UNTREF UNIVERSIDAD NACIONAL DE TRES DE FEBRERO

Ingeniería en Computación

Diseño y Arquitectura de Sistemas de Computación

Trabajo Práctico N°5:
GotenJS

Alumnos:

Capece, Cintia - capece35204@estudiantes.untref.edu.ar
Araneda, Alejandro – araneda46618@estudiantes.untref.edu.ar

Práctica entregada:
2do Cuatrimestre de 2021
Viernes, 19 de noviembre.

Docentes:

Dr. Fernando G. Tinetti
Ing. Jorge C. Fossati

Introducción

De acuerdo con los diccionarios de la lengua inglesa, el término *boilerplate*, de aplicación en ámbito legal como en el del *software*, proviene etimológicamente de la industria de la prensa escrita y se refiere a los artículos y anuncios que las agencias nacionales de noticias distribuían a los periódicos locales sindicalizados para ser publicados sin cambio.

En el terreno de la programación, por *boilerplate* se identifica al código que aún cumpliendo con una funcionalidad, se reitera en distintos desarrollos sin más cambios que los identificadores correspondiente a la unidad lógica, de manera que a la vista del programador no agrega en la inspección del código fuente más información sobre el algoritmo diseñado. Ejemplos comunes son las "guardas" en *headers* del lenguaje C o los métodos *setter* y *getters* en el lenguaje Java.

Si bien en los lenguajes derivados o más modernos, muchas veces se intenta suprimir estos fragmentos de código mediante el uso de palabras reservadas (por ejemplo, `get` y `set` en C# para generar *accessors*) las soluciones más implementadas incluyen "macros", anotaciones y decoradores, las que pueden ser englobadas bajo el concepto de metaprogramación.

Los antecedentes de la metaprogramación se remontan a la distinción entre las arquitecturas Harvard y Von Newman: uso de espacio de memoria separada o integrada para el almacenamiento temporal de datos e instrucciones. Incluso uno de los primeros lenguajes de programación como LISP, bajo el paradigma funcional y declarativo, incluye en su diseño la posibilidad de pasar de la representación simbólica de su código a una expresión ejecutable del lenguaje: el desarrollo de la función `eval` permitió la creación de uno de los primeros intérpretes de código así como el acuñar el término "REPL" para un entorno de ejecución dinámico.

Es cierto que la comunidad que administra y mantiene el lenguaje Python siempre ha sido refractaria a los niveles de abstracción que ofrece el paradigma declarativo, sin embargo, el señalamiento que hace Ian Bicking (creador de `pip` y `virtualenv`, los componentes de la administración de paquetes de Python) sobre los desafíos que impone la metaprogramación, parece pertinente: la reducción de código a través de "macros" agrega un nivel de complejidad aumentando la curva de aprendizaje para el programador, impactando muchas veces en el mantenimiento.

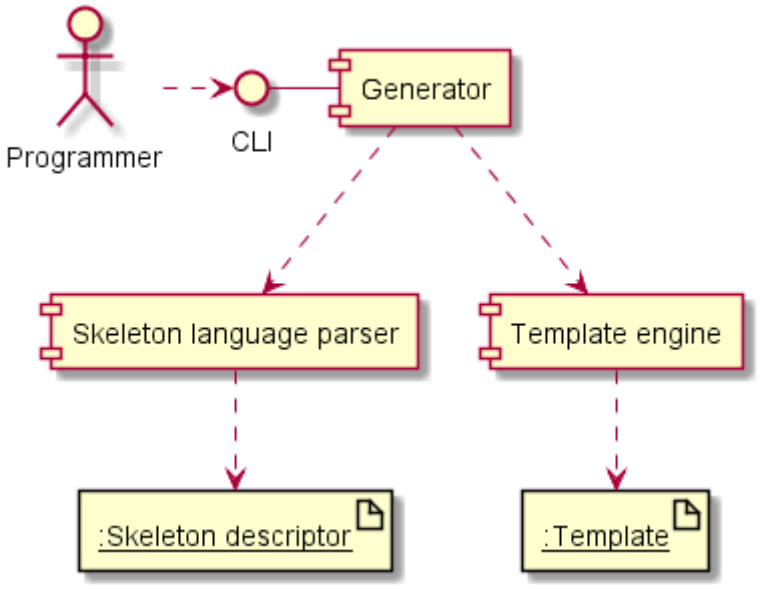

***Figura 1**: componentes de un generador de esqueletos de proyectos.*

Distintas tecnologías han extendido la metaprogramación con generadores de esqueletos de proyectos y los han incorporado a sus interfaces por línea de comandos (CLI, en inglés). Aquí, el término esqueleto para referirnos al aspecto estático de la arquitectura, como la organización de directorios y archivos, resulta más pertinente que el de "plantilla", a nuestro entender. Por ejemplo: `ng generate` en Angular y `dotnet new` en .NET.

En la Figura 1 exponemos lo que son los componentes habituales de los generadores de proyectos a partir de esqueletos incluyen además de la interfaz por línea de comandos, un intérprete de un lenguaje descriptor de esqueletos que permita la extensión de los modelos existentes y un motor de "población" de plantillas, el que en general permite algo más que el simple reemplazo de identificadores al adoptar algún lenguaje de *scripting*.

Arquitectura *de* GotenJS

GotenJS es un *framework* que se estructura en módulos que interactúan dependiendo de la tecnología a utilizar como se puede visualizar en la Figura 2.

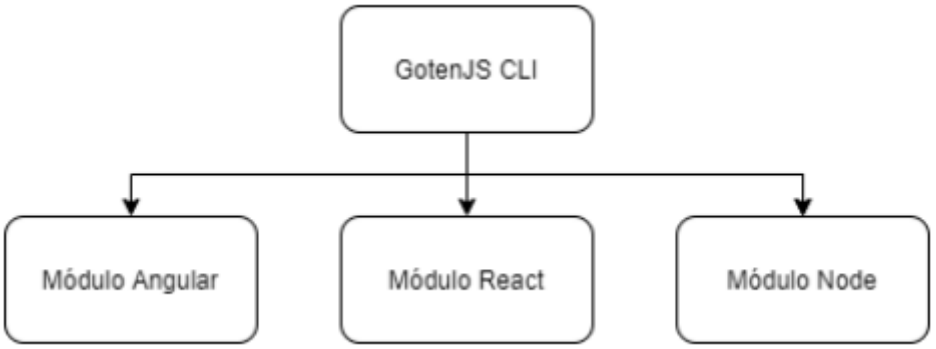

***Figura 2**: Estructura modular de GotenJS*

Módulo GotenJS CLI

El módulo CLI de GotenJS genera proyectos con una estructura predefinida. Estos proyectos son aplicaciones web conectadas a una base de datos (MySQL, MariaDB o MongoDB), un *backend* (en NodeJS, con Express), y un *frontend* (React, Angular).

Cada comando realiza una acción coherente con el nombre del propio comando y la tecnología especificada cómo parámetro. Se detallan a continuación las acciones que soporta el módulo GotenCLI:

- **Creación de Proyectos Base:** GotenJS provee el comando **`goten new`** para poder crear proyectos base dependiendo de la tecnología especificada.
- **Alta-Baja-Modificación** El concepto de "Alta, Baja y Modificación", puede ser referenciado por las siglas ABM, obviando la operación de "obtener" siendo utilizadas por las operaciones CRUD (crear, leer, actualizar y borrar). De esta forma, se lleva a cabo la persistencia en las bases de datos. Para esto, se provee el comando **`goten abm`**.
- **Autenticación:** Se provee el comando **`goten auth`** para agregar la autenticación con JWT y permisos basados en roles/atributos del lado del *backend* y del lado del *frontend* la pantalla correspondiente al *login* para ya ingresar el *username* y el *password* generados desde el *backend*.

- **Ícono de página:** Se provee el comando **`goten favicon`** para asignar un ícono en el *browser* de la Single Application (SPA) generada en React y/o Angular.

Descripción estática

La descripción de la estructura estática utilizada está compuesta por el siguiente directorio de archivos:

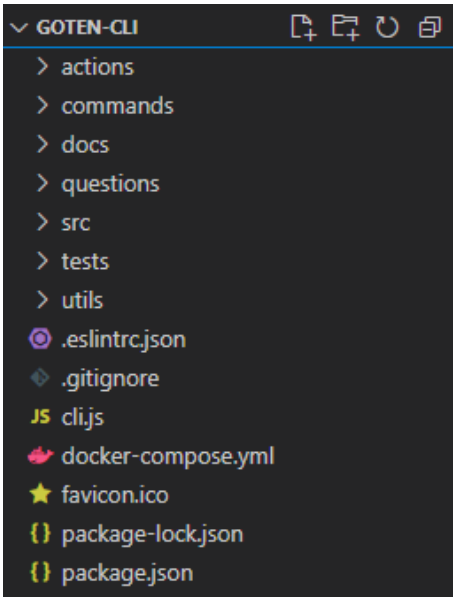

***Figura 3**: Estructura estática de GotenJS.*

La siguiente tabla describe la funcionalidad básica de cada componente de la estructura estática visualizada en la Figura 3:

Nombre y ubicación	Descripción
goten-cli/actions/*	Acciones de los comandos de los módulos CLI's correspondientes a cada tecnología.
goten-cli/commands/*	Firma de los comandos de los módulos CLI's correspondientes a cada tecnología.
goten-cli/docs	Documentación correspondiente a GotenJS.(versiones español/inglés)
goten-cli/src/*	Creación de los módulos CLI's correspondientes a cada tecnología.
goten-cli/test/*	Archivos de Tests para verificar que el framework funciona correctamente.
goten-cli/utils/*	Validaciones y métodos genéricos sobre modificaciones y manejo de archivos.

goten-cli/.eslintrc.json	Archivo de configuración de ESLint.
goten-cli/.gitignore	Archivo de configuración de Git. (qué archivos o carpetas ignorar en un proyecto)
goten-cli/cli.js	Archivo raíz donde se crea el módulo Goten-Cli, enlazado con los otros módulos CLI.
goten-cli/docker-compose.yml	Define la construcción de los contenedores mediante una imagen específica y la configuración dedicada a dicho contenedores (servicios, nombre, puertos, volúmenes, comandos, interacción y red).
goten-cli/favicon.ico	Imagen .ico default para probar el comando *goten-favicon*
goten-cli/package-lock.json	Se genera automáticamente para cualquier operación en la que npm modifique el node_modules o package.json. Describe el árbol exacto que se generó, de modo que las instalaciones posteriores pueden generar árboles idénticos, independientemente de las actualizaciones de dependencia intermedias.
goten-cli/package.json	Contiene metadatos legibles por humanos sobre el proyecto (como el nombre y la descripción del proyecto), así como metadatos funcionales como el número de versión del paquete y una lista de dependencias requeridas por el framework.

***Tabla 1**: archivos de la framework GotenJS.*

Como se mencionó en la Tabla 1, el archivo **docker-compose.yml** define la construcción de los contenedores con el objetivo de generar ambiente para poder desarrollar y/o utilizar GotenJS mediante la siguiente estructura:

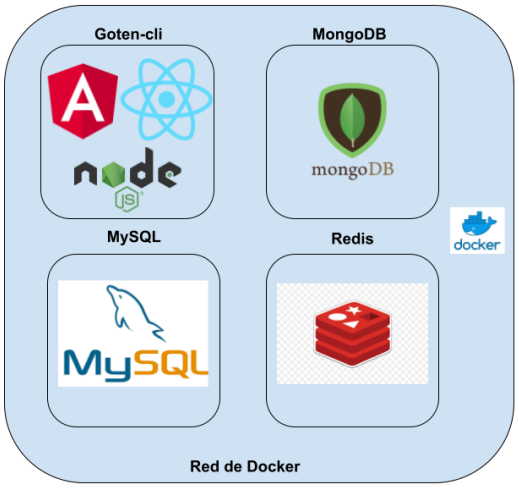

***Figura 4**: Infraestructura mediante docker-compose.*

La infraestructura visualizada en la Fig consta de los siguientes contenedores:

- **goten-cli :** container construido a partir de una imagen *node,* dedicado a la unificación de ambientes para la puesta en marcha del CLI de Goten. Para que la aplicación generada tenga conexión con las bases de datos, se linkean los demás contenedores a través de la *network docker goten-cli*.
- **MongoDB:** container construido a partir de una imagen *MongoDB.*
- **MySql:** container construido a partir de una imagen *MySql.*
- **Redis** container construido a partir de una imagen *Redis.*

Arquitectura *generada por* GotenJS

GotenJS define una arquitectura base de manera tal que el proceso de construcción de aplicaciones sea estandarizada, habiendo definido para cada módulo ciertas estrategias de diseño y desarrollo que se detallarán a continuación.

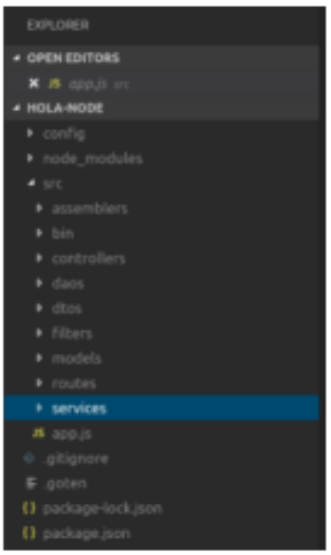

***Figura 5:** Estructura del proyecto de Node.*

Módulo NodeJS

El módulo Node permite generar la estructura base del proyecto dividiéndola en las carpetas **config** (configuraciones pertinentes), **routes** (definición de las rutas correspondientes), **controllers**(conjunto de funcionalidades), **services** , **dao** (interacción directa con la base de datos), **models** (los modelos que se representan en la base de datos), **dtos** (recurso para asignar los atributos del modelo), **assemblers** (funciones utilizadas para crear los modelos), **filters** (creación de filtros para realizar búsquedas en la base de datos); tal y como se puede visualizar en la Figura 5.

Módulo Angular

El módulo correspondiente a Angular trae las posibilidades de crear una aplicación con la estructura base del proyecto conteniendo lo generado por el comando **ng new**, e instala las dependencias de Bootstrap4 y fontawesome.
En la Figura 6 se muestra la arquitectura basada en Angular y como GotenJS interactúa con sus componentes.

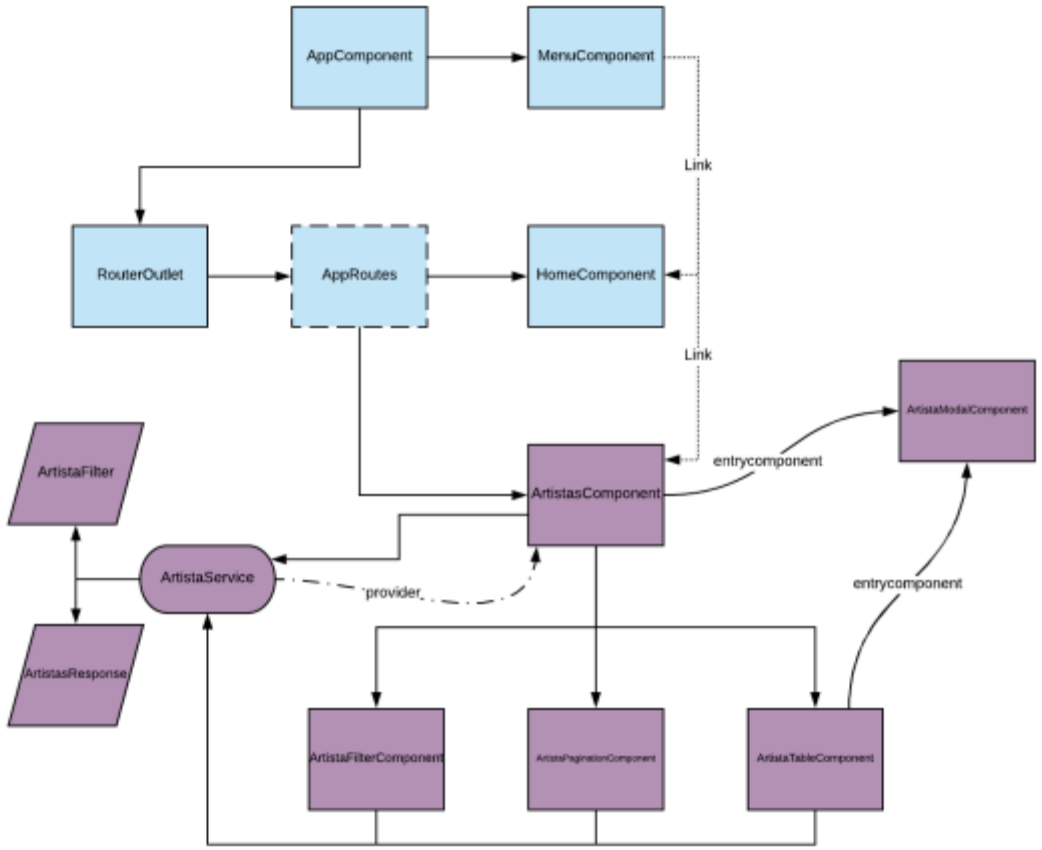

Figura 6: *Estructura del proyecto de Angular y la relación entre componentes.*

Cuando se ejecuta el comando goten new-angular se genera la estructura base del proyecto de Angular compuesto por directorios y archivos de base según la siguiente descripción:

- **proyecto/src/app/components/:** Directorio donde se agregan los componentes autogenerados.

- **proyecto/src/app/components/home.component/ :** Componente home, pantalla inicial de la aplicación.
- **proyecto/src/app/components/menu.component/ :** Componente menu, donde se registran los distintos tabs y a donde se enlazan los mismos.
- **proyecto/src/app/components/generics/ :** Componentes base que utilizan los componentes autogenerados.
- **proyecto/src/app/dtos/:** Directorio donde se agregarán los datos de las entidades que se consuman de la API.
- **proyecto/src/app/dtos/filters/:** Directorio donde se agregarán los filtros que se utilizan en los forms.
- **proyecto/src/app/services/:** Directorio en la que se ubican los servicios autogenerados.
- **proyecto/src/app/app.routes.ts:** en este archivo se registran las rutas y los componentes a los que cada una referencia, se tienen cuatro líneas de comentarios (//**<imports>,** //**<imports>,** //**<routes> y** //**</routes>**) que sirven para registrar las nuevas rutas cada vez que goten genera un ABM, es por esto que las mismas no deben ser modificadas/eliminadas por el usuario.

Conexiones inter módulos Angular-Node

La conexión entre la arquitectura genera en un proyecto Angular y Node está basada en la arquitectura clásica de una aplicación MEAN CRUD (Figura 7)

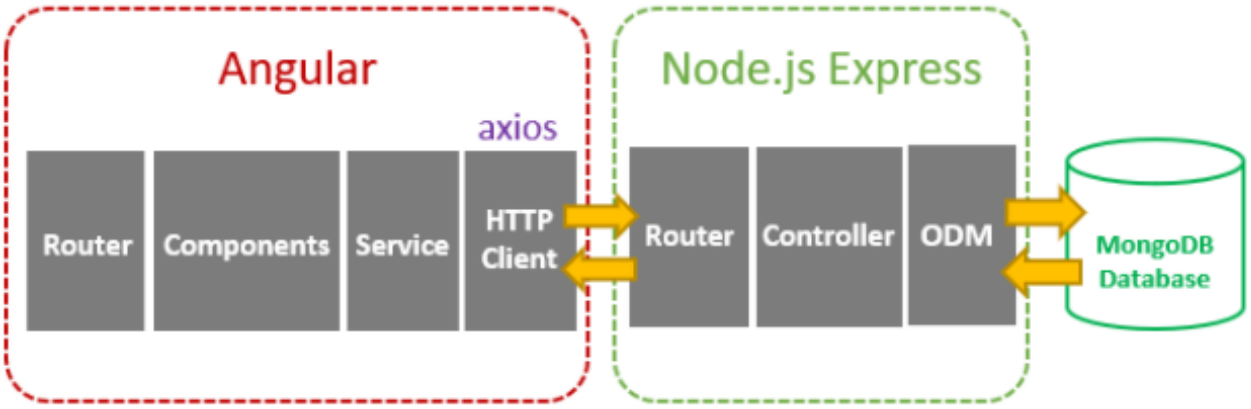

***Figura** 7: Arquitectura MEAN*

Desde Node Express, se exporta las API REST (consumida por las solicitudes HTTP de Angular).Mientras que desde el DAO se interactúa directamente con la base de datos MongoDB utilizando Mongoose. En el caso, de elegir como base **mysql**, se ubica la arquitectura SEAN interactuando con dicha base mediante Sequelize ORM.

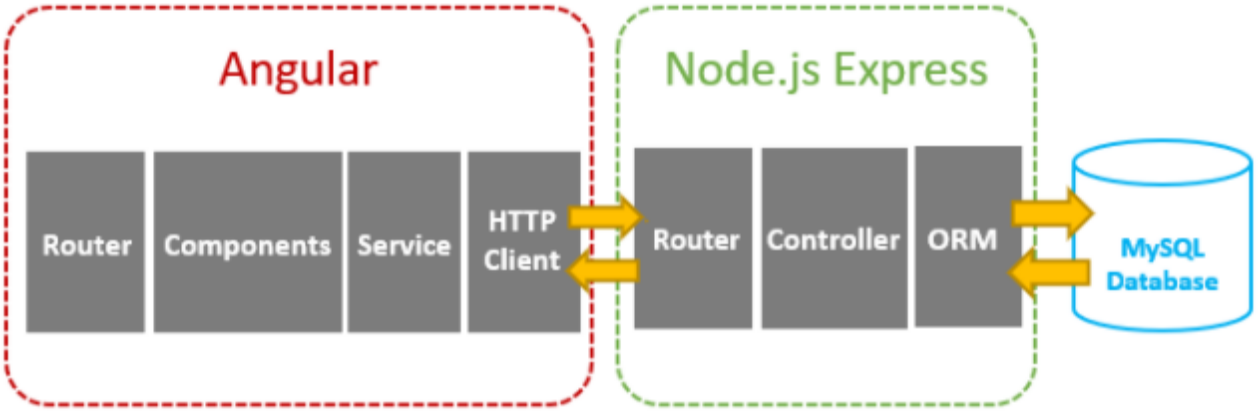

Figura 8: *Arquitectura SEAN*

Desde el Angular Client, se envían solicitudes HTTP y se recuperan respuestas HTTP usando HTTPClient para consumir datos sobre los componentes. Mientras que el enrutador angular se utiliza para navegar a las páginas.

Módulo React

El módulo correspondiente a React facilita la posibilidad de generar la arquitectura del proyecto tanto con **Context** como con **Redux**.

Context: La estructura del proyecto creado con Context se distribuye entre los directorios config , layout (componentes reutilizables), modules (componentes que representan los datos del backend) e utils (funciones y constantes útiles). La aplicación se crea utilizando **create-react-app** y en la Figura se muestra lo que genera el CLI de GotenJS.

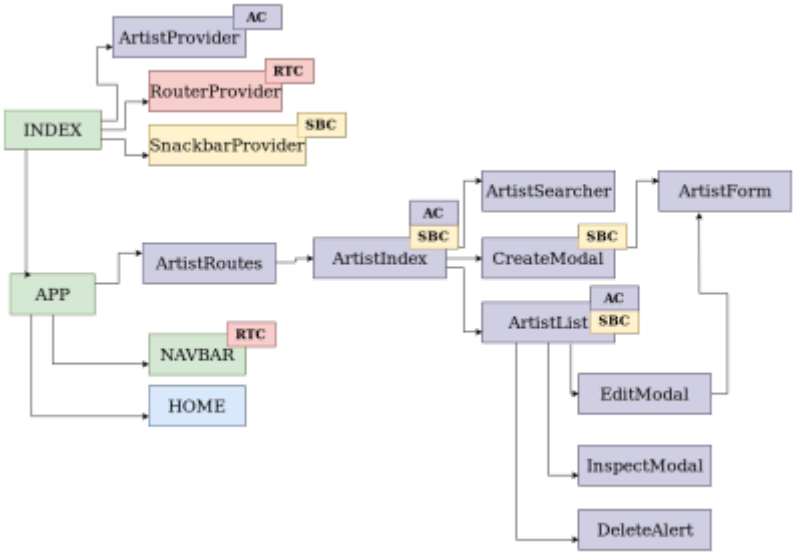

Figura 9: *Estructura del proyecto de React-Context y la relación entre componentes.*

En **verde**, están los componentes que no hace falta que el usuario modifique. El componente de App y el Navbar son bastante estáticos, dado que el CLI se ocupa de generar las rutas al crear módulos, y de agregar los providers correspondientes a index.js.
En **azul** está el componente Home. Este tiene que ser modificado por quien sea que use el CLI, dado que no tiene configuración alguna y está hecho puramente como relleno (que se supone necesario para todo proyecto).

En **rojo** está representado el provider del contexto de Router(RTC). En amarillo está representado el provider del contexto de Snackbar(SBC).
En **violeta** está representado un modelo sobre el cual se hizo el ejemplo de lo que generaría el CLI. Este módulo es de Artistas, y tiene ciertos componentes que representan el modelo definido en la base de datos.

Redux Luego, para el uso de Redux se define una estructura **'duck'**. Esta permite escalar la aplicación de forma incremental, de modo de no tener problemas con archivos de gran tamaño. La estructura a grandes rasgos se compone por una carpeta duck para cada m´odulo, independientes unas de otras y representarán cierta funcionalidad, y en estas se encontrará lo típico de Redux (constantes de tipos, acciones y action creators, y un concepto nuevo que se llama operations). La estructura es similar a la creada con Context.Está dividida en los módulos **config**, **layout** (componentes reutilizables), **modules** , **utils** (funciones y constantes útiles, entre ellas) y **redux**, En la Figura se muestra lo generado por CLI de GotenJS

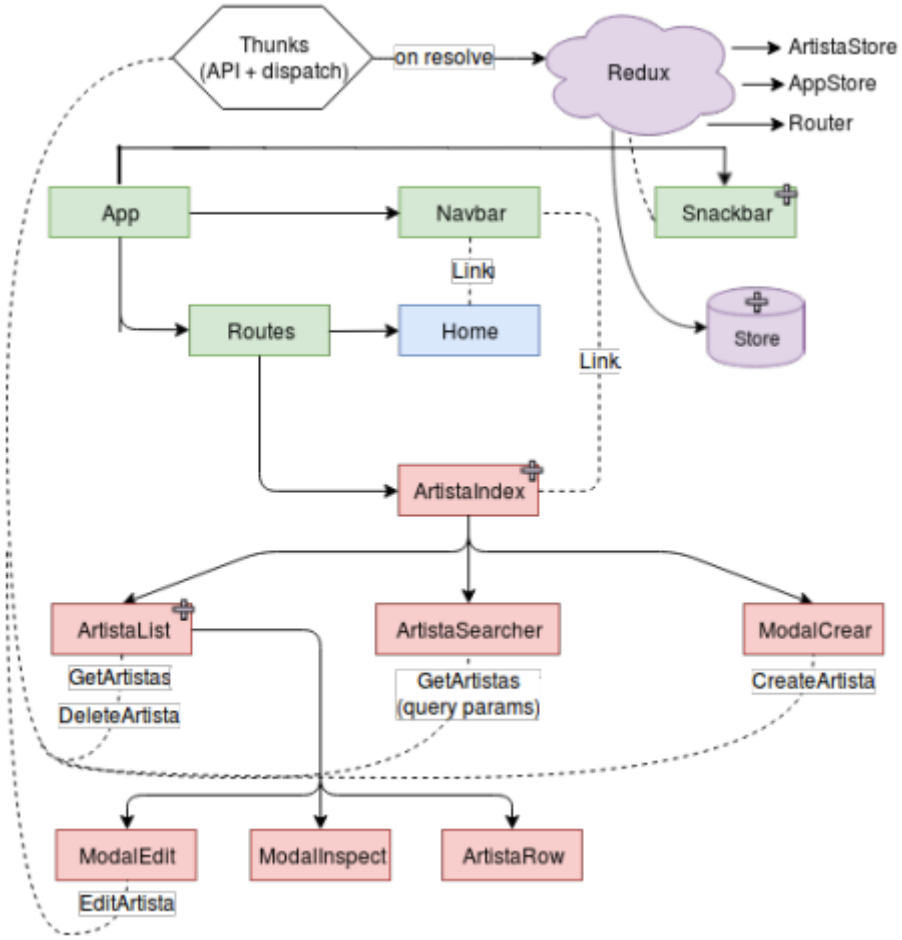

Figura 10: *Estructura del proyecto de React-Redux y la relación entre componentes.*

En **verde**, los componentes que no serán modificados (o no necesariamente) por el usuario.
En **azul**, el componente Home.
En **violeta** está representado Redux y su store. Algunos componentes escuchan al store de Redux (se los marca con una pequeña cruz violeta arriba a la derecha) mientras que otros estarán haciendo dispatch para hacer pedidos a la API, o en el caso del snackbar para mostrar y cambiar su mensaje.
En **rojo** se representa el modelo generado por CLI. Este módulo es de Artistas, y tiene ciertos componentes que representan el modelo.

Conexiones intermódulos React-Node

La conexión entre la arquitectura genera en un proyecto React y Node está basada en la arquitectura clásica de una aplicación MERN CRUD (Figura 11)

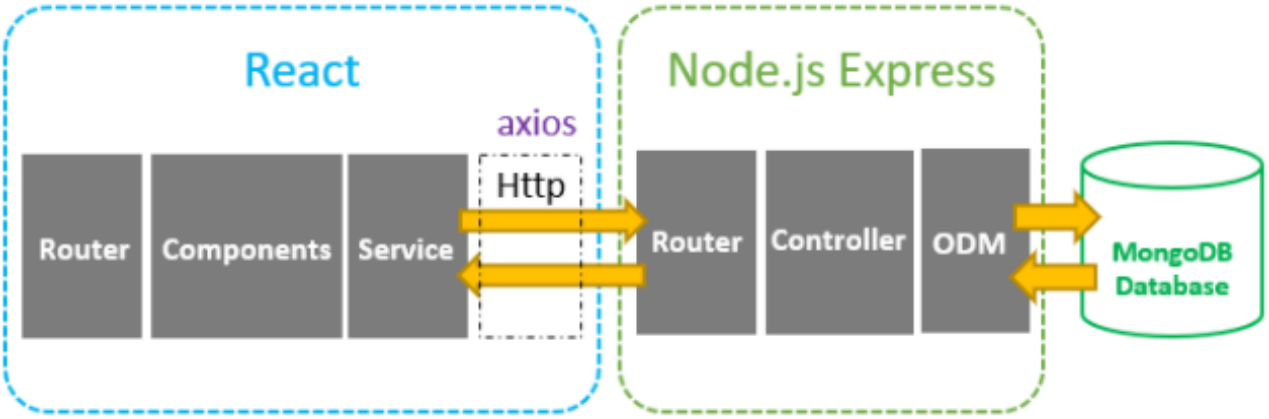

Figura 11: *Arquitectura MERN.*

Desde Node Express, se exporta las API REST (consumida por las solicitudes HTTP de Angular).Mientras que desde el DAO se interactúa directamente con la base de datos MongoDB utilizando Mongoose.

En el caso, de elegir como base **mysql**, se ubica la arquitectura SEAN interactuando con dicha base mediante Sequelize ORM.

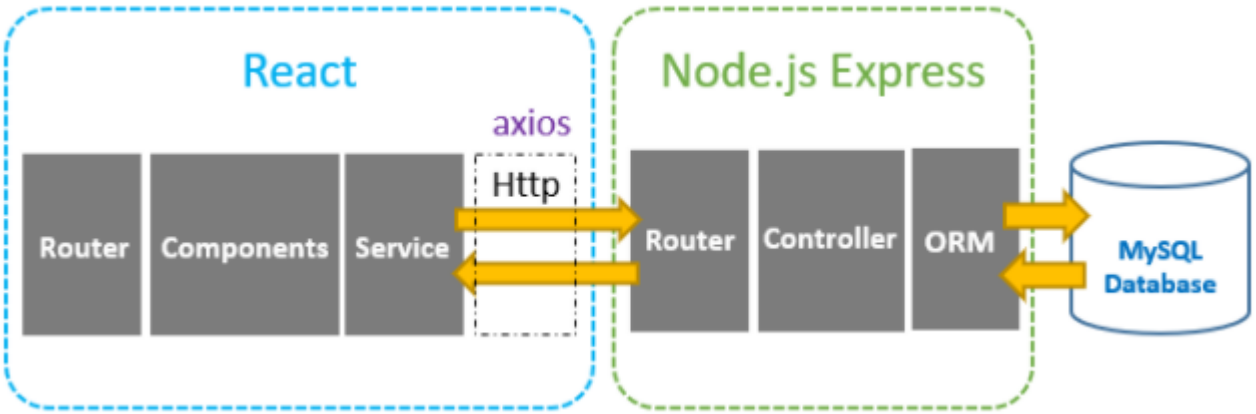

Figura 12: *Arquitectura SERN.*

Desde el React Client, se envían solicitudes HTTP y se recuperan respuestas HTTP usando Axios para consumir datos sobre los componentes. Mientras que el Router React se utiliza para navegar a las páginas.

En Context, la conexión a la API es sencilla debido a que se definieron las funciones a utilizar en la carpeta `utils/api/api<Nombre_del_modulo>.js`, y desde Context lo que se hizo es la lógica correspondiente para el manejo del store, pero siempre usando estas funciones.

En Redux, el querer realizar un GET, POST, etc. a la API,se maneja con **funciones thunk**. Estas son funciones que devuelven funciones, simplemente. Entonces, se tiene configurado el store de Redux con middlewares para que se pueda hacer un dispatch de estos thunks. ¿Para qué sirven? Para que en un mismo llamado a la API se puedan hacer varios dispatch según la respuesta. Es más cómodo que definirlas en el componente ya que queda menos acoplado el código y es más fácil de modificar.

Conclusiones

La búsqueda de métricas que permitan un seguimiento contínuo efectivo de la productividad en el desarrollo de *software* precede tanto a la aparición de la metodologías "ágiles" (Maxwell, 2000) cómo incluso a la explosión del paradigma de la programación orientada a objetos (Duncan, 1989).

Las metodologías de trabajo ocupan una parte importante de la agenda discusión en el ámbito de la arquitectura de sistemas de cómputo, como lo reflejan las diversas charlas con desarrolladores de la industria organizadas por la cátedra, así como las presentaciones de referentes del campo como Patterson y Hennessy.

La complejidad creciente de los sistemas de cómputo, debido en parte a las exigencias de fiabilidad que les imponen los delicados sistemas físicos puesto bajo su control, y en parte la búsqueda de la integración producto de su proliferación y a pesar de su heterogeneidad, impulsa a la creación de herramientas de *software* que enfoquen tanto el problema de la seguridad como de la escalabilidad.

A la vez que viejos conceptos de la ciencia de la computación, como aquellos provenientes del paradigma declarativo, vienen al rescate del moderno desarrollo de *software*, se observa la tendencia a elevar el nivel de abstracción: para el desarrollo de *hardware,* los lenguajes de descripción (HDL) y la síntesis de alto nivel (HLS); para la infraestructura, la descripción por código (IaC); y para el *software,* la herramientas de la metaprogramación.

Referencias

SUSE (2021). "*What is Software-Defined Infrastructure?*" [en línea]. Recuperado de http://www.suse.com/suse-defines/definition/software-defined-infrastructure/

Duncan, Anne Smith (1989). "*Software Development Productivity Tools And Metrics*". DOI:10.1109/ICSE.1988.93686

Maxwell, Katrina D. (2000). "*Benchmarking Software Development Productivity*". DOI:10.1109/52.820015

Bicking, Ian (2004). "*The Challenge Of Metaprogramming*" [en línea]. Recuperado de https://web.archive.org/web/20200621083557/www.ianbicking.org/the-challenge-of-metaprogramming.html

Google (2021). "*Angular - ng generate*" [en línea]. Recuperado de https://angular.io/cli/generate

Microsoft (2021). "*dotnet new command - .NET CLI*" [en línea]. Recuperado de https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-new

Capece, Cintia Tatiana (2021). "*GotenJS: Framework para generar la estructura base de una aplicación web*".

Gonzalez Labarta, Rubiel. (2014). "*Javascript Sean Stack: Una Mirada Al Futuro Del Desarrollo Web"*.

Germain Ramírez, Cristal Esmeralda. (2020) "*Desarrollo de aplicaciones web utilizando JavaScript.*"

Yebra Acero, Daniel. (2017) "*Desarrollo multiplataforma de tipo Full Stack: creación de un Back*"

Khue, Trinh Duy, Nguyen, Thanh Binh, Jang, UkJIn, Kim, Chanbin, Chung, SunTae. (2017). *"Design and Implementation of MEARN Stack-based Real-time Digital Signage System."* 20(5), 808–826. https://doi.org/10.9717/KMMS.2017.20.5.808