Getting started with Ruby

First of all you should decide which kind of integration to use. You have two types of possible integrations:

- **Centralized server:** The restaurants interacts with an application server
- **Point to point:** The restaurants interacts directly with PedidosYa API

Note: When you are using point to point method then you need an extra pair of credentials. This extra pair of credentials correspond to the restaurant itself.

Authentication

For accessing the API you need to create a *Credentials* object.

Centralized credentials example

```
credentials = ReceptionSdk::Credentials.new
credentials.client_id = "your_client_id"
credentials.client_secret = "your_client_secret"
credentials.environment = ReceptionSdk::Environments::DEVELOPMENT
```

This kind of credentials will work for all the restaurants handled by the third party application.

Point to point credentials example

```
credentials = ReceptionSdk::Credentials.new
credentials.client_id = "your_client_id"
credentials.client_secret = "your_client_secret"
credentials.username = "restaurant_username"
credentials.password = "restaurant_secret"
credentials.environment = ReceptionSdk::Environments::DEVELOPMENT
```

This credentials will only work for the specified restaurant.

Environments

This SDK supports multiple environments:

- **Development:** Default environment, used for development only.
- **Staging:** Pre-production environment, used before every release for final testings.
- **Production:** Production environment, used for real life.

Note: Please be sure that you deploy the application with the right environment.

Basic usage

Whenever you want to use the SDK, you must download the reception_sdk gem and use the classes that you need.

```
require 'reception_sdk'

credentials = ReceptionSdk::Credentials.new
```

Once you have the *Credentials* object created you must create an *ApiClient* object using the previous credential object.

```
credentials = ReceptionSdk::Credentials.new
credentials.client_id = "your_client_id"
credentials.client_secret = "your_client_secret"
credentials.environment = ReceptionSdk::Environments::DEVELOPMENT

begin
  api = ReceptionSdk::ApiClient.new(credentials)
  delivery_times = api.order.delivery_time.get_all
rescue ReceptionSdk::ApiException => ex
  puts "Uups an error has ocurred!"
end
```

Important: You must instantiate the *ApiClient* at the very beginning of your application and save that instance of *ApiClient* (for ex: in an static variable) and use that instance in the whole life cycle of your application.
Please do not instantiate the *ApiClient* inside a for loop for example.

Orders operations

First of all, you must import our delivery times and rejection messages, what is a delivery time? and a

rejection message?.

- **Delivery time:** Promised interval that order should be delivered.
- **Reject message:** If the restaurant can't accept the order then it must be rejected with one of the predefined reasons.

Important: *DeliveryTimes* and *RejectMessages* are dynamic, as well as their attributes, so do not hardcode them.
A good practice is to import them every day or on application start up, and save them somewhere in your application.

You can get all the delivery times using this operation

```
delivery_times = api.order.deliver_time.get_all
```

List of delivery times

Code	Description ES	Description PT
1	Entre 15' y 30'	Entre 15' e 30'
2	Entre 30' y 45'	Entre 30' e 45'
3	Entre 45' y 60'	Entre 45' e 60'
4	Entre 60' y 90'	Entre 60' e 90'
5	Entre 90' y 120'	Entre 90' e 120'
6	24 horas	24 horas
7	48 horas	48 horas
8	72 horas	72 horas
9	Entre 120' y 150'	Entre 120' e 150'
10	12 horas	12 horas
11	Entre 150' y 180'	Entre 150' e 180'

Then, you can get all the reject messages

```
reject_messages = api.order.reject_message.get_all
```

There are two important attributes in *RejectMessage* that you have to consider: forLogistics, forPickup.

When rejecting an order that has the flag logistics in true, you must not use/show reject messages that have forLogistics in false.
The same rule applies to orders with the flag pickup in true.
The field to show in your system is *DescriptionES*.

Example:

```
my_saved_reject_messages = saved_reject_messages()
reject_messages_for_this_order = my_saved_reject_messages.clone

if order.logistics
    reject_messages_for_this_order.each do |reject_message|
      if !reject_message.for_logistics
        reject_messages_for_this_order.delete(reject_message)
      end
    end
end

if order.pickup
    reject_messages_for_this_order.each do |reject_message|
      if !reject_message.for_pickup
        reject_messages_for_this_order.delete(reject_message)
      end
    end
end
```

Getting new orders

You should add an entry in your server cron that ask for new orders every minute.

Get new orders to iterate over them:

```
orders = api.order.get_all(ReceptionSdk::OrderState::PENDING, ReceptionSdk:
:PaginationOptions.create)
orders.each { |order|
  do_something(order)
}
```

When getting pending orders, the second parameter (PaginationOptions) is ignored and **all** pending orders are returned.
Pagination options is considered when requesting confirmed or rejected orders.

Or you can use lambdas to process each order:

```
on_success = ->(order, api) {
  if everything_ok(order)
    api.order.confirm(order.id, delivery_time_id)
    return true # If the order is processed, false otherwise
  end
}
on_error = ->(ex) {
  raise ex
}
api.order.get_all(ReceptionSdk::OrderState::PENDING, ReceptionSdk::Paginati
onOptions.create, on_success, on_error)
```

Note: Keep in mind that you will receive one order per callback so be aware that your import mechanism shouldn't take so long.

Regular orders

The client order for food when the restaurant is opened. These kind of orders are for immediately delivery and they have the properties *pickup*, *preOrder* and *logistics* in false. The restaurant should delivery the order approximately at the time specified in *deliveryDate*

Pickup orders

Some orders are not for delivery, instead the client will pick up the order in the restaurant. All the pick up orders have the property *pickup* in *true*, *address* in *null* and the estimated pickup time in *pickupDate*.

Anticipated orders

The client can order for food before the restaurant it's opened, if that's the case then the *preOrder* property has the value *true*. The restaurant should delivery the order at the time specified in *deliveryDate*.

Logistics orders

This only applies if the restaurant delivery it's handled by PedidosYa's fleet, otherwise you should ignore this.
All orders with logistics are identified by the property *logistics* in *true*. In this case you must pay attention to the *pickupDate* property, this field contains the time that one rider will pick up the order to be delivered.

Orders with discounts

We handle two kinds of discounts as specified in the property *type* in the *Discount* Model:

- **PERCENTAGE:** Discount specified by a percentage value
- **VALUE:** Discount specified by an amount of money

The Order model contains a list of discounts, with the corresponding priority of each discount.

Online paid orders

If the order was paid with an online payment method then the property *online* in the *payment* property of the order is in *true*.

Confirm an order

Once you processed the order and it's ready to be cooked you must confirm it indicating the estimated delivery time.

Note: An order must be confirmed before three minutes.

Note: For *Regular orders* and *Pick up orders* the method for confirmation it's the same, it's mandatory to specify the delivery time when the order is going to be confirmed, otherwise you must confirm the order without a delivery time, if you specify one, you are going to receive an error.

```
begin
  delivery_time_id = restaurant_selected_time()
  order = order_to_confirm()

  if order.pre_order || order.logistics
    result = api.order.confirm(order.id)
  else
    result = api.order.confirm(order.id, delivery_time_id)
  end

  if result
    confirm_order_locally()
  end
rescue ReceptionSdk::ApiException => ex
  if ex.code == ReceptionSdk::ErrorCode::INTERNAL_SERVER_ERROR
    # Wait 5 seconds and retry again
    # Don't retry more than three times
  else
    raise ex
  end
end
```

You can use order and delivery time ids directly, or an object with an id property.

Reject an order

If you can't process the order or the order cannot be delivered for some reason you should reject it.

```
begin
  reject_message_id = restaurant_reject_reason()
  order_id = order_to_reject()

  result = api.order.reject(order_id, reject_message_id)

  if result
    reject_order_locally()
  end
rescue ReceptionSdk::ApiException => ex
  if ex.code == ReceptionSdk::ErrorCode::INTERNAL_SERVER_ERROR
    # Wait 5 seconds and retry again
    # Don't retry more than three times
  else
    raise ex
  end
end
```

You can use order and reject message ids directly, or an object with an id property.

You can also reject an order with a text note.

```
begin
  reject_message_id = restaurant_reject_reason()
  order_id = order_to_reject()
  reject_note = 'Rejection note'

  result = api.order.reject(order_id, reject_message_id, reject_note)

  if result
    reject_order_locally()
  end
rescue ReceptionSdk::ApiException => ex
  if ex.code == ReceptionSdk::ErrorCode::INTERNAL_SERVER_ERROR
    # Wait 5 seconds and retry again
    # Don't retry more than three times
  else
    raise ex
```

```
  end
end
```

Note: An order must be rejected before three minutes

Dispatch an order

Once the order is cooked and it is ready to deliver, you must call this method to indicate the order is on the way to the client address.

```
begin
  order_id = order_to_dispatch()

  result = api.order.dispatch(order_id)

  if result
    dispatch_order_locally()
  end
rescue ReceptionSdk::ApiException => ex
  if ex.code == ReceptionSdk::ErrorCode::INTERNAL_SERVER_ERROR
    # Wait 5 seconds and retry again
    # Don't retry more than three times
  else
    raise ex
  end
end
```

Retrieve orders of the day

It's possible to retrieve the last confirmed or rejected orders of the day. For example, to retrieve the last confirmed orders you should do it this way:

```
orders = api.order.get_all(ReceptionSdk::OrderState::CONFIRMED, ReceptionSd
k::PaginationOptions.create)
if orders.length > 0
  do_something(orders)
end
```

In case you want the rejected ones just change *OrderState::$CONFIRMED* with *OrderState::$REJECTED*

Note: This will not return **all** the orders, the response is limited to the last 15 orders as explained

in the pagination section.
A good practice it's to save the order id in your system.

If you have the order id then you can ask for a particular order.

```
order_id = order_id()
order = api.order.get(order_id)
if order
  do_something(order)
end
```

Retrieve order tracking info for Logistics orders

It's possible to retrieve the order tracking info for Logistics orders. This could be used to have a more accurate date for which the driver retrieves the order from the store. This is useful for updating the order's pickupDate and deliveryDate fields. The available information is :

- The driver's coordinates and the driver's name.
- The pickup date: Date for when the driver picks up the order at the store. Updated until the state is PREPARING (included). With this data it's possible to update the order's pickupDate.
- The estimated delivery date: Estimated date for when the driver arrives at the user's destination and delivers the order. Updated until DELIVERED state. This field corresponds with the order's deliveryDate field and that's the one that should be updated. The first time the tracking info is retrieved, it's probable that the order's deliveryDate coincides with this field. With this data it's possible to update the order's deliveryDate.
- The state of the tracking: These are the possible states for the order tracking:
    - FAILURE: State of the tracking when it failed. The rest of the fields will be null.
    - REQUESTING_DRIVER: A driver is being requested to pickup and deliver the order. The pickupDate and the estimatedDeliveryDate will be equal to the order's pickupDate and deliveryDate fields respectively.
    - TRANSMITTING: The order is being sent to your system. All the information is available.
    - TRANSMITTED: The order was finally sent to your system. All the information is available.
    - PREPARING: The order is being prepared. All the information is available.
    - DELIVERING: The order is being delivered by a driver. Pickup date is null.
    - DELIVERED: The order was finally delivered. All data is null.
    - CLOSED: The order is closed and no live tracking happens. All data is null.

The pickup date and the estimated delivery date are the necessary values to update in the order. The driver's name and coordinates are part of the aditional information attached to the order tracking info.

To do that, you should do it in this way:

```
order_id = order_id()
order_tracking = api.order.get_tracking(order_id)
```

```
unless order_tracking.nil?
  tracking_state = order_tracking.state
  # driver class
  driver = order_tracking.driver
  # location class, lat & lng attributes for the driver's coordinates
  location = driver.location
  if tracking_state == TrackingState::PREPARING
    update_pickup_date(order_tracking)
  end
end
```