Ingeniería de aplicaciones web

Universidad Nacional de La Plata
14 de Enero de 2024

Documento de diseño

Especificaciones técnicas de la aplicación

Resumen **2**
1. General **2**
1.1. Objetivo 2
1.2. Alcance 2
2. Arquitectura del sistema **2**
2.1. Infraestructura de la aplicación 2
2.2. Tecnologías utilizadas 4
2.3. Modelo de datos 5
2.3.1. Modelo Orientado a Objetos 5
2.3.2. Descripción de atributos 6
2.3.3. Comentarios sobre el modelo 7
3. Decisiones de diseño **8**
3.1. Módulo de creación de tareas (CronJobs) 8
3.2. Módulo de extracción de datos (Crawler) 13
3.3. Módulo de Frontend 15
4. La aplicación web **16**
4.1. Configuración 16
4.2. Acceso 18
5. Documentación básica para la ejecución de pruebas E2E **18**
5.1. LoginLogout.cy.js 19
5.2. CRUDWebsite.cy.js 19
5.3. WebsiteTasksSnapshots.cy.js 19
6. Repositorio del proyecto **20**

Resumen

Este documento tiene como objetivo principal describir las decisiones de diseño tomadas durante el desarrollo del proyecto. Además, busca proporcionar información detallada sobre los casos de prueba que serán utilizados para validar y verificar el correcto funcionamiento de la aplicación web.

1. General

1.1. Objetivo

El propósito del trabajo planteado durante la cursada de la materia **Ingeniería de Aplicaciones Web (2023)** fue el de implementar una solución que permita extraer información de sitios, almacenarla y permitir su búsqueda mediante el uso de microservicios.

La aplicación web resultante será entonces un servicio **crawling** que visite las páginas de los sitios y las páginas conectadas con tags Anchor (<a/>) para extraer la información.

1.2. Alcance

El documento aborda aspectos clave del diseño, incluyendo la arquitectura general, el modelo de datos, descripción de las tecnologías utilizadas, decisiones de diseño específicas y aspectos clave para realizar las pruebas de forma exitosa.

2. Arquitectura del sistema

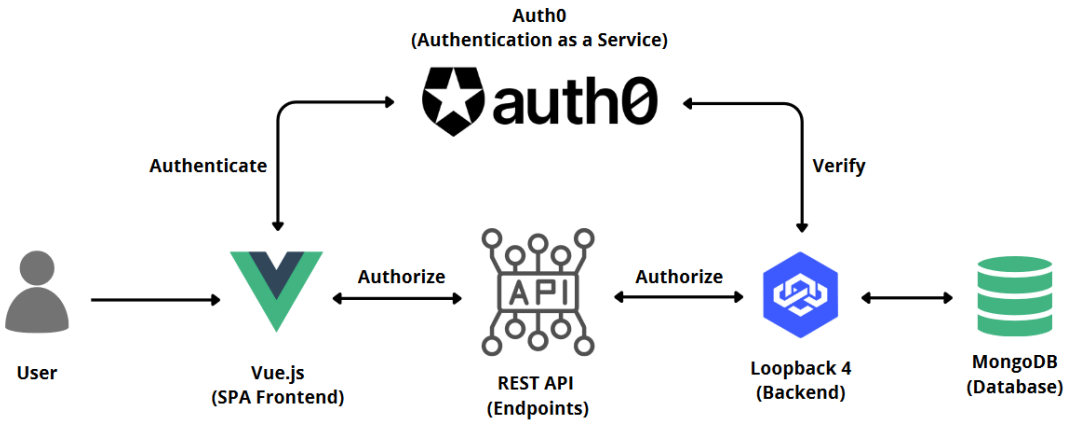
2.1. Infraestructura de la aplicación

1. **Backend (LoopBack 4):**
    a. Se utiliza LoopBack 4 para desarrollar una **API REST** que facilita la comunicación eficiente entre el frontend y la base de datos.
    b. La ventaja principal al usar Loopback es que permite desarrollar los controladores, modelos y repositorios de una forma sencilla mediante el uso del **CLI**. Las operaciones CRUD se gestionan a través de esta API, proporcionando endpoints bien definidos pero al mismo tiempo flexibles de forma tal que sin escribir demasiado código se pueden realizar queries más complejas.

2. **Frontend (Vue.js):**
    a. La interfaz de usuario (UI) se construye como una Single Page Application (SPA) utilizando el framework Vue.js.
    b. La SPA realiza consultas por medio de la **API REST** desarrollada en LoopBack 4 para obtener y enviar datos.

3. **Base de datos (MongoDB):**
    a. La persistencia de datos se gestiona mediante una base de datos MongoDB, conocida por su flexibilidad y capacidad de escalar horizontalmente. En la aplicación desarrollada se hace uso de esta flexibilidad, en particular en el atributo **Documentos** del modelo **Snapshot** como veremos más adelante.

4. **Gestión de usuarios - Autenticación (Auth0):**
    a. Es por el uso de esta herramienta que no tenemos un modelo de **usuario** en nuestro sistema ya que Auth0 facilita la implementación de un sólido sistema de autenticación y autorización mediante funciones altamente probadas y eficaces como un inicio de sesión seguro, gestión de usuarios y control de acceso.



Modelo de infraestructura

2.2. Tecnologías utilizadas

Aunque recién hicimos mención de algunas tecnologías en específico, a continuación se listan todas aquellas herramientas y/o dependencias que fueron utilizadas para la construcción integral del sistema.

Herramienta/Dependencias	Rama	Descripción
LoopBack 4	Backend	Framework de Node.js para la creación de APIs REST
Vue.js	Frontend	Framework para construir nuestra SPA
MongoDB	Backend	Base de datos NoSQL escalable y flexible
Auth0	Ambas	Plataforma de autenticación y autorización
Cypress	Frontend	Nos permite hacer pruebas E2E
Axios	Ambas	Cliente HTTP basado en promesas para realizar solicitudes HTTP
SweetAlert2	Frontend	Usado para crear alertas y notificaciones con mejor interfaz y estilos
Vue3-Json-Viewer	Frontend	Dependencia para visualizar JSON de manera interactiva (Usado para visualizar los documentos capturados)
Clipboard	Frontend	Biblioteca para gestionar operaciones de copiar y pegar en el portapapeles (Usado para copiar los documentos capturados)
OpenAPI	Ambas	Especificación para la creación de APIs REST de manera estandarizada
Bootstrap	Frontend	Usado para diseñar y desarrollar las interfaces web de forma más rápida y sencilla.
is-url-http	Backend	Usado para verificar si una URL tiene el esquema HTTP
Cheerio	Backend	Usado para analizar y manipular datos en documentos HTML con una sintaxis similar a jQuery

2.3. Modelo de datos

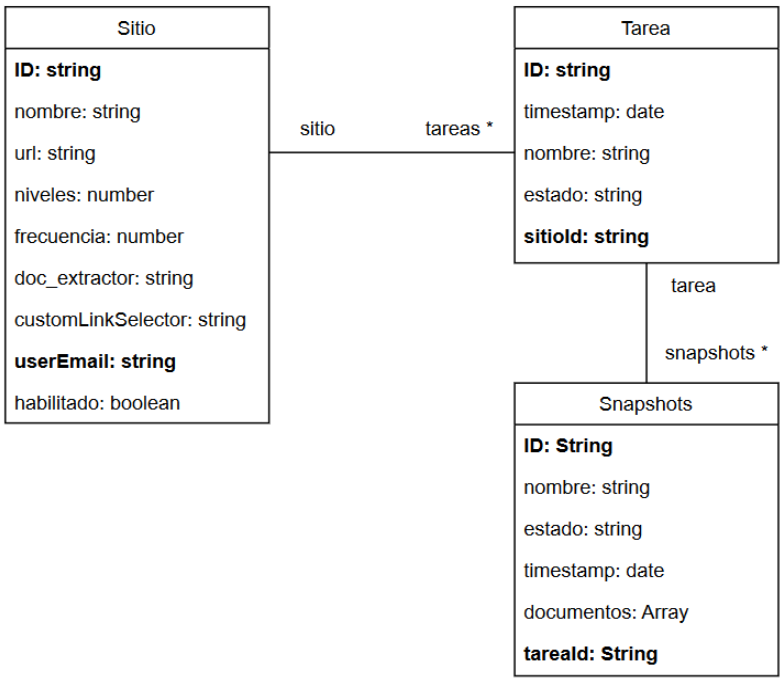
MongoDB fue seleccionado como la opción principal para este proyecto debido a que es una base de datos flexible, escalable y de fácil integración con nuestra aplicación backend, en especial con Loopback.

Además, encuentro la ventaja principal en esta base de datos NoSQL, en la misma finalidad de nuestra aplicación. Nuestro objetivo es capturar documentos de los sitios web y la información que queremos registrar no siempre es igual para todos los sitios registrados… es decir son atributos dinámicos.

Con **MongoDB** podemos guardar la información capturada (JSON) en cualquier estructura en la que fue registrada simplemente como un **Objeto**. Con esta solución tan simple podemos representar los datos sin mayores complicaciones.

De hecho, aprovechando esta misma flexibilidad, la solución propuesta en esta versión es un arreglo de objetos en la clase **Snapshot**. Esto lo veremos a continuación.

2.3.1. Modelo Orientado a Objetos



Esta solución tiene algunas diferencias con respecto al planteo inicial utilizado en los primeros pasos después de la creación del proyecto. De hecho llegar a esta solución requirió de varias iteraciones de revisión.

2.3.2. Descripción de atributos

Entidad Sitio

Atributo	Descripción
ID	Identificador de la entidad (generada por Mongo)
nombre	Nombre del sitio registrado. Ingresado por el usuario.
url	URL del sitio web que será usado como punto de partida para realizar la navegación durante el scrapping.
niveles	Indica la cantidad de niveles de hojas que vamos a inspeccionar.
frecuencia	Indica cada cuanto visitar el sitio, es decir, cada cuanto se deberá ejecutar el algoritmo encargado de realizarlo. (CronJob)
doc_extractor	Snippet javascript que extrae los datos desde las páginas.
customLinkSelector	Indica dónde se realizará la búsqueda de los links. Puede ser útil a la hora de buscar recursivamente en las hojas.
userEmail	Email del usuario. Sirve para identificar qué sitios pertenecen a un usuario.
habilitado	Indica si el sitio registrado está habilitado o no. En caso de que no se encuentre habilitado no se ejecutará el servicio de crawling para este sitio, es decir, no se tendrá en consideración en el algoritmo ni ejecutara CronJobs.
tareas	Arreglo donde se registran las tareas ejecutadas para ese sitio. Representa un CronJob en proceso, finalizado o interrumpido.

Entidad Tarea

Atributo	Descripción
ID	Identificador de la entidad (generada por Mongo)
timestamp	Fecha y hora de registro de la tarea
nombre	Nombre de la tarea. Puede ser editado posteriormente por el usuario pero en principio el nombre estará compuesto de la forma *Tarea-${sitio.nombre}-${totalTareas + 1}*
estado	Indica en qué estado se encuentra la tarea (En proceso, Finalizado, Interrumpido)
sitioId	Indica a qué sitio pertenece la tarea
snapshots	Arreglo que agrupa todos los snapshots (capturas) obtenidas durante la ejecución de la tarea

Entidad Snapshot

Atributo	Descripción
ID	Identificador de la entidad (generada por Mongo)
nombre	Nombre del snapshot/captura. De momento no puede ser editado y el formato de registro es: Snapshot-${tareaId}-${new Date().toISOString()
estado	Indica en qué estado se encuentra el snapshot. Por como está planteado el algoritmo este siempre será *Finalizado*.
timestamp	Indica la fecha y hora de registro del snapshot
documentos	Arreglo de objetos donde se registra la información obtenida de los sitios (clasificados por nivel).
tareaId	Indica a qué tarea pertenece el snapshot

2.3.3. Comentarios sobre el modelo

- El modelo no contempla la implementación de una entidad **Usuario** ya que se delega esa responsabilidad a **Auth0**, específicamente mediante el uso de **Social Login** con el que podremos iniciar sesión en la aplicación usando nuestra cuenta de Google.
- En un sitio se identifica al usuario mediante el registro de su **email** como atributo del mismo sin embargo queda pendiente la modificación de email por su **googleID**. El modelo de sitio fue el primero en realizarse y todavía no tenía un conocimiento más extenso sobre cómo funcionaba exactamente el sistema de Social Login.
  Como cada email es único esta solución funciona aunque podría considerarse como deuda técnica.
- Un sitio debe estar si o si habilitado para que se realice la extracción de los datos (ejecución del algoritmo crawler).
- Los documentos originalmente iban a ser una entidad propia, sin embargo se hace uso de la flexibilidad de **MongoDB** y se simplifica el modelo evitando agregar nuevas relaciones y simplemente agregando el atributo **documentos** como parte del mismo.

3. Decisiones de diseño

En esta sección vamos a profundizar acerca de las decisiones tomadas para el desarrollo de la aplicación web.

Cada uno de estos *módulos* detallados a continuación son parte fundamental del proyecto y fueron, en consecuencia, las secciones en donde se dedicó más tiempo por sobre todas las demás.

3.1. Módulo de creación de tareas (CronJobs)

Es una parte fundamental de toda nuestra aplicación. Este módulo contempla la creación de un **CronJob** que será el encargado de verificar cada cierto periodo de tiempo si hay nuevos sitios registrados sobre los que se deberá ejecutar el algoritmo de crawling.

Los CronJobs son tareas programadas que nos permiten ejecutar un script de manera automática en un momento determinado (fecha, hora, o en base a una frecuencia determinada). En nuestra aplicación tendremos un **CronJob principal** (MyCronJob.ts) que será el responsable de consultar por los sitios registrados en la base de datos cada **10 segundos**. Además este se registra en *application.ts* permitiendo que sea visible en la aplicación y pueda ejecutarse.

Sin embargo hay algunas consideraciones que se tomaron en cuenta para poder cumplir con la tarea solicitada en la especificación de los requerimientos del proyecto.

Si observamos el código, podemos notar que no solo contamos con la definición de la tarea *Cron* a ejecutar representada en el constructor de la clase MyCronJob sino que contamos con varios métodos auxiliares y una cantidad importante de comentarios dada la extensión del mismo.

Podemos ir desglosando el algoritmo para explicar algunos puntos clave:

1) Los sitios a tomar en cuenta para la ejecución deben estar habilitados:
   Esto significa que debemos verificar antes de cualquier otra operación si el sitio obtenido cambió de estado entre la última consulta la BD y la nueva iteración de la tarea programada.

```
// Busco los sitios en la BD (sitioRepository)
 const sitios: Sitio[] = await sitioRepository.find();
 sitios.forEach(async sitio => {
   if (!sitio.habilitado) {
     // Busco la tarea correspondiente y la detengo si existe
     const tareaEnEjecucion = this.runningJobs[sitio.getId()];
     if (tareaEnEjecucion) {
       this.completionStatus = 'Interrumpido';
       tareaEnEjecucion.stop();
       delete this.runningJobs[sitio.getId()];
     }
     return;
   }
```

   En primer lugar vemos que se obtienen todos los sitios de la BD y se empieza la iteracion principal en donde se realizarán todas las acciones necesarias para la extracción de los datos.

   Inmediatamente después de entrar en el loop, preguntamos si el sitio está habilitado o no (el usuario lo deshabilita desde la aplicación en Vue) entonces debemos buscar la tarea correspondiente para ese sitio en una variable global que contiene un arreglo con todos los Jobs registrados y la detenemos para evitar que se siga ejecutando la extracción. Posteriormente lo eliminamos de este arreglo indicando que la tarea ya no está más en ejecución.

2) Verificación de condición interna y estado de la tarea

Si la tarea se encuentra registrada entonces significa que está en ejecución. En ese caso se ejecuta una función auxiliar **verificarJobInterno** que verifica si el Job para ese sitio especifico cumple o no alguna condición que lo obliga a terminar la ejecución.

```
if (this.runningJobs[sitio.getId()]) {
   // El JOB ya se encuentra en ejecución
   await this.verificarJobInterno(sitio, this.runningJobs[sitio.getId()]);
}
```

En nuestro caso únicamente se verifica que no haya pasado más de media hora desde el inicio de la ejecución pero podrían agregarse más casos de prueba para evaluar la continuidad de la tarea.

Los sitios en nuestra aplicación cuando son creados toman como unidad de tiempo únicamente los minutos. Es decir, no fue contemplado en la idea original la frecuencia preestablecida *Diaria, Semanal, Mensual* sino que el usuario puede ingresar la cantidad mediante un input, por lo que hay ciertas cuestiones que debieron ser tenidas en cuenta al plantear esta solución e incluso puede ser que se necesiten más verificaciones:

- El registro de la unidad de tiempo (segundos, minutos, horas) no puede ser mayor a 60 de su respectiva unidad en la definición del cronJob.
- La verificación de tiempo (30 minutos) obliga a que existan por lo menos 2 tareas registradas en la base de datos por cada iteración del sitio.
    - Esto está pensado para distribuir mejor las capturas (snapshots) en aquellos sitios con frecuencias muy bajas.

      Supongamos que el usuario quiere obtener capturas muy rápidamente y configura la frecuencia a 1 minuto. Durante el periodo que dura una tarea (máximo de una hora) se obtendrían 60 capturas y dependiendo del algoritmo establecido podemos estar hablando de una gran cantidad de documentos, tantos que sería ineficiente poder navegar y buscar datos en ellos.
    - De esta forma prácticamente dividimos a la mitad la cantidad de datos por tarea y se realiza un mejor balanceo de los datos sin perder la información que quiere el usuario ya que el registro de las tareas no es de interés del mismo sino que éste está interesado en los documentos obtenidos.

- Aunque esta implementación funciona con frecuencias muy bajas, es verdad que falta una verificación para frecuencias iguales o mayores al tiempo máximo establecido (30 minutos). Podría incorporarse a futuro una simple solución que consista en consultar durante la verificación del job interno si el sitio tiene configurada una frecuencia mayor a ese valor (una sentencia if para no ejecutar el bloque de código). En este caso no será necesario hacer un balanceo ya que serán tareas con información muy acotada.

3) El Job no se encuentra en ejecución (no está registrado)
   Si evitamos entrar en la consulta if anterior, se procede a la creación del **CronJob** (tarea). Se setea un nombre por defecto compuesto por el string `Tarea-${sitio.nombre}-{totalTareas + 1}.`
   Posteriormente se llama a la función *crearNuevaTarea* en donde queda registrada en la base de datos con el estado 'En proceso'. La configuración de este estado es importante ya que en base a este se permiten realizar acciones en la aplicación Vue (La eliminación y edición de una tarea se realiza únicamente si la misma se encuentra en estado **Interrumpido** o **Finalizado**)

4) Creación del CronJob Interno (Tareas por cada sitio)
   Se realiza la configuración del nombre de la tarea (la misma que se definió anteriormente) y del **cronTime** obtenida mediante el atributo **frecuencia** del sitio que se está procesando

   Se define la función que deberá realizarse en la frecuencia configurada en donde se realiza una llamada al método **runProcess** encargado de realizar el proceso de extracción mediante el uso de otro algoritmo recursivo que visita los sitios.
   A este método debemos el sitio, la tarea creada, un conjunto donde se guardan los urls visitados (para evitar volver a entrar en ellos recursivamente por nivel) y la url del sitio.

5) Configuración de estado al completar la tarea
   Como dijimos, el estado de la tarea es importante para definir acciones futuras, sobre todo aquellas que deberá realizar el usuario en la aplicación Vue.
   Es por esto que hacemos uso de la propiedad **onComplete** provista por los CronJobs. Esta recibe una función anónima que ejecuta un update de la tarea con el nuevo estado. El estado podrá ser **Finalizado** o **Interrumpido**.

El primero ocurre cuando se finaliza la ejecución de una tarea de forma exitosa (el tiempo se cumplió).

El segundo caso ocurre cuando sucede algún evento que fuerza la detención de la tarea de forma anómala:

- Un usuario realiza una deshabilitación del sitio cuando este estaba aun ejecutando una tarea. Aquella tarea que se encontraba aún en ejecución, en la próxima iteración de búsqueda de sitios entrara en la verificación que vimos en el punto 1 y pondrá su estado en **Interrumpido**
- Si ocurre un evento externo, por ejemplo un error o reinicio del servidor, las tareas quedarían en el estado 'En proceso' de forma indefinida a menos que se realice una corrección del estado. Para ello se define una función **actualizarEstadoEnTareasInterrumpidas** que se ejecuta una sola vez al iniciar el cronJob principal. Este método busca aquellos sitios en la BD que quedaron con el estado 'En proceso' y los actualiza al nuevo estado **Interrumpido**.

6) Insertar Job en arreglo de registro

Cuando la tarea está en proceso se debe realizar el guardado en un arreglo definido en una variable de instancia para tener constancia de la ejecución de la misma. Esto es importante ya que será de utilidad para las verificaciones anteriores y si se debe registrar un nuevo Job.

```
// Guardo el nuevo job creado en el arreglo previamente definido para
// tener el registro y posibilidad de detenerlos posteriormente
this.runningJobs[sitio.getId()] = sitioJob;
```

3.2. Módulo de extracción de datos (Crawler)

Este módulo contempla la funcionalidad principal de la aplicación, la extracción de los datos mediante un servicio crawling.

El método **runProcess** es el que llamamos desde el **CronJob** que definimos anteriormente. Dentro de él, definimos un objeto que representa un arreglo clave valor donde guardamos el nivel de donde estamos realizando la extracción de los datos y los resultados obtenidos (que a su vez es otro arreglo con objetos clave valor).

Esto se abstrae en la interfaz **Nivel** que a su vez representan a los **documentos**, es decir, a los datos obtenidos que son resultados del scrapping y que pertenecen a un **Snapshot** (captura).

Método runProcess

```
export async function runProcess(sitio: Sitio, tarea: Tarea, urlsHttp: Set<String>,
  baseUrl: string) {

  const resultadosPorNivel: {[key: string]: {nivel: number, resultados: {[key: string]: object}[]}} = {};

  // Inicia el proceso con el sitio inicial
  await processWebsite(sitio, tarea, urlsHttp, baseUrl, 1, resultadosPorNivel);

  // Crea el Snapshot al finalizar todas las llamadas recursivas
  const documentos: Nivel[] = Object.values(resultadosPorNivel).map(entry => entry);
  const nuevoSnapshot = await crearNuevoSnapshot(tarea.getId(), documentos);
  await guardarSnapshot(nuevoSnapshot);
}
```

Este método, como se puede observar, llama a la función **processWebsite**. Esta es la función recursiva que realiza concretamente la captura de los datos.

Dada la recursividad, y ya que es la función principal de la aplicación, este algoritmo está comentado de forma extensa en el código por lo que no se va a proceder con la explicación detallada paso a paso, sin embargo hay algunos puntos a destacar:

1) Obtención del documento y manipulación de elementos
   El algoritmo hace uso de la librería **Cheerio** para cargar el documento html del sitio en cuestión y nos ofrece la capacidad de manipular los elementos mediante sentencias sencillas. Esta herramienta es fundamental para realizar la recolección de los datos.

2) Selector de link personalizado

En la definición de los atributos del sitio habíamos mencionado la propiedad **customLinkSelector**. Hacemos uso de esta propiedad en la función. Su objetivo es poder definir con mayor precisión de qué sección en particular vamos a querer realizar la obtención de los links.

Supongamos que tengo un sitio web a analizar y quiero obtener únicamente los links que se encuentren en una determinada sección del documento HTML (por ejemplo los links dentro del tag section → sitio.customLinkSelector = section a)

De esta forma nos aseguramos de obtener datos de forma más específica y evitando la visita de urls que no son de nuestro interés ya que no tienen sentido dentro de la lógica de nuestro scrapping (de acuerdo a la definición de nuestro algoritmo). Es un atributo opcional, por lo que si el sitio no lo tiene registrado se buscará por defecto en todos los links como puede verse en el código a continuación.

```
// Agregar el resultado al array correspondiente al nivel actual
resultadosPorNivel[nivelKey].resultados.push(result);

if (niveles <= sitio.niveles) {
  // Obtenemos todos los enlaces en la pagina
  // Vemos si el selector personalizado este seteado, sino usamos 'a' default
  const linksSelector = sitio.customLinkSelector ? sitio.customLinkSelector : 'a';
  const links = $(linksSelector);
  const linkPromises: Promise<void>[] = [];

  links.each((index, link) => {
    // const procesarLinks = async () => {
    const href = $(link).attr('href');

    if (href) {
      // Resolver la URL relativa a la URL base del sitio
      const resolvedUrl = new URL(href, sitio.url).toString();
```

3) Acumulacion de resultados (datos) por nivel

El algoritmo está desarrollado de forma tal que todos los documentos capturados terminan siendo agrupados por nivel. De esta forma el usuario podrá visualizar mejor cómo es que se recopilaron los datos y poder buscarlos mejor

```
// Acumulamos los resultados para este nivel
const nivelKey = `Nivel ${niveles}`;
if (!resultadosPorNivel[nivelKey]) {
  resultadosPorNivel[nivelKey] = {
    nivel: niveles,
    resultados: [],
  };
}

// Agregar el resultado al array correspondiente al nivel actual
resultadosPorNivel[nivelKey].resultados.push(result);
```

3.3. Módulo de Frontend

Aunque nuestra aplicación de Vue no requirió tanta énfasis en la lógica de desarrollo como sucede con los métodos en el backend, hay algunas consideraciones a destacar de forma general en cuanto al diseño o implementación de algunas soluciones.

1) Views como contenedores de lógica

   Se buscó como objetivo principal de las views (InicioView, JobsView, SearchView, SnapshotsView) que sean componentes que contengan toda (o la mayor cantidad) de lógica posible. Es decir, definir los métodos encargados de hacer fetch a los endpoints y las variables de estado que se usarán para renderizar nuevos componentes dependiendo la interacción del usuario o la respuesta del servidor a una petición.

   Esto es con la idea de mantener los componentes propiamente dichos, aquellos que son importados en las Views, lo más cercano posible a componentes que no manejen estados… aunque al final la mayoría terminó definiendo cierta lógica la misma fue reducida de manera significativa al delegar dichos métodos en estos contenedores.

2) Separación de componentes de acuerdo a funcionalidad

   Se agrupan los componentes en carpetas correspondientes a la misma vista (carpeta inicio, carpeta jobs, carpeta search, etc) y se dividen las funcionalidades que deberían tener la vista en varios componentes que al ser importados en el componente contenedor van a renderizar la totalidad de la vista.

   Esto nos sirve principalmente para tener componentes modularidad, más fáciles de desarrollar, entender y debuggear. Por ejemplo:

   - SnapshotsView
     - SnapshotsHead
     - SnapshotsList
     - SnapshotsSearch

3) Una sola vista, pocas redirecciones

   El desarrollo de la aplicación Vue fue pensado para tener la menor cantidad de redirecciones posible. Es decir, siempre que se pudo se hizo un componente que renderice condicionalmente el html o que aparezca por sobre los demás en estilo *Modal* (facilitado por la librería de estilos bootstrap).

Este último caso se usó siempre que se solicitó la creación de una entidad (sitio) y/o la edición de la misma (sitio, tarea).

La idea es que el usuario pueda realizar todas las acciones necesarias relacionadas con el contenedor (Views) que está visualizando sin necesidad de estar haciendo redirecciones por cada acción realizada. Todo debería estar disponible en una única vista que renderice (aunque sea oculto) los elementos necesarios para poder realizar la operación requerida.

Además, esto soluciona un problema que ocurrió puntualmente con **Pinia** y el guardado/consulta del estado global almacenado (token, información del usuario). Y es que resultó imposible guardar el estado del token obtenido desde **Auth0** haciendo uso del método provisto por la cátedra durante las primeras etapas del desarrollo y desde muy temprano se decidió optar por consultar, siempre que sea necesario, el token de usuario para setearlo de la siguiente forma:

```
// Configurando token
const configureToken = async () => {
  const token = await getAccessTokenSilently();
  client.defaults.headers['authorization'] = `Bearer ${token}`
}
```

Al tener prácticamente todos los componentes en una misma vista y sin necesidad de estar haciendo redirecciones inoportunas, podemos reducir al mínimo la necesidad de configurar el token de esta forma. De hecho teniendo en cuenta todos los componentes desarrollados únicamente fue obligatorio hacer esta configuración un máximo de 4 veces.

4. La aplicación web

4.1. Configuración

Antes de poder levantar la aplicación y acceder al sistema debemos tener en cuenta algunos archivos de configuración y variables de entorno que no se incluyeron en el repositorio remoto. Sin embargo a continuación se detalla el paso a paso para poder configurar la aplicación correctamente y así eventualmente poder hacer uso del sistema.

1) Clonar el repositorio:

   Antes que nada, debemos clonar el repositorio desde GitHub

   - git clone https://github.com/GutierrezS-JC/IAW-2023
   - cd IAW-2023

2) Configuración del frontend:

   Vamos a realizar la configuración para el frontend:

   - cd frontend
   - code . → Abrimos el editor de código
   - touch .env → Creamos el archivo .env. También puede hacerse de forma manual.

   Dentro del archivo **.env** debemos agregar los siguientes datos:

```
VITE_API_SERVER_URL=http://localhost:5173
VITE_AUTH0_DOMAIN=your-auth0-domain
VITE_AUTH0_CLIENT_ID=your-auth0-client-id
VITE_AUTH0_CALLBACK_URL=http://localhost:5173/
VITE_AUTH0_AUDIENCE=https://your-auth0-audience/api/v2/
```

   También debemos agregar el siguiente archivo **cypress.env.json**:

```
{
  “GOOGLE_USERNAME”: "your-google-username"
  “GOOGLE_PASSWORD”: "your-google-password"
  “GOOGLE_NAME”: ""
}
```

   Aunque con estos archivos debería bastar para realizar la configuración, por las dudas si llega a surgir algún error (en especial con el uso de cypress) se recomienda agregar el siguiente archivo **auth_config.json** dentro de la carpeta **src:**

```
{
  “domain”: "your-auth0-domain"
  “clientId”: "your-auth0-client-id"
  “audience”: "https://your-auth0-audience/api/v2/"
}
```

3) Configuración del backend

   En el backend únicamente debe crearse el archivo .env dentro de la carpeta raíz. Este archivo se usará para setear las variables de entorno que serán usadas, en este caso, para realizar la conexión a la base de datos **MongoDB.**

   Se deberá crear un usuario en https://www.mongodb.com/es/atlas y completar el archivo con sus datos.

```
SECRET_KEY: "your-secret-key"
USER: "your-user"
```

4) Instalación de dependencias

   Tanto para el frontend como para el backend se deberá ejecutar el comando de instalación de dependencias.

   - npm install

4.2. Acceso

Una vez termine de completar los pasos de configuración deberá levantar las respectivas aplicaciones:

Frontend: npm run dev
Backend: npm start

Podrá acceder a la aplicación **Vue** ingresando en: http://localhost:5173/
Podra visualizar las API Docs ingresando en: http://127.0.0.1:3000/explorer/

5. Documentación básica para la ejecución de pruebas E2E

Como mencionamos antes, para poder utilizar cypress se debe realizar una breve configuración. Una vez finalizada debería funcionar sin inconvenientes aunque son algunos detalles a tener en cuenta.

En primer lugar, para abrir **cypress** deberá usar el comando **npx cypress open** ya que no fue configurado el script *npm run cypress:open*.

Por otro lado, los tests únicamente funcionan sin inconvenientes en el navegador **Edge** que se abre desde la interfaz de Cypress.

Tanto en Chrome como en Firefox los tests presentan algunas dificultades, no en cuanto a la correctitud de los mismos sino más bien en otras cuestiones ajenas al trabajo que, por cuestiones de tiempo, no pudieron ser indagadas más en profundidad.

A continuación se detalla brevemente cual es el propósito de cada test (Se recomienda eliminar los sitios previamente registrados, si es que los tiene, para asegurar el correcto funcionamiento de estas pruebas)

5.1. LoginLogout.cy.js

Este test tiene como objetivo probar que tanto el Inicio de Sesión como el cierre de la misma funcionen correctamente. Es un test sencillo pero que también verifica que efectivamente tanto la aplicación frontend como el servidor funcionan (junto con la integración de Auth0).

Como un detalle, este test originalmente estaba configurado para ser probado con un email de prueba que no fue provisto en este documento, por lo que el test de verificación de usuario que se encuentra justo antes del cierre de sesión seguramente de un resultado fallido ya que se espera el string **Hola, Ange Postecoglou** siendo este el nombre de la cuenta de gmail usada para las pruebas.

Si quiere que todos los *Assert* sean verdaderos puede cambiar el nombre del test en el archivo por el de la cuenta de gmail que esté utilizando para realizar las pruebas.

5.2. CRUDWebsite.cy.js

Este test tiene como objetivo probar la creación, visualización, deshabilitación, edición y eliminación de un sitio (Operaciones CRUD).

5.3. WebsiteTasksSnapshots.cy.js

Este test tiene como objetivo probar la creación de sitios, tareas, snapshots y búsqueda en los documentos. Además se realiza un segundo caso de prueba para testear la edición y eliminación de una tarea.

El test está configurado para crear dos sitios con una frecuencia de **1 minuto**, es decir, en algún punto del test se realizará un **delay** de un poco más del tiempo establecido para darle tiempo al sistema a extraer la informacion del sitio http://books.toscrape.com/

Una vez pasa el minuto establecido, se detiene la ejecución de los sitios creados (deshabilitación) y se revisan las tareas. Cada sitio debería haber al menos 1 tarea registrada.

Se ingresa a la vista de tareas, se hace click sobre la última registrada para ver los snapshots (capturas) y se procede a ver los detalles del mismo junto con los documentos extraídos.

Posteriormente se realiza una búsqueda de un libro **Light in the attic** y el test comprueba que se ha devuelto al menos 1 resultado. Por último se vuelve al inicio y se cierra la sesión.

En el **segundo test** se busca probar la edición y eliminación de las tareas. Finalmente se eliminan los sitios registrados y se cierra la sesión.

6. Repositorio del proyecto

https://github.com/GutierrezS-JC/IAW-2023