UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Informe de Práctica Profesional I

DESARROLLO WEB - EVOL SERVICES

Sergio Daniel Ignacio Romero Véliz

Ingeniería Civil en Computación

Correo: sergioromeroiv@gmail.com
Teléfono: +56 9 5681 4697
Empresa: Evol Services S.A.
Periodo de realización: Enero - Febrero 2025

Supervisor: Edison Delgado Aguilera
Correo: edison.delgado@evol.energy
Teléfono: +56 9 6593 8746

CC4901 Práctica Profesional I
Fecha de entrega: 06 de Abril de 2025

Índice

1. Resumen

En el presente informe se describe el proceso de práctica en Evol Services, una empresa de consultoría y de gestión de contratos energéticos, donde el practicante participó en la refactorización de un proyecto en React + TypeScript de 3 años de desarrollo y ~500k líneas de código, con una gran parte de sus componentes con bugs y malas prácticas de código.

Se discute la metodología para afrontar el problema, el trabajo en equipo con desarrolladores de la empresa basado en SCRUM, la propuesta de soluciones y las complicaciones que se tuvo durante el proceso de práctica. El cómo se llegó a completar la refactorización, mejoras al proceso de testeo y building de la aplicación y finalmente nuevas interfaces para la aplicación web de la empresa.

Se realiza una reflexión sobre la ética profesional en el trabajo realizado, la confianza puesta en la relación practicante/empresa y los dilemas encontrados durante el proceso.

El informe concluye que los objetivos propuestos inicialmente en el plan de práctica fueron parcialmente completados, dado que el trabajo se enfocó principalmente en el desarrollo en el frontend, llegando a un aprendizaje mucho más profundo en este sector.

2. Introducción

Evol Services S.A., anteriormente llamada Ecom Energía, es una empresa de comercialización y administración de contratos de energía, adquirida en 2022 por el holding Empresas Lipigas.

En el departamento de Desarrollo y Tecnología de Evol, se realizan labores de mantención y desarrollo de nuevas funcionalidades del sistema que utiliza la compañía. El sistema se compone de variados servicios modularizados para gestionar la lógica de negocio y facilitar el trabajo a los otros departamentos de la empresa, principalmente a Gestión de Contratos Energéticos, Riesgo y Regulación y Atención al Cliente.

El pilar de este sistema es una aplicación web (EMER) que gestiona la mayoría de la lógica de negocio de Evol. Desarrollada inicialmente por Altiuz (ahora llamada Atuz), era en una apliacción web con una arquitectura monolítica[1] hecho con Java 8, Spring y AngularJS. Se encarga de gestionar clientes, contratos, medidores energéticos, mediciones y del envío masivo por e-mail de reportes mensuales para cada cliente, detallando su consumo, ahorro, incidencias, etc.

[1] Monolito: Arquitectura que combina la lógica de negocios, interfaz de usuario y capa de acceso de datos en una sola unidad desplegable.

Con el paso del tiempo, la evolución del negocio y escala del mismo, las responsabilidades de desarrollo y mantención del sistema pasaron desde Altiuz hacia el departamento de desarrollo y tecnología de Evol Services. Manteniendo el backend en Java, marcando la segunda iteración del proyecto, EMER 2.0, el frontend pasó del framework AngularJS a un proyecto aparte con React y TypeScript, separando las responsabilidades de frontend/ backend en repositorios distintos de la aplicación web.

El proyecto en Java con Spring, ahora dedicado al backend, está estructurado en distintos módulos (Restful, Server, DOM y Backend), permitiendo la escalabilidad y evolución de la lógica empresarial, pero con el costo de altas cantidades de módulos obsoletos hechos por desarrolladores que ya no tienen relación con la empresa, y de altos tiempos de espera para tareas complejas como la compilación de los reportes. En cuanto al frontend del EMER 2.0, donde se enfocó este proceso de práctica profesional, es un proyecto desarrollado con React que utiliza Vite como herramienta de compilación y servidor de desarrollo, TypeScript para tipado estático, tests E2E[2] implementados en Cypress, Axios como cliente HTTP y Bootstrap como framework CSS. Además, el ecosistema tecnológico de Evol incluye otros componentes como bots lectores de facturas energéticas que automatizan la carga de valores a la base de datos relacional PostgreSQL.

3. Descripción del problema

Al comenzar el proceso de práctica, el frontend en React utilizaba mayoritariamente la Context API de React, con una arquitectura compleja basada en múltiples Providers, Contexts, contextTypes, useProvider hooks y barrel files para manejar la lógica y los datos provenientes del backend. Aunque esta infraestructura permitía que la aplicación funcionara correctamente en el momento, presentaba limitaciones en cuanto a extensibilidad, legibilidad y mantenibilidad, lo que dificultaba el desarrollo de nuevas funcionalidades y la corrección de errores existentes o introducidos por actualizaciones de dependencias. Por esta razón, el equipo decidió migrar progresivamente a un modelo de gestión de estado centralizado con Redux, que resultaba más flexible y mantenible para este proyecto en particular.

Durante las seis primeras semanas, la tarea principal del practicante fue avanzar con esta migración, llevándola del 40% de completitud al 90%, minimizando el uso de la Context API y garantizando una implementación más uniforme en los componentes del proyecto. Debido a la existencia de componentes legados extensos y altamente complejos, la migración completa no era factible dentro del tiempo disponible. El esfuerzo requerido para migrar estos componentes no justificaba los beneficios obtenidos, por lo que se optó por hacer correcciones puntuales cuando fuese requerido. Dado que estos componentes tenían

[2]E2E: Metodología de pruebas que verifica el flujo y experiencia de la aplicación de principio a fin (end to end).

una estructura frágil, su modificación o reescritura tenía altas probabilidades de introducir errores difíciles de detectar y corregir, lo que hacía inviable su migración.

En las últimas semanas del proceso de práctica, el estudiante se enfocó en desarrollar nuevas interfaces para una funcionalidad de la aplicación relacionada con un nuevo tipo de contrato que la empresa iba a ofrecer a sus clientes. A partir de un documento extensivo de requerimientos y un mockup, desarrolló las interfaces requeridas utilizando los componentes reutilizables del proyecto y siguiendo la estructura de manejo de estados con Redux. Este desarrollo permitió que semanas después, una vez lista la lógica de negocio en el backend y la base de datos, el frontend estuviera preparado para una integración fluida, reduciendo la carga de trabajo de los desarrolladores encargados de esta nueva funcionalidad, que ya llevaba cinco meses en desarrollo.

Además, a lo largo de todo el proceso, se realizaron correciones y mejoras en componentes y módulos del proyecto, incluyendo bugs que llevaban años sin ser atendidos. El estudiante implementó una mejora en la detección de errores en el código, mediante un wrapper[3] del compilador de TypeScript que realizaba un análisis de tipos en todo el proyecto antes de realizar push al repositorio remoto. Esta mejora resultó fundamental, ya que anteriormente el proyecto había experimentado problemas por importaciones de componentes fallidas que, debido a las optimizaciones del proceso de compilación de Vite, no eran detectadas hasta que el código ya estaba integrado en la rama de desarrollo.

4. Objetivos

4.1. Objetivo General

El objetivo general de este proceso de práctica fue que el estudiante participase en el desarrollo y mantenimiento del sistema EMER en Evol Services como desarrollador fullstack, contribuyendo a la mejora del software mediante la optimización del código, la corrección de errores y la implementación de nuevas funcionalidades.

4.2. Objetivos Específicos

- Analizar los desarrollos previos del frontend para entender cómo estaba funcionando y qué cosas debían cambiar.
- Corregir bugs en los desarrollos legados en parches o en las nuevas versiones de los componentes.
- Mejorar la detección de bugs en fase de desarrollo del repositorio del frontend.
- Refactorizar módulos obsoletos y/o defectuosos aplicando el nuevo estándar del equipo para desarrollar en el front.
- Crear interfaces para nuevas funcionalidades de la aplicación.

[3]Wrapper: programa o fragmento de código que encapsula y llama a subprocesos. Se usan para adaptar o agregar funcionalidad al subproceso

5. Metodología

La metodología de trabajo utilizada para completar los objetivos de la práctica fue la siguiente:

El equipo seguía una metodología de Daily SCRUM, que se basaba en reuniones diarias cortas (10 a 20 minutos), fuera por Microsoft Teams o en persona, para discutir los avances, compromisos diarios y dificultades en el trabajo de cada uno. Esta forma de trabajar impulsa fragmentar las tareas más generales y complejas en tareas pequeñas, realizables y medibles para cada día. Este sistema funciona bien siempre y cuando las tareas se podían fragmentar efectivamente y que no se repitieran por más de 1 día, lo que no siempre era el caso. El estudiante considera que esta metodología fue útil para no perder el hilo y la percepción del tiempo en cuanto a los objetivos más generales y poder hacer ajustes a los desarrollos por faltas de tiempo, lo que fue muchas veces necesario por aparición de bugs, vacaciones de miembros, enfermedades u otras incidencias.

En cuanto a la metodología para ocupar GIT, era la siguiente:

1. La rama main (producción) era intocable, sólo el coordinador de Desarrollo y Tecnología (supervisor de prácica) tenía permiso de hacer cambios a main mediante pull requests.
2. La rama desarrollo siempre debía estar funcionando correctamente, era el punto común de encuentro entre desarrolladores.
3. Para introducir cambios, debía crearse una nueva rama salida de desarrollo. Al terminar se realizaba un Pull Request detallando todos los cambios y con la checklist pre-revisión completada (haber realizado una revisión propia del código, probar la aplicación en local y crear scripts SQL en caso de ser necesario). Luego, el Pull Request era revisado y mergeado por el coordinador de Desarrollo y Tecnología a la rama de desarrollo.
4. Cada Pull Request debía introudcir un cambio discreto hacia la rama de desarrollo, como una nueva funcionalidad o una corrección específica. Esto era para controlar los posibles efectos secundarios de los cambios y mantener un historial claro de qué cambios se han hecho.

La metodología ocupada para trabajar específicamente en la refactorización del frontend fue migrar por partes la aplicación, por ejemplo, el menú de administración -> menú de cargas. Cada menú tenía entre 5 y 8 módulos asociados, los cuales eran en su mayoría páginas relacionadas a CRUD (create, read/retrieve, update y delete), las que son mantenedores de las entidades de la base de datos. Para cada módulo, el estudiante leía tanto su versión obsoleta como otros módulos ya migrados que eran similares para luego reescribir los componentes. Se probaba manualmente el funcionamiento y el manejo de estado y después se automatizaba el proceso con Cypress. Al terminar y pasar todos los tests existentes, se creaba un Pull Request hacia la rama de desarrollo.

Los tests en Cypress se hacían al final dado que era una tarea intensiva para la computadora entregada por la empresa hacia el estudiante, por lo que se testeaba exhaustivamente usando la aplicación en el servidor de desarrollo local y luego se escribían tests automatizados, que finalmente tenían la utilidad de romperse en caso de que un cambio externo modificara el funcionamiento de la página.

En cuanto a los desarrollos de nuevas interfaces, se siguió una metodología similar, aplicando el mismo estándar de desarrollo. Se utilizaban los componentes específicos que ocupaba el proyecto con datos estáticos en el front, pero desarrollando la interfaz anticipando las tareas de integración con el backend, para minimizar los cambios y correcciones del código de la interfaz.

6. Descripción de la Solución

La refactorización propuesta para el frontend que se había puesto en marcha desde antes del proceso de práctica fue cambiar la estructura de cada módulo. La idea era pasar desde tener sólo un directorio de components y providers a separar cada responsabilidad en un directorio distinto. Los componentes debían encargarse únicamente de definir el formato de cada pantalla, features se encargaba de manejar el estado de Redux, definiendo las funciones requeridas para usar la libería, services guardaba las llamadas HTTP hacia el backend de manera localizada, dado que anteriormente se guardaban en otro directorio mucho más arriba en el árbol de archivos. Los direcotrios types y utils eran para guardar las declaraciones de tipos de TypeScript y funciones a utilizar dentro de los otros directorios respectivamente. El archivo `index.tsx` tenía la única responsabilidad de cargar cada componente de manera On-Demand para llamarlos desde el Router.

Formato anterior

```
.
└── Módulo/
    ├── components
    ├── provider
    └── index.tsx
```

Nuevo formato

```
.
└── Módulo/
    ├── components
    ├── features
    ├── services
    ├── types
    ├── utils
    └── index.tsx
```

La estructura anterior de cada modulo seguía un patrón simple. En el directorio components se alojaban los archivos React TypeScript que se encargaban de las plantillas HTML + CSS, llamando a los providers de cada contexto que fuese necesario; el directorio de provider contenía la lógica de contexto de la entidad que representaba el módulo, por ejemplo, si el módulo era de Distribuidores, en provider debía estar un archivo llamado `ApiDistributorProvider.ts`, que exportaba funciones para manejar las llamadas a la API relacionadas con distrubidoras, como crear, modificar y eliminar.

Para la lectura de varias distrubuidoras, se tenía un archivo aparte de provider llamado `ApiDistributorWorklistProvider.ts` que se encargaba de llamar a la API y llevar los datos a una estructura reutilizable definida para todas las listas de entidades de la base de datos. El problema con esta estructura era que en el archivo `index.tsx` se envolvía un componente principal del directorio components en todos los contextos que el módulo iba a utilizar. Además, cada módulo utilizaba una sola URL en el Router, por lo que para cambiar de acción dentro del componente, se había escrito una estructura de montaje de componentes en pantalla, la cual era dificil de leer y debugear dado que para agregar o cambiar cosas de la estructura, había que modificar más de un archivo para lograrlo.

El trabajo del estudiante fue llevar esta migración hasta completarla en 2 meses de trabajo, separando las responsabilidades de cada archivo para seguir la nueva estructura que era mucho más amigable para realizar extensiones y correcciones, ya que cada parte de la funcionalidad se implementaba en lugares separados. Antes habían declaraciones de tipos y funciones de manejo de estado dentro de `components`. En las primeras 2 semanas el estudiante se enfocó en familiarizase con la librería Redux y TypeScript, tecnologías que no había ocupado antes. Cada módulo le tomaba al rededor de 4 o 5 días en reescribirlo. Durante las siguientes semanas este tiempo se vería reducido a 2 o 3 días por módulo. A mitades de Febrero el 90% del front estaba migrado a la nueva estructura.

Una vez finalizada la refactorización de los principales módulos del EMER 2.0, al estudiante se le encomendó implementar una maqueta funcional en el front para un nuevo módulo de gestión de contratos, se le fue entragado un mockup[4] y un documento de especificación sobre el nuevo módulo. Debía tener 3 tabs, una para gestionar los datos generales del contrato, una tab para gestionar los periodos del contrato y condiciones asociados a este, y por ultimo una tab de anexos. En la tab de periodos también debía haber un submódulo de bloques horarios para manejar el cóbro de energía por bloques horarios, siguiendo las condiciones descritas por un formulario dinámico. Siguiendo la estructura de manejo de estados con Redux, la maqueta quedó con la siguiente estructura de archivos, preparada para su integración con el backend que estaría listo semanas después:

[4]Mockup: Una representación visual similar a un prototipo que simula el aspecto final de un diseño

```
.
└── PPAContracts/
    ├── components/
    │   ├── tabs/
    │   │   ├── contract/
    │   │   │   ├── ContractForm.tsx
    │   │   │   └── Chatbox.tsx
    │   │   ├── periods/
    │   │   │   ├── energy-blocks/
    │   │   │   │   ├── forms/
    │   │   │   │   │   ├── FixedPriceForm.tsx
    │   │   │   │   │   ├── MarginalCostForm.tsx
    │   │   │   │   │   └── DecoupledForm.tsx
    │   │   │   │   ├── EnergyBlockForm.tsx
    │   │   │   │   └── EnergyBlockWorklist.tsx
    │   │   │   ├── PeriodWorklist.tsx
    │   │   │   └── PeriodModalForm.tsx
    │   │   └── annexes/
    │   │       ├── AnnexesFrom.tsx
    │   │       └── AnnexesWorklist.tsx
    │   ├── VersionFilter.tsx
    │   └── Main.tsx
    ├── features/
    │   └── PPAContractSlice.ts
    ├── types/
    │   ├── PPAContract.ts
    │   ├── Period.ts
    │   ├── EnergyBlock.ts
    │   └── Annexes.ts
    ├── utils/
    │   ├── utils.ts
    │   ├── pagination.ts
    │   └── validations.ts
    └── index.tsx
```

Después de desarrollar esta maqueta, el supervisor de práctica le comentó al estudiante que hubo errores en la rama de desarrollo por imports defectuosos. El estudiante tomó la tarea de investigar lo sucedido y buscar una forma de solucionarlo.

Para comenzar el análisis, leyó los commits que habían sido la solución de los imports defectuosos y se percató de que la única diferencia eran las mayúsculas en los directorios, en vez de decir `import some-component from ./components` decía `import some-component from ./Components`. En el servidor local de Vite este import era resuelto sin problemas pero en un servidor de pruebas de Ubuntu, el proyecto no compilaba. Investigó sobre el problema y encontró que en el sistema de archivos de Windows, el sistema operativo que usan todos los computadores de los trabajadores de la empresa no era *case sensitive*, es

decir, que `Components` y `components` para el sistema de archivos de Windows no tenían ninguna diferencia, sin embargo en los sistemas operativos basados en Linux, como Ubuntu, estos directorios eran diferentes. Este tipo de problemas en específico ocurrían cuando un desarrollador nombraba por error el directorio `Components` pero la convención era llamarlo `components`. Al intentar renombrar el directorio desde un Windows, cambiaba el display del nombre en el explorador de archivos, pero por debajo, el directorio no era renombrado correctamente. Una solución a este problema era pasar por un nombre intermedio como `components-temp` y luego renombrar a `components`, pero esta solución apuntaba a un error ya encontrado, quedaba el problema de detectar estos errores antes de llegar a la rama de desarrollo o posiblemente a producción. Investigando más sobre la situación, el estudiante notó que cada vez que pasaba esto, el compilador de TypeScript mostraba un mensaje diciendo que el import se había resuelto, ya que la diferencia con el nombre real del directorio era una sola letra. Además, al construir la aplicación con `npm run build`, aunque la aplicación tuviera errores de TypeScript, el comando se ejecutaba sin error alguno. Al buscar en la documentación de Vite se mencionaba que el builder que se ocupaba estaba optimizado para la rapidez de construcción, por lo que no se realizaba ningún chequeo de tipos o errores en tiempo de building. Vite mencionaba que era responsabilidad del desarrollador realizar un chequeo de tipos y errores antes de construir el proyecto. El estudiante evaluó distintas alternativas para hacer un chequeo estático de los errores del proyecto como ESLint, pero esta opción no fue factible, pues al ejecutar ESLint se arrojaban más de 86 mil warnings y errores en el proyecto. Afortunadamente, TypeScript tiene un comando para ejecutar el compilador sin crear archivos de output en JavaScript, por lo que el estudiante hizo una prueba utilizándolo como type-checker antes de construir el proyecto. Agregó los siguientes comandos al `package.json` del proyecto:

```
scripts: {
    ...
    "linter": "tsc --noEmit",
    "build:dev": "tsc --noEmit && vite build --mode dev",
    "build:prod": "vite build --mode prod"
    ...
}
```

Esta solución detectaba correctamente estos errores en tiempo de compilación del proyecto durante las pruebas locales. El compilador de TypeScript sólo detecta errores más graves a comparación de ESLint, que incluye sugerencias de buenas prácticas en el código, que al no seguirlas, las marca como errores. A pesar de que `tsc` era menos estricto que `ESLint`, el compilador de TypeScript detectaba alrededor de 200 errores de tipado e imports en el proyecto. Al revisar los errores, el estudiante se percató de que la gran mayoría eran fáciles de corregir, puesto que la aplicación en sí no estaba rota, sólo había que declarar más explícitamente algunos casos bordes, como declarar variables, como `let x: number | null` en vez de sólo number.

El proceso de arreglar todos estos errores de compilación le tomó alrededor de 4 días. Una vez hecho se integró hacia la rama de desarrollo como parte del proceso de pruebas antes de crear un Pull Request.

Días después el supervisor de práctica se contactó con el estudiante para revisar un problema en la rama de desarrollo, el problema era que uno de los imports defectuosos de los que se debería encargar el script de linting con TypeScript se había colado en la rama de desarrollo y el comando de `npm run build:dev` no funcionaba. El estudiante sugirió a su supervisor que corriera el script de linting, pero el comando no decía nada. Esto es porque el comando de `tsc --noEmit` es silencioso y sólo imprime líneas en la consola en caso de haber un error.

Esto causa un problema de usabilidad, porque el equipo del supervisor de práctica estaba funcionando extremadamente lento durante esos días, entonces parecía que el comando no estaba haciendo nada.

Finalmente el import defectuoso que se había colado en desarrollo era un error que ya había sido arreglado en el Pull Request de introducir el comando de linting, sin embargo, otro Pull Request fue aprobado y la corrección de ese error fue sobreescrita, probablemente por no haber actualizado la rama del nuevo Pull Request con la rama de desarrollo antes de hacer el merge.

Pero el problema de usabilidad estaba presente de todas formas, no había un feedback hacia el usuario de que el comando estuviese haciendo su trabajo o se había quedado congelado. Para solucionar este problema, en vez de llamar directamente a `tsc --noEmit`, el estudiante creó un wrapper del comando en JavaScript, mostrando un spinner similar al de `npm install` para mostrar que el comando estaba siendo ejecutado correctamente mientras en un sub-proceso se ejecutaba `tsc --noEmit`, al terminar, el script leía el output del comando y si no se encontraban errores, mostraba un mensaje de éxito. Por otro lado, cuando encontraba errores, imprimía exactamente lo mismo que `tsc --noEmit`, que imprimía el tipo de error, el archivo y la línea en donde se encontró el error, junto con un recuento total de los errores del proyecto y links para abrir directamente los archivos defectuosos en el editor. De esta forma había un feedback mucho más rico entre la herramienta y el desarrollador.

La decisión del estudiante de crear el script con JavaScript fue principalmente para hacerlo multiplataforma, pues el equipo de desarrolladores tenía computadores con windows pero el proyecto se construía en un ambiente de linux, por lo que un shell script no era la mejor opción. En cambio, los scripts de JavaScript corren sobre el motor de NodeJS, lo que los hace multiplataforma.

7. Reflexión

Durante el transcurso de la práctica, el estudiante experimentó el trabajar en un equipo de desarrolladores profiesionales en una empresa consolidada y con procesos burocráticos, como lo fue instalar software en el computador empresarial para trabajar. Para hacerlo, se debe iniciar un proceso por la Mesa de Ayuda Integral de empresas Lipigas, dado que nadie tiene permisos de administrador en los computadores de la empresa. Cada vez que se requiere instalar un software hay que sacar un ticket virtual por la plataforma, esperar aprobación y que un técnico se conecte de manera remota al computador para tipear la clave de administrador. Este proceso hizo que los primeros 3 días de trabajo el estudiante no pudiera hacer nada más que ver por GitHub cómo estaban estructurados los repositorios y leer un poco de la documentación.

Gracias a la modalidad híbrida de trabajo, el estudiante pudo convivir con el equipo de desarrolladores de manera presencial en una oficina. La costumbre de todas las mañanas era completar la planilla con la planificación y salir a comprar desayuno a un kiosco cercano, tiempos de convivencia que acercaron al estudiante con el equipo y ayudaba a crear un ambiente de trabajo en equipo grato.

En un principio la práctica profesional iba a ser enfocada en el desarrollo FullStack, empezando primero por el frontend, que suele ser más amigable para un principiante y luego participar en el desarrollo de backend o AWS. Sin embargo, dado el rendimiento del estudiante y todo el trabajo que necesitaba el frontend, el proceso terminó enfocándose en éste area. Dado esto, el estudiante logró familiarizarse con TypeScript y React de una manera mucho más profunda y así, resolver problemas más complejos de los que podría haber encontrado si le hubiese dedicado menos tiempo.

Además, durante la práctica, el estudiante colcluyó que trabajar en el frontend, si bien no le disgustaba o parecia aburrido, la mayoría de problemas no son tan complejos en su naturaleza, sino que son más de seguir una estructura predefinida y poder escribir código más rápido. Hubo un momento en el que el estudiante sintió que ya no estaba aprendiendo tanto y se sintió estancado. La solución que encontró a este problema fue que, si bien la forma de refactorizar está bien definida, la tarea de reescribir todos los componentes seguía presente, y muchos de estos eran muy parecidos, no había mayor dificultad en eso, por lo que debía simplemente reescribir los componentes más rápido. Entonces el practicante creó sus propios snippets[5] para rellenar el código boilerplate[6] de los componentes. Gracias a eso el estudiante pudo optimizar su tiempo, demorandose menos en tareas repetitivas en donde no tenía que pensar en lo absoluto, a pasar más tiempo solucionando bugs, mejorando los

[5]Snippets: Fragmentos de código rellenables rápidamente

[6]Boilerplate: Término que se usa para referirse a código que se repite en varios bloques a lo largo de tu proyecto o inclusive en diferentes proyectos

componentes reutilizables y en general, dedicando más tiempo a pensar en cómo hacer el proyecto mejor.

Por otro lado, a ambos practicantes del equipo (el estudiante y otro practicante de otra universidad) los trataron más como desarrolladores comunes que sólo practicantes. Todo su trabajo estaba siendo incorporado directamente en la rama de desarrollo, y en un futuro integrado a producción. Por ende, era responsabilidad de cada practicante testear de manera exhaustiva sus desarrollos porque la empresa no contaba con un departamento de QA y testing de software. Además, como desarrolladores, siempre tuvieron acceso a un dump de la base de datos relacional del último mes para fines de desarrollo, estos dumps se compartían libremente por el chat de Devs, por lo que cualquiera puede descargar los códigos fuentes y el dump desde su computadora personal. Como dilema ético, es importante reconocer el valor de los datos, el dump contenía información de contacto de todos los clientes de la empresa, diversas mediciones, etc. Como estudiante de Ingeniería y Ciencias de la Universidad de Chile, el estudiante no guardó información valiosa de la empresa en su computadora personal porque consideraba que no era lo correcto.

Indagando más sobre la ética profesional, el compromiso del estudiante con la calidad del trabajo fue siempre perseguir la excelencia sobre todos sus desarrollos. Reconociendo que dada su poca experiencia con las tecnologías, quizás las soluciónes del estudiante no fuesen las óptimas para todas las situaciones, y que quizás en 2 años la solución óptima sea otra, siempre se mantuvo un estándar alto en todo trabajo del estudiante, lo que fue reconocido por su supervisor.

En cuanto a cómo ayudó la universidad en el desempeño del estudiante en su primer proceso de práctica profesional, 2 ramos son los destacados por sus enseñanzas:

- CC3002 - Metodologías de Diseño y Programación, donde se aprende a trabajar en proyectos de una escala un poco mayor a la de los cursos anteriores y se hace una breve introducción a cómo trabajar con GIT. Ambas cosas le fueron de gran ayuda al estudiante al llegar a trabajar en un proyecto de casi medio millón de líneas de código de sólo frontend, en donde más de un desarrollador trabaja en el repositorio.

- CC5002 - Desarrollo de Aplicaciones Web, un curso electivo en donde se aprenden todas las bases de lo que es el desarrollo de aplicaciones web de manera agnóstica a las tecnologías. El estudiante considera que si no fuese por este ramo electivo, probablemente no hubiese conseguido el puesto de practicante en primer lugar y señala que estos acercamientos a las áreas de trabajo de ciencias de la computación son muy importantes a la hora de buscar práctica profesional.

8. Conclusiones

Recapitulando, el objetivo principal propuesto de la práctica fue refactorizar módulos obsoletos del frontend para eliminar el uso de Context API en el proyecto, objetivo que fue completado con éxito dentro del periodo de práctica. Quedaron residuos de uso de Context API asociados a componentes demasiado complejos para reescribir en el periodo de práctica, pero más de un 90% del proyecto quedó implementado usando React Redux. También se presentó la oportunidad de crear nuevos componentes para la empresa, que fue un logro extra de la práctica.

Durante todo el proceso el estudiante logró entender mejor lo que es trabajar en un equipo de profesionales, organizar un proyecto, cómo convivir y pedir ayuda en el trabajo de desarrollo de software.

Por otro lado, el estudiante no llegó a participar en otras áreas del desarrollo de software como el backend, computación en la nube o la toma de requerimientos, debido al acotado tiempo de práctica y la gran cantidad de trabajo que requería el front. A cambio de esto, ganó un entendimiento superior sobre el desarrollo de interfaces de usuario y manejo de herramientas y lenguajes como TypeScript y Cypress.

Algunos puntos de mejora de la solución creada pueden ser el manejo más idiomático de ciertos casos en el frontend con TypeScript, como el manejo de posibles valores nulos en formularios cuando se llenan con datos provenientes de la API, pues la solución creada mantuvo un estilo agnóstico de programación, dada la poca familiaridad y experiencia con el lenguaje, por lo que para un experto en TypeScript, algunas implementaciones podrían parecer torpes y que exista una mejor forma de hacerlo según la documentación o experiencia con TypeScript. Por otro lado, el testing con Cypress se podría mejorar con un dump de la base de datos específico para pruebas y correr los tests en un servidor aparte para minimizar errores asociados al rendimiento del equipo de en dónde se ejecuten y los asociados a modificaciones de los datos con el tiempo. Un set de datos que no cambie en el tiempo abre las puertas para un testing más profundo de las funcionalidades y realizarlo de manera automática.

Finalmente se agradece a Evol Services, al señor Edison Delgado, Coordinador de Desarrollo y Tecnología, y a todo el equipo de desarrollo y tecnología por la oportunidad de aprender y experimentar el desarrollo profesional en un entorno real y seguro como practicante.