UNIVERSIDAD DE ANTIOQUIA

DESARROLLO DE UNA PLATAFORMA DE GESTIÓN DE EVENTOS PARA EL ÁREA DE GESTIÓN HUMANA Y ADMINISTRATIVA DE CEIBA SOFTWARE

Autor

David Jaramillo Bolívar

Universidad de Antioquia

Facultad de Ingeniería, Departamento de Ingeniería de Sistemas

Medellín, Colombia

2019

Desarrollo de una plataforma de gestión de eventos para el área de gestión humana y administrativa de Ceiba Software

David Jaramillo Bolívar

Informe de práctica
como requisito para optar al título de:
Ingeniero de Sistemas

Asesor

Jeysson Pérez Gómez
Ingeniero de Sistemas, Especialista en Gerencia Integral

Universidad de Antioquia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas
Medellín, Colombia
2019.

Índice

Resumen	3
Introducción	4
Objetivos	6
Marco Teórico	7
Resultados y análisis	12
Conclusiones	24
Referencias bibliográficas	25

Resumen

Ceiba eventos es una aplicación web para dispositivos móviles que surgió de la necesidad de digitalizar un proceso manual que se viene haciendo en Ceiba, en la gestión y realización de eventos que ejecuta el personal de gestión humana cuando se quiere conmemorar y realizar una integración con sus empleados en celebraciones y fechas especiales de cualquier clase.

Es una aplicación que se construyó bajo el paradigma de la programación funcional y objetual, en Scala y Typescript, implementa las buenas prácticas en ambos frentes, desde lo visual, hasta el core y el núcleo de la aplicación, a través de un conjunto de patrones y reglas o pautas que hacen del código una estructura natural y mantenible: la arquitectura hexagonal, el diseño dirigido por el dominio, y en general, un código fácil de entender e interpretar. Estos lenguajes naturalmente los acompañan un framework, Angular y Play son herramientas que no solamente le dieron un esqueleto al Software, sino que también le dotan al desarrollador de agilidad al problema en común y en cuestión: la construcción de una REST API.

Con el desarrollado realizado, los eventos en Ceiba podrán ser administrados de una forma transparente, y gestionarlos será un proceso fácil de efectuar. El área de gestión humana podrá visualizar a detalle los gastos de un evento, los participantes, y sus insumos y recursos, al igual que les posibilita la oportunidad de tener un repositorio de eventos, a lo largo del tiempo, del cual cada registro estará atestiguando cómo fue la ejecución y administración del evento.

Palabras clave

Arquitectura hexagonal, DDD (Domain Driven Design), Angular, Play, Typescript, Scala.

Introducción

Ceiba es una casa de desarrollo que se sirve de las metodologías ágiles para la construcción e implementación de soluciones tecnológicas. Es un deber de la empresa resolver las problemáticas del negocio del cliente a través de un producto tecnológico echo a la medida. Ceiba, aunque se enfoca en el desarrollo, también le genera un valor agregado a los clientes mediante consultoría de procesos, coaching en agilismo, análisis de datos y diseño y experiencia de usuario. Es una de las firmas más grandes de desarrollo en Colombia, con un número aproximado a los 350 empleados, y 14 años de experiencia en el mercado sirviendo a reconocidas empresas del territorio: Sura, Bancolombia, EPM, entre otras.

Ceiba no solamente busca la prosperidad en el negocio del cliente, sino también el bienestar de su personal, y esta tarea recae, en cierta medida, sobre el área administrativa y el área de gestión humana. Se han implementado varias estrategias para mejorar el ambiente laboral, por ejemplo, en dar obsequios con el fin de promover la integración en fechas especiales, tradiciones o celebraciones de cualquier tipo, es decir, se agendan y programan eventos de cualquier tipo.

El problema que se quiso abordar, y al que se le quiso dar una solución que mejorará el proceso fue el siguiente: de gestión humana una persona se encarga de dirigir y administrar el evento, ella tiene que comprar el material, extender la invitación a todos los empleados, registrar los participantes, y generar un reporte que permita avalar los gastos monetarios. El procedimiento es visto y manifestado como un tarea tediosa, y propensa a errores que no encaja con el conjunto de herramientas tecnológicas que normalmente se tienen para ejecutar los procedimientos laborales y regulares de la empresa (como lo es por ejemplo, la intranet de Ceiba, Jira y Google apps).

Se hizo entonces una aplicación web móvil en Scala y Javascript, que fuera capaz de digitalizar completamente el proceso. Fue esencial que la aplicación tuviera la estructura y los patrones que garanticen la mantenibilidad del mismo, a lo que se quiso implementar la arquitectura hexagonal, y el diseño dirigido por el dominio. Fue tarea del practicante levantar los requisitos de la aplicación, de elegir las tecnologías, los frameworks, las herramientas, la arquitectura y los patrones y las prácticas de codificación que habrían de implementarse en el proyecto. Esto implica una fase de aprendizaje, y una fase de construcción e implementación.

El proyecto concibe el desarrollo de un sistema de información para la gestión de actividades que involucren la entrega de algún inventario, recursos o materiales adquiridos al interior de la compañía. Esta nueva solución, mejora el tiempo invertido al registrar debidamente la información en una plataforma capaz de generar reportes y administrar en vivo la entrega del material u obsequio destinado al empleado. De esta forma el personal del área administrativa y de gestión humana cuenta con una herramienta que automatiza la operaciones y que se llevan a cabo en los eventos que organiza la compañía.

Finalmente, cabe resaltar que el desarrollo del aplicativo se hizo utilizando Scrum, una metodologia ágil personalizada para el desarrollo de una sola persona, además de contar en el proyecto con el acompañamiento constante mensual del asesor de la universidad, esto fue un reporte mensual que tenía por objetivo dar cuentas del trabajo hecho hasta la fecha de la entrega, y contrastar las metas con el cronograma del proyecto y así priorizar y replantear las entregas posteriores.

Objetivos

Objetivo General

Diseñar, desarrollar y desplegar un sistema de administración de inventario y gestión de eventos para mejorar las labores del área de gestión humana y del área administrativa de Ceiba Software.

Objetivos Específicos

1. Levantar y diseñar las características principales (requisitos) del sistema a desarrollar.
2. Definir y documentar el stack de tecnologías y herramientas, patrones arquitectónicos, principios y prácticas o técnicas de desarrollo a implementar en el sistema.
3. Desarrollar el módulo de gestión de eventos en tiempo real.
4. Desarrollar el módulo de reportes de inventario e historial de eventos.
5. Entregar la aplicación y desplegar el sistema a los servidores de producción.

Marco Teórico

Ceiba eventos es una aplicación cliente servidor, móvil web, y una RESTful API. El backend se construyó en arquitectura hexagonal, y el diseño dirigido por el dominio. El flujo, en su descripción más sencilla es: el usuario interactúa a través de la interfaz, el frontend envía las peticiones http al backend, la aplicación construida en Scala las procesa y modifica o consulta información, se retorna una respuesta, y es esta respuesta la que se manifiesta en la interfaz.

La arquitectura hexagonal es un término que se acuñó en un artículo del año 2015 por Dr. Alistair Cockburn y es, según Garrido (2018), un "patrón que promueve el desacoplamiento de las tecnologías y sus frameworks" (Garrido, 2018). Aquí, en el proyecto, es la mayor propuesta a desarrollar: de explorar los lenguajes e integrar las tecnologías de tal forma que la construcción del software garantice la mantenibilidad del mismo al experimentar cambios fuertes en su estructura, es decir, que pueda ser "puesta a prueba de forma independiente hacia los dispositivos externos que la aplicación depende: bases de datos, servidores de la aplicación, etc" (Garrido, 2019).

La arquitectura hexagonal connota exclusivamente la solución del negocio, sin intermediarios, es decir, peculiarmente se caracteriza porque el hexágono representa una aplicación agnóstica a la tecnología que solamente le concierne lógica del negocio. En el hexágono no se impone una estructura en particular; se es libre de implementar patrones, capas, o cualquiera estrategia que se crea idónea (por ejemplo DDD) (Garrido, 2019).

La arquitectura hexagonal está compuesta por actores, puertos y adaptadores:

- Los actores son, según Garrido (2019), el "ambiente de la aplicación, del mundo externo. Estas cosas incluyen humanos, otras aplicaciones y dispositivos de hardware/software". Se categorizan en dos tipos: principales y secundarios. Los actores principales inician la interacción con la aplicación, los secundarios enmarcan mecanismos y herramientas que la solución interactúa para completar funcionalidades.
- Los puertos son interfaces que ofrece la aplicación para materializar la interacción con los actores, (exclusivamente allí ocurren los intercambios de la información, y es para el actor secreto el detalle de implementación), o interfaces que son requeridas por el cliente (el actor) y que necesita la aplicación para ejecutar la lógica del negocio (Garrido, 2019).
- Los adaptadores son los artefactos por el cual los actores se relacionan con los puertos de la aplicación: "un adaptador es una componente de Software que permite que las tecnologías interactúen con los puertos del hexágono" (Garrido, 2018). Un adaptador en concordancia con el actor y el puerto, podrá o usar una interface o implementarla: usar una interface cuando se precisa convertir una petición tecnológica dependiente a una petición agnóstica y del negocio, e implementar una interface cuando se quiera convertir una petición independiente a tecnologías en una petición tecnológica dependiente (Garrido, 2019).

Esta arquitectura, según Fidao (2019), naturalmente se expresa en módulos: “la arquitectura hexagonal define capas conceptuales de código y sus responsabilidades, y luego señala las formas de desacoplar el código entre las mismas” [6]. Son tres capas, la capa del dominio, la aplicación, y del framework o infraestructura:

- El dominio define las políticas, es el núcleo de la aplicación, el que define el comportamiento y las restricciones, es la lógica del negocio que le proporciona un valor único a la aplicación.
- En la aplicación se exponen los “casos de uso”. Los casos de uso son las acciones que se pueden ejecutar en la aplicación, son las bondades del sistema, de la solución. Es una capa intermedia entre el dominio y la infraestructura: adapta las peticiones de la infraestructura, y delega las responsabilidades al dominio, es un orquestador.
- La capa de la infraestructura se sienta al final, es contigua a la aplicación. En la aplicación se encuentran las librerías y terceros, es decir, los adaptadores.

En la figura 1 se muestra el posicionamiento de las tres capas, y las tecnologías más importantes que se integran en el proyecto.

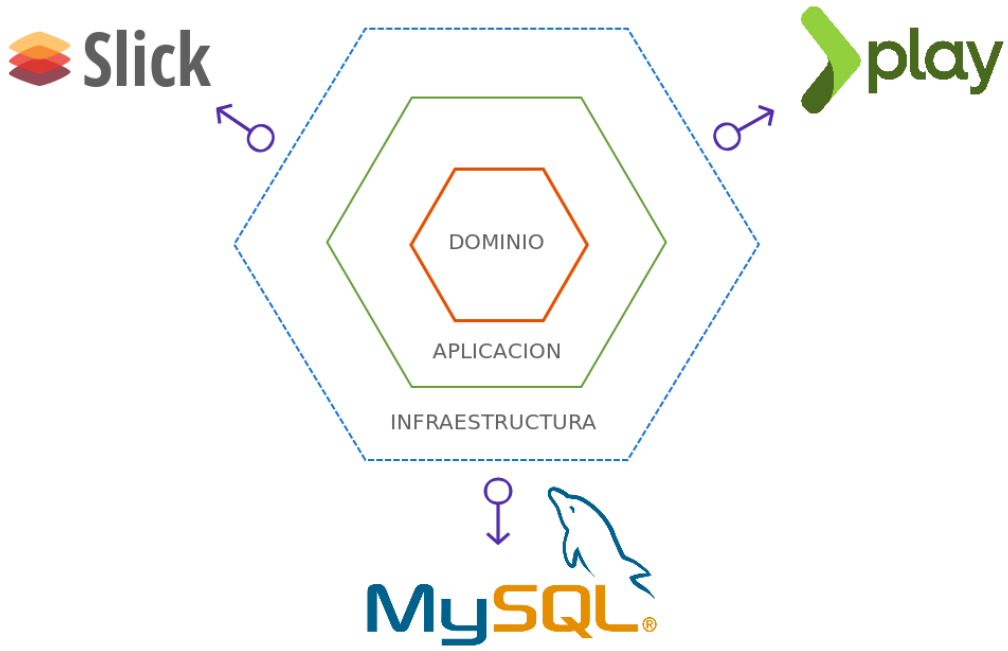

Figura 1. Arquitectura hexagonal y dependencias tecnológicas..

Con la idea de proporcionar una estructura a la lógica del negocio, se implementa DDD. La arquitectura hexagonal no le impone restricciones al dominio, sin embargo, es de acompañarse con

esta metodología que comprende un conjunto de reglas y recomendaciones que hacen que los artefactos de código tengan un alto grado de afinidad con el negocio: "DDD abarca un lenguaje en común, técnicas, patrones y hasta una arquitectura. El foco del diseño está puesto es en el dominio y en modelar los problemas que se están tratando resolver" (Apiumhub, 2019). DDD unifica el negocio, lo transforma en un idioma en el que los analistas y los desarrolladores se puedan entender (un lenguaje ubicuo), y la vez, suministra los mecanicas para construir la aplicación, desde el código que alude exclusivamente al negocio.

Son varios los "bloques de construcción" que utiliza DDD para estructurar la aplicación desde la perspectiva del negocio, un diccionario de términos y conceptos (Lelonek, 2019); dominio, lenguaje ubicuo, invariantes, contexto limitado, adaptador, agregados, raíz del agregado, entidad, objetos de valor, servicios del dominio y repositorios (y algunos otros más, que si bien se relacion con DDD, de por si no se asociacian a el). La aplicación logra implementar cuatro de las componentes mencionadas anteriormente: los agregados, las entidades, los objetos de valor y los servicios. Evans (2004), los describe de la siguiente manera:

- Los agregados pretenden darle una solución a las dificultades que puedan presentarse en la persistencia de la aplicación, en la consistencia de los datos. Los modelos son un conglomerado de tablas o entidades que se coordinan a través de, por ejemplo, las claves foráneas (unos a muchos, muchos a uno, etc), más no definen un contexto, una frontera, es decir, no determinan el alcance de una transacción, un agregado por lo tanto contextualiza y le proporciona una estructura a las entidades abstractas del negocio: "los agregados son un cluster de objetos asociados que se tratan como una unidad para los cambios de datos. Un agregado tiene una raíz y un límite, el límite define qué es lo que hay en el agregado."
- Al modelar los objetos, se podrían distinguir dos tipos, los objetos que se identifican no por sus atributos si no por su identidad, y los que no tienen identidad y se distinguen únicamente por el valor que contengan sus atributos. Al primer arquetipo se les llaman entidades, al segundo, los objetos de valor. Las entidades alteran las decisiones del diseño en el dominio, nos ayudan a proyectar como los objetos pueden ser identificables a través de sus distintos ciclos de vida en la aplicación, y definir las operaciones que permitan revisar la igualdad, no al comparar sus atributos, si no la esencia del mismo.
- Los objetos de valor describen las características de un elemento, no necesitan tener las propiedades que precisan las entidades de un hilo, de trazabilidad, y son por el contrario objetos efímeros y transeúntes que por lo general utilizan las entidades para comunicarse entre sí. Los objetos de valor, son objetos "que representan los aspectos descriptivos del dominio sin una identidad conceptual. Los objetos de valor son instanciados para representar objetos del diseño por el cual solamente nos importa por qué son, y no por lo que eran o lo que van a ser".
- Las operaciones del dominio no deberían ser responsabilidad de las entidades y los objetos de valor, esto es, si son complejas y no pertenecen conceptualmente al modelo que se está implementando. Los servicios son entonces la solución natural que ofrecen una interfaz que se apoya por sí sola, pues "algunos conceptos del dominio no son naturales para modelarse como objetos, y forzar la funcionalidad requerida del dominio a ser una responsabilidad de

una entidad o un objeto de valor, distorsiona la definición de un objeto basado en el dominio, o agrega objetos artificiales sin sentido".

Se ha dicho entonces que la arquitectura hexagonal y el diseño dirigido por el dominio son el esqueleto de la aplicación, la piedra angular. Sin embargo, también es importante hablar un poco de los frameworks, ciertamente también construyen el molde de lo que es la aplicación.

Angular es una "plataforma y un framework para construir aplicaciones cliente en HTML y TypeScript" (Angular, 2019), empaqueta un conjunto de funcionalidades base y librerías opcionales que se importan en la aplicación, se les llaman módulos. Un módulo es construido ya bien sea por el framework, o el usuario. Angular seguramente se caracteriza principalmente por esta plantilla, esta estructura: un módulo se constituye por las componentes, las componentes son los átomos del DOM, de la página web, son quienes componen y construyen las vistas (por HTML, SaSS y Typescript). Las componentes a su vez, utilizan servicios. Los servicios proveen funcionalidades específicas que no le competen a las vistas. Las 3 piezas fundamentales que determinan al framework, los módulos, las componentes y los servicios. En los módulos se diferencian las funcionalidades principales de la aplicación, pues las funcionalidades se agrupan en módulos, en las componentes estan las vistas, y en los servicios, la lógica que consumen los servicios y envía peticiones al backend.

Play es un "framework para la construcción de aplicaciones web de alta productividad en Java y Scala que integra las componentes y las apis para el desarrollo de aplicaciones web modernas" (Play, 2019), suministra las herramientas necesarias para la exposiciones de servicios a través de un controlador, un servidor HTTP integrado y un mecanismo de enrutamiento que permite el acceso a los endpoints de la aplicación. Play se revela como un framework altamente asíncrono al utilizar el modelo de actores que implementa Akka, que hacen de él que sea un servidor sea ligero y sin estado al entregar un mecanismo de trabajo que garantice la predictibilidad y el consumo mínimo de recursos en memoria, hilos y CPU; "gracias a su modelo reactivo, las aplicaciones pueden escalar naturalmente de ambas formas, horizontal y verticalmente" (Play, 2019). Y finalmente es un framework "sin opinión" a la persistencia, se integra fácilmente con cualquier paradigma, incluyendo NoSQL y otros ORMs.

Así es entonces cómo está construida la aplicación, un desarrollo hecho en Play framework (lenguaje Scala), Angular (lenguaje Typescript), con arquitectura hexagonal y algo de DDD.

Metodología

Scrum es una de las metodologías más populares para el desarrollo de software iterativo (Kulbacki, 2018), es simple, directa, está muy bien documentada y es fácil de entender. Es un marco ideal, sin embargo, está descrito para trabajar en equipo, y si se quiere aplicar al desarrollo de una sola persona, hay que modificarlo un poco. Se realizaron las ceremonias tradicionales de los sprints en Scrum, las reuniones diarias (dailies), las retrospectivas (sprint retrospective), las entregas (sprint release), y los planeamientos (sprint plannings), y se hicieron los mismos insumos habituales, el product backlog, y el sprint backlog. Como se tiene que algunas ceremonias requieren de al menos 3 participantes, la persona encargada asume más de un rol: el rol del scrum master y el rol del equipo desarrollador. Los dailies y las retrospectivas se plantean como unas pequeñas reflexiones escritas que sirven de apoyo para mejorar la productividad y generar nuevas estrategias. Los sprints tuvieron una duración de 2 semanas, las entregas (sprint release) fueron de media hora y no se consolidaron tiempos fijos como se haría tradicionalmente en los planeamientos, las retrospectivas, y los dailies, ya que son procedimientos que se realizan a nivel personal en donde lo importante es cumplir con los objetivos que manifiestan las ceremonias de scrum.

Los requisitos del sistema a construir (objetivo esp. número 1) se establecieron antes de cualquier sprint. Se realizó una reunión en donde el product owner y el desarrollador precisaron las necesidades que debió tener la solución esperada. Esta fase sentó las bases para iniciar el proyecto. Luego, se inició una etapa investigativa y de planeamiento (objetivo esp. número 2); se edificó el esqueleto, se definieron las tecnologías y los patrones a implementar, y se concibió el plan de los sprints a desarrollar (claro está, un plan sujeto a cambios). Esto fue un sprint, y se le llama el sprint cero. Seguidamente, se inició la codificación del sistema (objetivo esp. número 3 y 4) acorde al itinerario que se especificó en el sprint 0. Finalmente, se configuró parcialmente el ambiente para los servidores de producción (objetivo esp. número 5), y adicionalmente se validó la aplicación: se terminaron y pulieron algunos detalles que están pendientes por terminar.

Al final se tiene entonces que del objetivo específico número 2 al objetivo específico número 4, hay una cantidad “5” de sprints. El objetivo específico número 1 y el objetivo específico número 5 son, respectivamente, las etapas de iniciación y cierre del proyecto.

Resultados y análisis

Al culminar, el producto final le brinda al área de gestión humana una herramienta que no solamente agiliza la gestión de eventos en Ceiba (entiéndase por gestión como ejecución del mismo, desde el momento en el que logísticamente se empieza a trazar un plan a ejecutar, hasta el cierre o clausura), sino que también le suministra los medios necesarios para tener una meticulosa administración de los gastos monetarios a través de un historial en detalle que documenta, durante y después de su vigencia, la cantidad de participantes, personal, y material. Al así registrarse un evento, el proceso es completamente transparente; será posible contabilizar el dinero que se ha invertido en periodo de tiempo, de las actividades recreativas o de bienestar que se propician para el trabajador regular de la empresa, su frecuencia, sus participantes, su asistencia, entre otros aspectos.

La solución final es una aplicación web. El roadmap se estructuró bajo la metodología Scrum, que prioriza las entregas y permite que al final de cuentas se pueda construir el producto mínimo viable. En primer lugar, hay una reunión con el *Product owner*, el *Product owner* especifica las funcionalidades que debería tener la aplicación. El practicante levanta las historias, las puntua, les asigna una prioridad, las plasma en un tablero, y construye un calendario a cumplir con dichos entregables. La planeación está compuesta por varias fases: la primera fase es de configuración, como se quiere trabajar en el sprint, cuantos puntos tiene el sprint, bajo qué criterios se piensan estimar las historias, cuanto es la duración del sprint, y qué metodologías o qué estrategias se piensan implementar. Se documentan, y son puestas en práctica para la fase número dos.

La segunda fase consistió en realizar una investigación inicial de las arquitecturas a desarrollar, y un stack de las tecnologías a implementar. Se escogieron dos patrones: la *arquitectura hexagonal* y el *diseño dirigido por el dominio* (*ddd*). Son recurrentes en los proyectos más actuales de Ceiba, y se promueven como excelentes prácticas en el desarrollo que hacen de cualquier aplicación, una aplicación resiliente a los cambios, con un alto grado mantenibilidad y de fácil extensión. Naturalmente, para poder iniciar el desarrollo, se eligen las herramientas de soporte: el código fuente se almacenaría en Github (repositorio), el *backend* se desarrollaría en IntelliJ, y el *frontend* en Visual Studio Code, la integración continua en Travis CI, y el despliegue continuo en Heroku. Fue de vital importancia que las herramientas fueran de uso gratuito y que el desarrollo no generará un costo adicional que necesitará de las suscripciones anuales de un software con licencia. Las lenguajes con los que se eligió realizar el desarrollo, son Scala y Typescript. En Scala se codifica el *backend* de la aplicación. El backend tiene un framework, Play Framework (para la creación de apis), y una librería, Slick (para la inserción de los datos y persistencia). En Typescript se codifica el *frontend* de la aplicación. El *frontend* se construye a través de un framework, Angular 6. Se toma esta decisión por dos razones: Angular es una tecnología pujante en el mercado actual y Scala tiene las bondades de la programación funcional. Adicional a ello, son las herramientas con las que usualmente se trabaja en los proyectos de la empresa, un agregado que incrementa la mantenibilidad de la aplicación.

La tercera y última, es el desarrollo. En total fueron 7 funcionalidades las que se integraron en la aplicación (notas: solamente existe un tipo de usuario en la aplicación, se le llama un gestor y es la persona de talento humano que se encarga de gestionar el evento):

- Registro del evento: la inscripción de los eventos se podrá realizar por nombre, descripción, recursos (el presente a regalar) e insumos (materiales necesarios para iniciar o gestionar el evento, como lo es el personal adicional, o el transporte de los alimentos u obsequios). Es obligatorio el registro de al menos un recurso, y también es obligatorio que el recurso tenga un valor asociado. En dicho recurso es opcional incluir la cantidad disponible del mismo y además es posible elegir entre los recursos previamente ingresados cuál será el recurso favorito. Ambas funciones (de cantidad y de favoritismo), son articulaciones que soportan la funcionalidad que se explica a continuación, la gestión del evento.
- Diligenciamiento del evento: los gestores asignan los recursos, en tiempo real (esto es, una actualización casi inmediata en base de datos), a los empleados (los participantes) del evento. Esto involucra una búsqueda al LDAP de Ceiba. Como requisito principal, es necesario que un empleado tenga por lo menos un recurso asociado para ser vinculado a un evento. Fue importante contar con un mecanismo lo idóneamente inteligente para que el gestor tenga una retroalimentación del estado actual que tienen todos los recursos del evento. Para ello se manejaron varias estrategias: que recurso a recurso, la aplicación verifique y notifique la disponibilidad del mismo (si aplica, es decir, en caso tal que en el registro del evento, a dicho recurso se le hubiera asociado una cantidad máxima de recursos disponibles), y que exista un botón de actualización, a decisión del gestor, para actualizar la información que se despliega en pantalla (explicar por qué ?).
- Visualización de evento: el gestor puede visualizar el resumen de la administración de un evento. Esto es, el nombre, la fecha del dia que inició, la cantidad total de participantes registrados, la cantidad total de recursos consumidos versus el total (si aplica), y el detalle del participante: su nombre, la hora registrada de asistencia, los recursos asignados (con su nombre y cantidad), y el lugar en el que fue registrado junto con la persona que hizo el ingreso.
- Categorización de eventos: la categorización le permite al gestor acceder a un repositorio de eventos, por y sin terminar. Esta jerarquía diferencia las posibles acciones del evento: un evento por terminar tiene funcionalidades distintas a un evento que ya está finalizado. Los eventos que están en gestión son eliminables, y también son el punto de acceso hacia los detalles de gestión del mismo. Los eventos que ya se finalizaron, también son el punto de acceso hacia las estadisticas del mismo.
- Cerrar eventos: una vez que el gestor considera que el evento ya finalizó, y que por lo tanto no hay más participantes que ingresar, entonces es porque es el momento oportuno de hacer el cierre. Al cerrar un evento, ya pasa de estar en un estado en “gestión”, a un estado “finalizado”. Un evento al estar en estado finalizado, no puede volver a su estadio anterior (realmente iría en contra del propósito eliminar esta información que es de valiosa importancia para el objetivo final de la aplicación). Y es así como esta función se complementa con la “categorización de eventos”; cuando el evento tiene dicha marca, es entonces cuando se filtran automáticamente en la vista perteneciente a la funcionalidad

"categorización de eventos". Los eventos se cierran a través de la vista hecha en la historia "diligenciamiento del evento".

- Eliminar: como respaldo ante las equivocaciones de un gestor a la hora de crear un evento o ante la posibles cancelaciones, es sensato pensar que los eventos puedan ser descartados a demanda. Esta funcionalidad se acompaña del cierre, y está disponible en la misma vista de la "visualización de eventos".
- Inicio de sesión: los gestores hacen parte del personal de gestión humana en Ceiba. Por lo tanto, se tiene que tener un mecanismo de autenticación que restrinja el acceso a cualquier usuario (como lo es el desarrollador regular), es decir, con la potestad de reconocer en el LDAP de la empresa. Se inicia la sesión con el correo empresarial de Ceiba y la contraseña de red.
- Navegación: ha de construirse una navegación que integre las funcionalidades de la aplicación, es decir, la codificación de las componentes gráficas y ayudas visuales que se encarga del enrutamiento entre las vistas. Esto es un panel lateral, y menú de opciones en las vistas correspondientes a las funcionalidades principales (registro, diligenciamiento, visualización y categorización de eventos).
- Estadísticas de evento: las estadísticas de evento son el sumario de toda la información recolectada durante la vigencia del mismo. Contiene: quienes y cuantos asistieron, los gestores que administran el evento, los gastos monetarios totales y el inventario final sobre los recursos consumidos y los insumos adquiridos.

Algunas historias no se estimaron de la mejor forma, iniciar fue especialmente complejo debido a que se presentaron algunos retrasos en el desarrollo normal de las historias. Entender la arquitectura hexagonal y el diseño dirigido por el dominio fue particularmente difícil, las primeras tareas se tomaron más tiempo de lo acordado y generaron un retraso en el proyecto. La arquitectura hexagonal se implementó a la regla, sin embargo, no fue el caso para el diseño dirigido por el dominio. Probablemente dar un porcentaje fijo de implementación o de integración de DDD sea una tarea muy difícil, DDD es una técnica que ha sido ampliamente explorada, su literatura es vasta y sus procedimientos o metodologías son muchas. Por lo tanto aquí se implementó lo más habitual, lo más "ordinario", y lo necesario para ser DDD. Al respecto también conviene decir que el aprendizaje de los lenguajes y sus frameworks, no fue inmediato y solicitan un esfuerzo adicional.

Existieron algunos impedimentos que, de no haber sido por la arquitectura, hubieran imposibilitado el desarrollo normal de las historias; la conexión al LDAP. La coneccion al LDAP se simula, hasta cierto punto, y esto fue debido a que la herramienta es restringida a los administradores de Ceiba. Los cambios han de implementarse cuando la aplicación esté en producción o el desarrollo decida continuarse. Su implementación no tendría por que afectar el dominio, y la aplicación está construida de tal forma que tales cambios no implican refactorizaciones, si no que, por el contrario, hace que sea una integración sin esfuerzo.

Las tres capas de la arquitectura hexagonal se constatan en la carpeta *app* del proyecto, más una carpeta adicional (la excepción): la aplicación, el dominio y la infraestructura (Figura 2). Iniciemos

por la capa intermedia y sus componentes elementales; en la aplicación están las acciones, los objetos de transferencia y los transformadores.

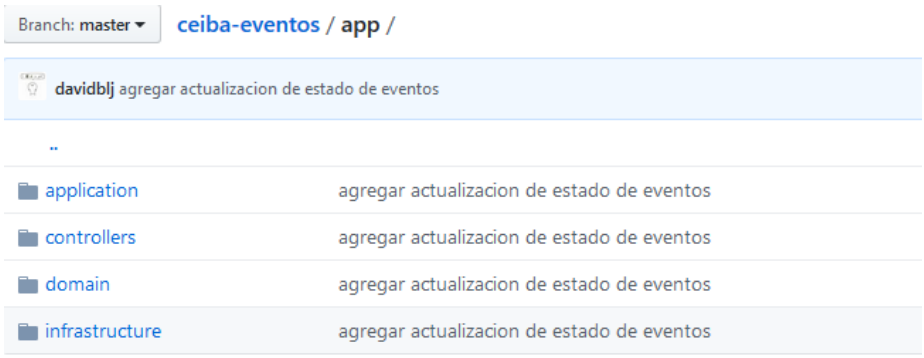

Figura 2. Arquitectura y paquetería principal.

Es preciso considerar el flujo de al menos una funcionalidad a través de sus capas y discernir los patrones en algo más entendible: la inserción de un participante. Los objetos de transferencia son modelos que no se conciben en la definición del dominio, y comunican el mundo exterior con la aplicación. Se define la clase *Attendant*, como un objeto de transferencia que encapsula la información entrante del participante: su nombre, la ubicación, y la lista de recursos asignados. Los recursos asignados son, a su vez, el identificador de un recurso, y la cantidad compartida o la cantidad obsequiada (*Figura 3*). Su representación es completamente distinta a la lógica del negocio, y de igual forma, diferente en las bases de datos. En efecto, necesitamos transformar ese objeto, a un objeto que pueda entender el dominio, y desde luego nacen los transformadores. Los transformadores son simples, tan solo mapean datos (*Figura 4*), y se necesita de un transformador que traduzca el participante a un objeto del dominio; el participante del dominio. Y luego están las acciones. Las acciones se empaquetan por entidades. La función *singUpAttendant* registra un participante, no sin antes convertir el objeto que proviene de la infraestructura al dominio (Figura 5) por medio del transformador. Lo que retorna el dominio se transfiere a la capa más externa, la infraestructura. Así se integran estas tres componentes.

```
case class Attendant(eventId: Int, employeeId: Int, locationId: Int, assignedResources:
List[AttendantAssignedResource])

case class AttendantAssignedResource(resourceId: Int, sharedAmount: Int)
```

Figura 3. Objetos de transferencia.

```
object AttendantTransformer {

  def toDomainObject(attendant: AttendantAppObject): Attendant = {

    val domainAttendantAssignedResourceObject = attendant.assignedResources
        .map(assignedResource => {
            AttendantAssignedResource(assignedResource.resourceId, None,
                                      assignedResource.sharedAmount, None,
                                      None)
    })

    Attendant(attendant.eventId, None, attendant.employeeId, None,
              attendant.locationId, domainAttendantAssignedResourceObject,
              None, None)
}
```

Figura 4. Objetos transformadores.

```
 def execute(attendant: Attendant): Future[Int] = {

    val domainAttendantObject = AttendantTransformer.toDomainObject(attendant)
    organizer.signUp(domainAttendantObject)
}
```

Figura 5. Acciones del participante.

Antes de iniciar, en la infraestructura, señalemos la simpleza que tiene su empaquetamiento (*Figura 6*); sin necesidad de entrar en detalles a escrutinizar el código, podemos manifestar que la aplicación utiliza 4 tecnologías o integraciones, ajenas al dominio, pero vitales y que complementan el sistema: Play, Slick, Juice y LDAP. Es Slick quien proporciona la implementación correspondiente a los temas de la persistencia, y Play, la REST API que configura, expone el servidor, y recibe las peticiones que provienen del mundo externo. Subyace en estos detalles, que las tecnologías pueden intercambiarse sin afectar el core de la aplicación, la lógica del negocio está en un lugar completamente aparte, en el dominio. El empaquetamiento del framework tiene 2 clases y 3 paquetes (Figura 7). Play framework necesita de los denominados *writes and reads*, funciones

que traducen los mensajes JSON en objetos de Scala; estas rutinas no solamente transforman, sino que también validan (*Validator.scala)*. Es importante manipular errores, los errores que se producen al ejecutar el dominio necesitan de una estructura, esta estructura la proporciona la clase *ErrorHandler*, y es la respuesta que obtiene el cliente cuando surgió lo inesperado. Y desde luego y lo más esencial, los controladores de Play que procesan las peticiones HTTP. Aún no se tienen los medios para cambiar la ruta por defecto en la que están ubicados. Fue Play quién género el cascarón de la aplicación, y es Play, al interior de sus librerías, que determina el lugar en el que se van a exponer los endpoints: bajo la carpeta app (la raíz del código fuente), y no en la carpeta infraestructura (de la arquitectura hexagonal). Desde luego, la carpeta controllers (Figura 2), debería estar en la ruta *infraestructure/play/controllers.*

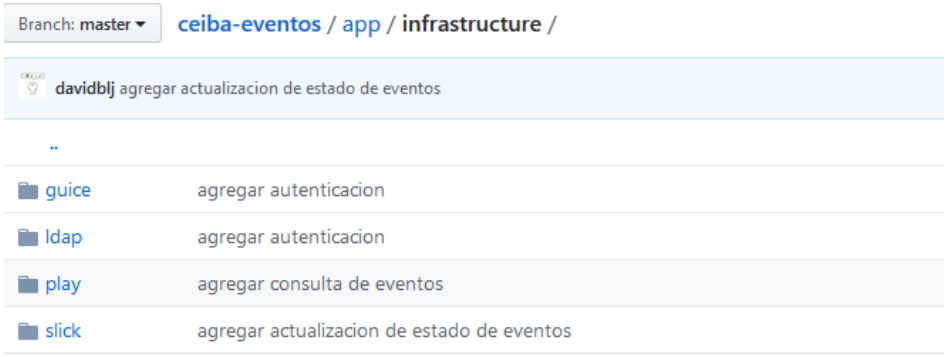

Figura 6. Paquetería de la capa infraestructura.

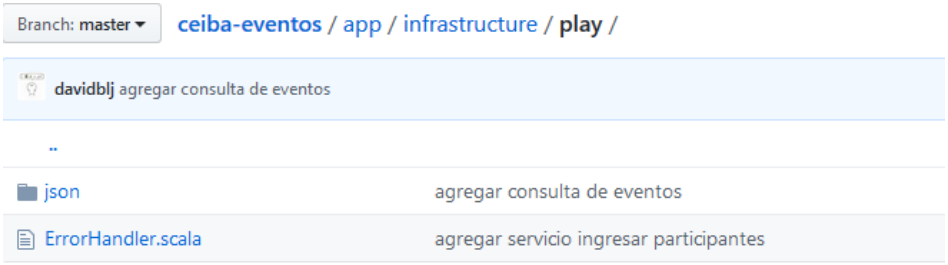

Figura 7. Paquetería Play Framework

La ventaja es evidente, las características del framework se expresan a través de la paquetería y se centralizan en un solo lugar (Figura 7), y se precisa advertir entonces que, sin mucha repercusión, este no es el caso para los controladores de la aplicación. Retomando, la inserción de un participante se visualiza en la figura 8. El controlador llama la acción *signUpAttendant*, y a su vez utiliza los modelos de transferencia, funciones y componentes propias de la capa de la aplicación.

```
def signUpAttendant(): Action[Attendant] = Action.async(validator.validateJson[Attendant]) {

    request => {

      val attendant = request.body
      SignUpAttendant.execute(attendant).map(_ => {
        NoContent
      })
    }
}
```

Figura 8. Controlador para la inserción de un participante.

Posteriormente, al final del proceso de inserción, está el dominio. Su arquitectura se gobierna por DDD. En la capa más contigua, es decir, en la aplicación, cualquier acción que necesite el dominio, tan solamente le corresponde usar lo que se exponga en los servicios (principalmente), e inicializar los objetos de valor y agregados que se necesitan para ensamblar su funcionalidad y así cumplir con su propósito final. Debido al diseño dirigido por el dominio, agregar un participante es elemental (Figura 9), y el grueso de la rutina, se oculta en los repositorios de la infraestructura.

```
def signUp(attendant: Attendant): Future[Int] = {
    eventRepository.add(attendant)
}
```

Figura 9. Inserción del participante en la capa del dominio.

Se han definido las tres capas; sus contenidos y responsabilidades, queda entonces por identificar cómo se logra una comunicación sin acoplar el código a implementaciones específicas: a través de la inyección de dependencias. La capa del dominio empieza por inyectar, definir y usar las interfaces (*Figura 10*). Examinemos cómo funciona esta dinámica en los repositorios. La implementación de un repositorio se define en la capa de la infraestructura (*Figura 11*), pues es la infraestructura quien define y, a su antojo, usa las tecnologías. Juice en tiempo de compilación (*Figura 12*), inyecta o suministra las clases que implementan las interfaces. Este procedimiento es primordial: sin importar que haga la infraestructura, el dominio no tiene por que sufrir cambio alguno, utiliza exclusivamente las interfaces y de esta forma es totalmente ignorante al mundo externo, en este caso, a la persistencia. El código se ha cerrado a la modificación, pero sigue abierto a la extensión, y el núcleo de la aplicación se limita a calcular rutinas exclusivas al negocio, y no a la inserción o la consulta de la información. Así se construye la arquitectura hexagonal, una técnica que se basa en una comunicación característica a través de interfaces, y en cascada por todas sus capas.

```
trait EventRepository {
  // ...
  def add(attendant: Attendant): Future[Int]
  // ...
}
```

Figura 10. Interfaces del dominio para la inserción de un participante.

```
class SlickEventRepository extends HasDatabaseConfigProvider[JdbcProfile] with EventRepository {

  // ...

  override def add(attendant: Attendant): Future[Int] = {
    slickAttendantRepository.add(attendant)
  }

  // ...
}
```

Figura 11. Implementación de las interfaces del repositorio.

```
class RepositoriesModule extends AbstractModule {

  override def configure(): Unit = {

   bind(classOf[EventRepository]).to(classOf[SlickEventRepository])
   // ...
}
```

```
class Organizer @Inject() (eventRepository: EventRepository, ...) {
  // ...
}
```

Figura 12. Configuración de Juice y la inyección de dependencias.

Hasta aquí se examinó la arquitectura estructural, y falta por revisar lo que se encuentra al interior de hexágono y la estructura que tiene la lógica de negocio como el resultado del diseño dirigido por el dominio. Se implementaron cuatro conceptos fundamentales: los servicios, los agregados, las entidades y los objetos de valor (*Figura 13*). En los servicios existe una clase que por el momento expone, en su mayor parte, las funcionalidades principales: la clase *Organizer.scala*. La inserción de un participante se expone en esta clase (Figura 8), y es debido a los agregados, que el código pueda ser tan expresivo; insertar un participante es en realidad guardar en un repositorio de eventos la inscripción del empleado. La entidad principal es la clase *Event.scala*. Los eventos se tratan como elementos con identidad única, y es a través del evento que se realizan las operaciones principales, es la entidad raíz. Como la entidad raíz, a través de un repositorio, se efectúa la inserción. El repositorio ya tendrá su propia implementación, y será tan extensa como lo sea el modelo de datos (Figura 14).

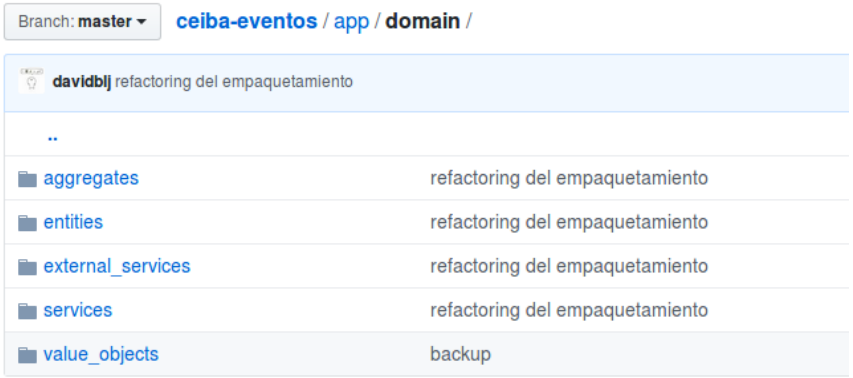

Figura 13. Paquetería de la capa del dominio.

```
class SlickEventRepository @Inject() (val slickAttendantRepository: SlickAttendantRepository) {

  override def add(attendant: Attendant): Future[Int] = {
     slickAttendantRepository.add(attendant)
  }
}

class SlickAttendantRepository @Inject(val attendantAssignedResourceRepository:
                                       SlickAttendantAssignedResourceRepository) (

  def add(attendant: Attendant): Future[Int] = {

    def createAttendant(): Future[Attendant] = {
      // ...
    }

    for {
      attendant <- createAttendant()
      _         <- attendantAssignedResourceRepository
                   .add(attendant.assignedResources, attendant.attendantId.get)
    } yield attendant.attendantId.get
}

// implementacion attendantAssignedResourceRepository ...
```

Figura 14. Implementación del repositorio para la inserción de participantes con Slick.

La entidad evento, en una base de datos relacional, es en realidad un conglomerado de varias tablas, sin embargo, esta implementación es transparente el dominio, y hace que la lógica del negocio sea totalmente transparente, y manifieste su verdadera intención.

Al finalizar el proyecto el practicante hizo 57 commits en el *frontend*, y 51 *commits* en el backend, para sumar un total de 108 commits total. Se realizaron 9 historias de usuario en 6 sprints. Se codificaron alrededor de 28,000 líneas en el frontend (incluyendo 10,000 lineas auto-generadas por Angular), y 6,000 líneas en el backend (incluyendo 1,000 líneas auto-generadas por Play). Las estadísticas se pueden evidenciar en la figura 15 estadísticas.

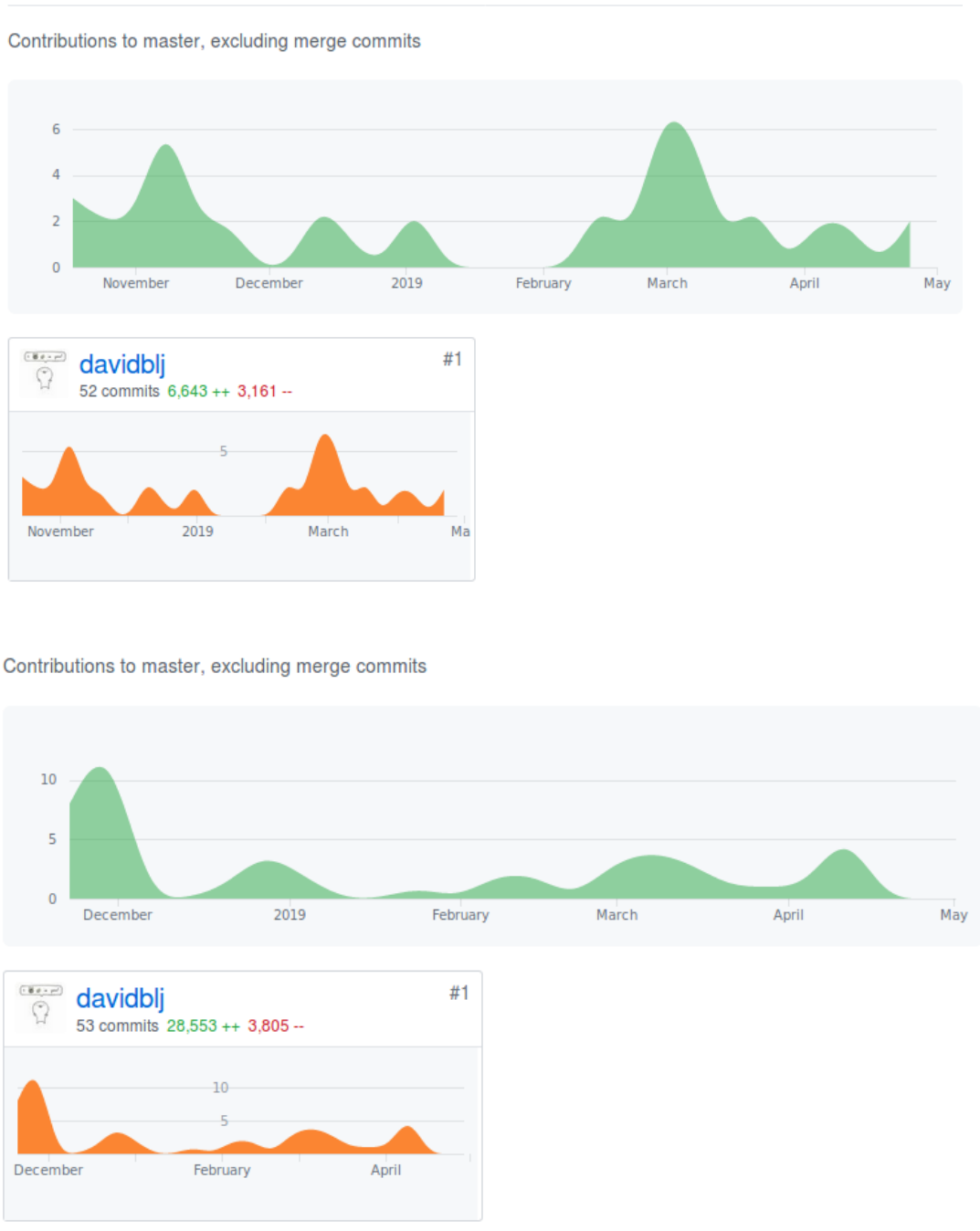

Figura 15. Evidencias de los repositorios; el backend y el frontend.

Conclusiones

El desarrollo del proyecto me permite razonar en las siguientes conclusiones:

- Scrum es una metodología esencial en el desarrollo de aplicaciones, inclusive si es de elaborarse un proyecto personal, pues las historias de usuario no solamente establecen un roadmap a desarrollar, si no que permiten estimar el proyecto en entregas comprensibles. De la misma forma, son prácticas y sirven como plan de apoyo o respaldo cuando no se entregan las historias con el calendario establecido; son la crónica del proyecto, en ellas se plasman los inconvenientes, las dificultades y los obstáculos presentados, para luego rendir cuentas y diseñar una estrategia que permita entrega el mínimo viable, sin afectar demasiado el resultado final.
- La arquitectura hexagonal es un patrón casi que mandatario en las aplicaciones web; la comunicación entre las capas es ideal para que las clases tengan un bajo acoplamiento y alta cohesión. Detrás de la utilización de interfaces como mecanismo de integración, se descubre la posibilidad de intercambiar, sin gran esfuerzo, la tecnologías de la aplicación.
- La arquitectura hexagonal se complementa con DDD, y es una combinación que al final permite que las responsabilidades sean mas atomicas, que el empaquetamiento sea mas claro, y que el código sea fácil de entender en comparación a unas estructuras o arquitecturas más ingenuas que no constituyen reglas en pro de la mantenibilidad, la extensión y en general, al prolongamiento de vida de la aplicación.
- DDD modela el dominio y establece una serie de reglas y condiciones que permiten que la lógica del negocio, a través de una estructura pueda separar adecuadamente los niveles de abstracción y se convierta en algo más comprensible.
- Play Framework es una herramienta apropiada para la construcción de REST APIs en Scala, con un alto nivel de madurez y los mecanismos necesarios y suficientes para exponer servicios web, configurar el servidor, y manipular la asincronía y escalabilidad de la aplicación, iniciar el desarrollo es facil en comparacion a frameworks similares.
- Angular es un framework que permite modularizar la aplicación en trozos pequeños fáciles de digerir, y aunque su aprendizaje es complejo, y requiere entender múltiples conocimientos en varios lenguajes (Typescript, Javascript, HTML y SCSS), los esfuerzos valen la pena; con el anhelo de crear agradables interfaces de usuario, la aplicación final es la integración fluida de diversas funcionalidades en una sola página, que es fácil de aprender y utilizar.

Referencias bibliográficas

Angular (2019) *Architecture overview*

Recuperado de: https://angular.io/guide/architecture

Apiumhub (2019) *An introduction to domain driven design & its benefits*

Recuperador de: https://medium.com/@Apiumhub/an-introduction-to-domain-driven-design-its-benefits-ff22e4e4b743

Evans, E (2003) *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Estados Unidos: Addison Wesley

Fidao, C. (2019) *Hexagonal Architecture*

Recuperado de: https://fideloper.com/hexagonal-architecture

Garrido, J.M. (2019) *Ports and Adapters Pattern (Hexagonal Architecture)*

Recuperado de: https://softwarecampament.wordpress.com/portsadapters/

Kulbacki C. (2019) *What's More Popular Than Scrum?*

Recuperado de: https://scrumstar.com/articles/the-most-popular-agile-methodologies

Lelonek, K. (2019) *DDD building blocks*

Recuperado de: https://blog.lelonek.me/ddd-building-blocks-for-ruby-developers-cdc6c25a80d2

Play (2019) *What is Play*

Recuperado de: https://www.playframework.com/documentation/2.7.x/Introduction