Universitat de Lleida

Upgrade Sprint 3 : Proxiprizes

Content:

Summary of general upgrades made in Sprint 3:

- More visual presentation
- In this sprint we have meet more times and did the proper documentation for every decision made
- Notifications developed
- Loadbalancer and node-cache
- ProApp with front-end updated
- Changes in economical viability document in order to reflect the received feedback

1. Technological section

The main changes that occurred during this second sprint were related to the back-end's settlement.

As we previously explained it in the architecture diagram, we chose to use node.js in order to implement the back end server so that we could use javascript for both client and server sides.

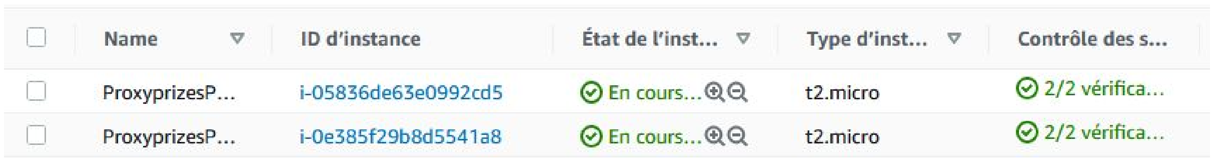
Deployment solution and LoadBalancer

One of the biggest upgrades we had to make during this last sprint was finding a solution to deploy our App. As we had already been using the AWS S3 bucket solution to store our images, we decided to have a look to the AWS solutions to deploy our servers.

After reading some documentations, we decided to choose the AWS EC2 solution.

We decided to create two different EC2 instances :

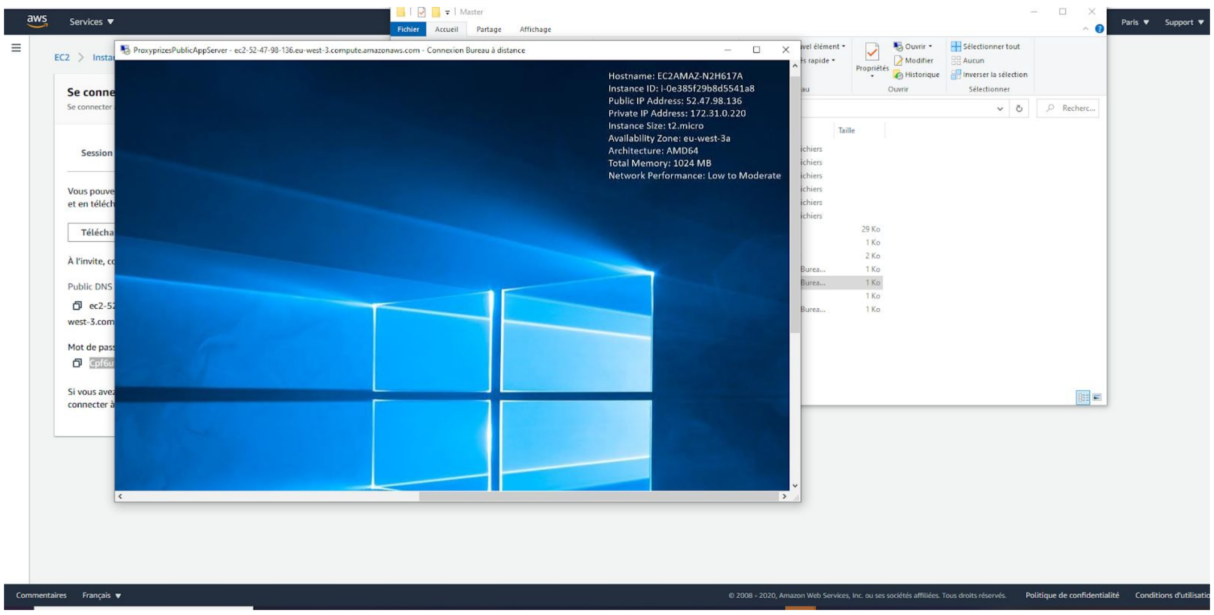
- One that will embody our public server
- The other for the pro server



	Name	ID d'instance	État de l'inst...	Type d'inst...	Contrôle des s...
☐	ProxyprizesP...	i-05836de63e0992cd5	✓ En cours...	t2.micro	✓ 2/2 vérifica...
☐	ProxyprizesP...	i-0e385f29b8d5541a8	✓ En cours...	t2.micro	✓ 2/2 vérifica...

Once the instance is launched, it is possible to connect it using a RDP client.

The RDP client allows us to access our instance remotely.

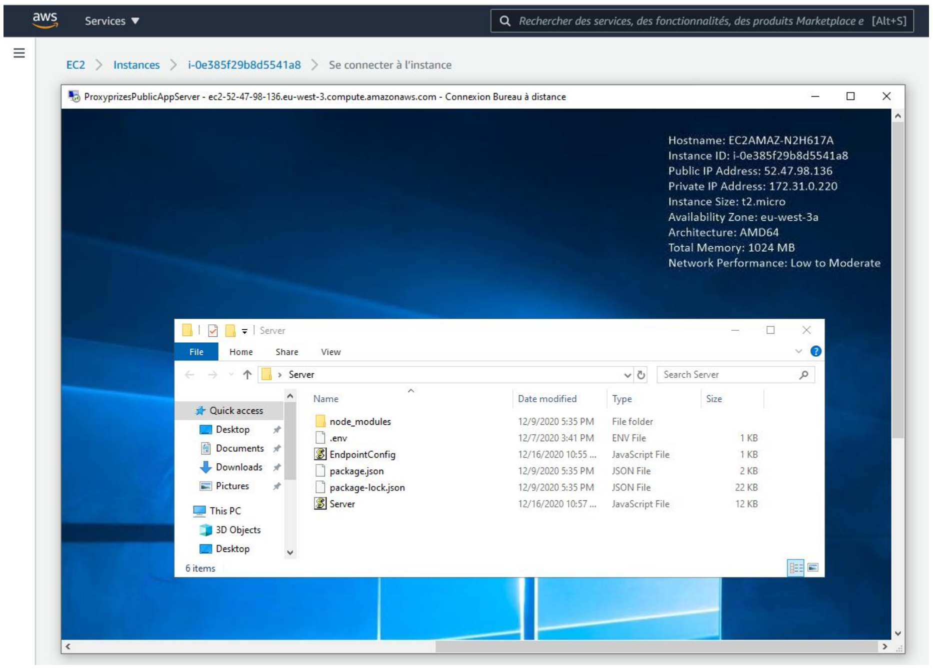


ProxyprizesPublicAppServer - ec2-52-47-98-136.eu-west-3.compute.amazonaws.com - Connexion Bureau à distance

Hostname: EC2AMAZ-N2H617A
Instance ID: i-0e385f29b8d5541a8
Public IP Address: 52.47.98.136
Private IP Address: 172.31.0.220
Instance Size: t2.micro
Availability Zone: eu-west-3a
Architecture: AMD64
Total Memory: 1024 MB
Network Performance: Low to Moderate

Once connected to the instance, we had the opportunity to install all the required environnement. We installed Node.js and WAMP.

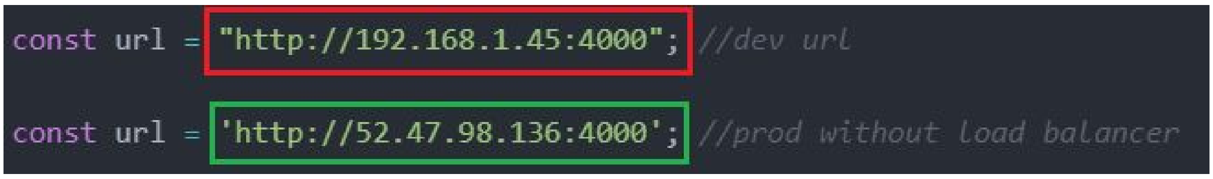
When then added our Servers scripts, installing within the EC2 instances the required dependencies such as express package or the MySql driver for instance.


aws
Services ▼
Rechercher des services, des fonctionnalités, des produits Marketplace e [Alt+S]
EC2 > Instances > i-0e385f29b8d5541a8 > Se connecter à l'instance
ProxyprizesPublicAppServer - ec2-52-47-98-136.eu-west-3.compute.amazonaws.com - Connexion Bureau à distance
Server
File
Home
Share
View
Search Server
Quick access
Desktop
Documents
Downloads
Pictures
This PC
3D Objects
Desktop
6 items

```
Hostname: EC2AMAZ-N2H617A
Instance ID: i-0e385f29b8d5541a8
Public IP Address: 52.47.98.136
Private IP Address: 172.31.0.220
Instance Size: t2.micro
Availability Zone: eu-west-3a
Architecture: AMD64
Total Memory: 1024 MB
Network Performance: Low to Moderate
```

Name	Date modified	Type	Size
node_modules	12/9/2020 5:35 PM	File folder	
.env	12/7/2020 3:41 PM	ENV File	1 KB
EndpointConfig	12/16/2020 10:55 ...	JavaScript File	1 KB
package.json	12/9/2020 5:35 PM	JSON File	2 KB
package-lock.json	12/9/2020 5:35 PM	JSON File	22 KB
Server	12/16/2020 10:57 ...	JavaScript File	12 KB

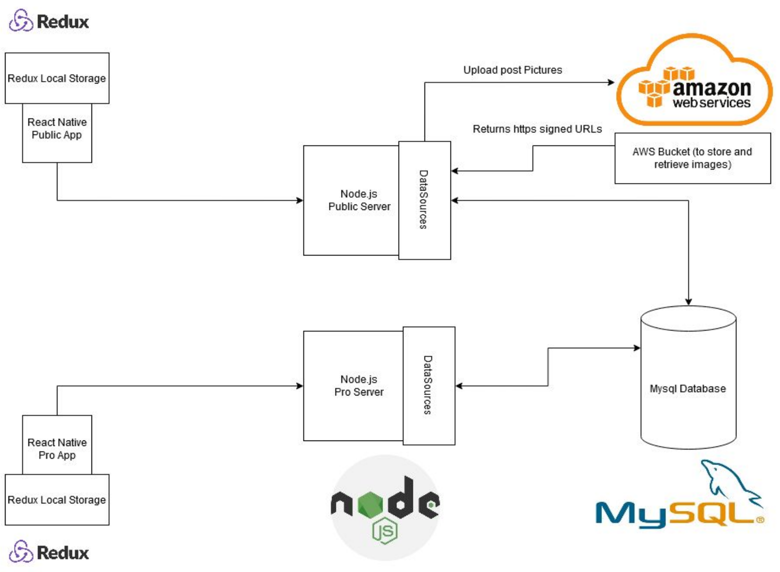
At this point we were able to access the EC2 instances from our client app by changing the local dev URL to the IP of the EC2 instance.



```
const url = "http://192.168.1.45:4000"; //dev url

const url = 'http://52.47.98.136:4000'; //prod without load balancer
```

In red is the dev URL that we were using to test functions from our local server.
In green is the ip address of the remote EC2 instance which heberges our server and DB.

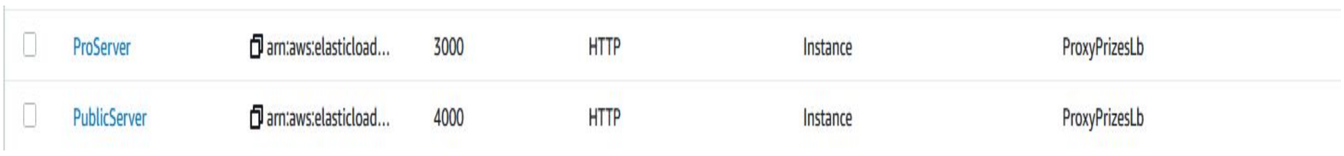
At this point, both pro and public servers were deployed but we still had such architecture :



We wanted our Pro and public App to hit the same machine and then to be redirected whether a request came from the pro or the public App.

We decided to configure an AWS LoadBalancer between or two EC2 instances.

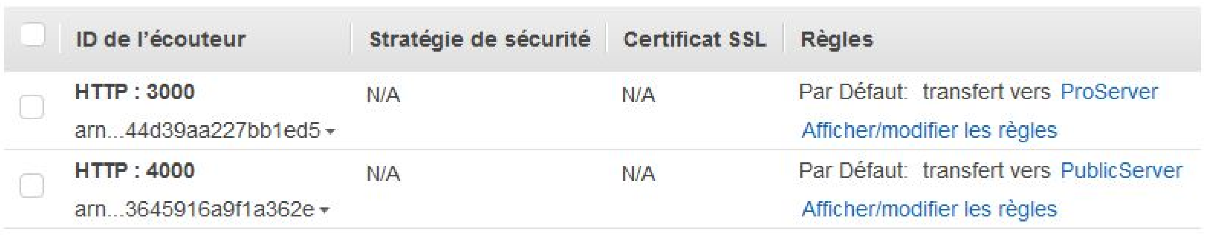
To do so, we first had to define target groups that will cluster our Pro and Public servers :



☐	ProServer	arn:aws:elasticload...	3000	HTTP	Instance	ProxyPrizesLb
☐	PublicServer	arn:aws:elasticload...	4000	HTTP	Instance	ProxyPrizesLb

Then we had to create the LoadBalancer.

For each AWS LoadBalancer, it is possible to define listeners with specific rules.

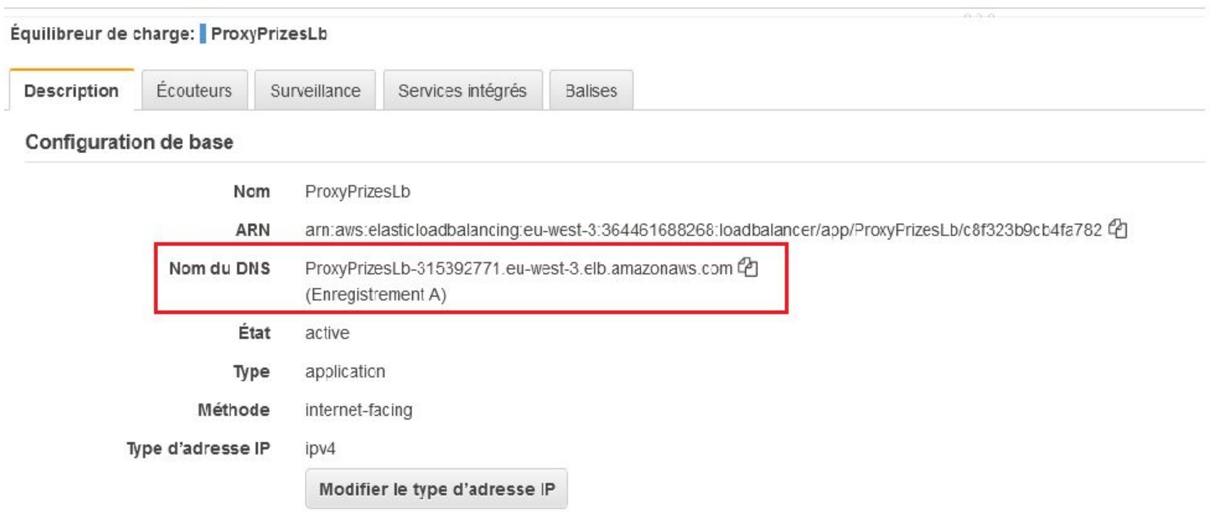


☐	ID de l'écouteur	Stratégie de sécurité	Certificat SSL	Règles
☐	**HTTP : 3000** arn...44d39aa227bb1ed5	N/A	N/A	Par Défaut: transfert vers ProServer Afficher/modifier les règles
☐	**HTTP : 4000** arn...3645916a9f1a362e	N/A	N/A	Par Défaut: transfert vers PublicServer Afficher/modifier les règles

Each time our LoadBalancer receives a connection request on a specific port, it will trigger a specific action to do.

In our case, if the LoadBalancer receives a request connection on port 3000, it will redirect the request to the Pro EC2 instance and if it receives a request connection on port 4000, it will redirect the request to the Public EC2 instance.

At this point, both our pro and public client applications are requesting using the same LoadBalancer DNS :



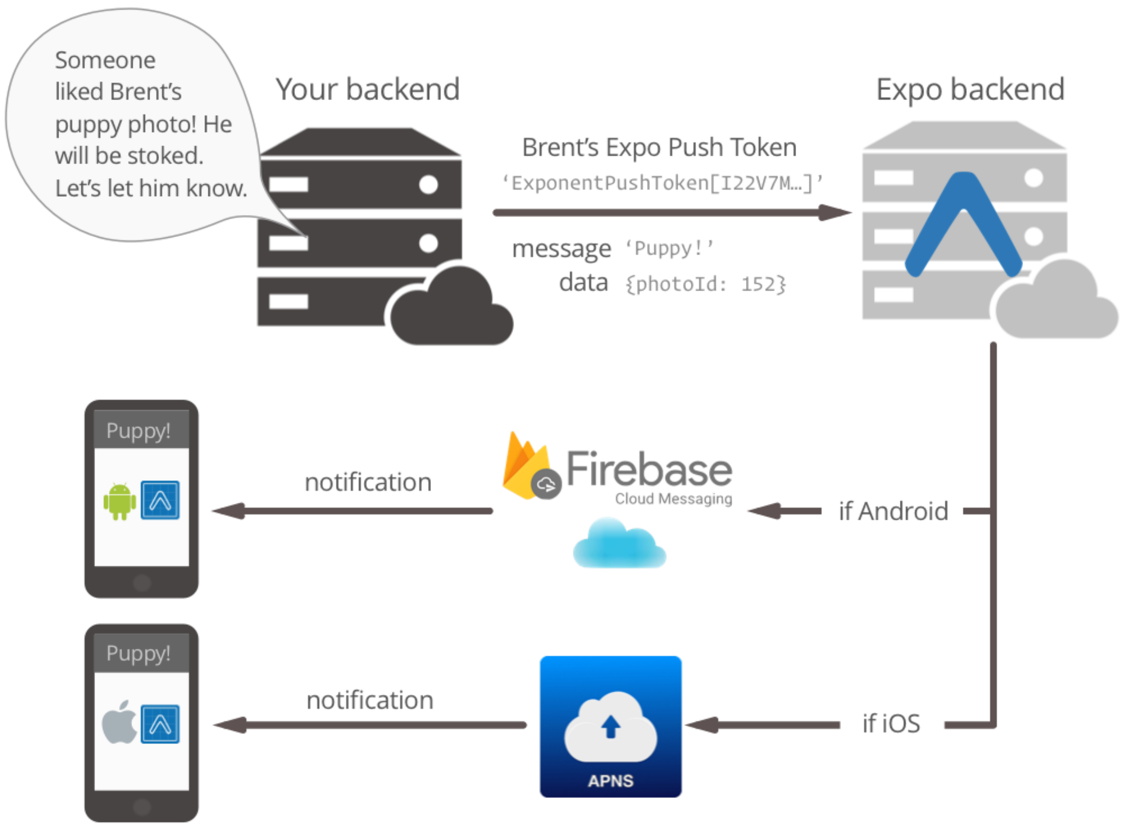
Équilibreur de charge: ProxyPrizesLb

Configuration de base

Nom	ProxyPrizesLb
ARN	arn:aws:elasticloadbalancing:eu-west-3:364461688268:loadbalancer/app/ProxyPrizesLb/c8f323b9cb4fa782
Nom du DNS	ProxyPrizesLb-315392771.eu-west-3.elb.amazonaws.com (Enregistrement A)
État	active
Type	application
Méthode	internet-facing
Type d'adresse IP	ipv4

Notifications

The feature of notifications was included in the App, the solution chosen was the Expo Notifications.

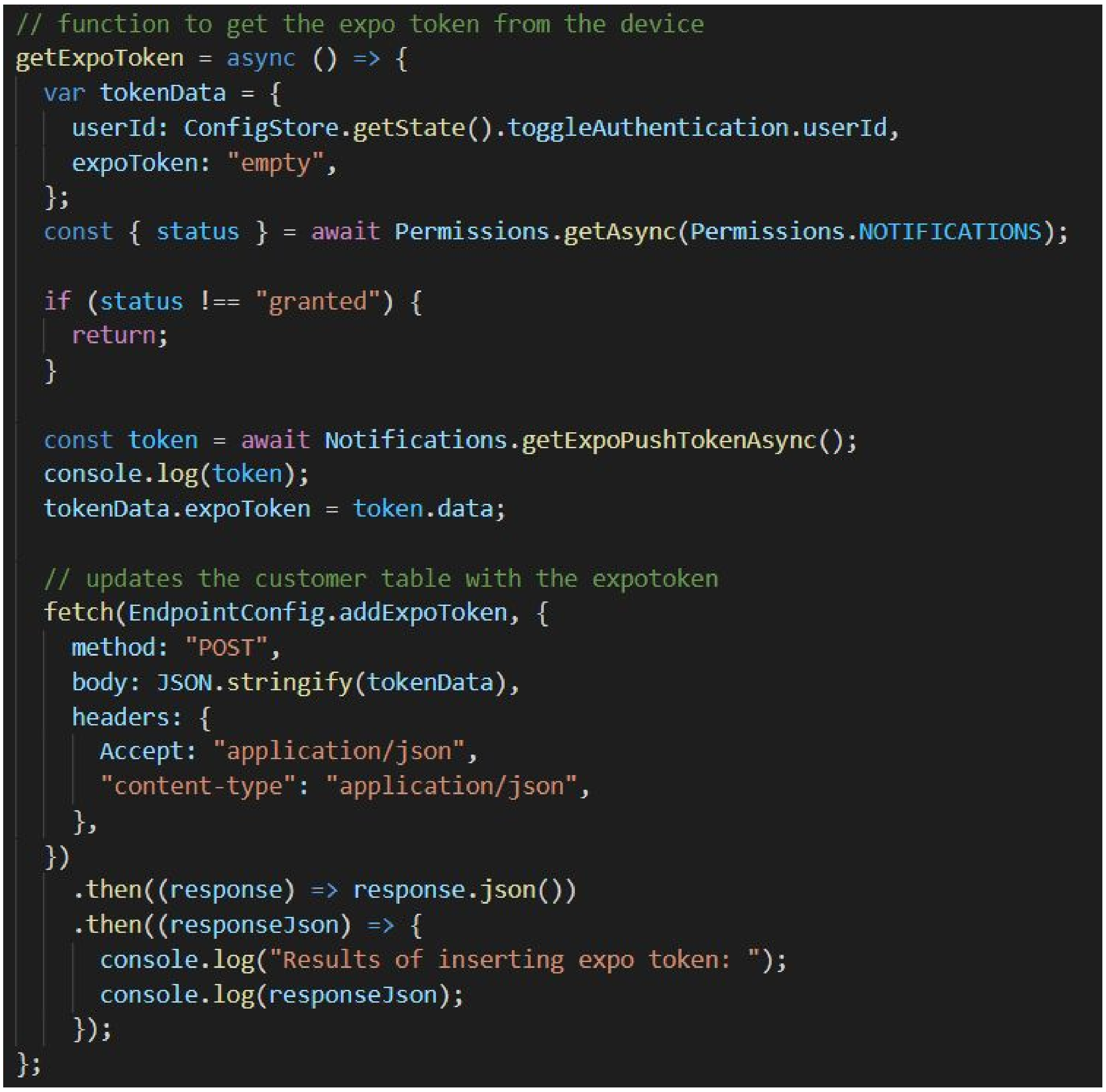


```
import * as Notifications from "expo-notifications";
import * as Permissions from "expo-permissions";
```

The good side of this component is that the expo backend handles the notification to both apple and android servers, you just need to send the requisition to the expo server with the notifications properties (like body, title, and to who are you sending the notifications).

For testing, you can use the tool Expo Push Notifications Tool to test sending one notification, to see if you properly got the correct token.

That said, we need to first get the ExpoToken from the users and store it in the database.



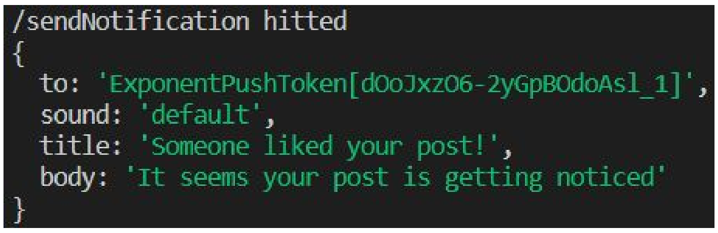
```
// function to get the expo token from the device
getExpoToken = async () => {
  var tokenData = {
    userId: ConfigStore.getState().toggleAuthentication.userId,
    expoToken: "empty",
  };
  const { status } = await Permissions.getAsync(Permissions.NOTIFICATIONS);

  if (status !== "granted") {
    return;
  }

  const token = await Notifications.getExpoPushTokenAsync();
  console.log(token);
  tokenData.expoToken = token.data;

  // updates the customer table with the expotoken
  fetch(EndpointConfig.addExpoToken, {
    method: "POST",
    body: JSON.stringify(tokenData),
    headers: {
      Accept: "application/json",
      "content-type": "application/json",
    },
  })
    .then((response) => response.json())
    .then((responseJson) => {
      console.log("Results of inserting expo token: ");
      console.log(responseJson);
    });
};
```

This function asks for the user the permission to send notifications, if its true, then insert the expoToken in the database for the corresponding user through the fetch to AddExpoToken.

To test this function, one of the features implemented was the like button function.
When pressed, besides updating the valuecounter in the database and also changing the state on the display, it tries to send a notification to the user who is the owner of the post that was liked.

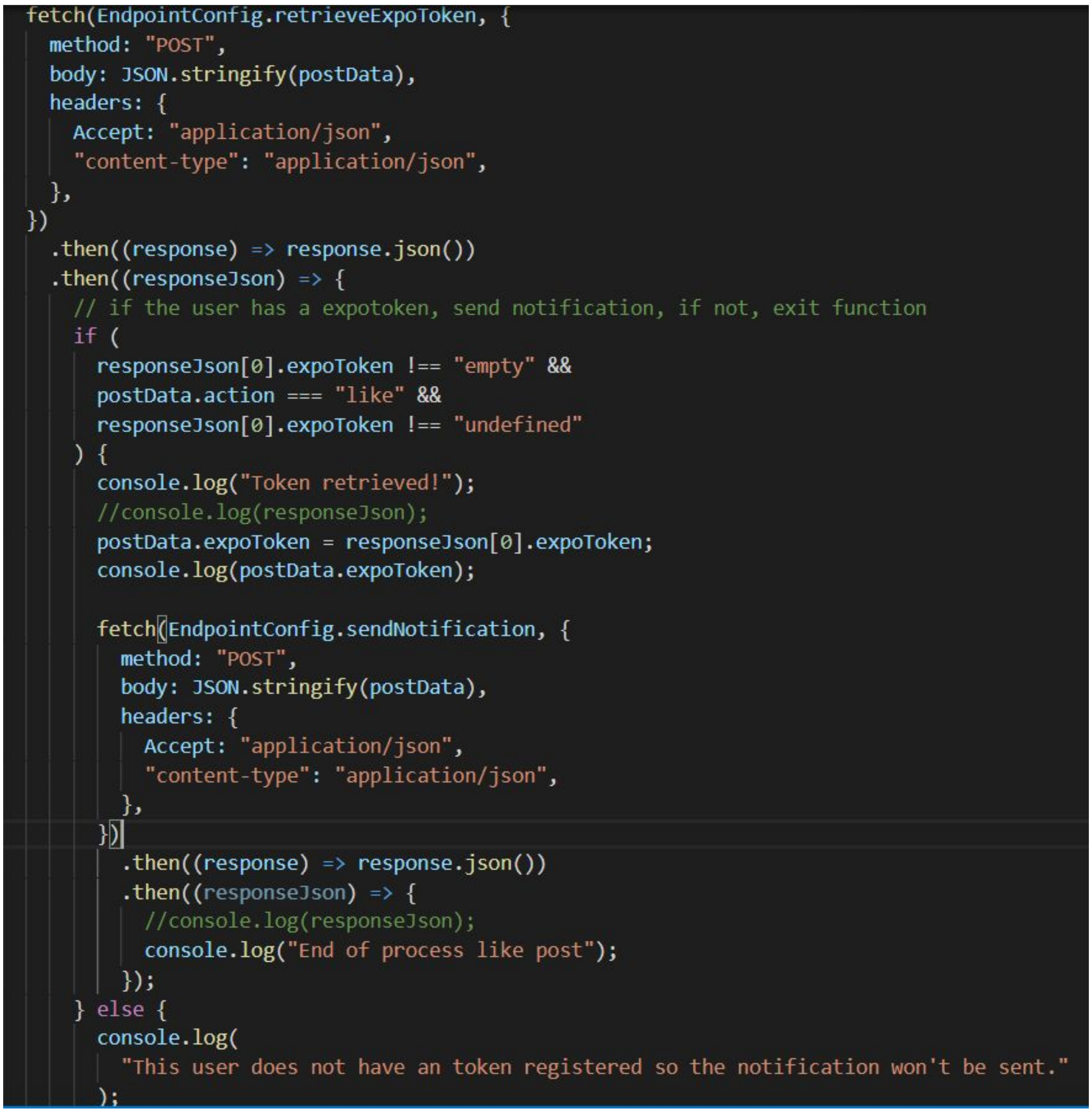
For that, we do the following:

1. Search for the expotoken of the destinatary user through the fetch retrieveExpoToken, if it exists we proceed, if not, we end the function here.
2. So now that we got the expoToken of the destinatary user, we send a request to our server to send the notification through the fetch sendNotfication. The format of the data we are sending is something like that:



```
/sendNotification hitted
{
  to: 'ExponentPushToken[dOoJxzO6-2yGpBOdoAsl_1]',
  sound: 'default',
  title: 'Someone liked your post!',
  body: 'It seems your post is getting noticed'
}
```

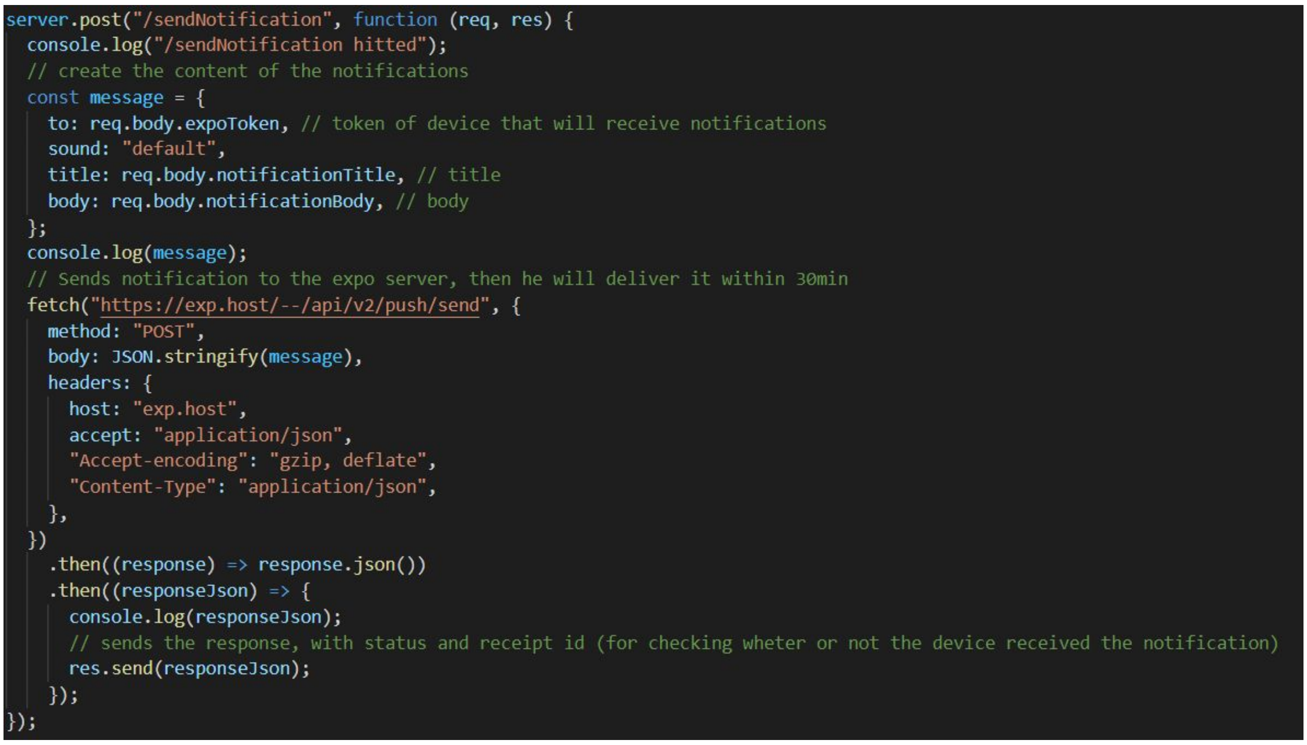
The front-end code:



```
fetch(EndpointConfig.retrieveExpoToken, {
  method: "POST",
  body: JSON.stringify(postData),
  headers: {
    Accept: "application/json",
    "content-type": "application/json",
  },
})
  .then((response) => response.json())
  .then((responseJson) => {
    // if the user has a expotoken, send notification, if not, exit function
    if (
      responseJson[0].expoToken !== "empty" &&
      postData.action === "like" &&
      responseJson[0].expoToken !== "undefined"
    ) {
      console.log("Token retrieved!");
      //console.log(responseJson);
      postData.expoToken = responseJson[0].expoToken;
      console.log(postData.expoToken);

      fetch(EndpointConfig.sendNotification, {
        method: "POST",
        body: JSON.stringify(postData),
        headers: {
          Accept: "application/json",
          "content-type": "application/json",
        },
      })
        .then((response) => response.json())
        .then((responseJson) => {
          //console.log(responseJson);
          console.log("End of process like post");
        });
    } else {
      console.log(
        "This user does not have an token registered so the notification won't be sent."
      );
```

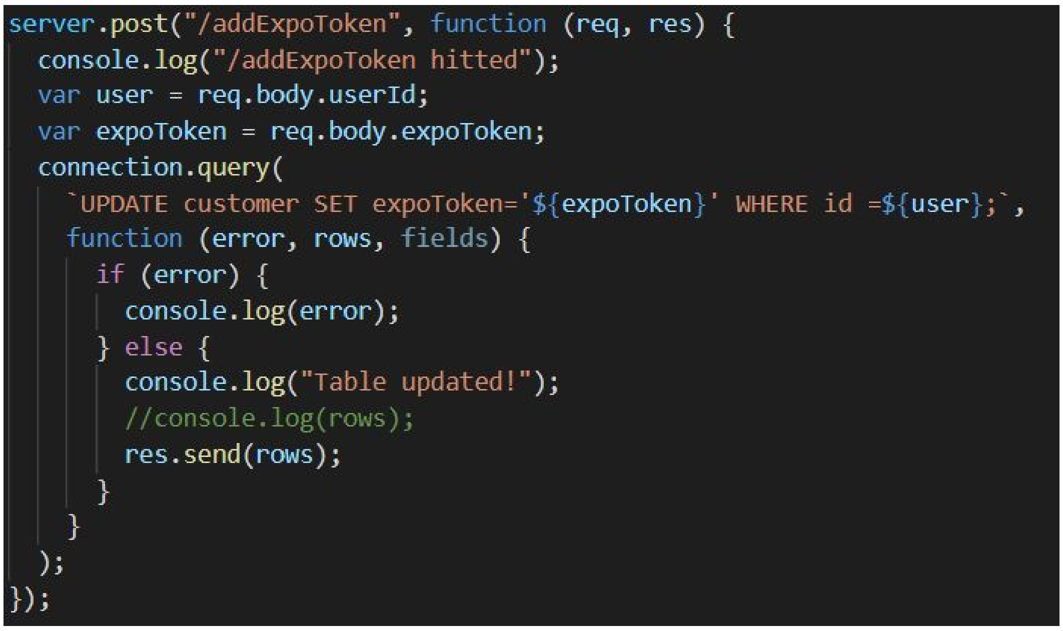
The back-end code:

Send notification function



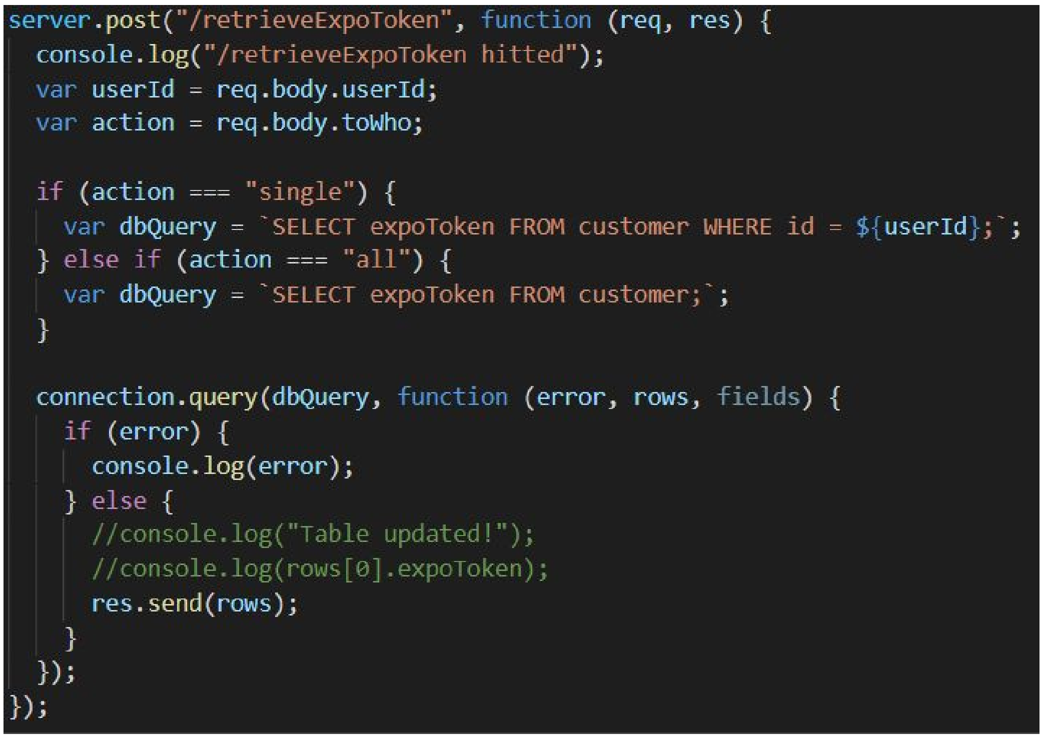
```
server.post("/sendNotification", function (req, res) {
  console.log("/sendNotification hitted");
  // create the content of the notifications
  const message = {
    to: req.body.expoToken, // token of device that will receive notifications
    sound: "default",
    title: req.body.notificationTitle, // title
    body: req.body.notificationBody, // body
  };
  console.log(message);
  // Sends notification to the expo server, then he will deliver it within 30min
  fetch("https://exp.host/--/api/v2/push/send", {
    method: "POST",
    body: JSON.stringify(message),
    headers: {
      host: "exp.host",
      accept: "application/json",
      "Accept-encoding": "gzip, deflate",
      "Content-Type": "application/json",
    },
  })
    .then((response) => response.json())
    .then((responseJson) => {
      console.log(responseJson);
      // sends the response, with status and receipt id (for checking wheter or not the device received the notification)
      res.send(responseJson);
    });
});
```

addExpoToken function



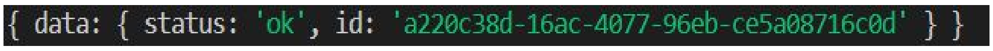
```
server.post("/addExpoToken", function (req, res) {
  console.log("/addExpoToken hitted");
  var user = req.body.userId;
  var expoToken = req.body.expoToken;
  connection.query(
    `UPDATE customer SET expoToken='${expoToken}' WHERE id =${user};`,
    function (error, rows, fields) {
      if (error) {
        console.log(error);
      } else {
        console.log("Table updated!");
        //console.log(rows);
        res.send(rows);
      }
    }
  );
});
```

retrieveExpoToken function



```
server.post("/retrieveExpoToken", function (req, res) {
  console.log("/retrieveExpoToken hitted");
  var userId = req.body.userId;
  var action = req.body.toWho;

  if (action === "single") {
    var dbQuery = `SELECT expoToken FROM customer WHERE id = ${userId};`;
  } else if (action === "all") {
    var dbQuery = `SELECT expoToken FROM customer;`;
  }

  connection.query(dbQuery, function (error, rows, fields) {
    if (error) {
      console.log(error);
    } else {
      //console.log("Table updated!");
      //console.log(rows[0].expoToken);
      res.send(rows);
    }
  });
});
```

If sending the notification was successful, then the server will respond like that:



```
{ data: { status: 'ok', id: 'a220c38d-16ac-4077-96eb-ce5a08716c0d' } }
```

The second parameter is a receipt id that you can use to check if the device has properly received the notification.

Back end caches.

One of the technological features we wanted to implement during this third sprint was adding back end caches. Back end caches allow to reduce the global latency of the system by adding back end storage.

One example of implementation we added in our application was related to the posts filters. In fact, during the previous sprint, we implemented a feature to allow the public app users to filter post thanks to a category attribute.

Thanks to this feature, the user was able to filter the posts displayed in the postScrollList screen.
The point we had to optimize was the following : each time the user clicked on a specific category, our app reached the **/filterPosts** endpoint of the server. The POST request passed the category selected within the request's body. Then, the server was requesting to retrieve all the posts with the selected category.

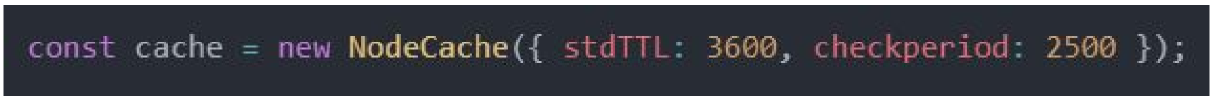
As we can see, each selection in the client side was generating a new request from the server to the DB in the back end side. This operation can fit for small application but would not be optimized in a bigger application so we wanted to implement back end caches to fix it.

The purpose of back-end caches is to avoid multiplying the request between the server and the database.

By reducing the number of requests and data exchanges between the server and the database, we are able to reduce the global latency of our application.

To implement the back-end cache, we used the node-cache npm Package.

We first generate a new instance of the NodeCache class using the class's constructor.



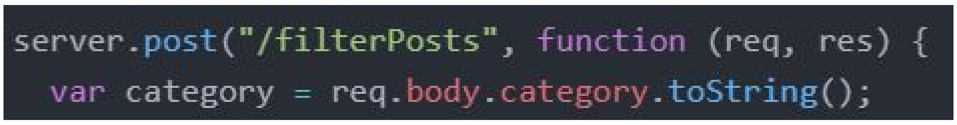
```
const cache = new NodeCache({ stdTTL: 3600, checkperiod: 2500 });
```

The first attribute stands for the time to live in seconds. It is concretely the time during which the data is going to be stored within the cache.

The node-cache package works using a key-value storage principle. The idea is to generate a key for each value that we want to store inside the cache.

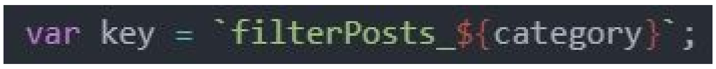
The working process is the following :

Hitting the **/filterPosts** endpoint, we start by retrieving the category selected by the user.



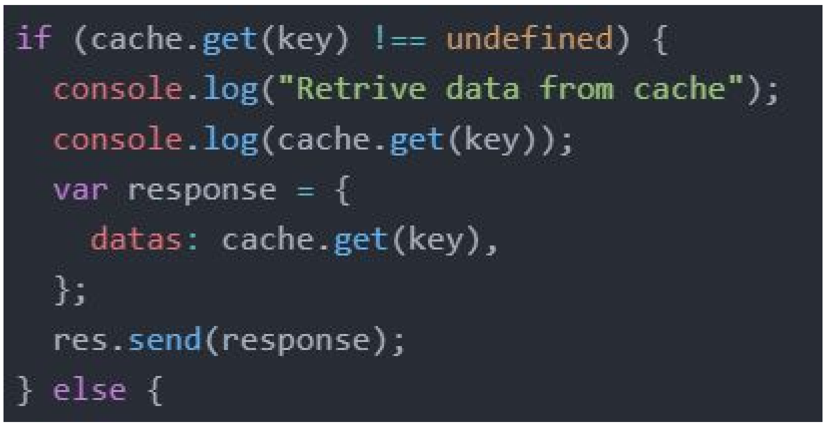
```
server.post("/filterPosts", function (req, res) {
  var category = req.body.category.toString();
```

We then generate a cache storage key using the previous category.



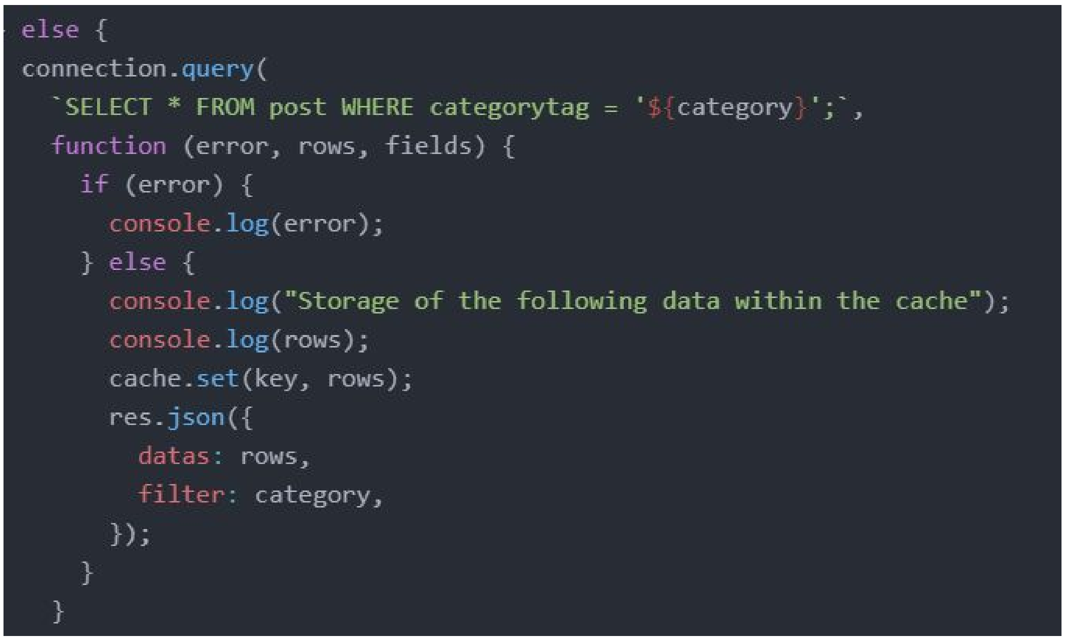
```
var key = `filterPosts_${category}`;
```

Then, we check if the cache already includes values associated with this key.
If the value associated with the key has already been stored, we directly retrieve it from the cache without querying the database.



```
if (cache.get(key) !== undefined) {
  console.log("Retrive data from cache");
  console.log(cache.get(key));
  var response = {
    datas: cache.get(key),
  };
  res.send(response);
} else {
```

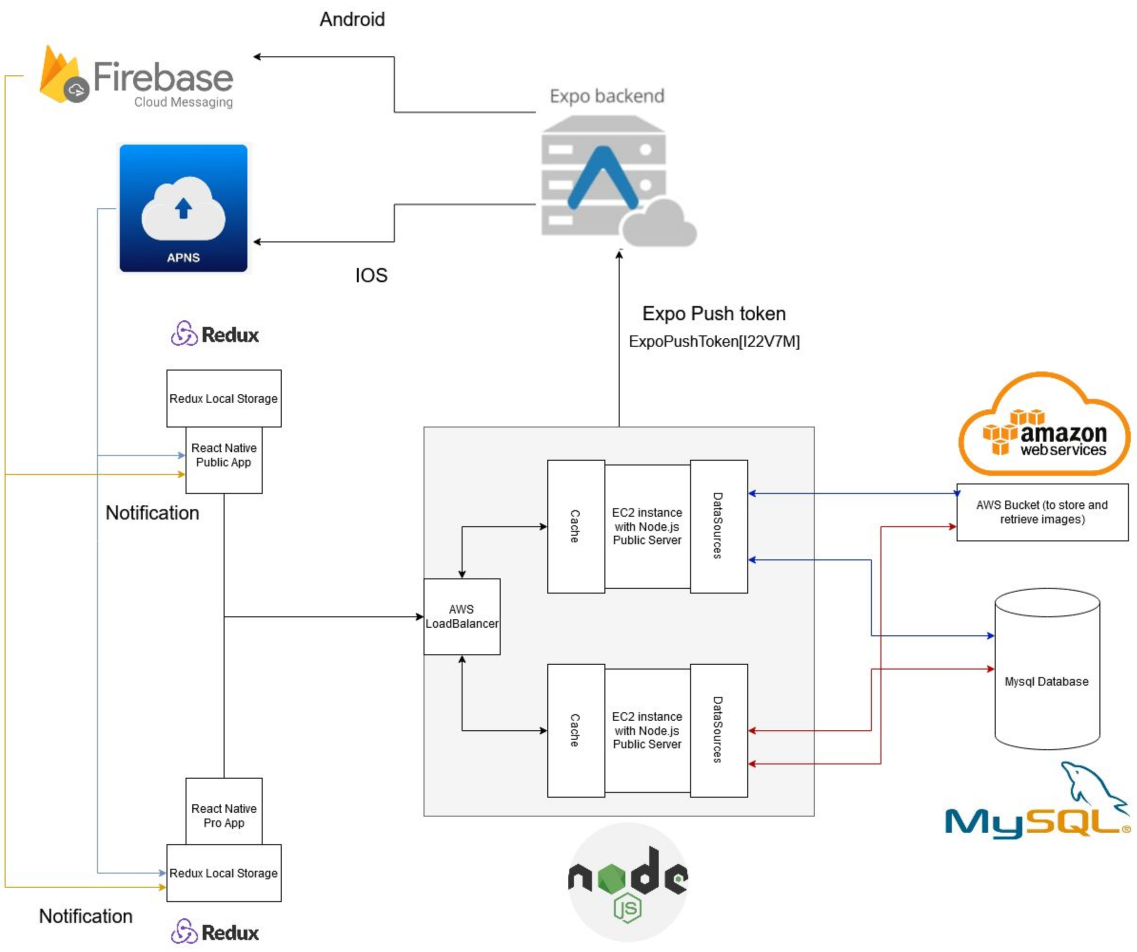
If we can't retrieve the key within the cache, then we make a request to the database.



```
else {
connection.query(
  `SELECT * FROM post WHERE categorytag = '${category}';`,
  function (error, rows, fields) {
    if (error) {
      console.log(error);
    } else {
      console.log("Storage of the following data within the cache");
      console.log(rows);
      cache.set(key, rows);
      res.json({
        datas: rows,
        filter: category,
      });
    }
  }
```

We then store the results from the database within the backend cache using the **.set()** method and linking it with the key.

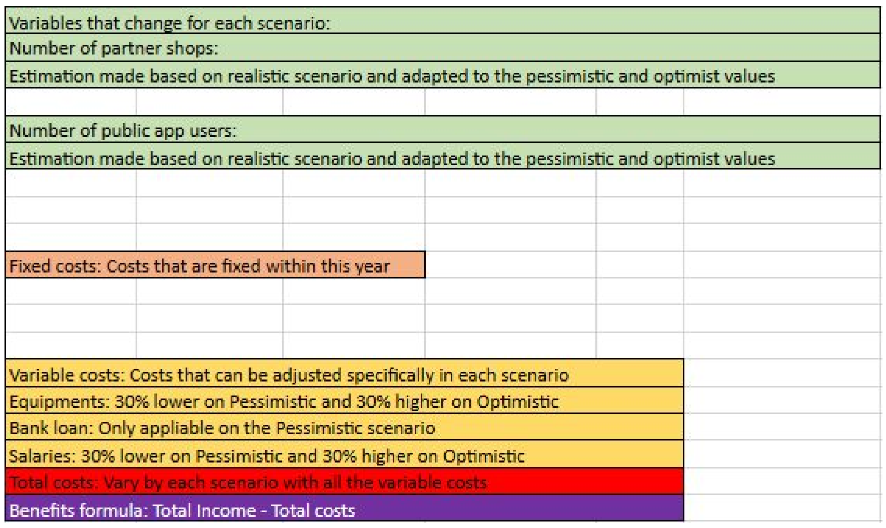
This is the current architecture of our App according to the sprint 3 modifications



2. Economic section

Major upgrades:

1. Added explanation about each variable in the what-if scenarios



Variables that change for each scenario:

Number of partner shops:

Estimation made based on realistic scenario and adapted to the pessimistic and optimist values

Number of public app users:

Estimation made based on realistic scenario and adapted to the pessimistic and optimist values

Fixed costs: Costs that are fixed within this year

Variable costs: Costs that can be adjusted specifically in each scenario

Equipments: 30% lower on Pessimistic and 30% higher on Optimistic

Bank loan: Only appliable on the Pessimistic scenario

Salaries: 30% lower on Pessimistic and 30% higher on Optimistic

Total costs: Vary by each scenario with all the variable costs

Benefits formula: Total Income - Total costs

2. Following the feedback that we decided to re-invest our cashflow in year 4 at the local small shops itself, through an helping entity funded by our company (ProxyPrizes), further details will be provided
3. Added crowdfunding page

Crowdfunding dsadas SIGN IN SIGN UP NOW

Get closer to your local shops

ProxyPrizes will be your go-to application when the matter is local shopping. Discover, post and share! You and other users will be part of a local community to

FUNDING

ProxyPrizes
1 campaign | Lleida

€60.000 EUR 4.320 backers

34% of 60.000 Fixed Goal 31 days left

BACK IT FOLLOW

How it works

- Discover new products of local shops
- Share you opinions about products
- Earn exclusive discounts
- Participate in sales and events
- Encourage your local shops

Explore your surroundings

All participating shops in your city will be acessible easily within the app

Scroll through recommendations and products

In your main timeline you will see the content best related to you and your local community, you can even filter through the items!

android iOS

GET IT ON Google Play Download on the App Store

Avaliable everywhere!

Android or iPhone? Don't worry, our application will work in both systems!

Contribute with:

- 10 euros: Our most humble thank you!
- 20 euros: Special thanks and + 2 exclusive discounts
- 30 euros: 5 exclusive discounts + access to early limited sales
- 50+ euros: All previous rewards + a Special surprise