Crear una Lightning App para recibir propinas utilizando el lenguaje Rust

Ejercicio para aspirantes al curso intensivo de Torogoz.dev

Introducción: El presente ejercicio, es parte de las evaluaciones integrarles que los estudiantes deben completar, para optar al curso intensivo, se evaluará la entrega y funcionamiento del ejercicio, como extras que cada estudiante decida agregarle, este ejercicio no asegura la aceptación, los organizadores harán una evaluación integral, en base a experiencia, compromiso, participación etc.,

NOTA: Todo lo recomendamos hacer en LINUX, pero también tiene opción de utilizar Windows, instalando Linux como Sub-System. Para saber cómo configurarlo de esta manera, puedes seguir los pasos en esté vídeo. https://www.youtube.com/watch?v=Cvrqmq9A3tA

Requisitos:

- Lightning POLAR (simulador de nodo LN) o utilizar el nodo LN si ya tiene uno instalado y configurado.
- Rust >= 1.65.0
- Framework web **rocket**
- Serde
- Cargo >= 1.65.0
- LND >= 0.14.2
- Dotenv (Adminsitrar Variables de Entorno)

- Docker Server and Docker Compose (cada uno por separado)
- PC con sistema operativo Linux o prepara una partición con Linux, se recomienda la distribución de UBUNTU con versión más reciente, pero puede ser cualquiera.
- Tener un espacio libre de al menos 512 GB

Instalando POLAR

Para instalar POLAR seguir las instrucciones que se detallan en el **GitHub** del proyecto: https://github.com/jamaljsr/polar

Primero debes instalar Docker Server para Linux: https://docs.docker.com/desktop/install/linux-install/

Luego instalar Docker Compose para Linux: https://docs.docker.com/compose/install/linux/#install-the-plugin-manually

Por último, instala Polar v1.4.0 para Linux

https://github.com/jamaljsr/polar/releases/download/v1.4.0/polar-linux-x86_64-v1.4.0.AppImage

Finalmente probar que POLAR levante y funcioné, puedes probar abrir canales, aquí hay un vídeo de ejemplo: https://www.youtube.com/watch?v=mb37durvPns&ab_channel=PolarLightning

En polar se pide configurar lo siguiente:

Crear una red con los valores por defecto, 3 nodos **LND**, (Alice, Bob y Carol), además de 1 nodo bitcoin core.

Para hacerlo, presionamos el botón crear red, una vez veamos en la app el gráfico donde aparecen nuestros nodos hacemos clic en el botón **comienzo** y espera unos segundos hasta que el indicador de cada nodo cambie de color a verde.

Para poder enviar pagos en Lightning es necesario que los nodos estén interconectados por medio de canales de pago, crear canales con Polar es muy sencillo. Solo necesitamos hacer clic con el mouse en una de las orejas del nodo Alice y arrastrarlo hasta una de las orejas del nodo Bob, inmediatamente te debe aparecer una ventana modal titulada **Abrir nuevo canal**, aumentamos la capacidad a 10_000_000 sats y presionamos el botón de abrir canal, repetimos la misma acción y creamos un canal de Bob a Carol, también de 10_000_000 sats.

El canal entre Alice y Bob, como fue creado por Alice, tiene los 10 millones de satoshis del lado de Alice. Alice puede realizar pagos, pero no recibir pagos, lo mismo ocurre con el canal creado desde Bob a Carol. Para permitir que todos los nodos puedan enviar y recibir creamos una factura por 5_000_000 sats con el nodo Carol y la pagamos con el nodo Alice.

Instalando Rust

Para instalar Rust debes seguir las instrucciones en su sitio oficial para Linux: https://www.rust-lang.org/tools/install

Luego de haber instalador **rust**, puedes proceder a validar la instalación incluyendo **Cargo**.

Puedes utilizar los siguientes comandos en Linux.

$ Rustc --version

$ Cargo –version

Luego puedes correr un programa llamado RANA creado en Rust, para validar que todo esté Ok.

Para poder correr el programa RANA, sigue los pasos detallados en el GitHub del proyecto.
https://github.com/grunch/rana

IMPORTANTE: Cuando llegues al punto de correr el proyecto para validar funcionamiento, **utiliza una dificultad menor a 10** para que el programa finalice rápido, una dificulta mayor a 30 podría llevar días de proceso, solo nos interesa ver que el programa corra, para validar que Rust y Cargo se instalaron bien.

Preparando la base para el proyecto de Propinas con LN

Luego de haber instalado todas las dependencias requeridas, procederemos a crear el esqueleto de aplicación de propinas.

Crearemos el repositorio del proyecto.

```
$ cargo new lntip
```

La aplicación generada tendrá la siguiente estructura de directorios:

```
├── Cargo.lock
├── Cargo.toml
└── src
    └── main.rs
```

Con un editor de texto abrimos el archivo **lntip/src/main.rs** y vemos lo siguiente:

```
fn main() {
    println!("Hello, world!");
}
```

Luego entramos al directorio simplemente corremos el server.

```
$ cd lntip
$ cargo run
```

Instalamos el **framework web rocket**, el cual nos permitirá que nuestro programa sea un servidor web.

```
$ cargo add rocket@0.5.0-rc
```

Sustituimos el contenido de **main.rs** con lo siguiente:

```
#[macro_use]
extern crate rocket;

#[get("/hola/<name>/<age>")]
fn hello(name: &str, age: u8) -> String {
    format!("Hola, tienes {} años y te llamas {}!", age, name)
}

#[launch]
fn rocket() -> _ {
    rocket::build().mount("/", routes![hello])
}
```

Ejecutamos nuevamente el proyecto:

```
$ cargo run  (debes estar dentro del directorio del proyecto)
```

A continuación, ve a esta dirección **http://localhost:8000** en el navegador para acceder a la aplicación creada.

Instalando y configurando cargo-watch

Para no tener que reiniciar el proyecto cada vez que realicemos un cambio en el código instalaremos **cargo-watch**: https://crates.io/crates/cargo-watch

```
$ cargo install cargo-watch
```

Vamos a la consola donde está corriendo **`cargo run`**, presionamos **`ctrl + c`** y volvemos a iniciar el proyecto, pero esta vez ejecutaremos **`cargo watch -x run`**.

Conectándonos a LND (Cliente de Lightning)

Para poder conectarnos a un nodo Lightning desde **rust**, utilizaremos la librería **tonic_openssl_lnd:** https://crates.io/crates/tonic_openssl_lnd

También instalaremos **dotenv** para administrar las variables de entorno: https://crates.io/crates/dotenv

```
$ cargo add tonic_openssl_lnd dotenv
```

En nuestro directorio **lntip** creamos un archivo llamado **`.env`**, debe contener estas variables:

```
LND_GRPC_HOST=''
LND_GRPC_PORT=''
# path to tls.cert file
LND_CERT_FILE=''
# path to macaroon file
LND_MACAROON_FILE=''
```

Volvemos a **Polar**, seleccionamos a Alice, el nodo al que nos queremos conectar, vamos a la pestaña "**Conectar**", copiamos el contenido de **Host GRPC**, al copiarlo obtendremos algo como esto `127.0.0.1:10001`, de aquí tomaremos la IP del Host (127.0.0.1) y la colocamos en la variable `LND_GRPC_HOST`, el puerto (10001) en `LND_GRPC_PORT`.

En la parte de abajo de la pestaña conectar seleccionamos **Rutas de archivo** y copiamos el contenido de **TLS Cert** y lo colocamos en la variable **`LND_CERT_FILE`** y finalizamos haciendo lo mismo con el **admin macaroon** en `LND_MACAROON_FILE`.

Ahora agregamos esta línea al archivo **`main.rs`** ubicado en la raíz del directorio de trabajo, debemos copiarlo en la primera línea del archivo.

```
use dotenv::dotenv;
use std::env;
```

Para hacer nuestro código un poco más legible vamos a crear un archivo para manejar las rutas **src/routes.rs**, ¡por ahora agreguemos una sola ruta **index** que nos devolverá un **"Hola mundo!",** el archivo quedará de esta manera:

```
use rocket::*;

#[get("/")]
pub fn index() -> &'static str {
    "Hola mundo!"
}
```

PUNTO DE VALIDACIÓN: Chequeamos http://localhost:8000 en el navegador y deberíamos poder ver **"Hola mundo!".**

Conectar el proyecto al nodo Lightning

Para conectar **rust** con nuestro **nodo lightning** creamos un nuevo archivo **`src/lightning.rs`** en el cual escribimos la función que realizará la conexión y nos retorna un cliente.

```
use dotenv::dotenv;
use std::env;
use tonic_openssl_lnd::{LndClientError, LndLightningClient};

pub async fn connect() -> Result<LndLightningClient, LndClientError>
{
    dotenv().ok();
    let port: u32 = env::var("LND_GRPC_PORT")
        .expect("LND_GRPC_PORT must be set")
        .parse()
        .expect("port is not u32");
    let host = env::var("LND_GRPC_HOST").expect("LND_GRPC_HOST must
be set");
    let cert = env::var("LND_CERT_FILE").expect("LND_CERT_FILE must
be set");
    let macaroon =
env::var("LND_MACAROON_FILE").expect("LND_MACAROON_FILE must be
set");
    // Connecting to LND requires only host, port, cert file, and
macaroon file
    let client = tonic_openssl_lnd::connect_lightning(host, port,
cert, macaroon)
        .await
        .expect("Failed connecting to LND");

    Ok(client)
}
```

Creando una factura Lightning

Ahora vamos a crear una función que crea una factura lightning y la llamaremos **`create_invoice()`:**

```
use tonic_openssl_lnd::lnrpc::{AddInvoiceResponse, Invoice}; // <--
al inicio

pub async fn create_invoice(
    client: &mut LndLightningClient,
    description: &str,
    amount: u32,
) -> Result<AddInvoiceResponse, LndClientError> {
    let invoice = Invoice {
        memo: description.to_string(),
        value: amount as i64,
        ..Default::default()
    };
    let invoice = client.add_invoice(invoice).await?.into_inner();

    Ok(invoice)
}
```

El archivo completo debe quedar así:

```
use dotenv::dotenv;
use std::env;
use tonic_openssl_lnd::lnrpc::{AddInvoiceResponse, Invoice};
use tonic_openssl_lnd::{LndClientError, LndLightningClient};

pub async fn connect() -> Result<LndLightningClient, LndClientError>
{
    dotenv().ok();
    let port: u32 = env::var("LND_GRPC_PORT")
        .expect("LND_GRPC_PORT must be set")
        .parse()
        .expect("port is not u32");
    let host = env::var("LND_GRPC_HOST").expect("LND_GRPC_HOST must
be set");
    let cert = env::var("LND_CERT_FILE").expect("LND_CERT_FILE must
be set");
    let macaroon =
env::var("LND_MACAROON_FILE").expect("LND_MACAROON_FILE must be
set");
    // Connecting to LND requires only host, port, cert file, and
macaroon file
    let client = tonic_openssl_lnd::connect_lightning(host, port,
cert, macaroon)
        .await
        .expect("Failed connecting to LND");

    Ok(client)
```

```
}

pub async fn create_invoice(
    client: &mut LndLightningClient,
    description: &str,
    amount: u32,
) -> Result<AddInvoiceResponse, LndClientError> {
    let invoice = Invoice {
        memo: description.to_string(),
        value: amount as i64,
        ..Default::default()
    };
    let invoice = client.add_invoice(invoice).await?.into_inner();

    Ok(invoice)
}
```

Para poder utilizar estas funciones debemos decirle al proyecto que **lightning.rs** existe, para ello vamos a **src/main.rs** y debajo de **mod routes**; agregamos **mod lightning**;.

Volvamos a nuestro archivo de rutas **src/routes.rs**, vamos a crear una nueva ruta que utilizaremos para crear nuevas facturas lightning network para recibir pagos en Bitcoin, la nueva **ruta /create_invoice.**

Agregamos al inicio use **crate::lightning;** y debajo de la función **index** escribimos la nueva ruta:
Nota: **crate** (son liberarías ya creadas por otros).

```
#[get("/create_invoice/<description>/<amount>")]
pub async fn create_invoice(description: &str, amount: u32) -> String
{
    let invoice = lightning::create_invoice(description, amount)
        .await
        .unwrap();

    invoice.payment_request
}
```

Solo falta un detalle más, en **`src/main.rs`** debemos agregar la nueva ruta, así que modificamos el método **`mount():`**

```
.mount("/", routes![routes::index, routes::create_invoice])
```

OJO: Al abrir la ruta: http://localhost:8000/create_invoice/factura/999 veremos una cadena de texto que comienza por **`lnbc...`**, si te aparece esto, felicidades!!! tu app ya puede interactuar con Lightning

Network.

Retornando JSON desde rocket.rs

La ruta /**create_invoice** nos retorna la factura, pero para verificar el pago vamos a necesitar el **hash** de la factura, esto lo podemos obtener fácilmente de la **struct `AddInvoiceResponse`**, crearemos una nueva **struct** que utilizaremos para retornar un **json** que contenga la factura (**invoice**) y el **hash**, para esto utilizamos **serde**.

Agregamos **serde** a nuestro proyecto:

```
cargo add serde
```

En **`src/routes.rs`** usamos **serde** al inicio del proyecto y agregamos una nueva **struct `InvoiceResponse`:**

```
use rocket::serde::{json::Json, Serialize};

#[derive(Serialize, Default)]
pub struct InvoiceResponse {
    payment_request: String,
    hash: String,
    paid: bool,
    preimage: String,
    description: String,
}
```

Modificamos **`/create_invoice`** para retornar **json** en lugar de **string**:

```
#[get("/create_invoice/<description>/<amount>")]
pub async fn create_invoice(description: &str, amount: u32) ->
Json<InvoiceResponse> {
    let invoice = lightning::create_invoice(description, amount)
        .await
        .unwrap();

    let hash_str = invoice
        .r_hash
        .iter()
        .map(|h| format!("{h:02x}"))
        .collect::<Vec<String>>()
        .join("");
```

```
    Json(InvoiceResponse {
        payment_request: invoice.payment_request,
        hash: hash_str,
        ..Default::default()
    })
}
```

Diseccionando la Factura Lightning

Como ya lo hemos estudiado en el curso de LN, una factura se compone de deferentes datos, Para entender la factura que acabamos de generar podemos ir a https://www.bolt11.org, pegarla en la página y ver la metadata incluida en la misma, todo el detalle lo podemos encontrar en el estándar de Lightning bolt11

Preparamos la vista (Front End)

Creamos el directorio **`templates`** en la raíz del proyecto y dentro el archivo **`layout.html.hbs`:**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap
.min.css" rel="stylesheet" integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC"
crossorigin="anonymous">
  <title>Lightning Network Tipping app</title>
</head>
<body>
  <div class="container">
    {{~> content}}
  </div>
  <footer class="footer"></footer>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.j
s" integrity="sha512-
894YE6QWD5I59HgZOGReFYm4dnWc1Qt5NtvYSaNcOP+u1T9qYdvdihz0PPSiiqn/+/3e7
```

```
Jo4EaG7TubfWGUrMQ==" crossorigin="anonymous" referrerpolicy="no-
referrer"></script>
  <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.b
undle.min.js" integrity="sha384-
MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJoMp4YLEuNSfAP+JcXn/tWtIaxVXM"
crossorigin="anonymous"></script>
  {{~> scripts}}
</body>
</html>
```

Hasta este punto, la estructura de archivos/directorios debería ser la siguiente:

```
├── src
│   ├── routes.rs
│   ├── lightning.rs
│   └── main.rs
├── templates
│   └── layout.html.hbs
├── Cargo.lock
├── Cargo.toml
└── .env
```

Creamos un formulario para generar la factura Lightning

Para recibir un pago en Lightning Network necesitamos un formulario donde el usuario indique el monto y la descripción, dentro del directorio **`templates`** creamos un archivo llamado **`index.html.hbs`** que contenga lo siguiente:

```
{{#*inline "content"}}
    <div id="form" class="mt-5 mb-5 collapse">
      <form autocomplete="off" class="row mt-3 g-3" id="form">
        <div class="col">
          <div class="form-floating mb-3">
            <input type="text" id="description" class="form-control">
            <label for="description">Descripción</label>
          </div>
          <div class="form-floating mb-3">
            <input type="text" id="amount" class="form-control"
required>
```

```
            <label for="amount">Monto</label>
          </div>

          <div class="mb-3">
            <button type="button" id="send-btn" class="btn btn-light
btn-lg">Enviar</button>
          </div>
        </div>
      </form>
    </div>
    <div id="invoice" class="mt-5 mb-5 collapse bg-light rounded-3
jumbotron">
      <h3 id="invoice-memo">LNTip</h3>
      <h3><span id="invoice-amount">0</span> SATs</h3>
      <p class="lead">Para continuar, haga un pago con una billetera
con soporte Bitcoin Lightning Network a la siguiente factura.</p>
      <p class="lead">Esta factura expira en 10 minutos.</p>
      <hr class="my-4">
      <h6>Factura:</h6>
      <p id="invoice-text" class="text-break"></p>
    </div>

    <div id="success-box" class="mt-5 mb-5 collapse bg-light rounded-
3 jumbotron">
      <div align="center">
        <iframe src="https://giphy.com/embed/BzyTuYCmvSORqs1ABM"
width="480" height="480" frameBorder="0" class="giphy-embed"
allowFullScreen></iframe>
      </div>
      <hr class="my-4">
      <p class="lead" align="center">¡Pago realizado correctamente!
😀</p>
      <hr class="my-4">
    </div>
{{/inline}}
{{#*inline "scripts"}}
    <script src="/public/main.js"></script>
{{/inline}}
{{~> layout~}}
```

En el archivo **`cargo.toml`** agregamos la dependencia para utilizar el **template engine handlebars con rocket,** agregamos lo siguiente:

```
[dependencies.rocket_dyn_templates]
version = "0.1.0-rc.2"
features = ["handlebars"]
```

En el archivo **src/main.rs** agregamos el soporte para **templates**:

```
use rocket::fs::{relative, FileServer}; // <--
use rocket_dyn_templates::Template; // <--

#[macro_use]
extern crate rocket;
mod lightning;
mod routes;

#[launch]
fn rocket() -> _ {
    rocket::build()
        .mount("/public", FileServer::from(relative!("static"))) //
<-- Seteamos un directorio para contenido estático
        .mount("/", routes![routes::index, routes::create_invoice])
        .attach(Template::fairing()) // <--
}
```

Incluyendo Javascript en el frontend

La manera más directa que tenemos para interactuar con el usuario es utilizando **javascript** en el web browser, para esto creamos un directorio para almacenar contenido estático llamado **static** en la raíz del proyecto, dentro de **static** creamos un archivo **main.js** que ya estamos llamando desde el **layout.html.hbs**, el contenido inicial de **main.js** es el siguiente:

```
// /static/main.js
$(() => {
  $("#form").collapse("show");
  $("#send-btn").click(() => {
    console.log("!hola");
  });
});
```

Hasta este punto, la estructura de archivos/directorios debería ser la siguiente:

```
├── src
│   ├── routes.rs
│   ├── lightning.rs
│   └── main.rs
```

```
├── templates
│   ├── index.html.hbs
│   └── layout.html.hbs
├── static
│   └── main.js
├── Cargo.lock
├── Cargo.toml
└── .env
```

Presionamos el botón **Enviar** y si todo está bien podremos ver en la consola un mensaje **!hola**, con esto ya podemos modificar este evento para que envíe la información a nuestra API, el archivo debe verse así:

```
$(() => {
  $("#form").collapse("show");
  $("#send-btn").click(sendBtn);
});

const sendBtn = async () => {
  const amount = $("#amount").val();
  const description = $("#description").val();
  $.ajax({
    url:
`http://localhost:8000/create_invoice/${description}/${amount}`,
    success: function (invoice) {
      console.log(invoice);
    },
    async: false,
  });
};
```

Actualizamos el formulario, ingresamos descripción y monto, al enviar debemos poder ver la factura y el hash en la consola del navegador.

Recibiendo el pago

Necesitamos saber si una factura ha sido pagada o no, vamos a crear una nueva función en **`src/lightning.rs`** en la que le solicitemos al nodo el estado actual de una factura.

```
use tonic_openssl_lnd::lnrpc::{AddInvoiceResponse, Invoice,
PaymentHash}; // <-- agregamos PaymentHash
```

```
pub async fn get_invoice(hash: &[u8]) -> Result<Invoice,
LndClientError> {
    let mut client = connect().await.unwrap();
    let invoice = client
        .lookup_invoice(PaymentHash {
            r_hash: hash.to_vec(),
            ..Default::default()
        })
        .await?
        .into_inner();

    Ok(invoice)
}
```

Creamos una nueva ruta que recibe el hash de la factura y consulta la función que recién hemos creado, pero antes instalamos una nueva dependencia para el manejo de hexadecimales:

```
cargo add hex
```

Y al archivo de rutas **src/routes.rs** le agregamos lo siguiente:

```
use hex::FromHex;
use tonic_openssl_lnd::lnrpc::invoice::InvoiceState;

#[get("/invoice/<hash>")]
pub async fn lookup_invoice(hash: &str) -> Json<InvoiceResponse> {
    let hash = <[u8; 32]>::from_hex(hash).expect("Decoding failed");
    let invoice = lightning::get_invoice(&hash).await.unwrap();
    let mut preimage = String::new();
    let mut paid = false;

    if let Some(state) = InvoiceState::from_i32(invoice.state) {
        if state == InvoiceState::Settled {
            paid = true;
            preimage = invoice
                .r_preimage
                .iter()
                .map(|h| format!("{h:02x}"))
                .collect::<Vec<String>>()
                .join("");
        }
    }
    Json(InvoiceResponse {
        paid,
        preimage,
        description: invoice.memo,
```

```
        ..Default::default()
    })
}
```

Para terminar la lógica en **rust**, agregamos la nueva ruta **`routes::lookup_invoice`** a **`src/main.rs`** como hemos hecho con las otras rutas, solo nos falta terminar el javascript.

Ahora en **`main.js`** creamos una función llamada **`waitPayment()`** que consulta si el pago ha sido realizado.

```
const waitPayment = async (hash) => {
  $.ajax({
    url: `http://localhost:8000/invoice/${hash}`,
    success: function (invoice) {
      if (invoice.paid) {
        console.log("pago realizado");
      }
    },
    async: false,
  });
};
```

Ahora nos encontramos con un problema, la función **`waitPayment()`** se ejecuta solo una vez, el usuario puede haber pagado y no le hemos podido indicar que su pago ha sido recibido, para esto utilizamos una función de javascript llamada **`setInterval()`** que nos permite ejecutar una función indefinidamente cada intervalo de tiempo que le hayamos indicado.

Modifiquemos las funciones **`waitPayment()`** y **`sendBtn()`** incluyendo **`setInterval()`** y **`clearInterva()`**, a continuación se muestra la versión final de static/main.js

```
let interval = null;

$(() => {

  $("#form").collapse("show");

  $("#send-btn").click(sendBtn);

});

const sendBtn = async () => {
```

```
  const amount = $("#amount").val();
  const description = $("#description").val();
  $.ajax({
    url: `http://localhost:8000/create_invoice/${description}/${amount}`,
    success: function (invoice) {
      $("#form").collapse("hide");
      $("#invoice-amount").text(amount);
      $("#invoice-text").text(invoice.payment_request);
      $("#invoice").collapse("show");
      $("#success-box").collapse("hide");
      interval = setInterval(waitPayment, 1000, invoice.hash);
    },
    async: false,
  });
};

const waitPayment = async (hash) => {
  $.ajax({
    url: `http://localhost:8000/invoice/${hash}`,
    success: function (invoice) {
      if (invoice.paid) {
        clearInterval(interval);
        interval = null;
```

```
        $("#form").collapse("hide");
        $("#invoice").collapse("hide");
        $("#success-box").collapse("show");
      }
    },
    async: false,
  });
};
```

FIN: Si luego de pagar la invoice creado en POLAR, puedes ver el mensaje en el Front End de **¡Pago realizado correctamente!** 😀 entonces, felicidades!!! lo has logrado, has terminado tu ejercicio.

PUNTOS EXTRAS: el ejercicio descrito hasta este punto es sencillo, efectuar un pago de un invoice y recuperar el hash para notificar del mismo.

Si quieres obtener puntos extras, te invitamos para que puedas incluir cualquier otra funcionalidad adicional que consideres, puedes también mejorar con css y las platillas del HTML para hacerlas visualmente más atractivas.

Si en lugar de utilizar POLAR, ustedes utilizan su nodo de LN que ya tienen en Mainnet o inclusive en Testnet, será genial, no solo debería tener los nodos sino canales abiertos para poder generar invoices.

AYUDA: si durante el proceso del ejercicio hay preguntas, no duden en escribir al grupo para ver como disolvemos las inquietudes.

REVISIÓN. Luego anunciaremos las fechas de las sesiones, en la cuales estaremos revisando en conjunto con el estudiante el ejercicio, para que nos muestren la DEMO, la cual debe ser funcional.

Si al final tienen dudas y quieren revisar el código de alguna clase o algún archivo, porque consideran que se equivocaron en algo o que le falta alguna parte de código, nos avisan, para que podamos comparar el archivo en cuestión.

Atentamente,

Team Torogoz Dev.