Instituto Politécnico do Cávado e do Ave

Escola Superior de Tecnologia

Arquitetura de Sistemas

Mestrado em Engenharia Informática

2º Trabalho Prático

Mobilidade Urbana

Barcelos,15 de Dezembro de 2019

Jéssica Macedo a6835

Fernando Correia a11199

**Atualizações ao documento:**

|  |  |
| --- | --- |
| Alterações | Data |
| Criação do documento | 15/12/2019 |
| Alterações da 2ª entrega | 05/01/2020 |
| Alterações da 3ª entrega | 01/01/2020 |

**Introdução**

O trabalho abordado no presente relatório foi desenvolvido no âmbito da unidade curricular Arquiteturas de Sistemas do mestrado em Engenharia de Sistemas Informáticos em Desenvolvimento de Aplicações. Tem como fundamental objetivo o desenvolvimento de um sistema distribuído que permite alugar veículos de mobilidade urbana, tendo por base uma API Restful que garante a integração entre a aplicação servidor e as várias aplicações cliente (um cliente agente, um cliente gestor, e um cliente dashboard).

**Descrição detalhada do problema a resolver**

**Servidor:**

O objetivo do sistema é agilizar o aluguer de veículos disponíveis, fornecendo:

· filtros para a localização dos veículos livres;

· gestão dos dados de cliente;

· registo do pagamento através de um saldo recarregável;

O servidor deverá contemplar a utilização de bases de dados onde toda a informação relacionada com o serviço disponibilizado será guardada.

Desenvolver um conjunto de serviços para garantir o acesso à informação da base de dados, de forma a responder aos pedidos dos diferentes clientes;

Disponibilizar uma documentação (Open API) e descrição acerca dos testes realizados à utilização dos serviços.

Publicação num ambiente cloud dos diferentes serviços desenvolvidos;

Seguir uma arquitetura baseada em micro serviços – containers

· a utilização de uma Gateway para facilitar a integração dos vários micro serviços;

· a disponibilização de um sistema integrado de logging global a todos os micro serviços.

**Utilizadores**

Neste trabalho estão presentes quatro tipos de utilizadores, dos quais são:

* **Utilizador não registado** - Trata-se de um utilizador sem qualquer

registo na plataforma;

* **Cliente** - Trata-se de um utilizador previamente registado, sendo considerado

um cliente (do serviço de aluguer de veículos). Tem as mesmas

funcionalidades que um utilizador não registado e mais algumas para

além deste.

* **Funcionário** - Trata-se de um funcionário da entidade responsável

pela gestão dos veículos, que tem a responsabilidade de fiscalizar os

estacionamentos.

* **Administrador** - Trata-se da entidade fiscalizadora da aplicação. Consulta

métricas, valida registos e configura os dados.

**Funcionalidades dos Utilizadores**

1. **Utilizador não registado**

a. Permite obter informação dos lugares de estacionamento (latitude e longitude), capacidade, quantidade de veículos;

b. Permite registar-se e consequentemente logar-se na aplicação

2. **Cliente**

a. Utilizador previamente registado

b. Permite obter informação dos lugares de estacionamento (latitude e longitude), capacidade, quantidade de veículos;

c. Permite pesquisar veículos detalhando o nome da rua ou raio de pesquisa

d. Consulta do saldo atual da conta

e. Fazer check-in do veículo (código veículo, método de aluguer [preço por minuto/pacotes de horas], hora inicio, preço estimado, código de aluguer)

f. Fazer check-out do veículo (hora fim, verifica posição estacionamento, cálculo aluguer )

g. Fazer consulta dos dados relativos ao aluguer ativo (tempo e custo até ao momento)

3. **Funcionário**

a. Registo de estacionamentos de veículos em locais impróprios

b. Notificar cliente de estacionamento impróprio

4. **Administrador**

a. Consultar dashboard com resumo dos dados e histórico de ocupação de lugares

b. Permitir a validação do pedido de registo de utilizadores

c. Configuração da localização dos lugares de estacionamento

d. Nice to have: envio de indicação aos clientes da aproximação do fim do saldo

**Plano para o desenvolvimento da solução (objetivos para próximas entregas)**

**Até 15 Dezembro:**

1. Geração dos modelos de dados

**Até dia 5 Janeiro**

1. Geração dos serviços CRUD para os diferentes serviços
2. Criação da documentação Swagger
3. Início da criação de algumas funcionalidades da aplicação frontend em React<https://reactjs.org/>

**Até dia 17 Janeiro**

1. Continuação da criação da aplicação frontend
2. Instalação do sistema em serviço cloud com o Heroku<https://www.heroku.com/>
3. Instalação dos micro-serviços em Docker

**Modelo de dados**

**Vehicle**

var vehicleSchema = new Schema({

code: {

type: Number,

required: [true,'code of the vehicle’']

},

description: {

type: String

},

Place: {

type: mongoose.Schema.Types.ObjectId,

ref: ‘Place’

}

});

**Client**

var clientSchema = new Schema({

firstName: {

type: String,

required: 'first name of the person '

},

lastName: {

type: String,

required: 'last name of the person '

},

rentals: [{

type: mongoose.Schema.Types.ObjectId,

ref: ‘Rental’

}],

balance: {

type: Number

},

Created\_data: {

type: Date,

default: Date.now

}

});

**User**

var userSchema = new Schema({

username: {

type: String,

unique: true,

required: true

},

email: {

type: String,

unique: true,

index: true,

required: true

},

password: {

type: String,

required: true,

select: false

},

role:{

type: String,

required: true,

default: ‘client’ ,

enum: [“guest”, “client”,”employee”,”admin”]

},

registeredBy: {

type: mongoose.Schema.Types.ObjectId,

ref: 'User'

},

valid : {

type: Boolean

}

});

**Rental**

var rentalSchema = new Schema({

startDate: {

type: Date,

// default: Date.now

},

endDate: {

type: Date,

// default: Date.now

},

price: {

type: Number,

required: true

},

rentalMethod:{

type: String,

enum: ['minutes', 'pack'],

default: ['minutes']

},

code: {

type: Number,

required: true

},

vehicle: {

type: mongoose.Schema.Types.ObjectId,

ref: ‘Vehicle’

}

});

**Place**

var placeSchema = new Schema({

location: [{

type: String,

coordinates: [Number],

required: true

},

{

range: Number

}

}],

capacity: {

type: Number

},

quantity: {

type: Number,

}

});

Repositório GitHub: <https://github.com/Knox316/MobilityProject>

**2ª entrega:  05/janeiro**

* Ponto de situação

Para esta fase tínhamos definido:

1. Geração dos serviços CRUD para os diferentes serviços
2. Criação da documentação Swagger
3. Início da criação de algumas funcionalidades da aplicação frontend em React<https://reactjs.org/>

Geramos os serviços CRUD que nos foram propostos no enunciado do trabalho e que podem ser consultados na coleção POSTMAN do projeto *MobilityProject.postman\_collection.json.*

Criamos uma base de dados não-relacional com o Mongodb, que pode ser consultada com o link: *mongodb+srv://admin:admin@cluster0-krbnl.mongodb.net/MobilityProject?retryWrites=true&w=majority.*

Criamos a documentação Swagger que pode ser consultada em http://{host}:3000/api-docs.

Fizemos algumas alterações aos modelos de dados previamente criados, pois sentimos necessidade de alterar alguns pontos à medida que íamos construindo os serviços.

No schema Rental adicionamos os campos “finalCost”,”previewCost”, “timeSpent” e alteramos a estrutura para “start” e “end”, onde no primeiro indicamos a localização e data do checkin e no segundo os dados de checkout.

No schema “user” adicionamos os campos “waitValidation” (que vai indicar se ainda necessita de validação do administrador), “firstname” e “lastname”.

No entanto nesta fase não criamos ainda funcionalidades da aplicação front-end em React.

* Plano para próximas entregas

Para as próximas entregas, iremos focar-nos na aplicação frontend, de forma a utilizar os serviços criados e construir o interface. Se necessário iremos ajustar os serviços já criados.

Vamos também dividir os serviços criados em diferentes micro serviços, para depois podermos criar diferentes containers em Docker e publicar no Heroku.

* Código desenvolvido até ao momento

O código pode ser visto no Repositório GitHub: <https://github.com/Knox316/MobilityProject>.

**3ª entrega:  04/Fevereiro**

Para a última entrega tínhamos definido:

1. Continuação da criação da aplicação frontend
2. Instalação do sistema em serviço cloud com o Heroku<https://www.heroku.com/>
3. Instalação dos micro-serviços em Docker

***Serviços Backend***

Ao longo do desenvolvimento fomos detetando algumas necessidades e por isso fomos alterando alguma lógica nos modelos de dados. Retiramos o modelo “Client” que achamos que não fazia sentido, usamos apenas o modelo “User” que pode conter todos os tipos de utilizadores da aplicação (administradores, clientes, funcionários e utilizadores ainda não validados). Isto é possível porque sendo uma base de dados não-relacional, os registos são feitos em documentos que podem ser diferentes uns dos outros, permitindo guardar informação diferente para cada tipo de utilizador.

Seguem abaixo quadros que ilustram mais ao pormenor os modelos de dados.

Tabela 1 – Modelo de dados “Places”

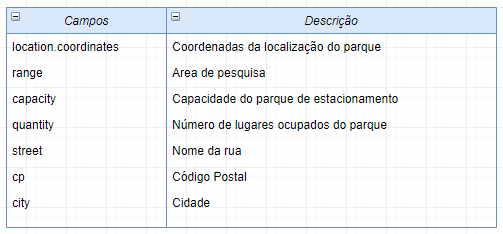


Tabela 2- Modelo de dados “Rental”

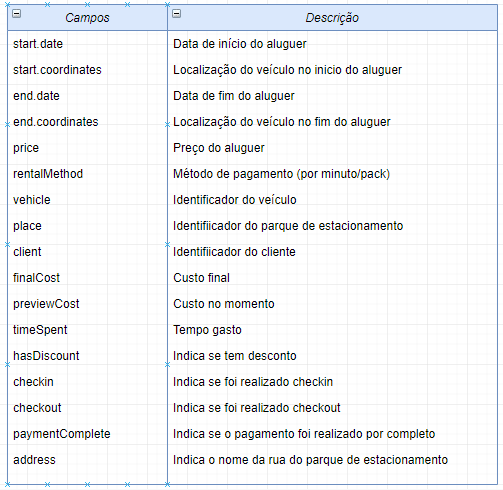


Tabela 3- Modelo de dados “User”

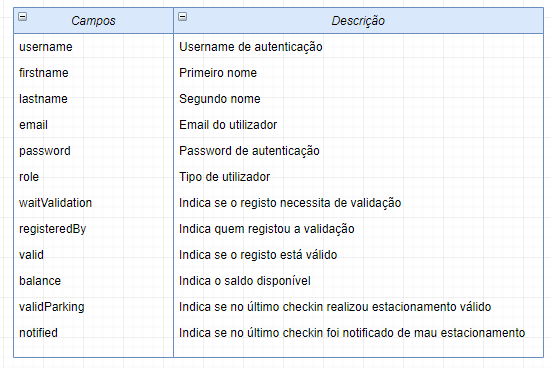
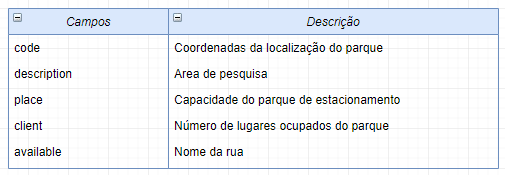


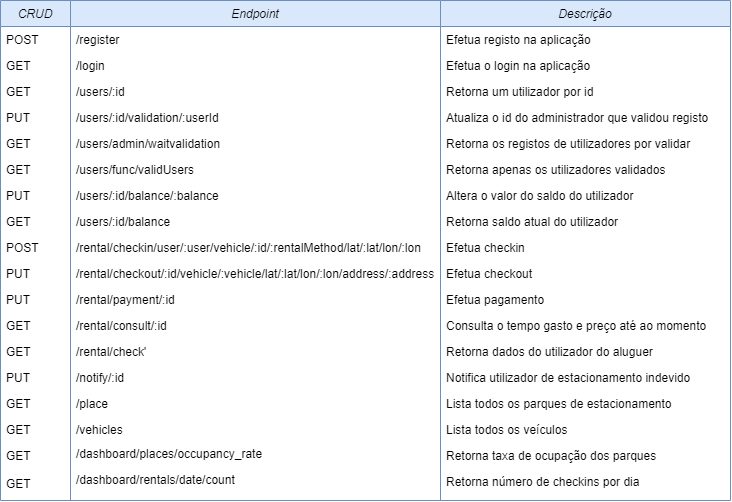
Tabela 4 – Modelo de dados “Vehicle”



Para os serviços foram criadas as seguintes rotas *RESTFUL API*:

**BaseURL: /api/v1**

Tabela 5 – Tabela dos serviços RESTFUL API



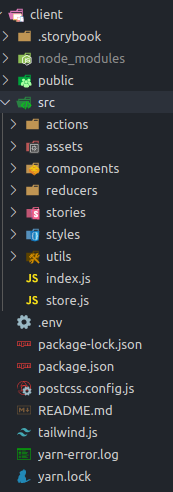
Frontend

Ao nivel de frontend utilizamos uma libraria chamada ReactJS para auxiliar na construção de componentes utilizando Redux para a gestão do estado da aplicação. A nivel de CSS utilizamos as frameworks Material-UI e Materializecss.

**React**

React é uma libraria que permite criar aplicações e componentes de forma modular para integrar em qualquer backend. React usa o conceito de “Virtual DOM” que é uma abstração do DOM em que apenas é renderizada as componentes na qual se altera informação em vez de fazer um full reload na página.

A estrutura base do nosso projeto em React é a seguinte:

[[1]](#footnote-15118)

**Redux**

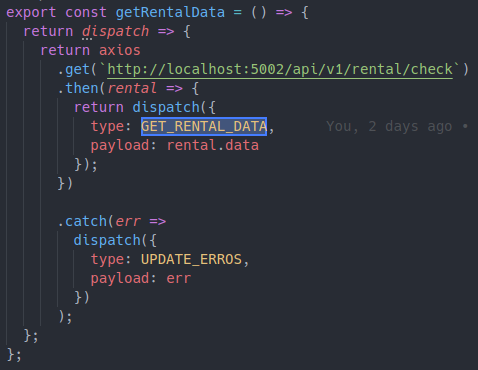
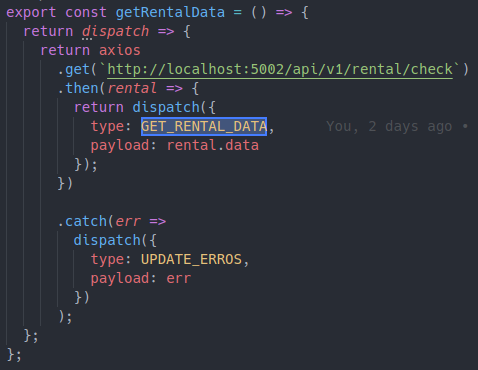
A modularidade de React permite que o utilizador tenha muita liberdade na forma como mostra os dados. Para evitar conflitos de dados entre endpoints usamos uma libraria chamada Redux.

Redux permite ter o estado da aplicação num unico ponto, normalmente uma “store” que armazena os objetos que vão armazenar os resultados das chamadas ao serviço.

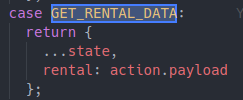
Redux usa “actions” para enviar pedidos para alteração do estado. É a unica forma de fazer alterações ao estado da aplicação, é emitindo uma Action a um reducer que vai receber o objecto da action e direcionar o objecto para a Action correta.

Exemplo:

**Action**

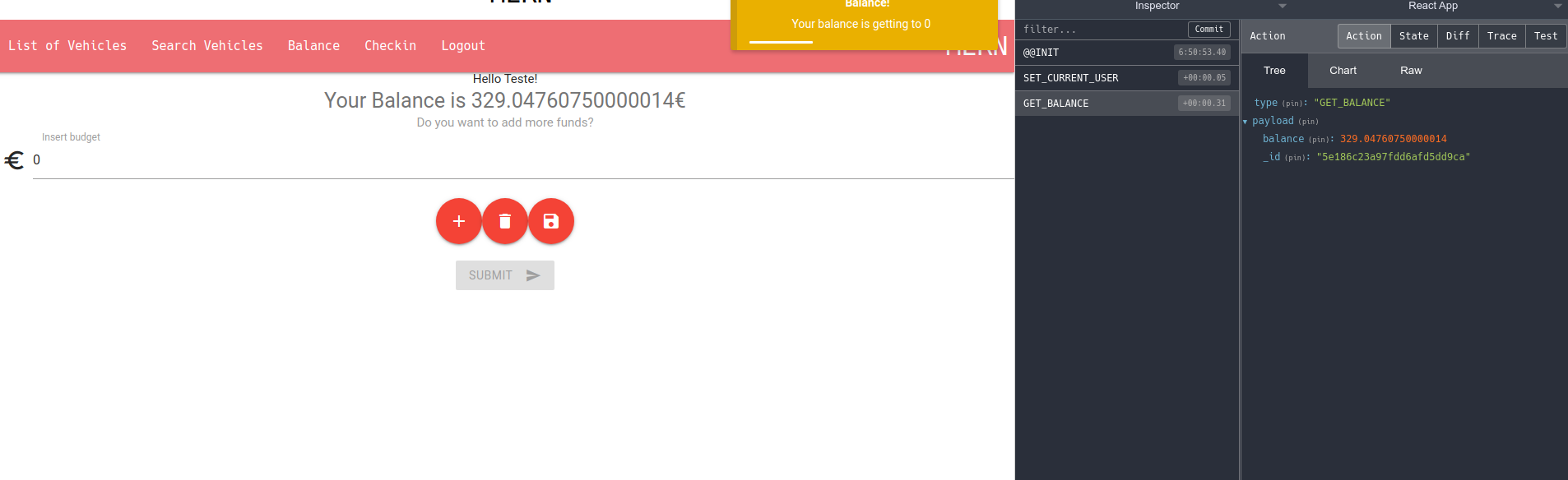


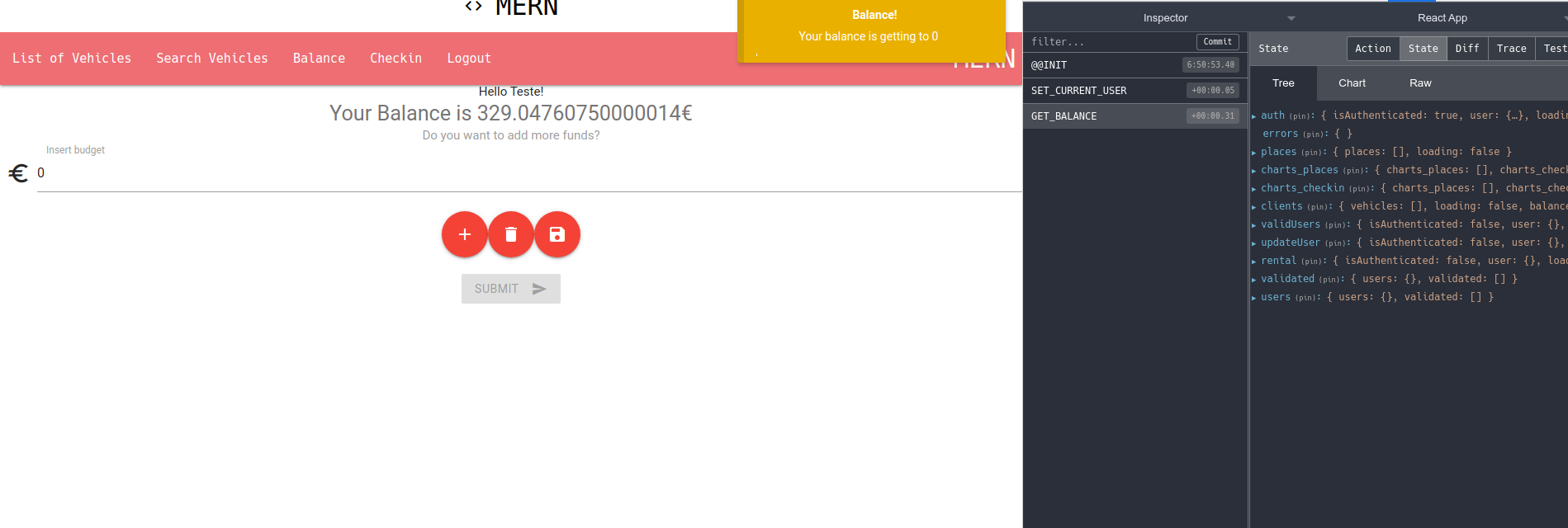
O Objecto “GET\_RENTAL\_DATA” vai guardar o valor de uma Action que é enviada para o reducer:



No reducer é recebido o objecto que vai devolver o estado anterior à mutação da action, e a action mutada pela Action e fazer render apenas do que foi alterado.

Isto permite a que o estado da nossa aplicação esteja concentrado apenas num ponto e seja mais fácil para controlar erros da aplicação e problemas com dados.





Outra das vantagens de usar Redux é a possibilidade de usar as ferramentas no Browser que permite verificar o estado da aplicação em determinado componente e ao longo do ciclo de vida do componente, bem como as actions que são passadas para mutar o estado da aplicação entre outras coisas.

A configuração da store para utilizar estas dev tools na nossa aplicação é a seguinte:

import { createStore, applyMiddleware, compose } from 'redux';

import thunk from 'redux-thunk';

import rootReducer from './reducers';

const initialState = {};

const middleware = [thunk];

const store = createStore(

rootReducer,

initialState,

compose(

applyMiddleware(...middleware),

window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

)

);

export default store;

Para poder utilizar React com Redux e todas as librarias externas que utilizamos neste projeto foram utilizados os seguintes Packages:

"dependencies": {

"@material-ui/core": "^4.8.3",

"@material-ui/icons": "^4.5.1",

"@opuscapita/react-async-select": "^2.6.0",

"@react-google-maps/api": "^1.8.2",

"@reduxjs/toolkit": "^1.2.3",

"@testing-library/jest-dom": "^4.2.4",

"@testing-library/react": "^9.3.2",

"@testing-library/user-event": "^7.2.1",

"axios": "^0.19.1",

"chart.js": "^2.9.3",

"classnames": "^2.2.6",

"google-maps-react": "^2.0.2",

"install": "^0.13.0",

"is-empty": "^1.2.0",

"jwt-decode": "^2.2.0",

"materialize-css": "^1.0.0",

"npm": "^6.13.6",

"pusher-js": "^5.0.3",

"react": "^16.12.0",

"react-canvas-js": "^1.0.1",

"react-chartjs-2": "^2.8.0",

"react-dom": "^16.12.0",

"react-dropdown": "^1.6.4",

"react-geocode": "^0.2.1",

"react-google-autocomplete": "^1.1.2",

"react-google-maps": "^9.4.5",

"react-materialize": "^3.5.9",

"react-notifications-component": "^2.2.7",

"react-places-autocomplete": "^7.2.1",

"react-pusher": "^0.2.0",

"react-redux": "^7.1.3",

"react-router-dom": "^5.1.2",

"react-scripts": "3.3.0",

"redux": "^4.0.5",

"redux-thunk": "^2.3.0"

},

**Particularidades de React**

Há mais 3 coisas importantes a referir sobre React:

***Props*** – são objectos descritivos do DOM.

***State*** – são objectos mutaveis que vão alterar o comportamento de um componente.

**Ciclo de vida** – são métodos executam qualquer coisa de acordo com a sua funcionalidade.

Os 3 mais usados nesta aplicação foram:

***ComponentDidMount*** - método executado imediatamente depois do componente ser inserido no node do DOM.

***ComponentWillMount*** – método executado imediatamente antes do componente ser inserido no node do DOM.

***ComponentWillUnmount*** – método executado quando o componente é removido do DOM.

Fonte: <https://reactjs.org/docs/react-component.html>

**Rotas**

<Provider *store*={store}>

<Router>

<div *className*='App'>

<Navbar />

<Route *exact* *path*='/logout' *component*={Logout} />

<Route *exact* *path*='/' *component*={Landing} />

<Route *exact* *path*='/register' *component*={Register} />

<Route *exact* *path*='/login' *component*={Login} />

<Route *exact* *path*='/places' *component*={Places} />

<Switch>

<PrivateRoute *exact* *path*='/charts' *component*={Charts} />

<PrivateRoute *exact* *path*='/dashboard' *component*={Dashboard} />

<PrivateRoute *exact* *path*='/main' *component*={Clients} />

<PrivateRoute *exact* *path*='/profile' *component*={Profile} />

<PrivateRoute *exact* *path*='/balance' *component*={Balance} />

<PrivateRoute *exact* *path*='/checkin' *component*={CheckIn} />

<PrivateRoute *exact* *path*='/checkout' *component*={Checkout} />

<PrivateRoute

*exact*

*path*='/searchVehicles'

*component*={SearchVehicles}

/>

<PrivateRoute

*exact*

*path*='/validateusers'

*component*={ValidateUsers}

/>

<PrivateRoute

*exact*

*path*='/checkParkings'

*component*={CheckParkingData}

/>

<PrivateRoute *exact* *path*='/marParkings' *component*={MapParkings} />

<PrivateRoute *exact* *path*='/notifyUsers' *component*={NotifyUsers} />

</Switch>

</div>

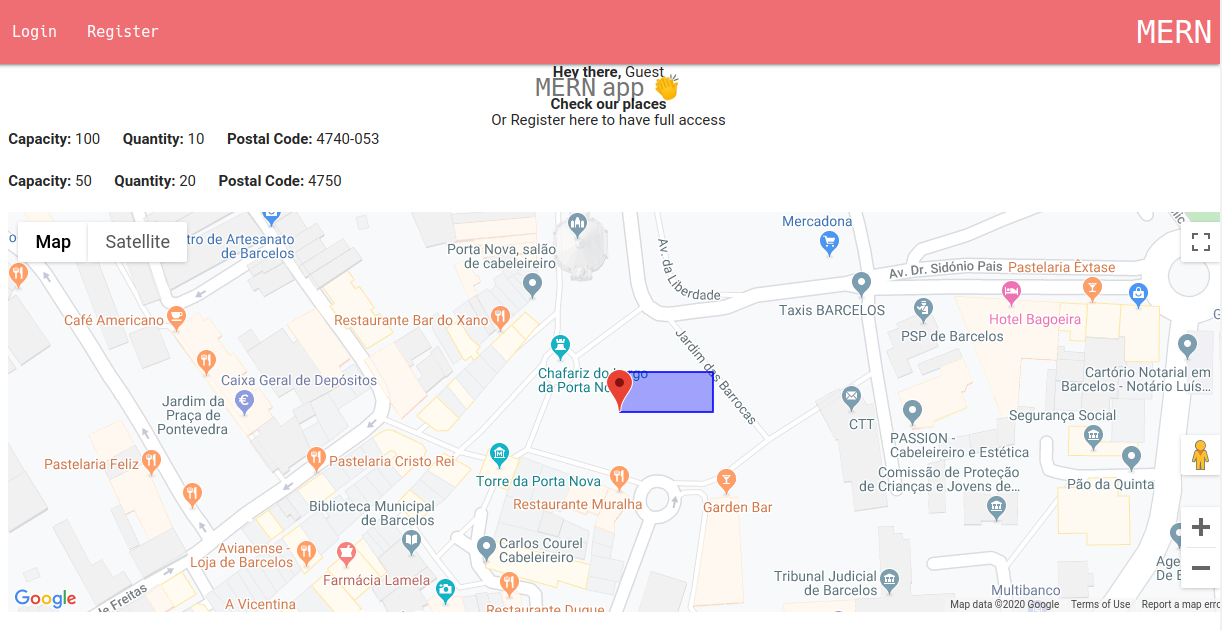
</Router>

</Provider>

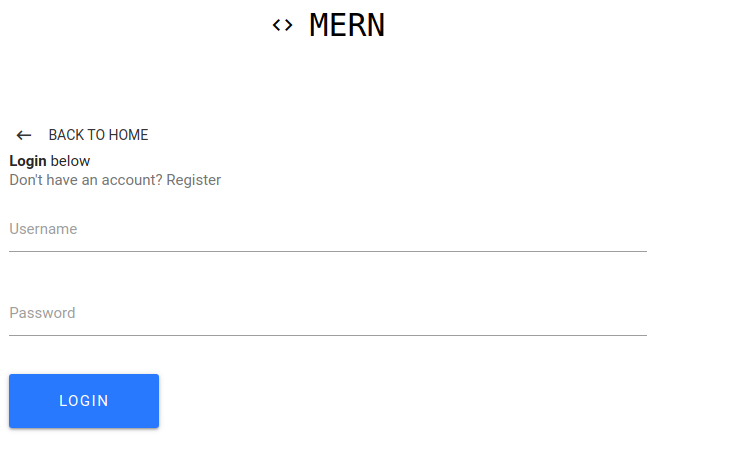
A nossa estrutura de navegação em App.js é apresentada da seguinte forma como se pode ver em cima.  
 A tag “Provider” vem da API de Redux e serve para fazer a ligação à aplicação com a Store.

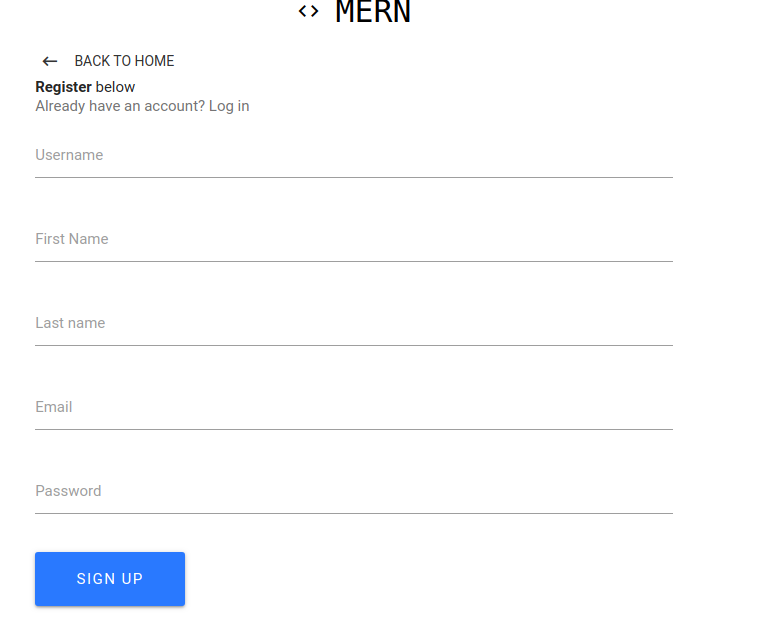
**Funcionalidades da aplicação**

**Utilizador não registado**



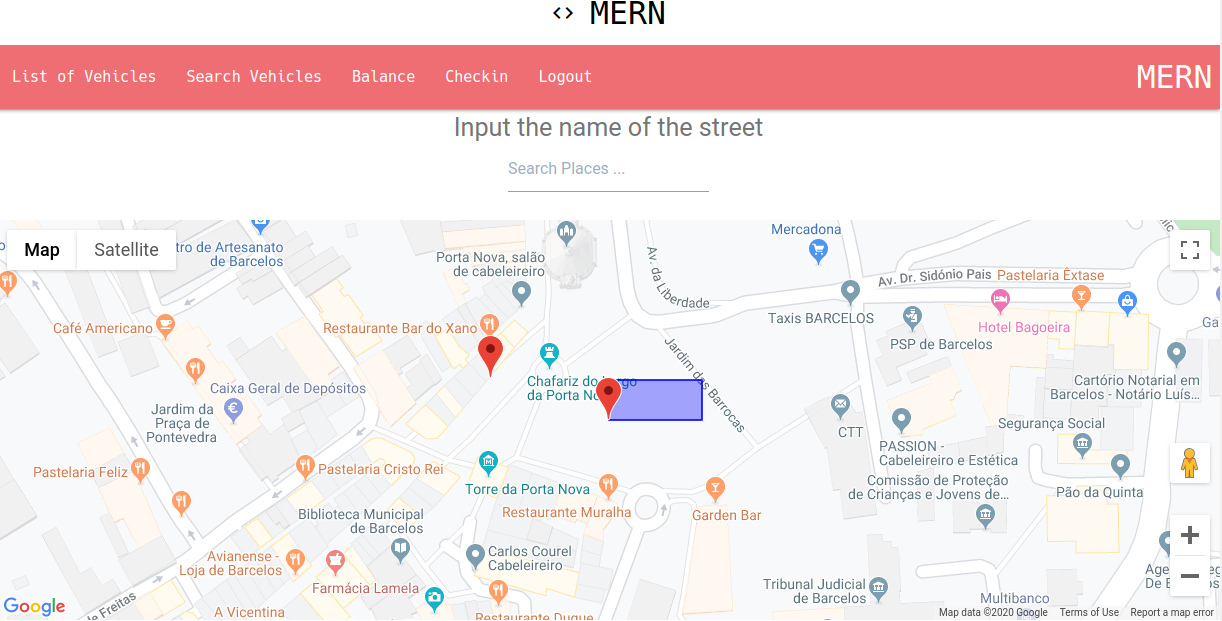
O utilizador não registado tem acesso a um ecra com os dados dos estacionamentos bem como o desenho de uma area de estacionamento. Tem também a opção para fazer o Login e o Registo.





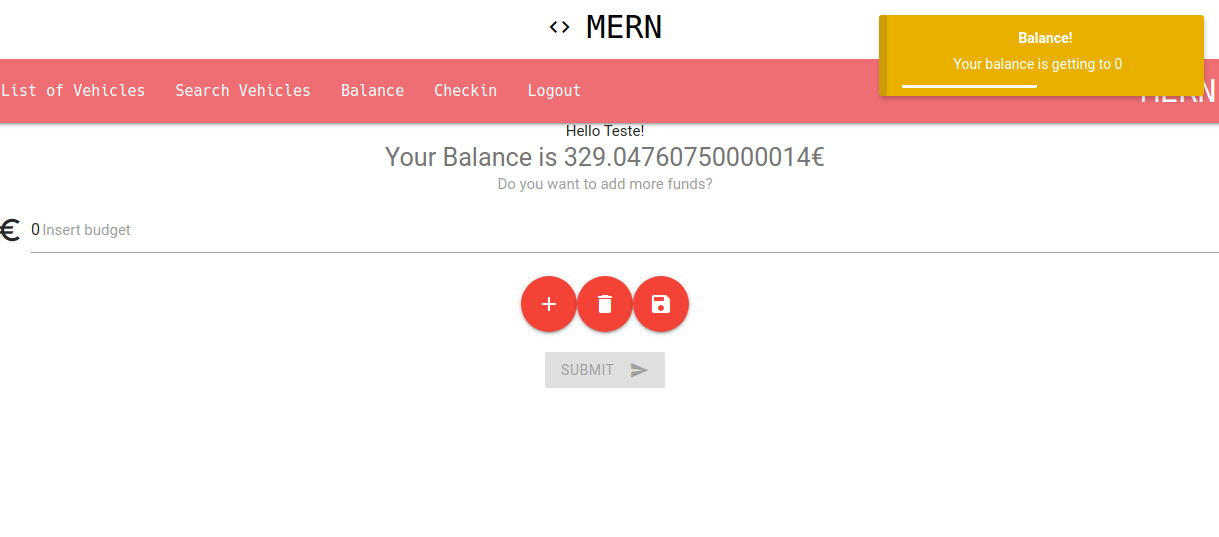
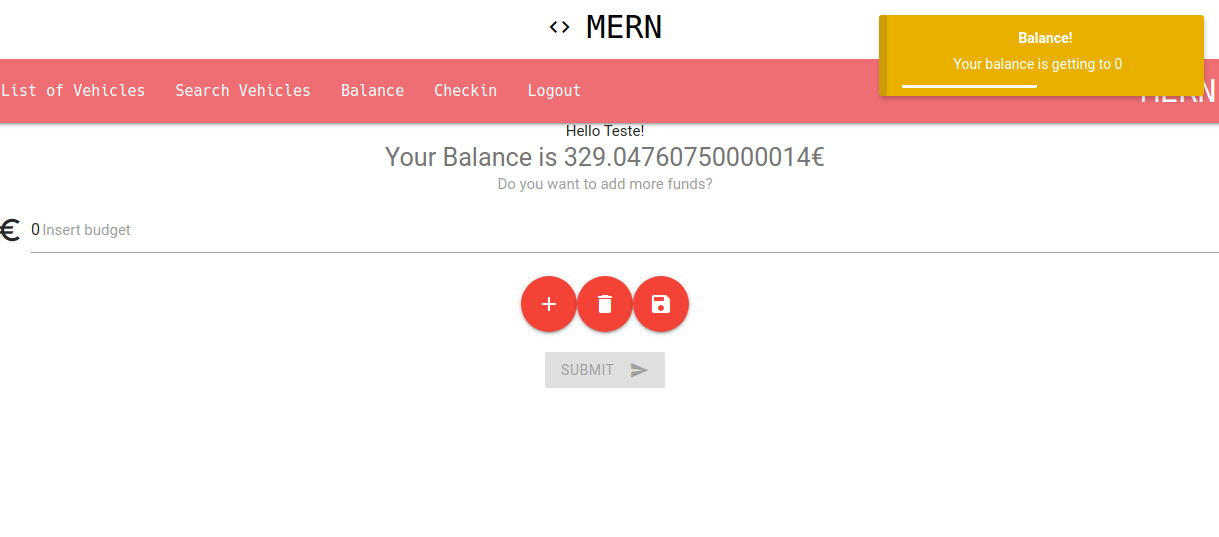
**Cliente**

Pesquisa de Veiculos

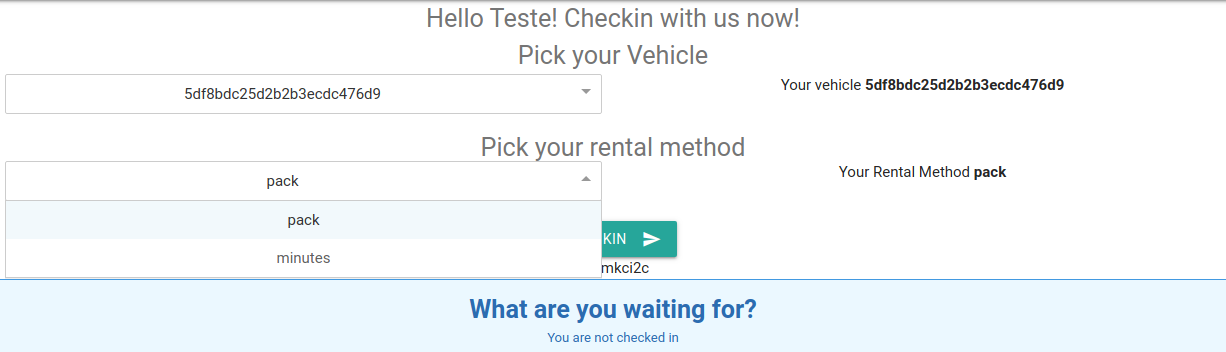


Neste ecra o utilizador pode pesquisar pela Ruas bem como ver os pontos de estacionamento.

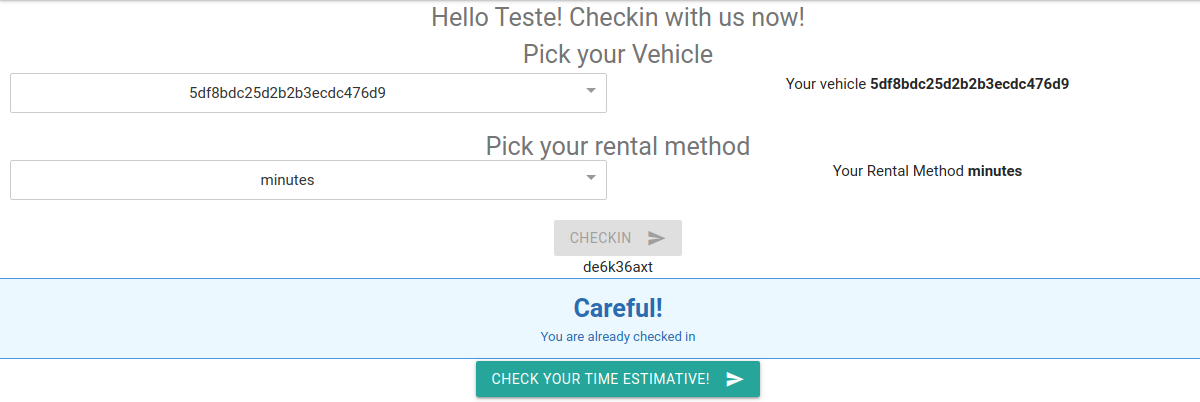
Saldo:



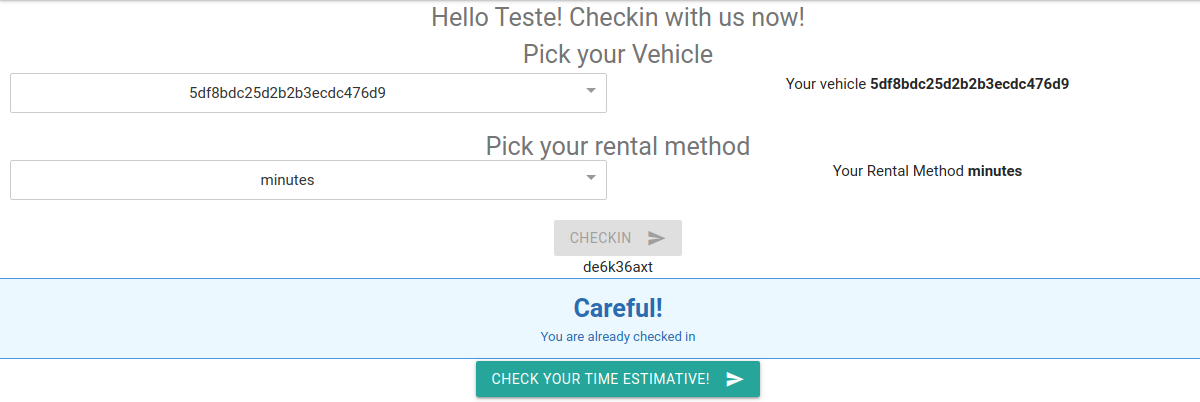
Neste ecra o utilizador pode consultar o seu saldo bem como recarregar os seus fundos. De notar que há uma notificação que apesar de ter a mensagem de aproximação a 0, para motivos de teste neste utilizador metemos a condição de se o utilizador tiver menos de 400€ de Saldo para mostrar a notificação.

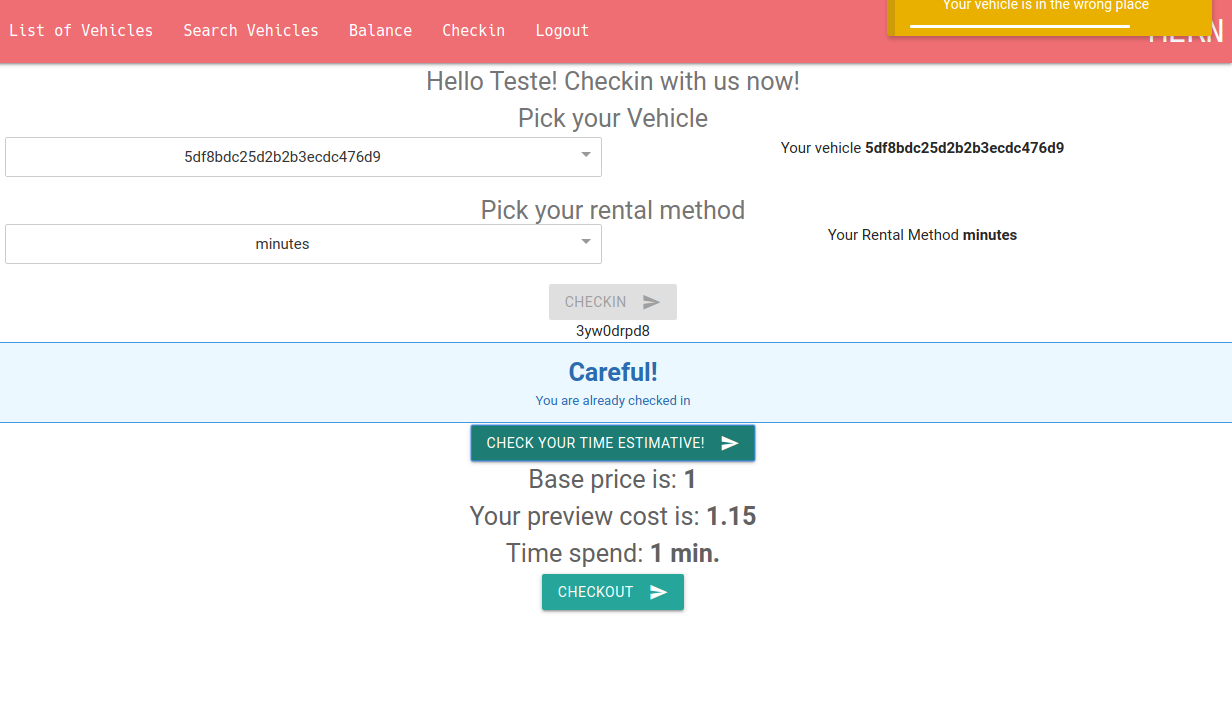
Iniciar:  


No menu de checkin o utilizador tem a possibilidade de escolher o Veiculo e o pack inicial.



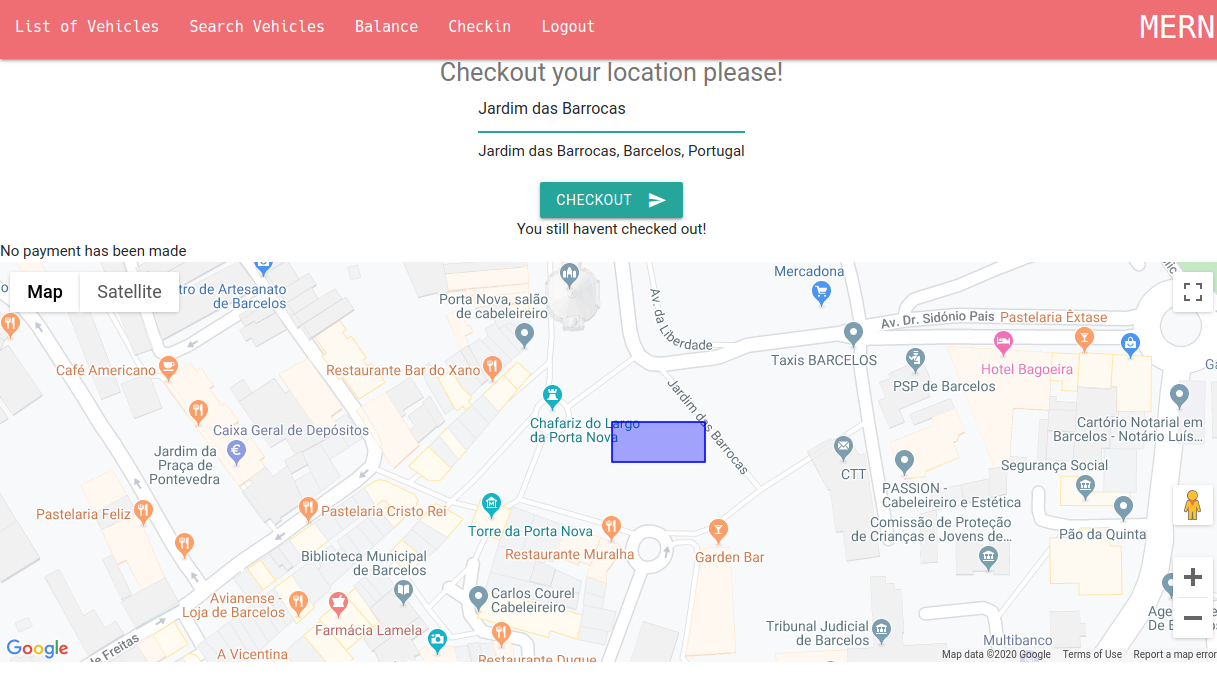
Após fazer o checkin pode fazer a estimativa baseada no tempo e no pack que escolheu.



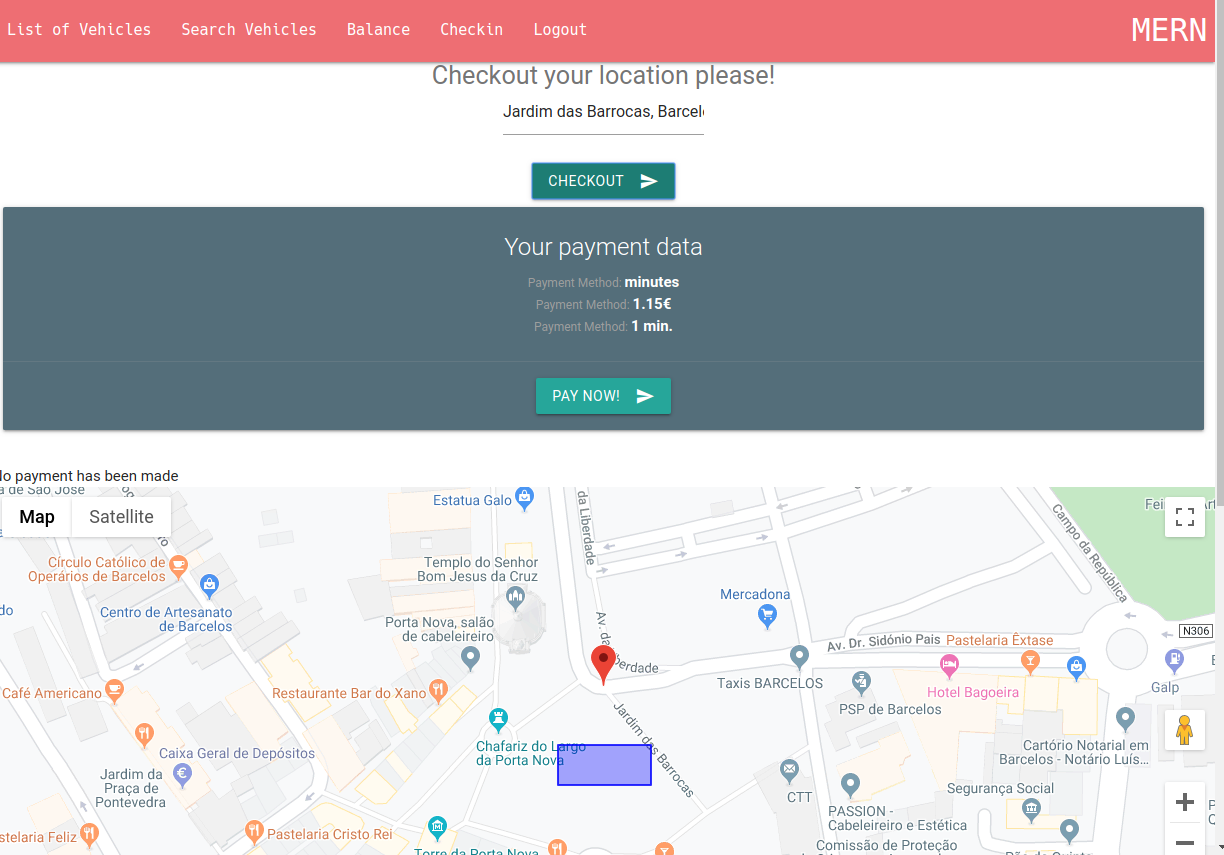
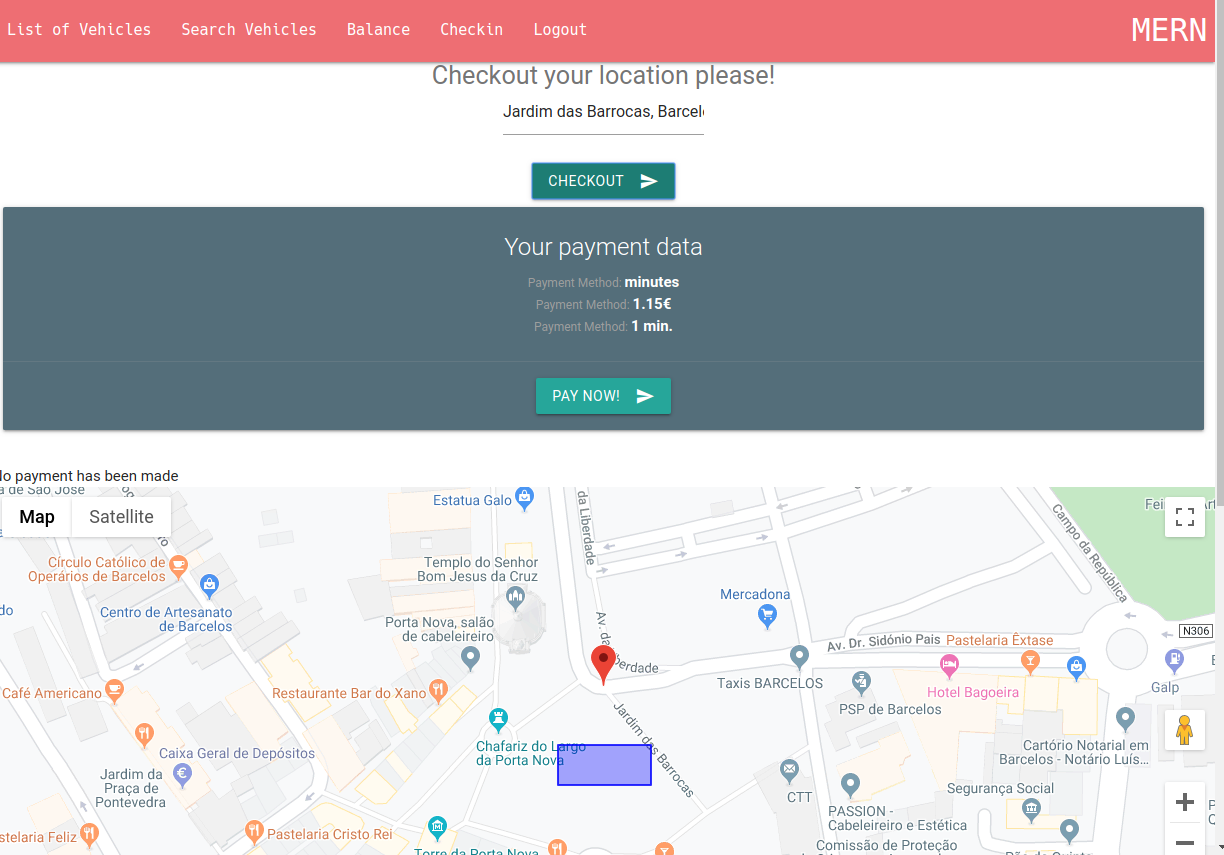


E só depois tem a opção para fazer checkout que o vai redirecionar para outro ecra.

Sair:

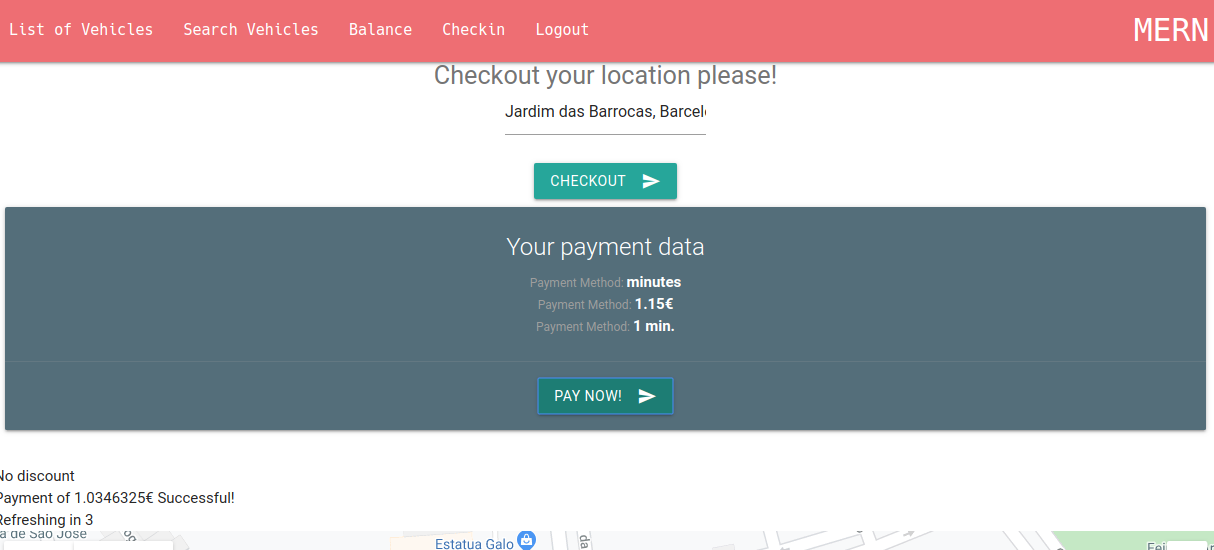


O utilizador tem a opção para escolher a rua para onde quer fazer o seu checkout.



Depois disso vai ter um card com o valor que tem a pagar.

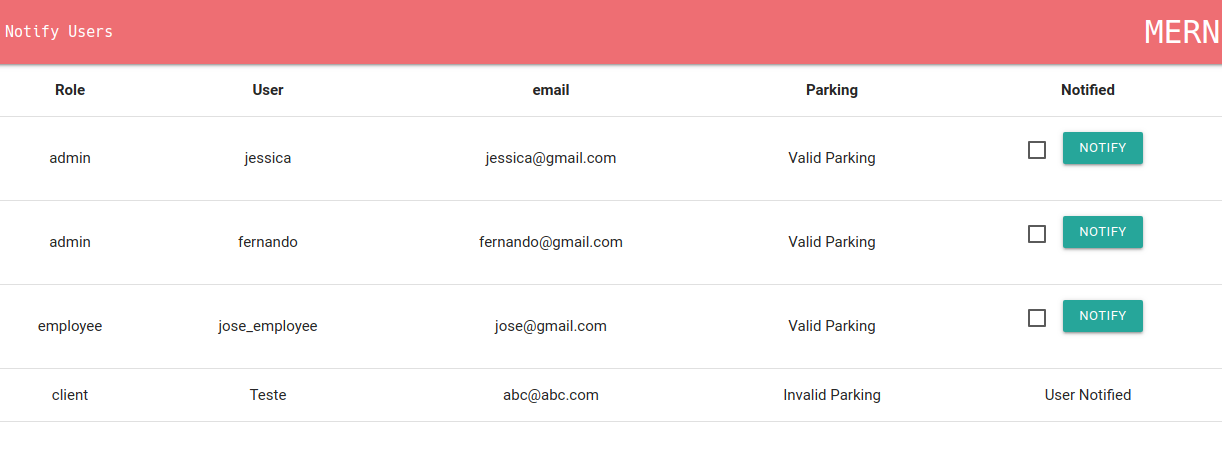
Se estacionar dentro do Poligono tem um desconto de 50 centimos. Neste caso não o tem.



O utilizador tem depois os dados certos de quanto pagou em baixo e é redirecionado para o seu saldo automaticamente após 3 segundos.

**Funcionário**

O funcionário na nossa aplicação tem apenas a função de validar um utilizador sendo que esse utilizador notificado quando logado vai receber uma notificação igual (mas com mensagem diferente) à que vimos em cima no Saldo.



**Administrador**

Dashboard

Na aplicação frontend foi criado um dashboard, que só o administrador da aplicação consegue visualizar. Estes dois dashboards funcionam em real-time, cada vez que é feita uma alteração na base de dados, é despoletado um trigger que imediatamente atualiza o gráfico. Para tornar isso possível foi utilizado o Pusher (<https://pusher.com/>).

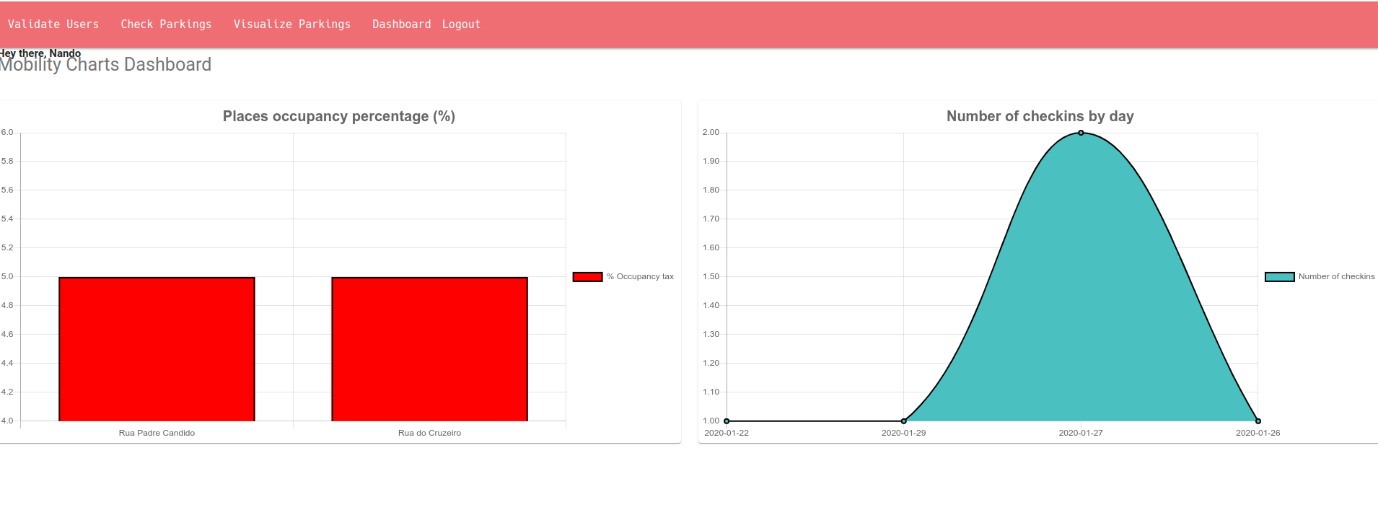
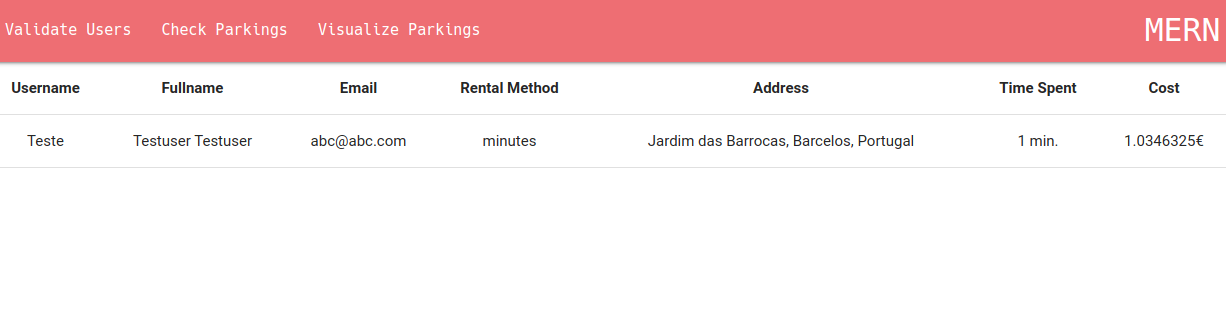
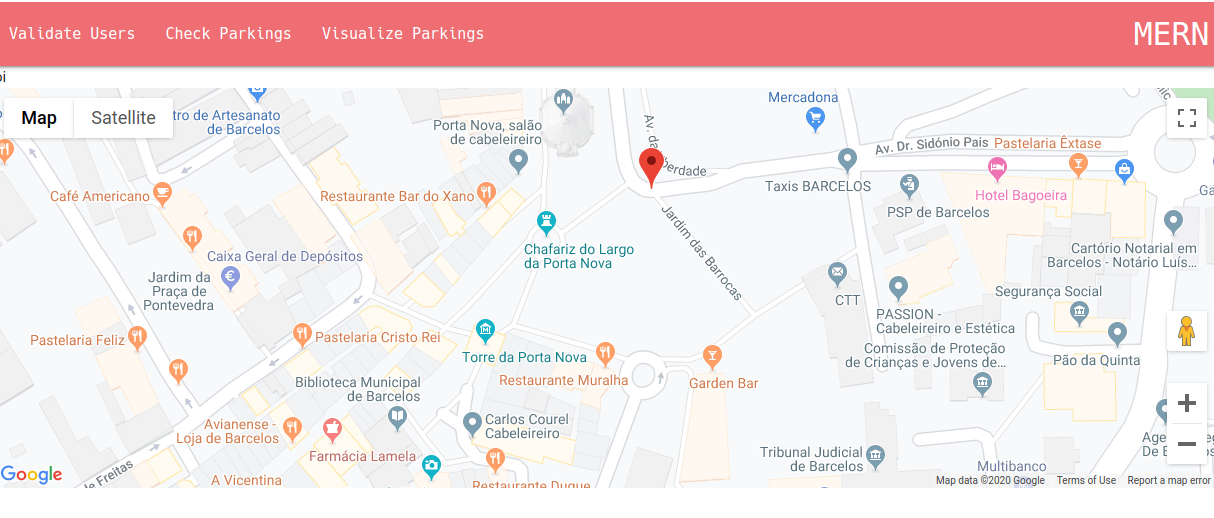


Figura 1 Dashboards

O administrador valida os utilizadores que ainda não tenham o seu login validado na base de dados para que possam aceder a todas as funcionalidades da aplicação.

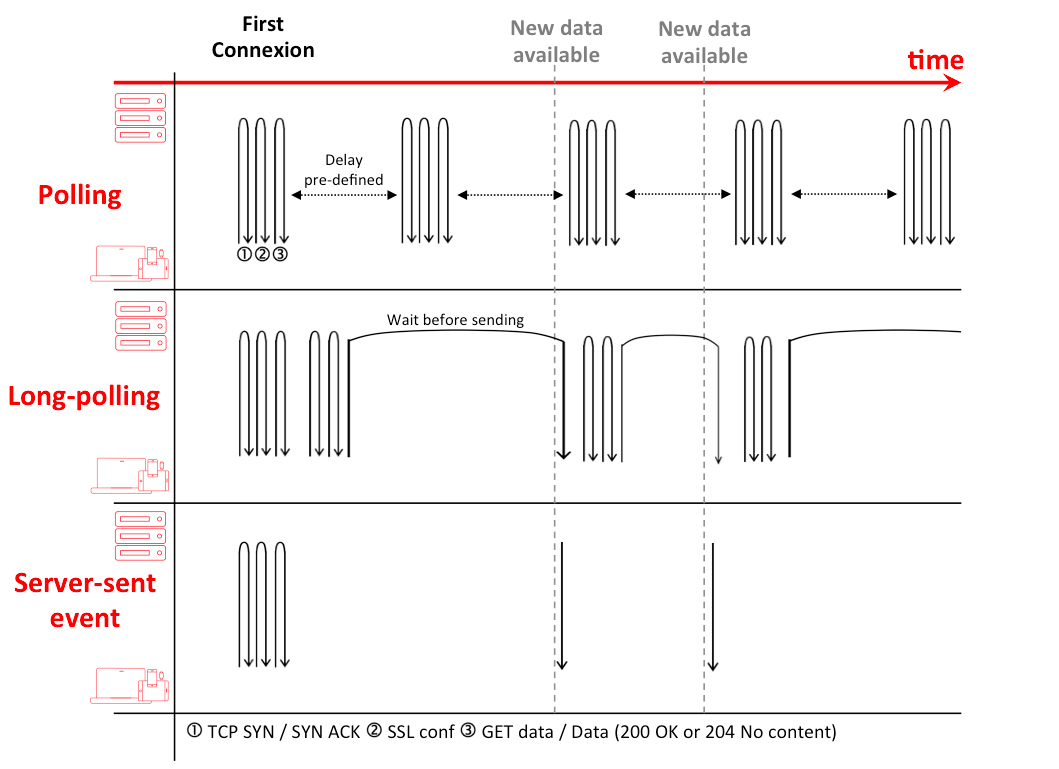


Pode também verificar os utilizadores que tenham feito Checkout



Bem como verificar os seus pontos no mapa como se pode ver na imagem acima.

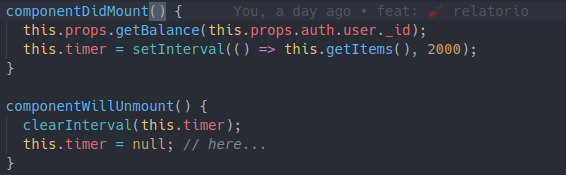
**Notificações**



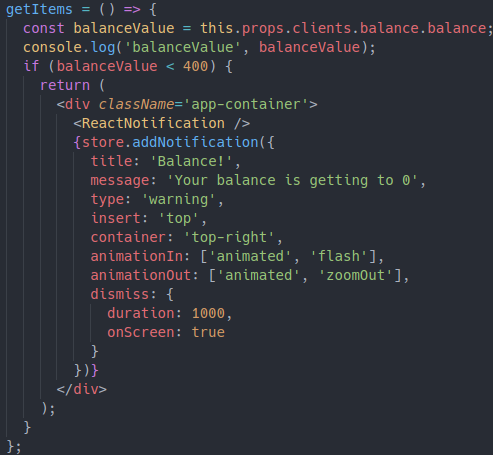
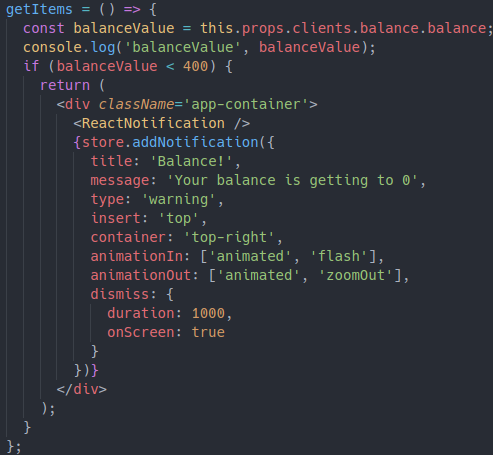
Para implementar o sistema de notificações a escolha recaiu sobre Polling visto que pode ser feito apenas do lado do cliente e de uma forma bastante fácil em react.

Fonte: <https://codeburst.io/polling-vs-sse-vs-websocket-how-to-choose-the-right-one-1859e4e13bd9>

O que fizemos para o nosso exemplo especifico foi ter a função dentro de um timer e estar à escuta do endpoint de X em X tempo como se pode verificar pelo exemplo de código em baixo.



O que fazemos é ter uma função executada de 2 em 2 segundos, neste caso a função getItems, que dentro executa uma condição para a notificação ser renderizada.

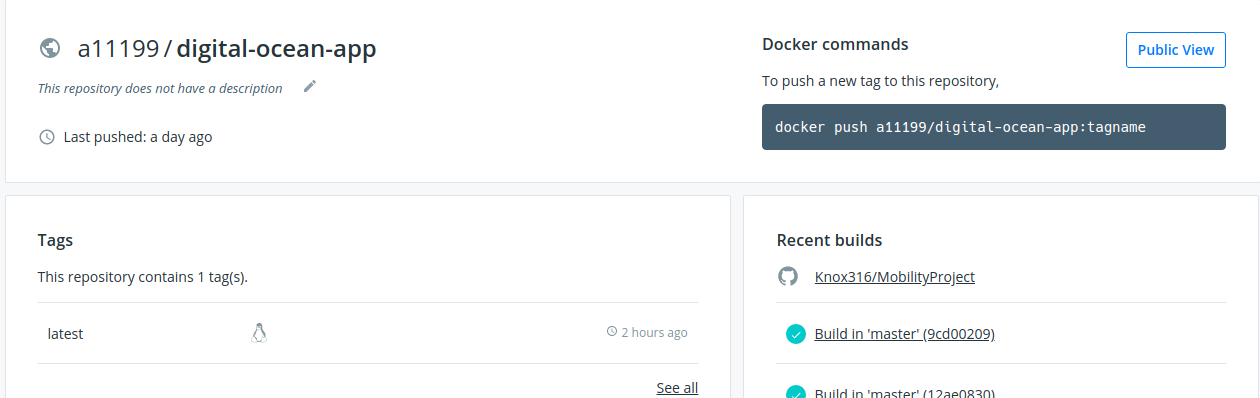


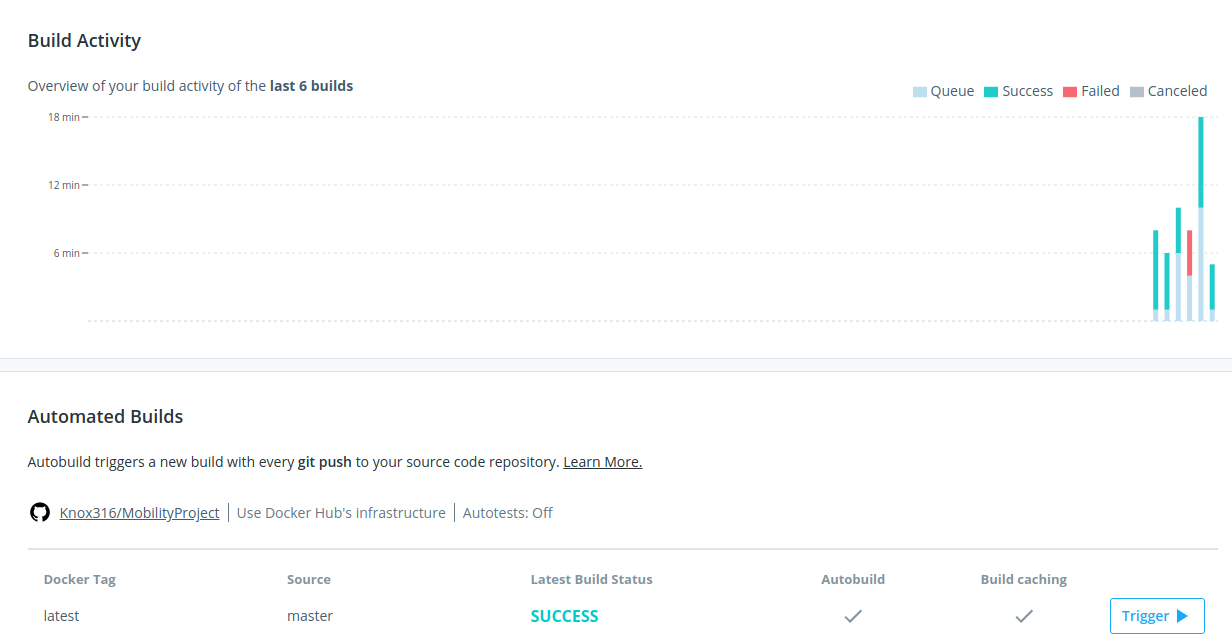
O componente ReactNotification é importado de:

<https://www.npmjs.com/package/react-notifications-component>

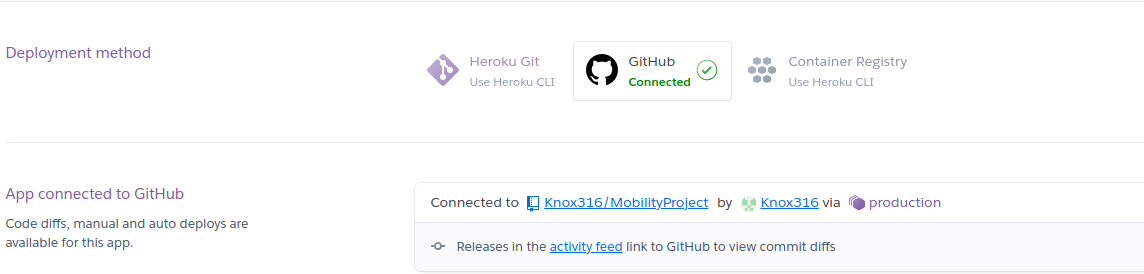
***Deploy e instalação***

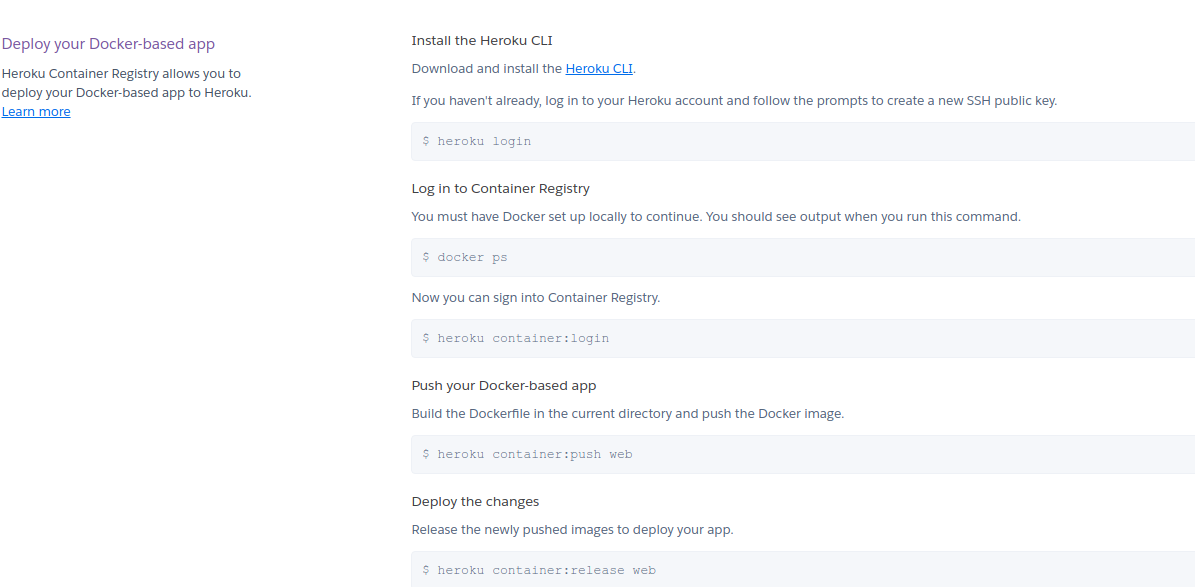
As imagens dos serviços estão guardados em containers no Docker Hub através do Docker. Estas imagens estão ligadas ao repositório do projeto para que possamos ter as imagens sempre atualizadas com o ultimo Push feito ao repositório.





Por sua vez o conteudo destas imagens está ligado a um container no Docker para que possamos fazer push das nossas builds para as builds no Heroku.



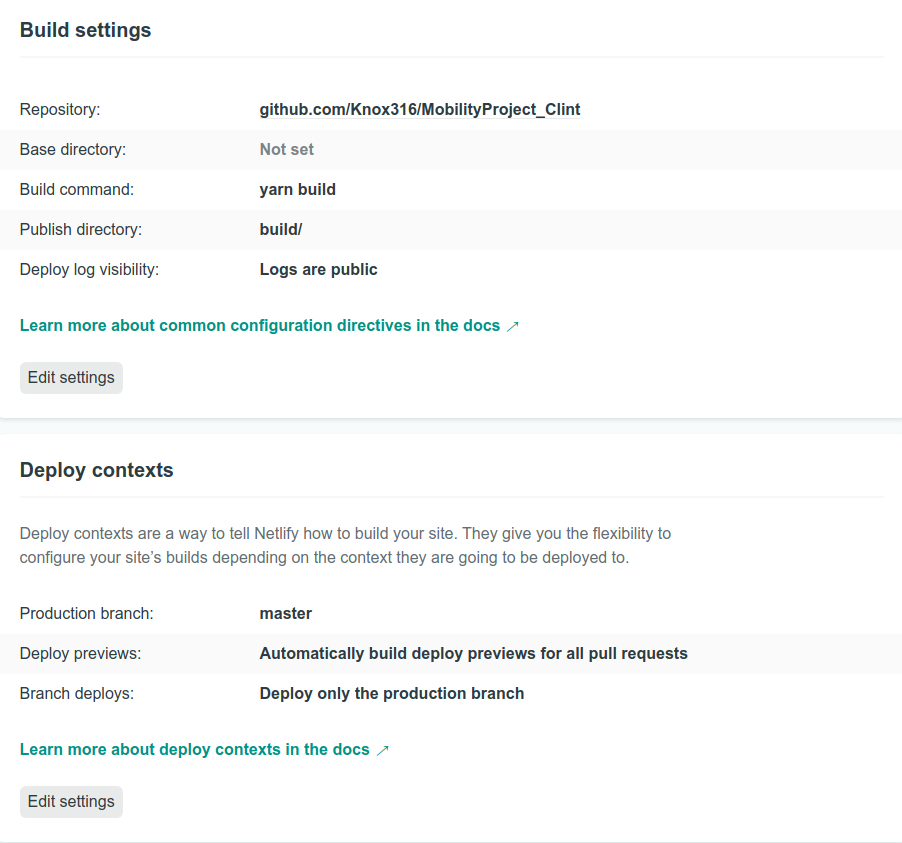


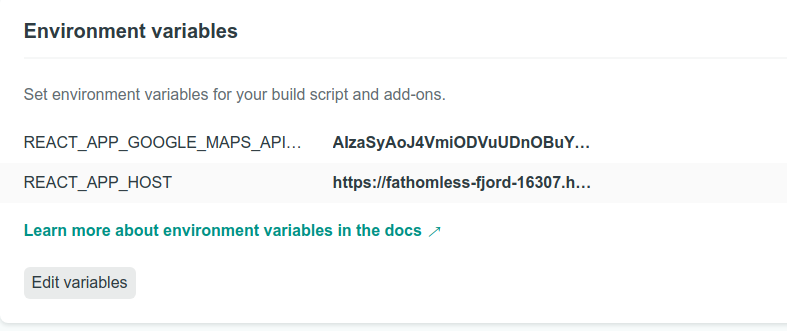
O processo é resumido através de:

1. **Pelo Github:**
   1. sudo git add .
   2. sudo git commit -m "."
   3. sudo git push heroku master
2. **Por container:**
   1. docker tag 65ac6e785094 registry.heroku.com/fathomless-fjord-16307/web (esta primeira alinah só é feita uma vez para indicar-mos qual é a imagem do docker que queremos linkar ao heroku através da tag 65ac6e785094)
   2. docker pull a11199/digital-ocean-app
   3. heroku container:push web --app fathomless-fjord-16307
   4. sudo heroku container:release web --app fathomless-fjord-1630
   5. heroku open --app fathomless-fjord-16307

Base url: <https://fathomless-fjord-16307.herokuapp.com/>

Para fazer hosting do client em React utilizados a plataforma Netlify <https://app.netlify.com/sites/quizzical-almeida-fe7a76/overview>





Base url: [quizzical-almeida-fe7a76.netlify.com](https://quizzical-almeida-fe7a76.netlify.com/)

Esta plataforma permite connectar com uma conta no Github e, entre outras coisas, configurar variaveis de ambientes e deployes baseados em actividade no repositório

Os builds são automatizados a partir de mudanças que haja no repositorio de backend e frontend tanto no Netlify como no Docker (Docker Hub).

**Pontos Futuros a melhorar:**

- Display de varios places em vez de apenas um;

- Display do nome do veiculos em vez do codigo no ecra de checkin;

- Correções a nivel do ecra do Checkin e Checkout quando se faz refresh. Entre um checkin, um refresh e um checkout pode haver erros depois para voltar a efetuar o checkin.

-Distribuição dos containeres de Docker por Microserviços;

1. [↑](#footnote-ref-15118)