

**Taller**  
**Patrón Guaranteed Delivery**

**Tutor**  
Ing. Eduardo Mauricio Campaña Ortega

**Integrantes**  
Diego Hiriart  
Luis Corales  
Christian Samaniego

**Fecha**  
20/07/2023

## Tabla de contenido

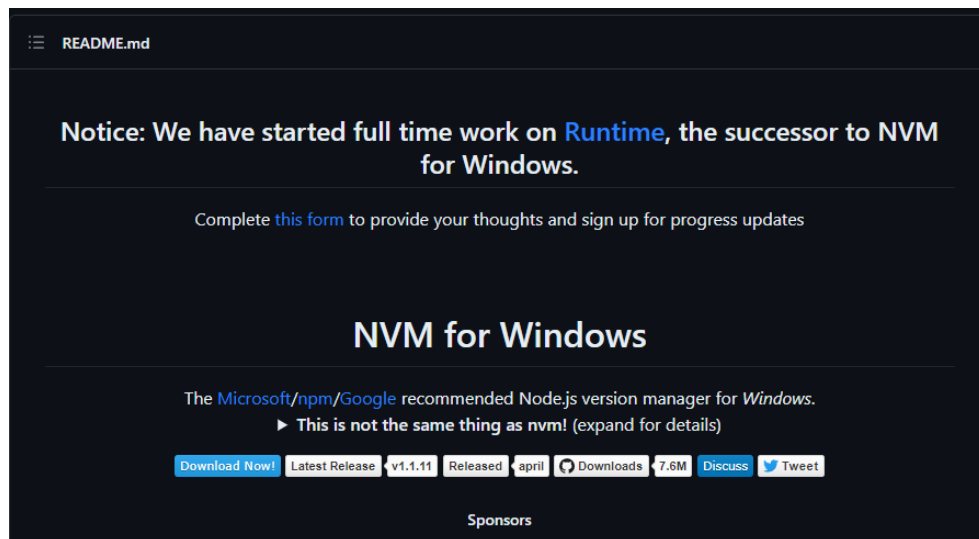
Pre-requisitos .....	1
Instalación Node.js .....	1
Desarrollo .....	4
Configuración Inicial .....	4
Cliente para recibir y procesar mensajes .....	4
Estructura y punto de entrada del sistema cliente .....	6
Estructura de mensajes recibidos .....	7
Recepción de mensajes .....	7
Rutas para que un servidor envíe mensajes .....	9
Servidor de envío de mensajes .....	10
Estructura y punto de entrada del sistema de servidor .....	11
Estructura de mensajes .....	12
Persistencia de mensajes .....	13
Envío de mensajes .....	13
Rutas de acceso a funciones .....	15
Ejecución .....	15
Conclusiones .....	19
Recomendaciones .....	19

## Pre-requisitos

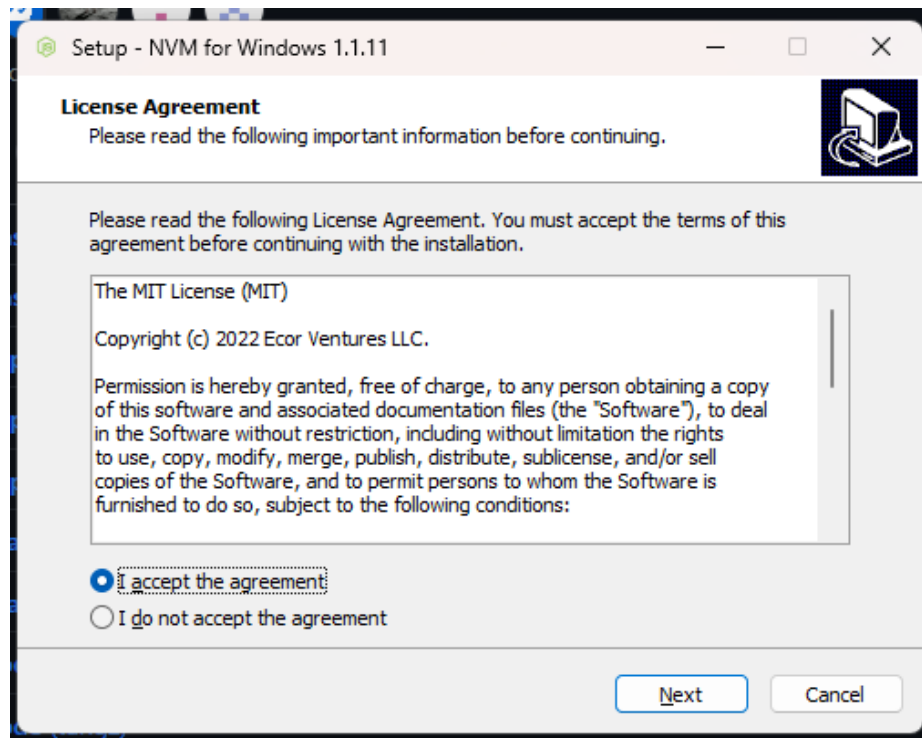
### Instalación Node.js

Para la implementación de esta solución es necesario contar con una instalación de Node.js, para lo cual se utilizará la herramienta nvm la cual permite instalar y gestionar distintas versiones de node de forma fácil.

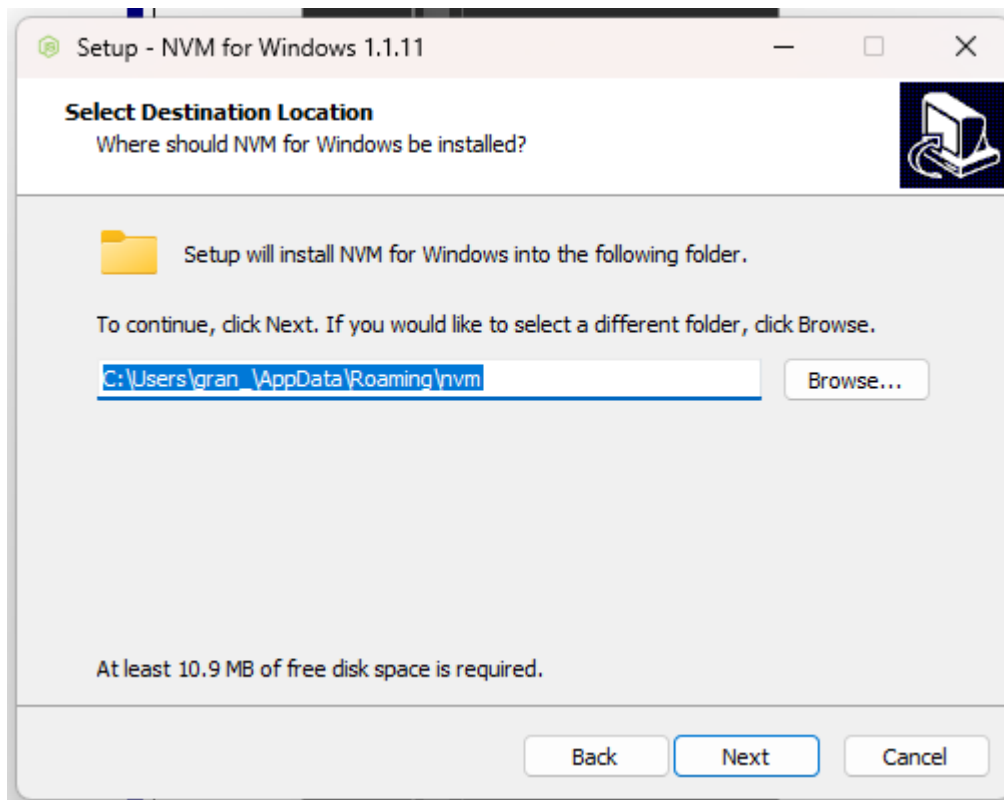
En el caso de utilizar Windows, se utilizará la herramienta nvm-windows. Para instalar la herramienta es necesario visitar el repositorio en github: <https://github.com/coreybutler/nvm-windows> y dar click en "Download Now!"



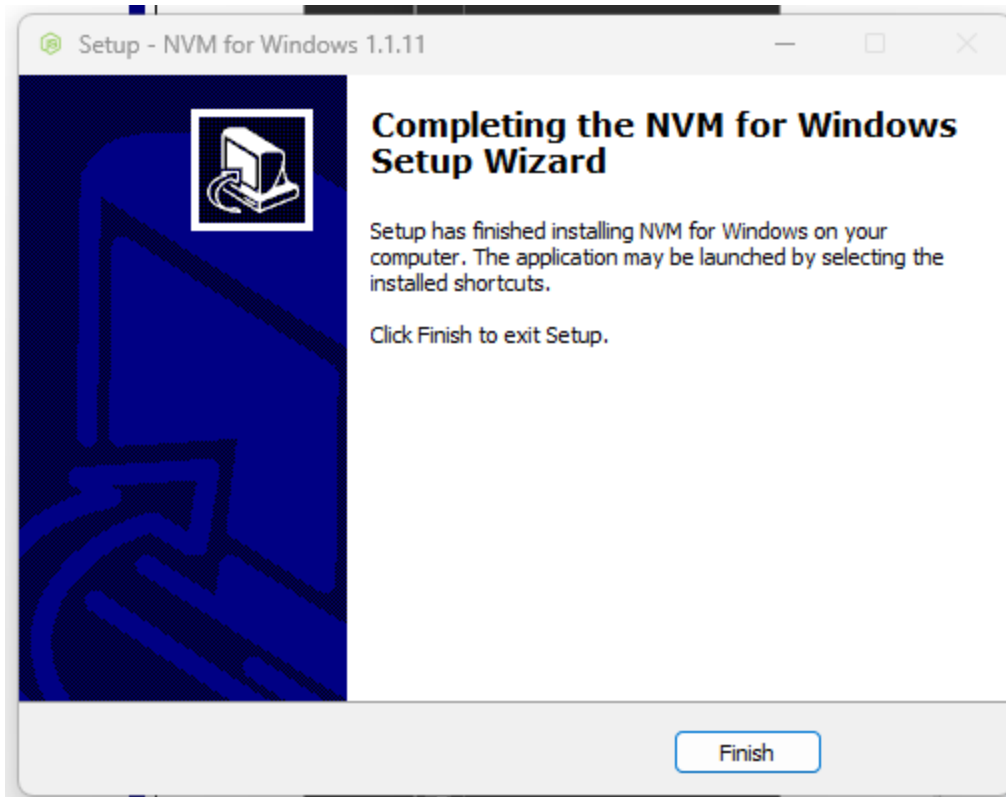
Descargamos el archivo nvm-setup.exe y lo ejecutamos. A continuación, se mostrará un wizard de instalación. Aceptamos y damos click en Next.



Seleccionamos el directorio de instalación y damos click en Next:



Tardará unos segundos en instalar y se mostrará la siguiente ventana de confirmación:



Podemos confirmar la instalación de nvm corriendo el siguiente comando en Powershell:

**nvm -v**

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\gran_> nvm -v
1.1.11
PS C:\Users\gran_> |
```

Ahora procedemos a instalar una versión de Node. En este caso instalaremos la versión estable más reciente corriendo el comando:

**nvm install lts**

```
PS C:\Users\gran_> nvm install lts
Downloading node.js version 18.17.0 (64-bit)...
Extracting node and npm...
Complete
npm v9.6.7 installed successfully.

Installation complete. If you want to use this version, type

nvm use 18.17.0
PS C:\Users\gran_> |
```

Finalmente corremos el comando **nvm use 18.17.0** para utilizar esta versión.

Al correr **node -v** podemos confirmar que se utiliza esa versión.

## Desarrollo

Este taller muestra una implementación básica de un patrón guaranteed delivery, con un sistema actuando como servidor y otro como cliente. El servidor guarda los mensajes como archivos (persistencia) que debe enviar al cliente y solo los elimina tras obtener confirmación de su recepción. De igual modo, el cliente guarda el mensaje de manera persistente para que se puedan acceder hasta ser procesados.

### Configuración Inicial

El código para este proyecto, en caso de necesitarse como guía, se puede encontrar en el siguiente repositorio:

- [Talleres-Integracion-HCS/Guaranteed delivery Hiriart Corales Samaniego at main · Diego-Hiriart/Talleres-Integracion-HCS \(github.com\)](https://github.com/Diego-Hiriart/Talleres-Integracion-HCS)

Cliente para recibir y procesar mensajes

El cliente debe alojarse en una carpeta cliente, y se genera el proyecto correspondiente corriendo este comando:

```
npm init -y
```

Esto genera un archivo **package.json** con las dependencias necesarias del proyecto, el contenido del mismo es el siguiente:

```
{
  "name": "guaranteed-delivery-client_hiriart-corales-samaniego",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "tsc --build",
    "start:dev": "nodemon"
  },
}
```

```

"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
  "express": "^4.18.2"
},
"devDependencies": {
  "@types/node": "^20.4.2",
  "nodemon": "^3.0.1",
  "ts-node": "^10.9.1",
  "typescript": "^5.1.6"
}
}

```

Para instalar las dependencias se debe correr el comando **npm install**, se requiere instalar la dependencia “express”. Con **npm install --save-dev** se debe instalar @types/node, nodemon, ts-node, y typescript.

Se utilizará **TypeScript** el cual es un superset de JavaScript que agrega tipado estático, interfaces, clases y genéricos, entre otras características. Permite detectar errores en tiempo de compilación y mejora la escalabilidad y legibilidad del código. El siguiente archivo contempla una configuración básica de typescript, se debe ubicar en la raíz del proyecto cliente y llamarse **tsconfig.json**.

```

{
  "compilerOptions": {
    "module": "commonjs",
    "declaration": true,
    "removeComments": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "target": "es2017",
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": "./",
    "incremental": true
  }
}

```

También se deben configurar las opciones de compilación, estas deben estar en la raíz de client, con los siguientes contenidos dentro de un archivo llamado **tsconfig.build.json**.

```

{
  "extends": "./tsconfig.json",
  "exclude": ["node_modules", "test", "dist", "**/*spec.ts"],
  "include": ["src/**/*.ts"]
}

```

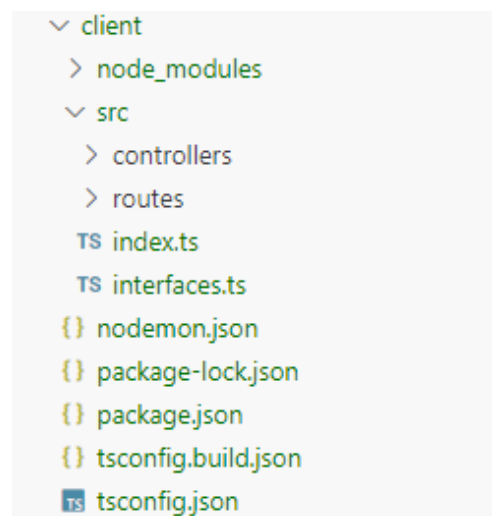
```
}
```

Para correr el proyecto localmente en modo de desarrollo, se utiliza **nodemon**, una herramienta desarrollo para aplicaciones de Node.js que facilita el proceso de desarrollo al monitorizar cambios en los archivos de tu proyecto y reiniciar automáticamente el servidor cada vez que detecta algún cambio. En el archivo **nodemon.json** en la raíz del proyecto cliente deben colocarse estos contenidos.

```
{
  "watch": [
    "src"
  ],
  "ext": "ts,json",
  "ignore": [
    ".git",
    "node_modules/"
  ],
  "exec": "ts-node ./src/index.ts"
}
```

#### Estructura y punto de entrada del sistema cliente

Para la creación del código que sigue, se debe conseguir la siguiente estructura de carpetas y archivos en el proyecto del cliente:



En la carpeta **src** se incluye todo el código necesario para correr el proyecto. Aquí se crea un archivo **index.ts** que es el punto de entrada de la aplicación, un archivo **interfaces.ts** que contiene la definición de tipos (estructura de los objetos de JavaScript, se podrían ver como clases). En las carpetas **routes** y **controllers** se incluirán las rutas para llamadas a funciones y las funcionalidades del sistema en sí, respectivamente. La carpeta **messages** se utiliza para almacenar los mensajes recibidos hasta poder procesarlos exitosamente. En el archivo **index.ts**, se debe incluir el siguiente código. Este importa la librería **express.js** (que proporciona un framework para aplicaciones web y APIs RESTful), y establece el puerto en el que correr la aplicación, así como las rutas disponibles.



```

import express, { Request, Response } from "express";

import router from "./routes";

const app = express();
const port = process.env.PORT || 4000;

app.use(express.json());
app.use(express.urlencoded());

// Root URL is /client
app.use("/client", router);

// If route not found
app.use((req: Request, res: Response) => {
  return res.sendStatus(404);
});

app.listen(port, async () => {
  console.log(`Running on: http://localhost:${port}`);
});

```

#### Estructura de mensajes recibidos

El cliente recibir mensajes que contienen un número, se puede definir el tipo que debe cumplir el mensaje. El siguiente bloque de código contiene la estructura del dato de tipo mensaje.

```

export interface Message {
  body: number;
}

```

#### Recepción de mensajes

Al recibir un mensaje, el cliente debe almacenarlo para su posterior procesamiento. Esto es en caso de que exista un fallo y el mensaje recibido se pierda y no se pueda recuperar sin recibir uno nuevo.

#### messages\_controller.ts

```

import { Request, Response } from "express";

import * as fs from "fs";
import { Message } from "../interfaces";
import { saveMessage } from "./persistence_controller";

export async function sendNewMessage(req: Request, res: Response) {
  //Create random number for message

```

```

let randNumber = Math.random();
//Create message
let newMessage: Message = { body: randNumber };
//Send message to client
let messageReceived = false;
try {
  const result = await fetch("http://localhost:4000/client/receiver", {
    method: "POST",
    body: JSON.stringify(newMessage.body),
  });
  if (result.ok) {
    messageReceived = true;
  } else {
    //Save created message if it was not received
    await saveMessage(newMessage);
  }
} catch (e) {
  //Save created message if it was not received
  await saveMessage(newMessage);
}
return res.sendStatus(200);
}

```

```

type Content = { body: number };

```

```

export async function resendFailedMessages(req: Request, res: Response) {
  const sentMessages: Content[] = [];
  const failedMessages: Content[] = [];
  const files = fs.readdirSync("./messages");

  await Promise.all(
    files.map(async (file) => {
      console.log(file);

      const message: Content = JSON.parse(
        fs.readFileSync(`./messages/${file}`).toString()
      );
      try {

```

```

        const result = await fetch("http://localhost:4000/client/receiver",
    {
        method: "POST",
        body: JSON.stringify(message.body),
    });

    if (result.ok) {
        //Delete stored message if successfully received by client
        fs.unlinkSync(file);
        sentMessages.push(message);
    }
    } catch (error) {
        failedMessages.push(message);
    }
    })
    );

    return res
        .status(200)
        .json({ sentMessages: sentMessages, failedMessages: failedMessages });
}

```

Rutas para que un servidor envíe mensajes

**index.ts**

```

import { Router } from "express";

import { receiveMessage } from "../controllers/messages_controller";

const router = Router();

router.post("/receiver", receiveMessage);

export default router;

```

## Servidor de envío de mensajes

Para la creación sistema de servidor en una carpeta “server” usando el siguiente comando:

**npm init -y**

Este comando generará un archivo **package.json** en el cual se especificarán las dependencias, sus correspondientes versiones y scripts:

```
{
  "name": "guaranteed-delivery-server_hiriart-corales-samaniego",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "tsc --build",
    "start:dev": "nodemon"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^20.4.2",
    "nodemon": "^3.0.1",
    "ts-node": "^10.9.1",
    "typescript": "^5.1.6"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Para instalar las dependencias se debe correr el comando **npm install**, se requiere instalar la dependencia “express”. Con **npm install --save-dev** se debe instalar @types/node, nodemon, ts-node, y typescript.

Para el servidor también se utilizará typescript, por lo que se debe contar con los siguientes contenidos en el archivo tsconfig.json. También se muestran los contenidos del archivo tsconfig.build.json.

### tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "declaration": true,
    "removeComments": true,
    "emitDecoratorMetadata": true,
```

```

    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "target": "es2017",
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": "./",
    "incremental": true
  }
}

```

#### **tsconfig.build.json**

```

{
  "extends": "./tsconfig.json",
  "exclude": ["node_modules", "test", "dist", "**/*spec.ts"],
  "include": ["src/**/*.ts"]
}

```

También se utilizará nodemon en el servidor, con lo siguientes contenidos en el archivo **nodemon.json**.

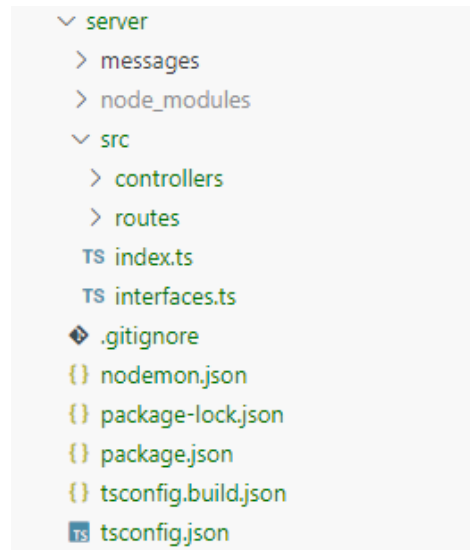
```

{
  "watch": [
    "src"
  ],
  "ext": "ts,json",
  "ignore": [
    ".git",
    "node_modules/"
  ],
  "exec": "ts-node ./src/index.ts"
}

```

#### Estructura y punto de entrada del sistema de servidor

Para la creación del código que sigue, se debe conseguir la siguiente estructura de carpetas y archivos en el proyecto del servidor:



El proyecto de servidor tiene la misma estructura que el cliente, con `src` para las funcionalidades, rutas y punto de entrada del programa. Se incluye también la carpeta `messages` que tiene los mensajes en persistencia hasta confirmar recepción. De igual forma, se requiere un archivo **index.ts** en la raíz del `server` para configurar la aplicación; se usa el código que sigue.

```
import express, { Request, Response } from "express";

import router from "./routes";

const app = express();
const port = process.env.PORT || 3000;

app.use(express.json());
app.use(express.urlencoded());

// Root URL is /server
app.use("/server", router);

// If route not found
app.use((req: Request, res: Response) => {
  return res.sendStatus(404);
});

app.listen(port, async () => {
  console.log(`Running on: http://localhost:${port}`);
});
```

#### Estructura de mensajes

Primero, es necesario generar los mensajes que se enviarán al cliente. Para este ejercicio se creará un número aleatorio como cuerpo del mensaje. Este mensaje primero debe definirse en el archivo **interfaces.ts**, añadiendo el siguiente código. El código contiene el cuerpo del mensaje.

```
export interface Message {
  body: number;
}
```

#### Persistencia de mensajes

Para la persistencia de mensajes, se necesita una funcionalidad que almacene los mensajes generados para mantener persistencia. El código para lograr esto se debe colocar en un archivo llamado **persistance\_controller.ts** en la carpeta controllers, y es el siguiente.

```
import { Message } from "../interfaces";
import { writeFile } from "fs";

export async function saveMessage(message: Message) {
  let currentTime = new Date().toJSON();
  //Save message in file with current time and date
  writeFile(
    `./messages/${currentTime.replaceAll(":", "-")}.json`,
    JSON.stringify(message),
    (error) => {
      if (error) {
        console.log("error saving");
      }
    }
  );
  return;
}
```

#### Envío de mensajes

Existen dos funciones que deben encargarse del envío de mensajes, cada una se llamará de manera separada. Primero, una función que genera un mensaje nuevo con el número aleatorio, esta llama a la función de persistencia para almacenar el nuevo mensaje en caso de problemas. Segundo, una función para reintentar el envío de mensajes almacenados, los mensajes que se logren enviar se eliminan de persistencia. El código para estas funcionalidades se coloca en la carpeta controllers, en el archivo **messages\_controller.ts**, el código es el siguiente.

```
import { Request, Response } from "express";
import * as fs from "fs";
import { Message } from "../interfaces";
import { saveMessage } from "../persistance_controller";

export async function sendNewMessage(req: Request, res: Response) {
  //Create random number for message
```

```

let randNumber = Math.random();
//Create message
let newMessage: Message = { body: randNumber };
//Send message to client
let messageReceived = false;
try {
  const result = await fetch("http://localhost:4000/client/receiver", {
    method: "POST",
    body: JSON.stringify(newMessage.body),
  });
  if (result.ok) {
    messageReceived = true;
  } else {
    //Save created message if it was not received
    await saveMessage(newMessage);
  }
} catch (e) {
  //Save created message if it was not received
  await saveMessage(newMessage);
}
return res.status(200).json(messageReceived);
}

```

```

type Content = { body: number };

```

```

export async function resendFailedMessages(req: Request, res: Response) {
  const sentMessages: Content[] = [];
  const failedMessages: Content[] = [];
  const files = fs.readdirSync("./messages");

  await Promise.all(
    files.map(async (file) => {
      const message: Content = JSON.parse(
        fs.readFileSync(`./messages/${file}`).toString()
      );
      try {
        const result = await fetch("http://localhost:4000/client/receiver",
{
          method: "POST",
          body: JSON.stringify(message.body),

```



```

    });

    if (result.ok) {
        //Delete stored message if successfully received by client
        fs.unlinkSync(file);
        sentMessages.push(message);
    }
} catch (error) {
    failedMessages.push(message);
}
})
);

return res
    .status(200)
    .json({ sentMessages: sentMessages, failedMessages: failedMessages });
}

```

#### Rutas de acceso a funciones

Para llamar a las funciones del sistema que se encargan del envío de mensajes, se necesita llamar a las rutas correctas de la API. Para esto se deben especificar las rutas en un archivo `index.ts` en la carpeta `routes`, misas que son referenciadas por **index.ts** de punto de entrada del sistema para exponer las funcionalidades. Estas rutas indican a que función llamar, y su código es el siguiente.

```

import express from "express";

import {
    sendNewMessage,
    resendFailedMessages,
} from "src/controllers/messages_controller";

const router = express.Router();

router.post("/send/send-new", sendNewMessage);
router.post("/send/resend-failed", resendFailedMessages);

export default router;

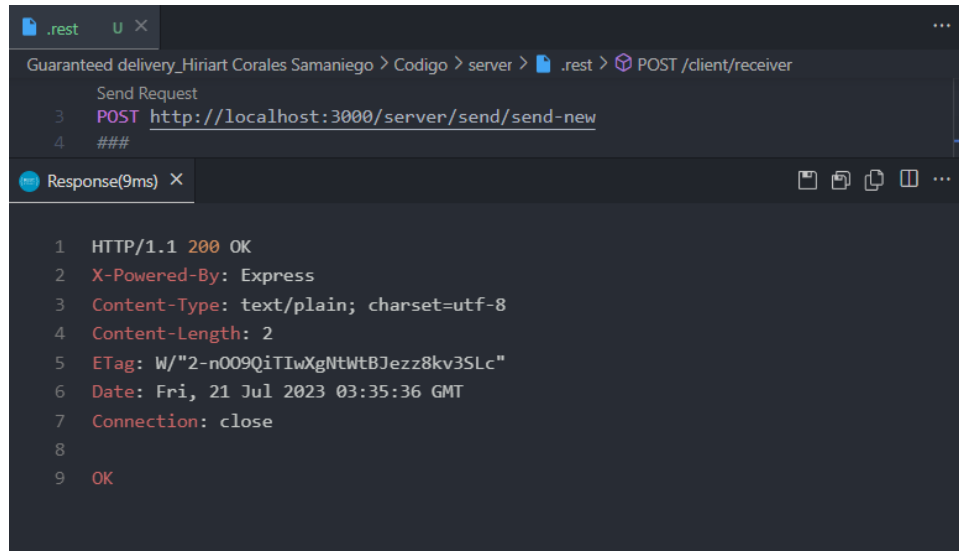
```

## Ejecución

En primer lugar, el servidor debe recibir una petición de método POST al endpoint `/send/send-new`, el cual permite enviar un mensaje al cliente que este puede almacenar como archivo.

Principalmente se envía una petición al servidor y está internamente hará otra petición al cliente para guardar el mensaje de un número aleatorio.

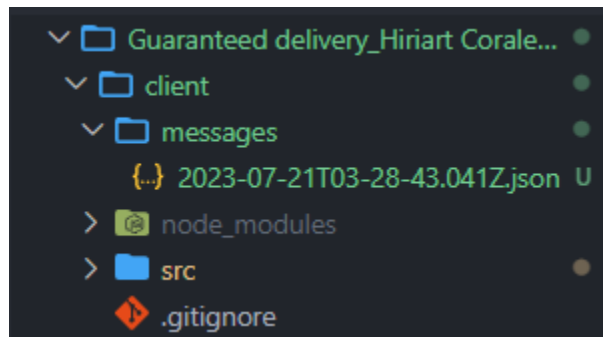
En caso de que el cliente reciba el mensaje, este guarda el archivo del número aleatorio.



The screenshot shows a REST client interface with a tab labeled ".rest". The breadcrumb path is "Guaranteed delivery\_Hiriart Corales Samaniego >Codigo > server > .rest > POST /client/receiver". The "Send Request" section shows a POST request to "http://localhost:3000/server/send/send-new" with a body of "###". The "Response(9ms)" section shows the following details:

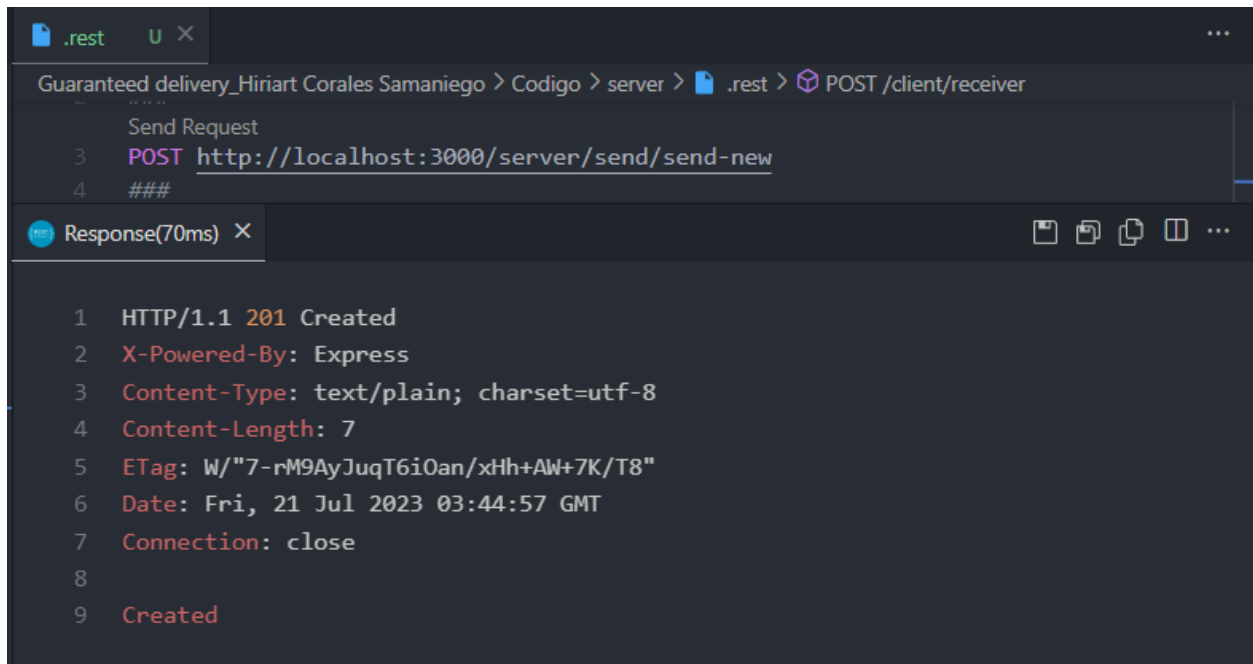
```
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: text/plain; charset=utf-8
4 Content-Length: 2
5 ETag: W/"2-n009QiIwXgNtWtBJezz8kv3SLc"
6 Date: Fri, 21 Jul 2023 03:35:36 GMT
7 Connection: close
8
9 OK
```

Envío de petición



Cliente guardando el mensaje

En caso de que no lo reciba, el servidor emite un mensaje 201 indicando que se creó el archivo del número aleatorio en el lado del servidor para luego ser reenviado al cliente.



The screenshot shows a REST client interface with a dark theme. The top bar indicates the current request is a POST to `/client/receiver`. The request body is `###`. The response status is `201 Created` with a response time of 70ms. The response headers are: `X-Powered-By: Express`, `Content-Type: text/plain; charset=utf-8`, `Content-Length: 7`, `ETag: W/"7-rM9AyJuqT6iOan/xHh+Aw+7K/T8"`, `Date: Fri, 21 Jul 2023 03:44:57 GMT`, and `Connection: close`. The response body is `Created`.

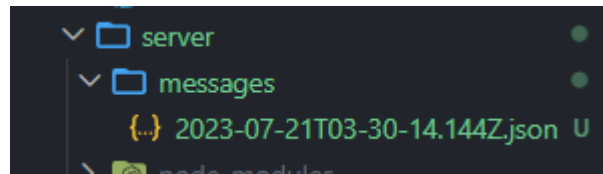
```
POST http://localhost:3000/server/send/send-new

###

HTTP/1.1 201 Created
X-Powered-By: Express
Content-Type: text/plain; charset=utf-8
Content-Length: 7
ETag: W/"7-rM9AyJuqT6iOan/xHh+Aw+7K/T8"
Date: Fri, 21 Jul 2023 03:44:57 GMT
Connection: close

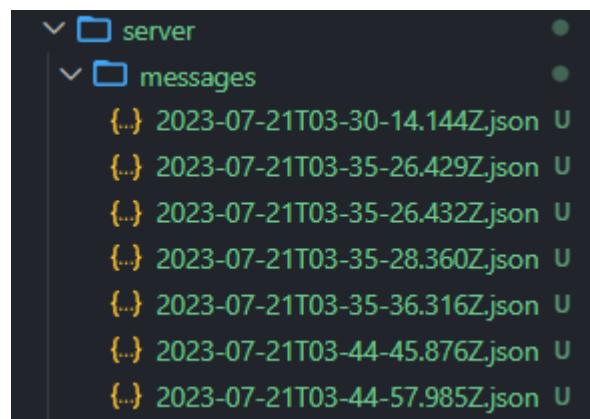
Created
```

Envío de petición con el cliente desconectado (crea un archivo en el servidor para enviar después)

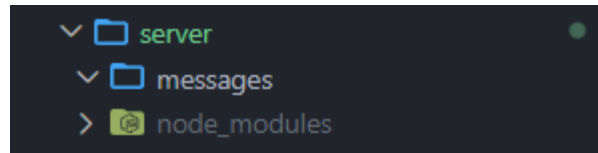


Archivo creado en el servidor

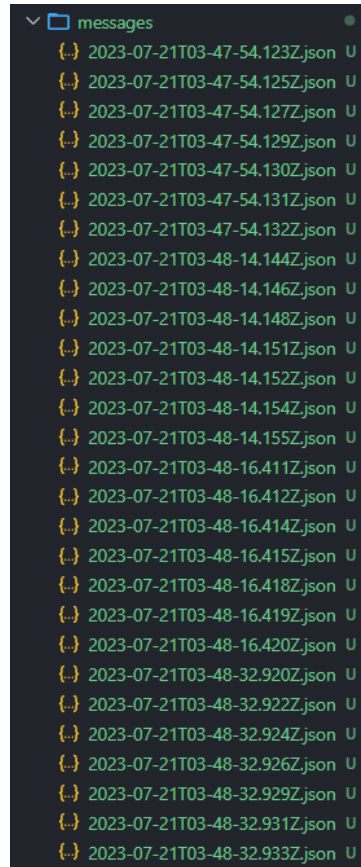
El servidor teniendo N cantidad de archivos que no pudieron ser enviados exitosamente puede recibir una petición para enviar los mensajes uno a uno al cliente. Esto se puede lograr con el endpoint `/send/resent-failed` que realiza la petición, en caso de enviar correctamente elimina los mensajes en el lado del servidor.



Los archivos antes de borrarse

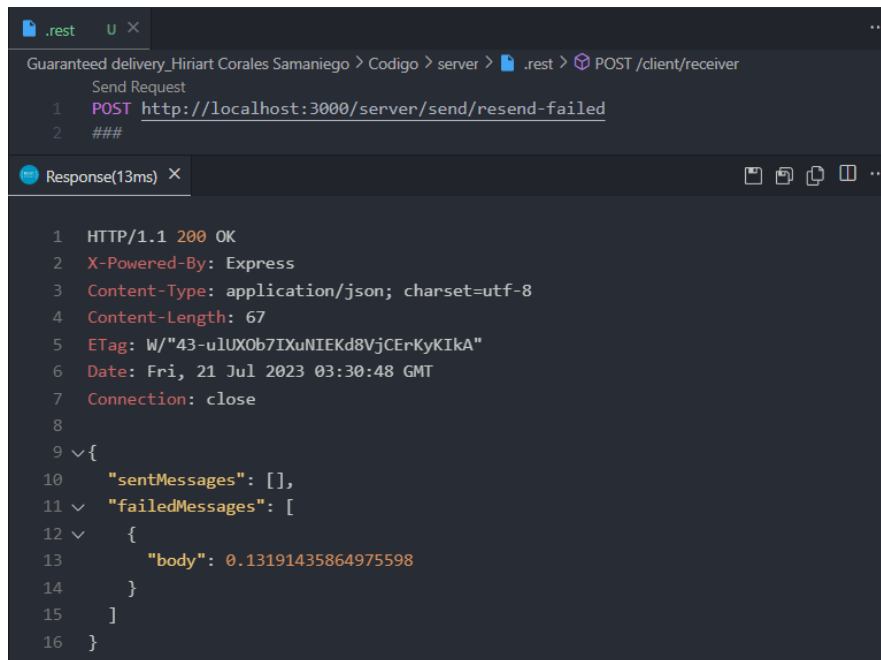


Se eliminan los archivos del servidor



Se crean en el cliente

En caso de que no se puedan enviar correctamente, la respuesta de la petición indicará cual falló.



```
.rest  U X
Guaranteed delivery_Hiriart Corales Samaniego > Codigo > server > .rest > POST /client/receiver
Send Request
1 POST http://localhost:3000/server/send/resend-failed
2 ###

Response(13ms) X
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 67
5 ETag: W/"43-u1UXOb7IXuNIEKd8VjCErKyKIKA"
6 Date: Fri, 21 Jul 2023 03:30:48 GMT
7 Connection: close
8
9 {
10   "sentMessages": [],
11   "failedMessages": [
12     {
13       "body": 0.13191435864975598
14     }
15   ]
16 }
```

En el caso de que falle se muestran los mensajes que no pudieron enviar al cliente

Demostrando que el mensaje puede que no logre enviarse en determinados momentos, pero el patrón guaranteed delivery permite que los mensajes puedan reenviarse en un futuro cuando el cliente esté disponible, asegurando el envío de los datos a toda costa.

## Conclusiones

- Existen patrones para resolver todo tipo de problemas de integración, y en este caso, el problema es el cómo asegurar que los datos se puedan enviar correctamente, aunque el receptor no esté disponible en todo momento.
- Guaranteed Delivery es una solución ideal para el manejo de datos críticos, se debe considerar como parte esencial del flujo de datos para transacciones o comunicaciones críticas.
- Typescript es un lenguaje de programación bastante versátil, simple de leer y capaz de mostrar ejemplos de patrones de integraciones de una manera eficaz y reflejando casos de la vida real.

## Recomendaciones

- Manejar correctamente los códigos de estatus al recibir las respuestas de las peticiones con el fin de que los receptores y los logs puedan describir correctamente las transacciones realizadas.
- Investigar sobre otros patrones similares que garanticen los envíos de datos a los clientes que no sean necesariamente Guaranteed Delivery.
- Para muchos de los patrones de integración se debe hacer un buen manejo del código asíncrono para evitar peticiones sin completar antes de que se envíen las respuestas respectivas a los clientes. Un mal manejo del código asíncrono podría resultar en peticiones incompletas o nunca respondidas y archivos creados erróneamente.