Gorgeous Food

ARQSOFT 2019 / 2020

1131485 Pedro Coelho

1150863 Henrique Maciel

1160907 João Freitas

1161258 Vitor Melo

Gorgeous Food

Achieved Architecture Design

The proposed problem required a solution that enabled the management of an inventory of items to be sold to interested internal and external members of a school. This solution needed to be available from a web browser and to be built under an open-source technology stack. A database and a user directory were already existent and provided access for communication with the solution. A prototype of the solution was designed and conceived which allows the registry, removal and purchase of an item in the inventory. This prototype was built in a period of five weeks, divided in three iterations. Given the context of the problem, it was designed that the solution needed to be assembled in four components, being these:

- Gorgeous Food Web Application, or GFAW for short, that has the responsibility to provide a graphical interface in which the user can interact with the inventory management. The technology used to develop this component was React.JS, and it was followed the JavaScript conventions promoted by Airbnb. [1]
- Gorgeous Food Application Business, or GFAB for short, that has the responsibility to produce the functionalities to manage the inventory. The team opted to provide these functionalities in the form of a REST API, as REST collections represent domain aggregators and resources represent the entities of collection domain. This component was developed in C# using .NET Core 3.0 which enabled the deploy of the application for both Windows and Linux environments, as well as Entity Framework for the mapping of objects to a relational database (ORM). It was followed the coding conventions of C# that are promoted by Microsoft https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions
- Database that provides an SQL interface for storing data in a relational database.
- User Directory Server that provides an LDAP interface for accessing user information.

These components are deployed in four tiers.

For GFAB it was applied the MVC (Model View Controller) architectural pattern, which allowed a clean responsibilities separation using three subcomponents. These components allow the definition of models that produce interfaces for consuming business logic functionalities (Model), the definition of model views that represent the requests and responses specified in the REST API (View) and the handle of the API requests and external communication (Controller).

GFAW on the other hand does not follow a standard architecture, as it is built using ReactJS. This technology does not promote the use of architectural patterns to design web applications, as well as the community does not adopt any preferred architectural pattern. This component was also divided in three subcomponents, similarly to GFAB, in which there is a component that the responsibility to handle any external communication (Controller), a component for structuring any model into a JavaScript object and also a component that represents the user interface, in which it handles the user events.

[1] https://github.com/airbnb/javascript

Architecture Tradeoff Analysis Method

Design

- The User Type was set as an Enumerator Type, which lowers the modifiability in exchange for better readability and easier input validation. (Risk)
- A 4 Tier deployment pattern was adopted, giving us better scalability and lower coupling in exchange for a slower performance due to the increase in operations. (Risk)
- User activity logging management functionalities was designed to be provided by GFAB, yet the existence of a new component that handle these functionalities would increase GFAB performance. (Risk)

GFAW(Front-End)

- Chance of memory leaks hurting the software's performance. Improved performance in exchange for time spent on implementation. (No risk)
- Lack of model correctness leading to bugs or unexpected errors. Less time spent debugging and more time on design and documentation. (Risk)
- Too much feedback about errors or actions that occurred in an unexpected manner or the total lack of it. Limitation of error cases and standardization of error messages doesn't impact time of implementation. (No risk)
- Wrong stylization of elements may cause a poor application experience. The use of stable libraries reduces implementation time and increases consistency of presentation while using a bit more of system resources. (No risk)
- Difficulties in data representation that impacts usability. One of the cases are the descriptors. On the user interface, because of the model, the user needs to start with the content, set the quantity unit and finally the quantity while he should fill in an inverse order. It is possible to force the inversion, but it would no longer reflect the designed model, generating confusion amongst developers. This has a huge impact and risk, for it is needed to revise the model and possibly refactor all layers in exchange of a better data representation. (Risk)
- Automatic synchronization after each backend operation leads to worse performance in exchange for instant data updates. (No risk)

GFAB(Back-End)

- Missing an implementation of local logs regarding the web API errors so in the event of one occurring, these records can minimize the time needed for its resolution. (Risk)
- Requests that require the creation of allergens, ingredients, meal types and descriptors use services that fetch the valid inputs for these models. These inputs are stored in memory and each time a fetch is performed the inputs data is copied, which results in heavy computational operations. (Risk)