Ruby job scheduler, to enqueue the task of generating all the pay stubs for the corresponding payroll. It was necessary to create a background job because generating numerous pdfs at the same time is time intensive and can cause a huge backlog in the system.

To address the scenario of reversal payrolls being run, I followed a set of steps similar to those described above. As for retroactive PTO, there was no preexisting event that indicated when a PTO was applied. Thus, I created an event that was emitted when time off requests were processed. I then created an event consumer that checked to see if the time off request was approved after its corresponding payroll was processed. If that requirement was satisfied, the event consumer would call a worker class to enqueue a background job, which generated the new version of the pay stub pdf.

4.4. User Interface Changes

Once the backend work for generating and persisting pay stubs was complete, I started integrating these changes into the user interface. Gusto's frontend interface was built on React, a Javascript library, and GraphQL, a data query and manipulation tool. I first created a GraphQL object to expose the PaystubFile model to the frontend framework. Exposing the backend model through GraphQL essentially lets developers make queries to fetch data from the model and display it to users. I created a query to retrieve PaystubFile objects and display them to users via a new url route. While I ran out of time to adequately test and deploy this feature onto production, I was able to get a majority of the foundational work done.

5. Results

The implementation of the newly designed feature is anticipated to ease the process of retrieving past pay stubs for users. Since the feature is not yet deployed on production, we have not been able to gauge the specific metrics that have come with this improvement. Deploying this feature will take away the need for users to call customer support when attempting to retrieve a previous version of their pay stub. Additionally, the frontend system will be more transparent in that users will understand why a specific pay stub was modified and has multiple versions. Overall, persisting pay stubs to S3 has taken away the pay stub related compliance issue Gusto originally had and it is anticipated to greatly improve Gusto users' experience.

6. Conclusion

Pay stubs are an integral part of any automated payroll management system because employees rely on pay stubs to view their compensation breakdown. Gusto's payroll and human resources management platform is a people-oriented service that prioritizes reliability and customer satisfaction. The persisting pay stubs feature was a significant improvement to Gusto's current system as it gave customers access to an accurate paper trail of pay stubs. It is imperative that Gusto has this feature in their system because without it they were non-compliant with the rules and regulations of managing payroll. By integrating this feature into Gusto's system, I was able to give users insight into payroll changes, which can help employers and employees with future business decisions.

7. Future Work

In terms of future work, the front-end display for viewing historical pay stub versions is still in development and should focus on having a simple interface for users to easily navigate to and from this feature. I worked with my team members to design the layout of this display page and started some initial work on it, but I was not able to completely integrate the backend changes to the existing frontend web page. Additionally, to maintain consistency with the system functionality, a backfill should be run on the generation of pay stubs so that all past pay stubs exist in S3. This will ensure that whenever a user requests to view a pay stub, the system will always retrieve it from S3. These changes should be adequately tested and approved before being shipped to production.

References

[1] Kritika Mahajan, Shilpa Shukla, Nitasha Soni. 2015. A Review of Computerized Payroll System. IJARCCE 4, 1. 67-70. https://doi.org/10.17148/ijarcce.2015.4113

[2] Jessica Helton, 2021. 10 Tips and Tricks To Successful Payroll Processing (May 2021). Retrieved October 15, 2022 from https://www.neosystemscorp.com/blog/top-10-tips-and-tricks-to-successful-payroll-processing-in-ukg-pro