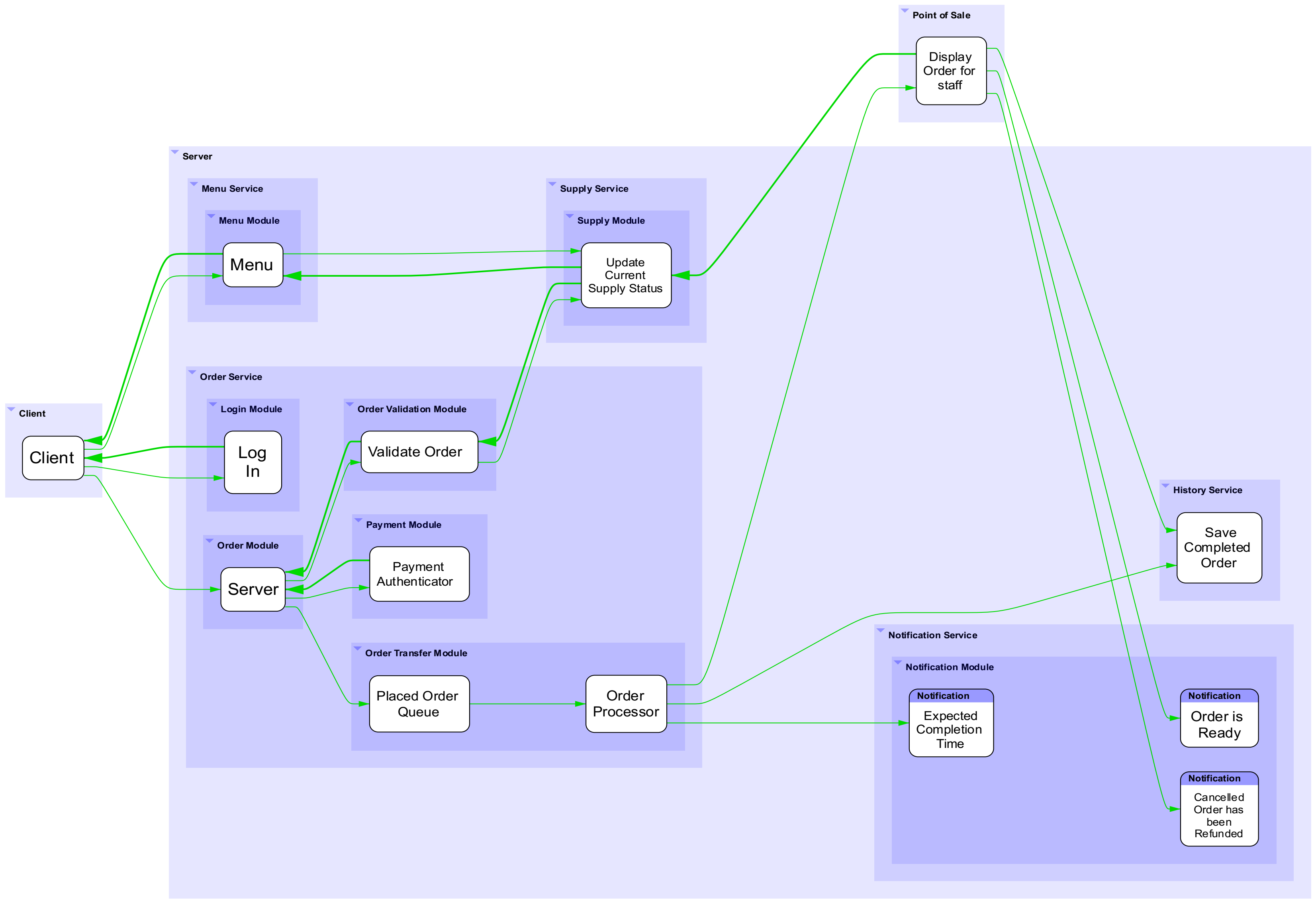
# Pizzeria Design — Miguel Muñoz



## Notes:

The Client communicates with modules, through two services: It calls the Menu Service to display the menu. And through the Order Service, it calls the Login Module to log in, and the Order Module to place an order. These are the services for which the client needs an immediate feedback. For example, if the payment fails to authenticate, the user must be told immediately.

### Menu Module

Upon first connecting, the Menu Module sends the current menu to the client. For speed, this can be stored in a Redis Cache, if necessary. The Menu Module also communicates with the Supply Module to find out which MenuItems have run out, although this doesn't get updated on the

### Login Module

The user may log in to save address and payment information, but doesn't need them to place an order.

### Client

The client has the complete menu, so it is capable of creating an order without input from the server. When the user is done creating the order, the client uses the Order Module to submit it to the server.

When the client places the order, they have the option of paying immediately or paying on pickup. If they are paying immediately, the Order Module validates the payment using the Payment Module, and tells the user if the payment fails.

Once the order and possible payment are validated, the server adds it to the Placed Order queue.

### Payment Module

The Payment Module checks the validity of the credit card or other payment package like PayPal, and authorized payment.

### Order Module

When the user places an order, the server first validates the order using the Order Validation Module, to determine if anything has run out. (The client could also conceivably do this by asking for menu updates as it builds the order.) It will also process the optional payment using the Payment Module. Once these are complete, the user may put their phone to sleep and resume what they were doing. The Order Module places the order on the Placed Order Queue.

## Order Validation Module

## Order Transfer Module

The Order Processor takes the order off the Placed Order Queue and submits it to the Point of Sale. It also sends the incomplete order to the History Module, which records it in the database as an incomplete order in the database. It also notifies the user when the item will be ready.

## Point of Sale

The Point of Sale server (not pictured) determines how long before the order is ready, and calls the notification service to notify the user when it will be ready. This must be done at the point of sale, because that's where they know which supplies have run out. It also processes cancellations, and calls the Notification Service to tell the user that their money has been refunded.

## Supply Module

The Supply Module communicates with the Point of Sale to learn of which MenuItems have run out for the day. It notifies the menu service, which can transmit the information to the client.

## History Module

The History Module keeps track of orders and receives updates when the order status changes.

## Notification Module

The Point of Sale notifies the user that the order is ready through the notification service. It also sends the updated order to the history service with the order status changed from incomplete to complete.

If a customer requests a refund, the Point of Sale process the refund, notifies the Payment Module to refund the money, and notifies the History Service that the order has been cancelled.

## Note on Modules and Services

These modules are broken up by functionality. Right now, many of them are combined into a single service, but sometimes it may make sense to break them out into separate microservices, but that's just an implementation detail. Some modules need to stay together. For example, the Login Service and the Order Service should be part of the same microservice to process user payments securely. The Menu Service, however, is essentially a read-only service, and doesn't need to be secure.

## Entities

```
User
  id:*           Integer
  firstName:     String
  lastName:      String
  address:       Address
  mobilePhone:†  String
  email:†        String
  payment:       PaymentMethod
  role:          enum [CUSTOMER, ADMIN]

Address
  id:*       Integer
  street:*   String
  city:*     String
  state:*    String
  postal:*   String

PaymentMethod
  id:*           Integer
  type:*         enum [CARD, PAYPAL] (Other options may be added later)
  creditCard:†   Card
  payPal:†       PayPal

MenuItem
  id:*               Integer
  name:*             String
  itemPrice:*        BigDecimal
  parentId:          Integer
  activeVersion:*    String
  inactiveVersion:   String
  duplicates:        Integer

OrderNode
  id:*           Integer
  orderId:       Integer
  parentTree:    OrderNode
  MenuItemId:*   Integer
  options:*      OrderNode[]

Card
  number:*   String
  expire:*   String
```

```
PayPal
  PaypalId:* String

Order
  id:*             Integer
  customer:*       User
  payment:         PaymentMethod
  status:*         enum (PENDING, COMPLETE, CANCELLED)
  purchaseTree:*   OrderNode[]

Graphic
  id:*           Integer
  dimensionx:*   Integer
  dimensiony:*   Integer
  imageData:*    String (path to image file)

SystemProperties
  MenuVersion*   String
```

* Required Field
† At least one field marked with † is required.

## Notes:

The database design needs to make a distinction between available options and chosen options. For example, A basic pizza can have options for pepperoni, olives, mushrooms, and green peppers. IDs for all these options would specify the pizza MenuItem as their parent MenuItem. But the pizza could also be an option for another menu item, like a Pizza Combo, which combines a pizza with a small salad. So each menu item would be the root of a tree. Each menu item could have options, and each option could have more options. To do this, each entity would link back to its parent entity. This is how the MenuItem entity works.

But a customer will only choose some of the available options. So the chosen items will also be a tree, but with only some of the options. To do this, I created the OrderNode entity. Each OrderNode wraps a single MenuItem. Each OrderNode has an optional parent OrderNode, and potentially children. So this forms a second tree structure that wraps nodes from the first structure.

Some menu items will need to be duplicated. For example, pizza toppings could appear on multiple size pizzas. Also, any size pizza could be ordered by itself, or as part of a combo with a salad. So the MenuItem table has a Duplicates field, which contains an id to another MenuItem. This allows an item to exist in multiple MenuItem trees, but which can be updated at a single point. This is because (as far as I know) an entity can't be in a many-to-many relationship with itself.

To illustrate all of this, imagine the following order:

1 Personal Pizza Combo, with Green Pepper and Olives, and Ranch Dressing for the salad.
1 26-inch Pizza with Pepperoni, Sausage, Ham, Mushrooms, and Green Peppers
1 2-Liter bottle of Coke

The order would look like this:

```
Order
├─ Personal Pizza Combo
│   ├─ 12-inch Pizza
│   │   ├─ Green Peppers
│   │   └─ Olives
│   └─ Green Side Salad
│       └─ Ranch Dressing
├─ 26-inch Pizza
│   ├─ Pepperoni
│   ├─ Sausage
│   ├─ Ham
│   ├─ Mushrooms
│   └─ Green Peppers
└─ 2 Liter Coke
```

Except for the parent node, which is an Order, each element of this tree is a OrderNode.

MenuItems have an activeVersion and inactiveVersion. Menu items that have been disable need to stay in the database. This way, when the menu changes, the history table can store past orders containing items that are no longer on the menu. This is accomplished by the activeVersion and inactiveVersion fields. For example, if the currentMenuVersion is 2.0, any menu items with inactive Version less than 2.0 are inactive, and any menu items with an activeVersion greater than 2.0 are not yet active. This eases rollout of a new menu. All items of menu version 2.1 can be added to the menu ahead of time, but the menu will not change until the actual menu version is bumped up to 2.1. Menu items are active when two conditions are met:
1. Their activeVersion number is greater than or equal to the menu version.
2. Their inactiveVersion must be null, or greater than the menu version.
So, when preparing to change the menu version from 2.0 to 2.1, you may schedule any menu items for removal by setting its inactiveVersion number to 2.1. You may schedule any new items for addition by setting their activeVersion number to 2.1. Since the current version is still 2.0, the active menu will not change. The changes will go into effect when the menu version is changed to 2.1.

The Graphic entity allow the html generator to specify the size of the image before it is downloaded, so loading of images doesn't continually alter the layout as images load.