Bricked Up

Francesco Schenone Sebestyen Deak Kacper Grzyb Ignad Bozhinov
Leonardo Gianola

17-12-2024

1 Introduction

The LEGO investment market has emerged as a profitable opportunity, with an average return of 11% [1]. However, the current process for analyzing prices, trends, and data on LEGO sets needs to be more cohesive and efficient. To address this, *Bricked Up* centralizes key information, providing interactive graphs, historical price comparisons, and tools to explore investment opportunities efficiently. This report outlines the development of the software, inspired by the Bloomberg Terminal, and its role in simplifying LEGO set investments for users.

2 Front-End

Before beginning to develop all of our views, as any respectable software engineer would do, we set out to create a prototype. As is industry standard, we used Figma to create our prototype, which included all the views we were planning to make and a general outline on how the application should look. Our intention was to use a component library both for the Figma prototype as well as during development, but given our unique design language it did not come to be. Since we were building a platform akin to Bloomberg Terminal (but for lego), we inspired our views by it, by using that color palette and a similar design language: very information dense, dark-mode only and a terminal-like view. Once the Figma prototype was developed, we reviewed it, and started developing the front-end.

As our framework of choice ended being Laravel, we tried to adopt as many Laravel defaults as possible, which was reflected in the way we developed our front-end. As templating engine, we used blade, a simple yet effective templating engine which allows for mix-and-matching HTML with PHP, dividing the application into components for a greater reusability and modularity. Blade is the default templating engine in Laravel, which meant we did not had to spend any time setting the project up but could go right to develop the views of our web app. We also ended up using alpine.js [2] in some parts of the application, when handling graphs or to create the scrolling bar with the prices for example, where vanilla Javascript was turning out to be too complicated or where inline PHP simply was not enough.

Many components were divided in reusable blade components, such as the navbar, all the elements of the dashboard, as to allow customization of such, the authentication components and other shared items. By making use of the components we greatly reduced repeated code, allowing us to adhere as much as possible to the DRY [3] principle. As an upside it also sped up development time considerably, as each team member could work on a component of the website indipendently, without needing to wait for pages to be complete yet.

The HTML we produced was heavily subdivided in with various `div` elements. That allowed again for a greater modularity and enabled to easily style different components in our web app with CSS. Most `div` element had a class associated with them, which then we would separatily style with a CSS document. We did not use a premade CSS stylesheet, but wrote our own, to adhere to our unique design language. All of

the CSS was kept in the default `app.css`. Given the extensive use of `div` elements with distinct classes, no conflicts emerged between components. Other HTML tags that were used are the `p` to hold text, `li` and `ul` tags to create lists and `img` tags to hold images correctly.

The appropriate use of Javascript, modern HTML and CSS together with a powerful templating engine like Blade, allowed for a visually pleasing end product, with what we believe to be an intuitive UI and UX.

Resources. Lectures 1 to 3.

Length. 2 columns.

3 Resource Management

Since we chose Laravel as our MVC framework, before we could define a resource management system in our project, we first needed a database. Our database solution of choice ended up being Supabase with PostgreSQL. We settled on it because it is very easy to set up, provides a generous free plan, has an intuitive UI, and can be easily scaled up by upgrading the database plan.
With the database set up, we created a schema for our application using draw.io (the ERD diagram can be found in the Appendix) and started writing Laravel migrations to implement the database. The central two models of our database are User and Set (stored in *users* and *sets* tables respectively) , the latter storing all of the most important data about a LEGO set. The rest of the models within our database center around adding additional typechecking or information to the sets, such as the set's price records.
As of the current state of the project, we populate the database from multiple different sources, and using multiple tools, but this could be streamlined if this application were to evolve.

- **Seeders** - We defined some basic database seeders for including static data such as the testing admin account (to be removed for production) or set availability types
- **Python Web Scraper** - A simple python console program utilizing *Selenium* to scrape Brickeconomy for set themes and subthemes
- **Playwright Web Scraper** - The main tool for obtaining the current prices of the sets within our database, an implementation of *Playwright* that scrapes eBay for price records
- **Admin Upload Data Page** - Albeit a temporary solution, this is currently the main tool for adding new sets into the database

Adhering to the MVC framework, we hydrate our views with data by the use of Laravel Controllers. Almost every single page has its own controller, so that the data can be custom formatted and optimized to the needs of that specific page. Our controllers perform different CRUD operations, and some of them can only be accessed by the admin user, as to comply with the project's requirements of user roles and application functionality. The operations include:

- **Account CRUD** - Before a user is logged in (checked by Laravel Breeze), they are only able to see our landing page with the ability to create an account. Most of the account management functionality was already pre-provided for us by Laravel Breeze, which made the development process a lot smoother, as we had to either user the pre-existing controllers or recycle their functions. Most note-worthy, the *RegisteredUserController* manages user account creation, and the *PasswordController* manages user authentication. The rest of the controllers within the Auth directory are used within the Settings page for editing the account details and deleting the account itself.
- **Set Details CRUD** - The admin-only Upload Data page allows the admin to upload CSV files that contain information about the set they want to add into the database. We created a small, initial dataset for our application, since we knew adding sets would require significant moderation. The data sanitization and creation is handled by the *FileUploadController*.
- **User Favourites CRUD** - In the settings page, a user can select their favourite sets, themes

and subthemes from all the available ones in the database. The controller responsible for this functionality is *SettingsController*.

- **User Inventory CRUD** - From the settings page, a user is able to add a set to their own set Inventory, where they can see a summary of all the sets they own. This is handled by the *InventoryController*.
- **User Dashboard Layout CRUD** - The *DashboardController* is responsible for both providing the data for the dashboard view, as well as saving the user custom created dashboard within the Edit Dashboard Layout page.

4 Authentication and Authorization

This project's third and final task is to incorporate authentication and authorization capabilities. This section should clearly describe:

- Authentication: The different users of the system and how it is implemented
- Authorization: Summarize the access of the different users in the system and how it is implemented
- Role table: Include a role table associating actions over the system (you can think of them as use cases) and users that can perform these actions.

Resources. Lecture 7.

Length. 2 columns.

5 Conclusions

The goal of the conclusion is similar to the introduction: it summarizes the work itself and the takeaways a reader should take when reading this work. However, it can use the information presented in the work to be more specific than the introduction.

In the context of the Web Technologies course, the conclusion should clearly describe:

- Summary: summary of the work and main takeaways. Also include a class diagram of the system (the models).
- Future Work: interesting directions on how the presented work can evolve in the future (it may be the starting point to choose individual extension topics)

Length. Half a column.

References

[1] Dmitry B. Krylov, "LEGO investing as an alternative asset class: Annualized returns of 11%," *ScienceDirect*, 2021. Available at: ScienceDirect Article.

[2] Alpine.js

[3] DRY principle