Project – FasTuga

1. OBJECTIVE

This project's objective is to implement a platform for the "FasTuga" restaurant, with a Single Page Application (SPA) on the frontend, developed with Vue.js framework, and a backend that includes a Restful API, developed with Laravel, and a Node.Js WebSocket server.

2. FASTUGA

"FasTuga" is a fast-food restaurant specialized in Portuguese food. The restaurant is located on a shopping mall and has a kitchen and a counter to deliver the food and beverages. It has access to the shared tables of the shopping mall but does not have their own tables. Customers of the restaurant will access the menu and make orders through the SPA application developed by this project. Orders are prepaid (orders can only be created after a successful payment) and each order may include several items (products of the menu) and a ticket number (cycle 1 to 99).

Chefs on the kitchen will receive a notification for each hot dish ordered (only hot dishes require cooking) and when the preparation of the dish is finished, they declare them as "ready". The employees on the counter (delivery employees) will receive notifications when the hot dishes are ready and will be responsible for completing the orders (orders can include hot dishes, cold dishes, drinks, and desserts) and deliver them to the customers. When an order is completed, the delivery employee responsible for that order will declare it as "ready", and the customer that made the order will receive a notification. The application should also include a "public board" with a list of all orders (each order is represented by a "ticket number") that are ready but not delivered yet.

Any user (including anonymous users) can make orders through the SPA application because orders only require a valid payment (the customer of an order can be "unknown"). However, users can be registered customers and have access to a point-based discount mechanism. The registered customer will get 1 point for each 10 € spent on one order (e.g., orders with total =

20,5€ or total = 29,5€ will get 2 points; total = 9,5€ will get 0 points) – points are gained on one order only – values spent on several orders are not accumulated (if a customer makes 20 orders of 9€ each, he will have 0 points). Each 10 points (only "blocks" of 10 points can be used) can be exchanged for a discount of 5€ on any order (e.g., if a customer has 23 points, he can exchange 10 points for a 5€ discount and maintain 13 points, or exchange 20 points for a 10€ discount and maintain 3 points; if the customer has 9 points, he cannot exchange them yet).

MENU

"FasTuga" menu includes hot dishes, cold dishes, drinks, and desserts. Each item (product) of the menu has a photo, a name, a description, and the current price. "FatTuga" managers are responsible for editing the menu – create, edit, or remove products, change the price of products, etc.

USERS AND CUSTOMERS

All registered customers and all employees (chefs, delivery and managers) of the "FasTuga" restaurant have an account on the platform and can login with their credentials (email + password). Each user account includes the type of user (customer, chef, delivery, manager), name, email, password, and photo. All users with an account have access and can change their profile and have access to their historical information (e.g., all orders of the customer, statistical information about prepared dishes or delivered order, etc.).

User accounts for employees are created by the managers, and user accounts for registered customers are created by the customers themselves through a register process. Managers can view and manage all accounts, including blocking/unblocking and deleting users. Registered customers are associated to a user (when registering a customer, the platform also creates the associated user) and include a phone, a NIF, the default payment type and reference, and the total points owned by him.

ORDERS

Customers (either anonymous or registered customers) compose orders by adding or removing menu items (products) to it. After the order is composed, the customer pays the order using one of the supported payment types. The order is only created after a successful payment. The initial status of the newly created order is "Preparing" (order is not ready yet). Other order status are: Ready (when order is ready to be delivered); Delivered (when order was delivered to the client)

and Cancelled (when the order was cancelled). Please note that any order may be cancelled by a manager for any reason they deem valid. When the order is cancelled, the restaurant will refund the customer the same amount that was paid and will revoke any points awarded or used in that order.

When the order is created, a "ticket number" is associated to it, so that the order is easily identified by the customer and employees. The ticket number cycles between 1 and 99 and is unique among the orders that are currently being handled (the restaurant cannot handle 99 orders simultaneously) – the order should also include a unique id that is the "real" and universal identification for that order (the orders primary key).

Each order can be associated to a registered customer (optional) and includes information about the date of the order, the payment type and reference, the price (total price of all items of the order), the total that was paid by the customer, the employee that delivered the order. It should also include information for the point-based discount mechanism, such as the points gained by the order, the total that was paid using points (discount) and how many points were used for the discount.

ORDER ITEMS

An order includes one or more items, each associated to a product and a price (the price of the item is the current price for the associated product). Each order may have several items associated to the same product (e.g., 3 items associated to "Coca-Cola") – this means that the items do not need to handle quantities.

If the product associated to the item is a hot dish, it must be prepared by a chef in the kitchen and its initial status is Waiting. When the chef starts to prepare it, the status changes to Preparing, and when the chef finishes the preparation, the status changes to Ready. If the product associated to the item is not a hot dish (it is a cold dish, drink or dessert), it does not need any preparation and its initial and only status is Ready.

Order items include a unique ID that identifies it univocally in the database. They should also include the information about the chef that was responsible for the preparation (only for hot dishes) and an order local number (1, 2, …) that identifies the item within the order that it belongs to. Using the order "ticket number" and the item "order local number" it is possible to identify a single item with the following notation: "24-3" (ticket number = 24; order local number = 3).

PAYMENTS

Order payments will be handled by an external payment gateway service (a simplified "fake" service will be provided), which the restaurant platform will consume so that the customer can pay for the orders, or receive refunds when orders are cancelled by a manager. The service supports 3 types of payments: Visa, MbWay and PayPal, and when creating a payment or a refund, it requires a payment reference. The reference for a Visa payment is the Visa Card ID with 16 digits; the reference for the MbWay is the mobile phone number with 9 digits and the reference for the PayPal is a valid email.

The external payment gateway service uses the provided payment information (type + reference) to validate and execute the transaction and returns whether the transaction was successful or not.

ADMINISTRATION

While the chefs and delivery (counter) employees are responsible for day-to-day operations on the restaurant, the managers are responsible for cancelling orders, editing the menu (products of the menu), and managing users accounts. The managers should also have access to platform usage and business-related statistical information.

NOTIFICATIONS AND REAL TIME DATA

Users of the application should receive notifications when events relevant to them happen. For instance, when a new hot dish is ordered, the chefs on the kitchen should be notified, when the ordered is ready for delivery, the customer of that order should be notified, etc. Also, relevant information that is shown on the application, such as (among others): the list of orders on the public board; the status of the customer's order and order items; the status of the hot dishes currently in the kitchen, should be updated automatically and at real time.

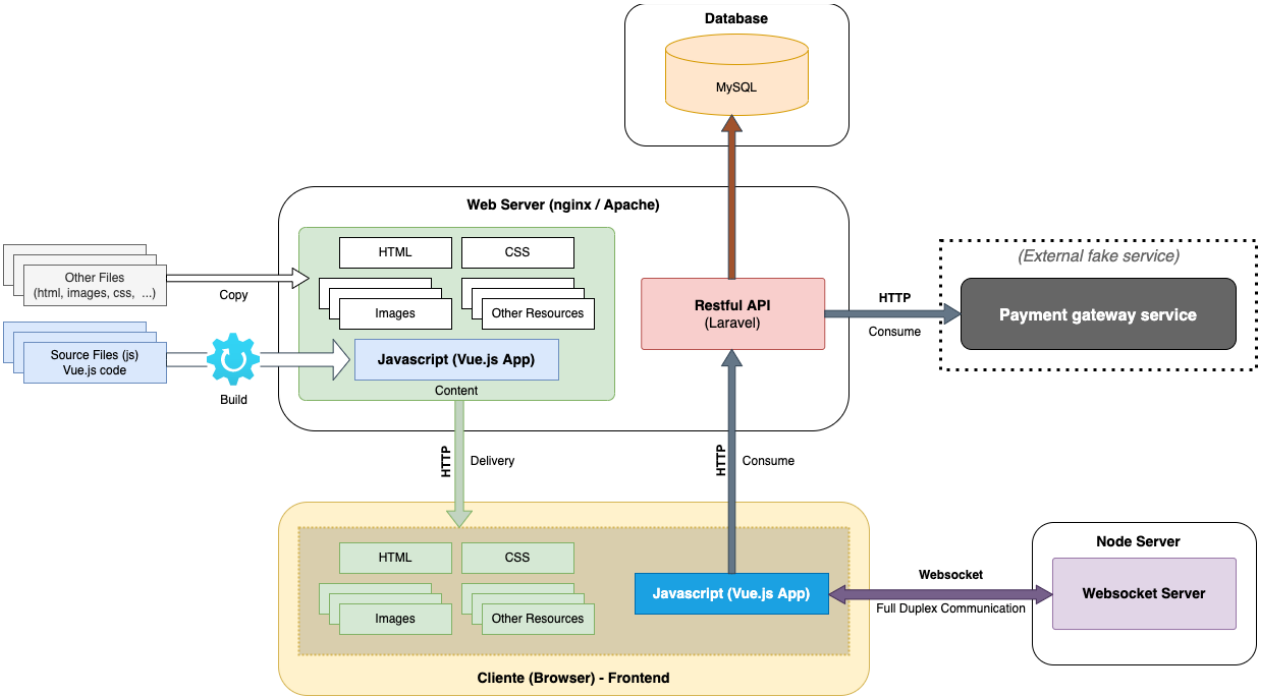
For this to work, employees are required to declare, within the application, all relevant status changes for the orders and hot dishes. For instance, the chef must use the application to declare that he has started or concluded the preparation of a specific hot dish, the delivery employee must declare that an order is ready to be delivered or has been delivered, etc.

3. PROJECT

The objective of this project is to implement the FasTuga platform, so that it complies with the business described previously. The platform includes the SPA frontend application (developed in Vue.js) and the backend with a MySQL database server, a Restful API implemented with Laravel and a Websocket server based on Node.js.

ARCHITECTURE

The following diagram describes the architecture of the project.



The client application source code will be partitioned in several Javascript source files (js), from which a build tool (webpack, vite or similar) will generate the application file (or files). The application file, which is also a Javascript file, and all the static assets (the html file, css files, images, etc.) are placed on the web server so that the application can be delivered through the Web – the application (JS file) is executed on the client (browser) but is delivered through a Web server. The application on the client (browser) will consume the Restful API service, which will be implemented in Laravel. The Laravel Restful API will read and store data on a MySQL database and use a payment gateway service (provided on a public HTTP server) to create payments and refunds. The project will also include a Websocket server, implemented in Javascript and running on a Node.js server, that will be responsible for establishing full-duplex communication between the clients (using the Node.js server as intermediary).

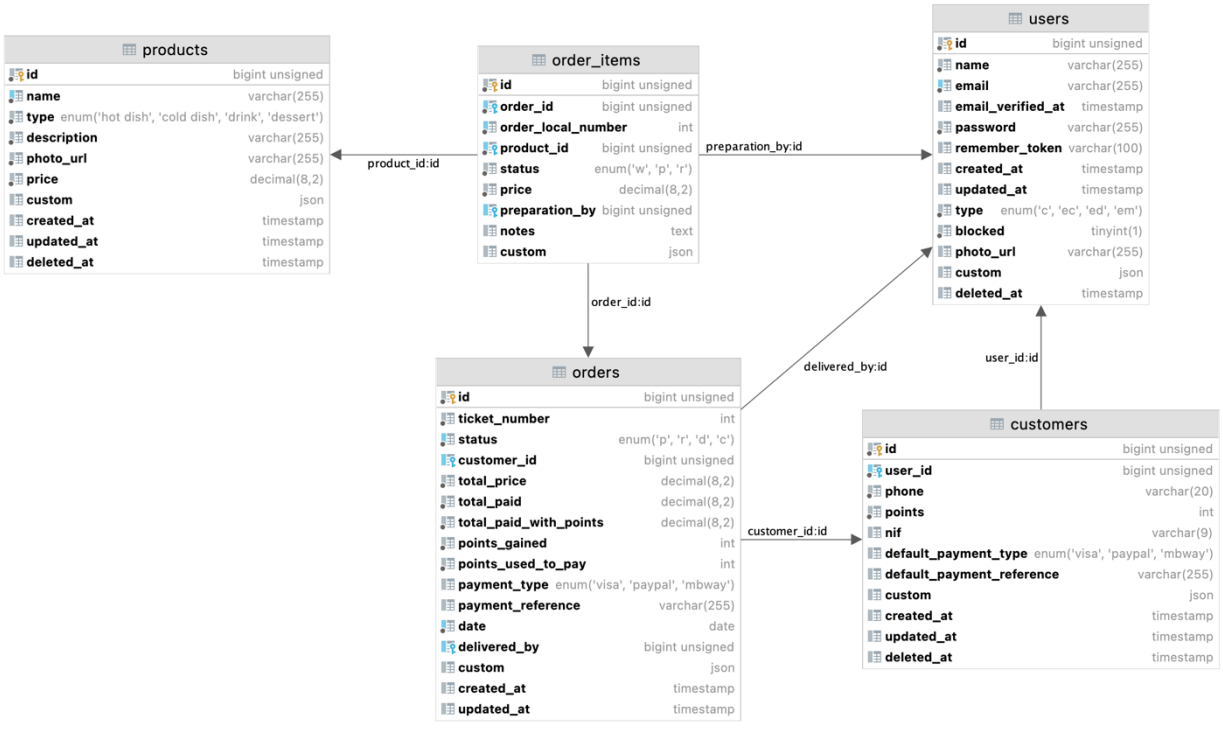
It is also possible to use other libraries, technologies or databases that complement the described architecture. For instance, “NoSQL” databases or cache technologies can be used to optimize the performance of storage operations, or queue servers to implement asynchronous operations.

DATABASE

The project must use a MySql relational database, which can be complemented by any Non-SQL database, created by the students, if they so decide. A MySQL database with sample data (created with a Laravel Migration and Seeder) is **provided** and can be used with or without further modifications. Alternatively, students can create a new MySQL database from scratch, as long as they use a MySql relational database, and the sample data simulates the application usage through 2 years with a minimum of 60 products, 2 000 registered customers, 50 000 orders and 100 000 orders items.

Please note that all user’s passwords of the provided data are “**123**”.

Provided database structure is:



- “users” types are: 'C' (Customer), 'EC' (Employee Chef), 'ED' (Employee Delivery) and 'EM' (Employee Manager);
- "Orders” status are: 'P' (Preparing), 'R' (Ready), 'D' (Delivered) and 'C' (Cancelled);
- “order_items” status are: 'W' (Waiting), 'P' (Preparing) and 'R' (Ready);

PAYMENT GATEWAY SERVICE

The platform will use an external payment gateway service to handle all payments and refunds, which is available on this URI: `https://dad-202223-payments-api.vercel.app`

The payment gateway service is a simplified "fake" service, that only simulates the payment and refund operations – no real transactions are created. The service has only 2 endpoints:

`post`	`<service URI>/api/payments`	Creates a new payment
`post`	`<service URI>/api/refunds`	Creates a new refund

Both endpoints expect a json object on the request payload with the following format (example):

```
{
    "type": "visa",
    "reference": "4283456893323321",
    "value": 34.01
}
```

When the payment or refund operation is created correctly, the service returns an HTTP Response with the status code `201 Created`. If the payment or refund operation is not executed correctly, because the request payload format is invalid or the reference is not recognized by the service or the value is over the limit, then the service returns an HTTP Response with messages that describe the problem and with the status code `422 Unprocessable Entity`. HTTP Responses can return other error status codes for other types of problems.

VALIDATION

Properties "type", "reference" and "value" of the request payload are all mandatory properties and have a set of validation rules that are verified on the service – they should also be verified on the client:

- type – only accepts these values: "visa", "paypal" and "mbway";
- reference – when type is "visa", accepts values with 16 digits (cannot start with the digit "0"); when type is "mbway", accepts values with 9 digits (cannot start with the digit "0"); when type is "paypal" accepts any valid email.
- Value – accepts any positive number (value > 0)

SIMULATION

To simulate the payment and refund operations, the "fake" service has a set of very simple rules that allows us to easily test the service. The rules **cannot be validated on the client** – they exist only to test the service.

- Visa payments are only valid when the reference starts with the digit "4".
- Mbway payments are only valid when the reference starts with the digit "9".
- Paypal payments are only valid when the reference (email) end with ".pt" or ".com".
- The limit (maximum value) for a mbway operation is 10€, for a paypal operation is 50€, and for a visa operation is 200€.

Note: When verifying the operation value is always rounded to a 2 decimal number (e.g.: 0.001 is rounded to 0.00; 0.006 is rounded to 0.01; 1.9993 is rounded to 2)

4. CONSTRAINTS

Project's **mandatory constraints** are the following:

C1. Frontend application must use exclusively the Single Page Application (SPA) paradigm.

C2. Frontend application must use the Vue.js framework for the client-side code.

C3. Restful API must be implemented with the Laravel framework.

C4. WebSocket server (if implemented) must be based on Node.js.

C5. Main relational database must use MySQL.

C6. Application must be published and accessible on the web through a desktop browser.

C7. The database of the published application must be seeded with data that simulates real life usage – both in size and content. Apply the provided seeder (Laravel seeder) - with the option "1-full" - to fill data on the published database or guarantee that the database includes sample data to simulate the platform usage through 2 years with a minimum of 60 products, 2 000 registered customers, 50 000 orders and 100 000 orders items.

5. NON-FUNCTIONAL REQUIREMENTS

Project's non-functional requirements are the following:

NF1. The client application code and structure must follow the principles and good practices of the Vue.js framework.

NF2. The server-side code and structure of the Restful API (implemented in Laravel) must follow the principles and good practices of the Laravel framework.

NF3. The Restful API must follow the principles and good practices of a RESTful service.

NF4. Visual appearance and layout should be consistent through the entire application and adapted to the applications' objective and usage context.

NF5. All implemented features must be correctly integrated and work consistently.

NF6. Application should have the best usability possible. This implies that information and available operations are clearly presented to the user, and that user interface is very simple, uses known patterns and requires the minimum user interaction possible.

NF7. Data inputted by the user should always be validated on the client and on the server, according to rules extrapolated from the business model and database.

NF8. Application should be reliable and have an optimized performance. For instance, performance can be improved by applying a mix of techniques to minimize internet bandwidth (reduce the number of HTTP Requests and the size of HTTP Responses) and client memory (limiting the size of datasets on the client) while maximizing the performance of data navigation and filtering on the client (if data is already on the client, navigation and filtering data can be very fast using JavaScript code on the client). Other techniques can be applied to the Vue.js code to improve the performance on the client, and to the Laravel and MySQL code to improve the performance on the server.

NF9. Application should be as secure as possible and it should guarantee the protection of users' data and privacy, ensuring that no unauthorized user can view or change any data or document it should not have access to. Authentication and authorization must follow the principles and good practices for authentication and authorization of Restful APIs.

NF10. Performance of the Web Socket full duplex communication should also be optimized. This implies minimizing the size of the messages and sending messages exclusively to the clients that really require it (avoid broadcasting messages to all clients if they do not need to receive them).

6. FEATURES

Students are free to implement the features they deem necessary so that the platform complies with the business model described previously. The platform may include features not directly described on this document, that make sense to include on the FasTuga business model.

7. DELIVERY AND PUBLISHING

The platform must be published in the provided virtual machine and accessible through ESTG intranet via a desktop browser (tests will be made on google chrome). **The functional evaluation will be made exclusively in the published application, so publication is not optional.**

When publishing the project, all features and configurations should be as close as possible to the application final production stage. Publishing should include all performance optimizations and must use data that **simulates real life application usage** (mandatory constraint C7).

Project delivery must include 2 files (code and report) per group and 1 individual report file per student. For example, if the group has 4 students, 6 files must be delivered (2 files related to the group and 4 individual report files) – one student delivers 3 files (2 group files and his own individual report), and the remaining 3 students deliver only their own individual reports. Files:

- prj_GG.zip – zip file that includes a copy of all the project's folders and files (source code and other files), except the folders "vendor", "node_modules" and ".git".
- grp_report_GG.xlsx – the group report file - includes group elements identification and information about the project implementation. The model for the report will be provided.
- individual_report_NNNNNNN.xlsx – the individual report file - includes self-assessment and peer review. The model for the report will be provided.

Note: replace GG with your group number and NNNNNNN with your student number

8. EVALUATION

The evaluation of the project will consider the compliance with the mandatory constraints and Non-Functional Requirements, the technical quality of the implementation and the quantity, quality and integration of the implemented features.

Weight of all evaluation criteria:

	Criteria	**Weight**
1	Non-functional requirements	**25%**
2	Technical quality of the implementation	**25%**
3	Quantity, quality and integration of the implemented features	**50%**