The Portal 2 Challenge Mode Leaderboards Project

Modern Web Approach

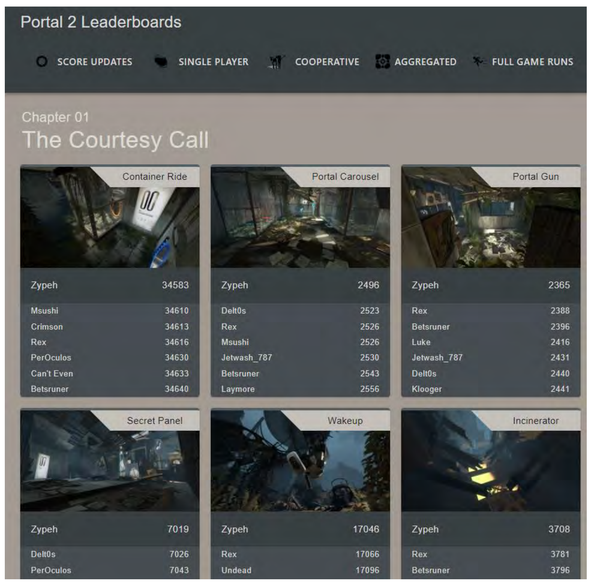
Daniel Bates, Josh Bednarz, Mitchell Baker, John Fiedler, Michael Murphy
University of North Texas – Computer Science

Introduction

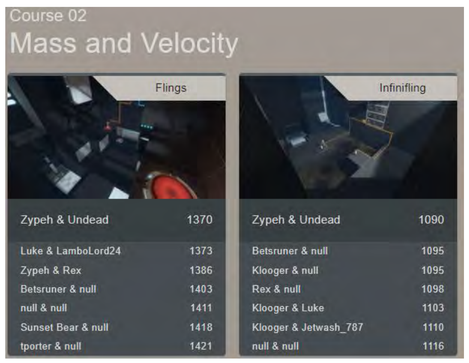
This project was intended to create a modern web approach to represent data and provide functionality to a community of competitors so that they can keep track of all the data associated with their challenge mode progression in Valve's "Portal 2", released on the Steam gaming platform in 2011. The community is made up of thousands of individuals across the world who have been playing for years, who approached our team about tackling this project. The goal was to add additional functionality to the current rankings they have, by redesigning an eight-year-old webserver with a modern and performant webpage that is built with scalability and adaptability in mind. The goal is to take the foundation we created this semester and build it out fully in the near future.

Examples

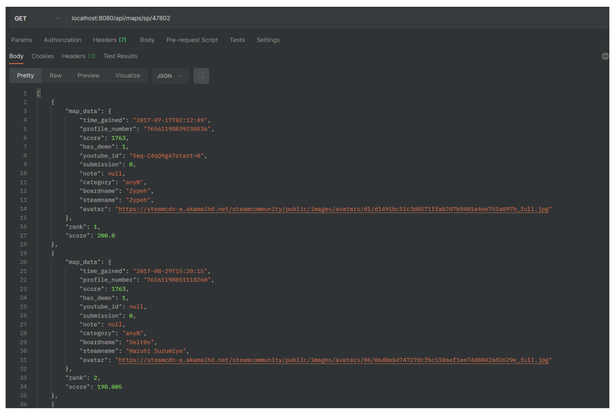
Single Player Example



Cooperative Example



REST API Example



Technologies

The stack for our project was built around speed, particularly on the server side. Our backend and server architecture utilize the Rust systems programming language, the Actix_web framework backing our REST API, and the Diesel ORM for connecting to our MySQL database. With how many calculations we're planning to tackle computing in the future, this sort of low-level speed is vitally important. Modern Rust also support async for web and is designed with a memory ownership model that guarantees you data race freedom. This allows us to easily parallelize our computations which for calculating aggregated scores for thousands of users every few minutes is crucial.

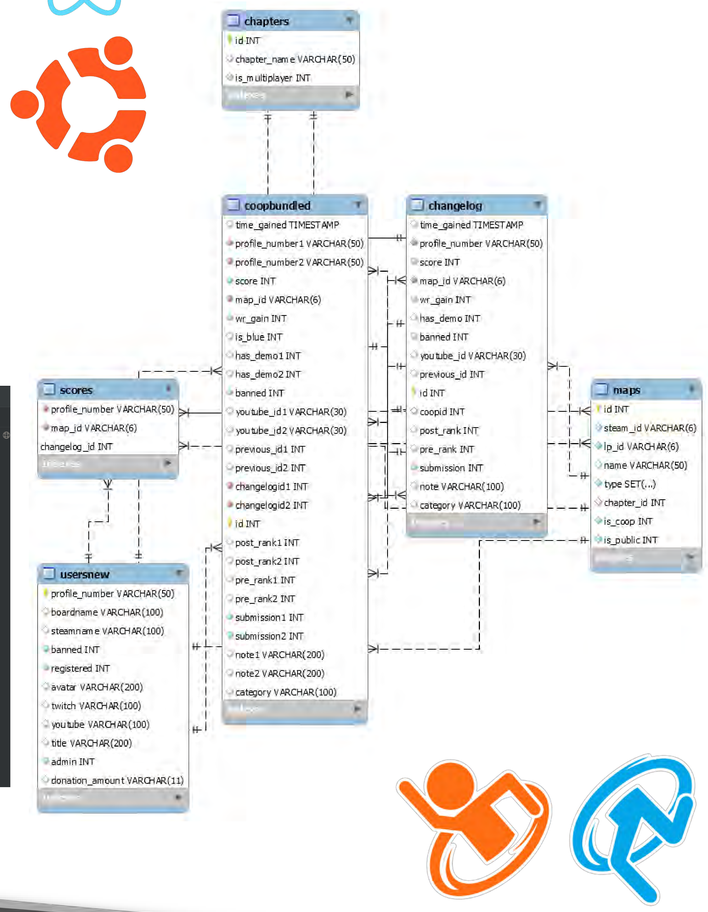
Our frontend is a modern React app that utilizes Material-UI and React Router Dom to modernize the app for the modern web, while keeping it lightweight and data-focused.

Our database is a legacy MySQL database that contains 10 years of history, we used a Python script to automate updating and adding specific fields, the biggest of which was creating the `coopbundled` table that combines two normal changelog entities into a combined entry.

Some of the performance compromises we had to make during this project were in relation to the still developing ecosystem surrounding Rust. Diesel had several limitations with its query builder, specifically with querying for distinct values, and for using aliases with queries to join on multiple of the same tables.

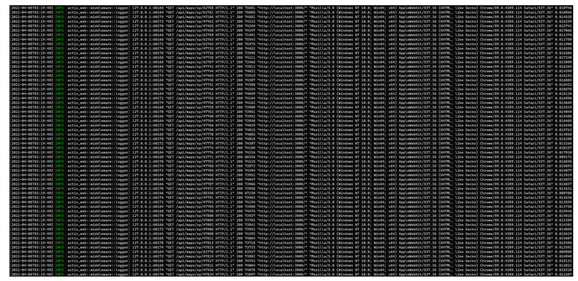
Even with our calls being much more performant than the current boards, it still leaves more performance on the table for when these are supported by diesel in the future.

Database Relations



Results

Our backend development pivoted halfway through the semester after all the basic outline for database interactions was finished. The backend team decided to switch to using a pool of database connections that all endpoints have access to, as well as moving all async API calls onto separate threads. Below is an example of the server handling 60 API calls almost simultaneously (each thread took roughly .01, across 12 cores). This performance boost alone is a massive success, not to mention the rewriting of specific routes to handle database calls in a manner that optimizes the amount of data the needs to be sent over the network.



Conclusion

In conclusion, we created an excellent foundation for the community who tasked us with creating this board. Members of our team will continue to develop additional functionality after graduation as well as open-sourcing the project to allow others to contribute, a current list of features is as follows.

Features

Database:

- Fixed foreign key structure for relations on legacy database
- Scripted in support for coop times being bundled
- Added support for different categories.

Web-server / Backend:

- Started our REST API
  - Communication to the database with diesel
  - GET endpoints modeled, all type-safe
  - Functionality for the POST, PUT, DELETE endpoints prototyped
- Server logging
- Authentication prototyped
- Steam API querying prototyped
- Async and multi-threading support using database pools for parallel database read operations.
- Performant data filtering to provide the front-end.

Front-end

- Footer / Header components created
- Material-UI theme support (light and dark)
- DOM Routing with React
- Body components created
  - About page
  - Aggregated Pages
  - Single Player & Cooperative Preview Pages
  - Single Player & Cooperative Map Pages
  - Changelog / Index Page
  - Donation and Wall of Shame (banned players) pages