Universidad ORT Uruguay

Facultad de Ingeniería

GxReq: Herramienta de trazabilidad de requerimientos para GeneXus

Entregado como requisito para la obtención del título Ingeniero en Sistemas

Álvaro Nicoli - 220159

Federico Banchero - 217730

Nicolás Eirin - 200111

Tutor:

Darío Macchi

2022

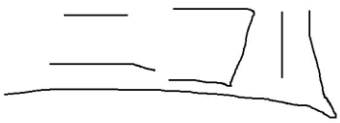
Declaraciones de Autoría

Nosotros, Federico Banchero, Nicolás Eirin, Alvaro Nicoli, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que: La obra fue producida en su totalidad mientras realizábamos el proyecto de grado de la carrera Ingeniería en Sistemas;

- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Federico Banchero
17 de marzo de 2022

Nicolás Eirin
17 de marzo de 2022



Álvaro Nicoli
17 de marzo de 2022

Agradecimientos

Queremos agradecer a nuestro tutor Darío Macchi por su dedicación y compromiso a lo largo de todo el proyecto, por habernos guiado y aportado sus consejos y conocimiento académico durante el transcurso del proceso.

A nuestras familias y amigos, quienes confiaron en nosotros, nos apoyaron incondicionalmente y fueron fuente de inspiración en todos estos años, y a lo largo de nuestra vida en general.

A nuestro cliente, GeneXus, quien nos dio la posibilidad de llevar a cabo este proyecto, nos brindó capacitación, nos dio una retroalimentación constante sobre nuestro avance, haciendo valiosas sugerencias y contestó con la mejor disposición todas nuestras dudas técnicas.

A docentes, catedráticos, funcionarios de la Universidad ORT Uruguay y personas que conocimos dentro de la Universidad en general, que de alguna manera influyeron en nuestra vida académica y profesional a lo largo de estos años.

Finalmente, a nuestros colegas de trabajo, que se interesaron e hicieron aportes al proyecto.

Abstract

Este proyecto surge del problema presentado por la empresa GeneXus al laboratorio de *Software Factory* de la Universidad ORT Uruguay.

GeneXus es una empresa de *software* de Uruguay fundada en 1984 que se dedica a la consultoría y conjuntamente desarrolla un ecosistema de herramientas para el desarrollo de *software low code* con el mismo nombre.

La filosofía de GeneXus se basa en el desarrollo iterativo incremental, tomando la visión de los usuarios, y capturando su conocimiento en la *knowledge base* o base del conocimiento en español. Con esta base del conocimiento, GeneXus es capaz de generar y mantener automáticamente la arquitectura de un entregable de *software*, creando por sí mismo todo lo necesario, como bases de datos, entidades de dominio, entre otras.

Si bien el producto GeneXus tiene una evolución de muchos años en el mercado, siendo su primer lanzamiento en 1989 con GeneXus 1.0, a la fecha no cuenta con un mecanismo para dar soporte a requerimientos escritos de manera que garanticen: la calidad de implementación, puedan actuar como artefacto para ser auditados y que puedan ser discutidos, compartidos, y entendidos por todos los involucrados.

Por lo expuesto anteriormente, entra en juego este proyecto, que se basa en una gran prueba de concepto que mediante una integración con las herramientas de GeneXus intenta permitir especificar una necesidad de negocio escrita correctamente en el lenguaje del cliente, con suficiente estructura para ser: útil en términos de automatización, completa, consistente, poco ambigua para actuar como contrato y flexible para adecuarse a la prototipación incremental.

Esta prueba de concepto consiste por un lado en un mecanismo de extensión para el entorno de desarrollo de GeneXus, que se integra con la suite de tests de dicho producto GxTest y GeneXus

Server. De ahora en adelante este mecanismo de extensión se denominará Extensión de GeneXus o simplemente la Extensión.

Por otro lado, cuenta con un sistema externo que consiste en una página web, que de ahora en más se denominará Cliente de Requerimientos y su respectivo *backend*. El Cliente de Requerimientos contiene un sistema de manejo de usuarios, proyectos, *features* y un editor de texto Monaco Editor[1] que también está integrado en el Mecanismo de Extensión, donde los usuarios pueden escribir requerimientos y estos requerimientos pueden ser exportados desde el entorno de desarrollo GeneXus por medio de la conexión entre el Mecanismo de Extensión y el *backend* del Cliente de Requerimientos.

[1] https://microsoft.github.io/monaco-editor/

Palabras Clave

Gherkin, Escenario, Ejemplo, Transacción, Procedimiento, Trazabilidad, Requerimiento, Plataforma Low Code no Code, GeneXus, GeneXus Server, Cliente de Requerimientos, Extensión, React Admin, JSON API .NET, Feature, Given, When, Then, Monaco, GxReq, Knowledge Base.

Glosario

- **Backend:** Pieza de *software* con la que el usuario no interactúa directamente. Generalmente encargada del almacenamiento y procesamiento de datos.
- **Backlog:** Lista de funcionalidades que el sistema a desarrollar debe tener.
- **BDD:** Acrónimo de *behavior-driven development* refiere a una metodología de *software* ágil donde se escribe el comportamiento de un sistema de software en base a ejemplos.
- **CLI:** Sigla en inglés para "*Command Line Interface"*. En español: "Interfaz de línea de comandos".
- **Commit:** Comando de Git que permite confirmar una instantánea [1].
- **Cucumber**: Herramienta de *software* que apoya el desarrollo haciendo uso de BDD.
- **Draw IO:** *Software* que permite la creación de múltiples diagramas. Es una aplicación web con diversas integraciones con otras herramientas y que permite el trabajo colaborativo entre múltiples usuarios, de forma concurrente.
- **Épica:** Es una gran historia de usuario que, o es demasiado grande para ser finalizada en una sola iteración, o que es lo suficientemente grande como para dividirla en historias de usuario más pequeñas [2].
- **Escala de Fibonacci:** Es una secuencia de números, fue utilizada como escala para estimar las tareas a realizar.
- **Escenario:** Es un elemento de la estructura de Gherkin, representa un ejemplo concreto que ilustra una regla de negocio [3].
- **Feature:** Es un elemento de la estructura Gherkin, esta proporciona una descripción de alto nivel de una función de *software* y agrupa los escenarios relacionados [3].

- **Frontend:** Pieza de *software* con la que el usuario interactúa directamente. Es la interfaz entre el usuario y el *backend* [4].

- **GeneXus:** Herramienta para el desarrollo de *software low code no code* orientada al entorno empresarial para aplicaciones web, dispositivos móviles y aplicaciones para escritorio de Windows [5].

- **Gherkin:** Lenguaje legible por el negocio usado para expresar el comportamiento de un sistema de *software.* El lenguaje puede ser interpretado por la herramienta de automatización Cucumber [6].

- **Git:** Herramienta de *software* de control de versiones.

- **GitFlow:** Modelo de creación de ramas en Git.

- **Historia de Usuario:** Explicación general e informal de una función de software escrita desde la perspectiva del usuario final o cliente [7].

- **HTTP:** Protocolo de comunicación que permite las transferencias de información a través de archivos en Internet [8].

- **Kanban:** *Framework* para metodologías de desarrollo ágiles.

- **Knowledge base:** En el contexto de GeneXus, es el nombre que reciben los proyectos de *software*.

- **Plataforma de desarrollo *Low Code/No Code*:** Herramienta que provee un entorno de desarrollo para crear aplicaciones de *software* a través de una interfaz gráfica en vez de escribir código [9].

- **Monaco**: Editor de código que impulsa Visual Studio Code. Tiene licencia MIT y es compatible con los principales navegadores.

- **Notion:** *Software* de productividad para el manejo de proyectos y tomar notas. Permite a los miembros de un equipo coordinar fechas límites, objetivos y asignación de tareas [10].

- **NuGet:** administrador de paquetes que permite a los desarrolladores compartir código reutilizable.

- **OneDrive:** Almacenamiento personal en la nube, permite sincronización de documentos entre múltiples dispositivos y usuarios de la aplicación [11].

- **Pipeline:** En *software*, se define como una cadena de elementos de procesamiento, donde la salida del actual es la entrada del próximo a procesar [12].

- **Procedimiento:** En GeneXus, un procedimiento define un programa que implementa un algoritmo. Este puede incluir el acceso a la base de datos y la actualización de datos [13].

- **Pull request:** Mecanismo que ayuda a debatir los cambios propuestos antes de integrarlo a la rama principal del repositorio.

- **QA:** Acrónimo de *Quality Assurance.* Conjunto de actividades que promueven la calidad tanto del producto como del proceso.

- **Requerimiento:** Propiedad determinada que un producto de *software* debe satisfacer.

- **Reflection:** Es la capacidad de un proceso para examinar y modificar su propia estructura y comportamiento.

- **Release:** Liberación de una versión del producto de *software*, que puede ser pública o privada.

- **Repositorio:** En el contexto de herramientas de control de versiones, hace referencia a la estructura que almacena los metadatos de un conjunto de archivos o directorios [14].

- **REST:** Es el acrónimo de *Representational State Transfer* y un estilo de arquitectura para sistemas hipermedia distribuidos [15].

- **SCM:** Acrónimo de *Software Configuration Management*. Área de la ingeniería *software* encargada de gestionar los elementos de la configuración.
- **Scrum:** *Framework* para metodologías de desarrollo ágiles.
- **SMART:** Acrónimo de *Specific*, *Measurable*, *Achievable, Relevant y Time-Bound.* Establece criterios para la definición de objetivos de un proyecto.
- **Snippet:** Pequeña parte de código reutilizable.
- **Sprint:** Periodo breve y definido de tiempo en el que el equipo trabaja en el proyecto.
- **Sprint Planning Meeting:** Reunión previa al comienzo del *sprint,* en esta se define que se realizará y cómo se hará.
- **Story point:** Unidad de medida usada para la estimación del esfuerzo que requerirá una tarea.
- **Syntax Highlighting:** Resaltado de sintaxis.
- **Transacción:** En GeneXus, una transacción describe un objeto o actor de la realidad, definiendo la estructura de la base de datos, las reglas comerciales y la interfaz de usuario para la manipulación de datos [16].
- **Trello:** Aplicación web que permite la creación de tableros al estilo Kanban, listas de verificación, entre otras funcionalidades.
- **UI:** Acrónimo de *User Interface.*
- **UpperCamelCase:** Estilo de escritura que aplica a palabras compuestas.
- **UX:** Acrónimo de *User Experience.*

Índice

Índice de figuras

Índice de tablas

1 Introducción

Este documento describe el desarrollo del proyecto realizado como requisito para la obtención del título de Ingeniero en Sistemas de la Universidad ORT Uruguay. Este proyecto tuvo la duración de un año, desde marzo de 2021 hasta marzo de 2022.

Este proyecto consta de una prueba de concepto propuesta por GeneXus, la cual tiene como objetivo investigar y realizar un desarrollo que permite agregar una trazabilidad entre la especificación de requerimientos y los incrementos resultantes de estos.

1.1 Estructura del documento

A continuación, se presenta un breve detalle de los diferentes capítulos con los cuales está constituido este documento.

1.1.1 Problema

Se presenta el problema para el cual se desarrolla la prueba de concepto, describiendo el contexto en el que se da, cada una de las actividades y herramientas utilizadas para su justificación y definición.

1.1.2 Solución

En este capítulo se presenta la solución propuesta para el problema antes descrito, se describen las principales funcionalidades desarrolladas, el alcance del proyecto, cuáles serían los grupos interesados y por último, un listado con los entregables brindados al cliente.

1.1.3 Metodologías

Se presentan las características de este proyecto, la metodología de trabajo, el cómo manejamos la incertidumbre, y por último las diferentes etapas por las que pasó el proyecto.

1.1.4 Ingeniería de requerimientos

Se describe el proceso empleado para definir los requerimientos funcionales y no funcionales, en conjunto con una descripción de estos.

1.1.5 Arquitectura y diseño

Se detallan cuáles fueron los motivos que guiaron la arquitectura, y se presenta ésta. También se describe cómo esta arquitectura ayuda a satisfacer los atributos de calidad importantes para nuestro proyecto. Se detallan y justifican las tecnologías elegidas.

1.1.6 Gestión del proyecto

En el presente capítulo se detalla la gestión del proyecto, describiendo así las herramientas utilizadas para esto, las etapas y principales hitos, la aplicación de diferentes marcos de trabajo y la gestión de riesgos y comunicación empleadas.

1.1.7 Gestión de la calidad

En este capítulo se define que es la calidad para este proyecto, se describen las prácticas aplicadas para el aseguramiento de la calidad, como son aplicación de estándares, revisiones y pruebas.

1.1.8 Gestión de la configuración

Se detallan los elementos de la configuración identificados. También se describen las herramientas utilizadas, la organización de los repositorios remotos y las técnicas utilizadas para la gestión de versiones y cambios.

1.1.9 Conclusiones

Este capítulo se refiere al cierre del proyecto, en este se evalúa el cumplimiento de los objetivos, se presentan las conclusiones y lecciones aprendidas. También se detallan los que consideramos los siguientes pasos a dar.

1.2 Descripción del cliente

El cliente del proyecto es GeneXus, empresa multinacional de *software* uruguaya. GeneXus fue fundada en 1984 por Breogán Gonda y Nicolás Jodal.

GeneXus ha creado una plataforma de desarrollo, que tiene el mismo nombre de la empresa, que lleva más de 30 años en el mercado, con la premisa de simplificar el desarrollo de *software,* automatizando todo lo automatizable. GeneXus sigue el paradigma *low code no code*.

Actualmente, la plataforma GeneXus, es usada en más de 9000 empresas y en más de 50 países.

GeneXus logra esto permitiendo que el desarrollador especifique un modelo abstracto e independiente de la tecnología, representativo de las necesidades de negocio.

Dado estos modelos abstractos, GeneXus genera código de forma automática. Para esta generación de código hace uso de generadores de código y algoritmos de IA. La generación de código se puede dar en diferentes lenguajes y con diferentes gestores de base de datos. GeneXus brinda facilidades para el uso de metodologías ágiles y DevOps.

Con esto se logra aumentar la productividad en el desarrollo de *software,* a la vez que las necesidades de negocio son el centro en todas las etapas del *software.*

1.3 Descripción del equipo

El equipo de trabajo está constituido por tres integrantes, los cuales son estudiantes de la carrera Ingeniería en Sistemas.

Figura 1: Integrantes del equipo

Cabe destacar que el equipo ha trabajado en conjunto con anterioridad, en diversas materias de la carrera. Esto hace que el trabajo sea más dinámico. También cabe destacar que cada uno de los miembros tiene habilidades variadas y diferentes, las cuales se complementan entre ellas.

Cada uno de los integrantes tiene experiencia laboral profesional en empresas del rubro de IT, a continuación se presentan los empleos actuales:

- **Federico Banchero**: *Ssr. Developer* en *Urudata Software*
- **Nicolas Eirin**: *Sr. Developer* en *Urudata Software*
- **Alvaro Nicoli**: *Sr. Developer* en *Vindow*

1.4 Objetivos

En esta sección se presentan los objetivos establecidos por el equipo, los cuales se dividen en tres subsecciones: objetivos académicos, objetivos del producto y, por último, objetivos del equipo. Estos objetivos permiten al equipo tener una meta en común.

Dada la importancia de éstos en el proyecto, se decidió usar la técnica SMART, para la definición de los mismos. SMART define que los objetivos deben ser Específicos, Medibles, Alcanzables, Relevantes y Temporales o Acotados en el Tiempo [17].

1.4.1 Objetivos académicos

1.4.1.1 Entregar un proyecto satisfactorio

Descripción: Entregar un proyecto que cumpla con las expectativas académicas, en cuanto a complejidad y desafíos técnicos permita la obtención del título de Ingeniero en Sistemas.

Medida de éxito: La nota del proyecto debe ser satisfactoria para la obtención del título, debe existir una carta de conformidad del cliente.

Propósito: Entregar el mayor valor al cliente y obtener una satisfacción personal por haber cumplido con el desafío.

1.4.1.2 Poner en práctica conocimientos adquiridos durante la carrera

Descripción: Poner en práctica los conocimientos adquiridos durante la carrera en un contexto real.

Medida de éxito: Haber utilizado más de un conocimiento adquirido en la carrera en el transcurso del proyecto. Entendiendo por conocimiento, a un conocimiento técnico, de gestión o habilidad blanda.

Propósito: Potenciar las habilidades adquiridas y darles un fin práctico.

1.4.1.3 Aprender nuevas tecnologías

Descripción: Cada miembro del equipo debe aprender una tecnología de la cual no tiene conocimiento previo.

Medida de éxito: Reconocer estructuras básicas propias de la tecnología y poder desarrollar un proyecto que englobe los conocimientos básicos de la misma.

Propósito: Manejar un *stack* de tecnología más amplio.

1.4.1.4 Aprender nuevas herramientas

Descripción: Cada miembro del equipo debe aprender una herramienta de productividad que actualmente no maneja.

Medida de éxito: Tener conocimientos básicos sobre una nueva herramienta de productividad.

Propósito: Conocer herramientas que permitan hacer el trabajo de una manera más óptima.

1.4.1.5 Aprender nuevos procesos

Descripción: Cada miembro del equipo debe aprender a utilizar Popcorn Flow, Dual Track Scrum y Kanban. Procesos que no fueron utilizados en asignaturas de la Universidad.

Medida de éxito: Tener conocimiento y poder aplicar dichos procesos en otro ámbito.

Propósito: Contar con conocimientos de procesos alternativos para manejar proyectos.

1.4.2 Objetivos del producto

1.4.2.1 Calidad Interna

Descripción: El equipo identificó de forma temprana que era sumamente importante mantener una buena calidad en el producto desarrollado, para asegurar este objetivo se establecen acciones a realizar.

- **Revisiones de a pares**
- **Cobertura de pruebas unitarias**
- **Pruebas *End to End***
- **Definición de historias de usuario**

Medida de éxito:

- **Revisiones de a pares**: Cada cambio en el código, y antes de que se fusionen con la rama de código estable, se establece que ambos miembros del equipo restante deben revisar el código realizado y aportar comentarios al respecto.

- **Cobertura de pruebas unitarias**: El equipo estableció tener un porcentaje no menor a 80% de cobertura de pruebas unitarias, en lo que respecta al código de la extensión GeneXus.
- **Pruebas *End to End***: Para el *backend* del Cliente de Requerimientos se establece que deben existir pruebas *End To End.*
- **Definición de historias de usuario**: Las historias de usuario deben contener criterios de aceptación que sean claros, detallados y verificables, garantizando así la completitud. En efecto la historia de usuario bien definida debe cumplir con los criterios INVEST[2].

Propósito:

- **Revisiones de a pares:** Prevenir errores en etapa de desarrollo y fomentar que todo el equipo conozca toda la solución.
- **Cobertura de pruebas unitarias:** Reducir la cantidad de errores, fomentar el *refactoring* y servir como documentación para futuros desarrolladores.
- **Pruebas *End to End:*** Asegurar la correctitud, la funcionalidad y los flujos de negocio del sistema.
- **Definición de historias de usuario:** Crear una asociación directa entre el rol, las funcionalidades y el beneficio requerido por el cliente.

1.4.2.2 Satisfacción de los interesados

Descripción: El proyecto desarrollado debe aportar el mayor valor posible para el cliente y los potenciales interesados.

Medida de éxito: Obtener una carta de satisfacción firmada por parte del cliente y obtener un *feedback* positivo.

Propósito: Aportar valor al cliente, demostrar el compromiso por el desarrollo realizado al cliente.

[2] https://www.agilealliance.org/glossary/invest/

1.4.3 Objetivos del proyecto

1.4.3.1 Medir la factibilidad técnica del proyecto

Descripción: Dado que se trata de una prueba de concepto, la factibilidad técnica del proyecto debe ser evaluada y medida.

Medida de éxito: La idea puede ser implementada en la duración del proyecto.

Propósito: Si es factible, convertir la prueba de concepto actual en prototipo o producto liberado al mercado.

2 Problema

A continuación se desarrolla el problema presentado por el cliente GeneXus en la feria de proyectos de la Universidad ORT Uruguay. Se describe el contexto, la justificación y la definición del mismo.

2.1 Contexto y definición

GeneXus es una herramienta de desarrollo de *software low code/no code* con años de evolución y madurez. Diseñada para simplificar al máximo el proceso de desarrollo de *software* y resolver automáticamente la implementación de las distintas capas de un sistema. Para lograr esto utiliza una técnica incremental, uniendo las visiones parciales de los distintos tipos de usuarios y generando a medida que son necesarios elementos visuales tangibles por los mismos. Si bien GeneXus ha evolucionado mucho en este tipo de aspecto, ha quedado rezagado respecto a la trazabilidad de los requerimientos.

Se observa un desfase entre la evolución que GeneXus tiene respecto de los mecanismos que incorpora para la generación de código, bases de datos y servicios, y los mecanismos para asegurar la calidad y trazabilidad de requerimientos.

Por un lado, el primer problema que el cliente presenta respecto de su herramienta, es que no tiene un mecanismo para dar soporte a requerimientos escritos que garanticen la calidad de la implementación y que actúe como artefacto para ser auditado, discutido, entendido y comprendido entre todos los involucrados.

El segundo problema es que no existe trazabilidad de una necesidad de negocio escrita con características de contrato en un lenguaje común a todos los involucrados.

Detectados estos dos problemas, GeneXus plantea la utilización de un lenguaje estructurado basado en la gramática Gherkin para especificar requerimientos [18]. Además, la herramienta debe permitir realizar revisiones que involucren a distintos actores del proyecto y aseguren la

calidad en términos de completitud, consistencia, no ambigüedad, correctitud, verificabilidad, rastreabilidad y modificabilidad [19].

Por último, aprovechar la estructura de estos requerimientos para la generación automática de objetos nativos GeneXus, como por ejemplo transacciones, procedimientos, pruebas unitarias.

2.2 Resumen

En resumen, el cliente busca satisfacer los siguientes puntos:

Especificar requerimientos de un modo estructurado

Para la especificación de los requerimientos de un modo estructurado, sugiere la utilización de la gramática Gherkin, la misma que se utiliza en *frameworks* de trabajo como Behaviour Driven Development y Specification by Example.

Mantener trazabilidad entre el requerimiento y su implementación

Mantener la trazabilidad entre el requerimiento y los objetos que fueron generados para implementar dicho requerimiento. Es decir, crear una conexión o vínculo entre el requerimiento y los objetos GeneXus relacionados al mismo.

Realizar revisiones que involucren a distintos actores del proyecto y aseguren la calidad

Contar con un mecanismo que permita revisar los requerimientos a los distintos actores que participan en un proyecto. Lo que el cliente busca de esta parte es que a través de un mecanismo con las características antes mencionadas los requerimientos sean completos, consistentes, no ambiguos, correctos, verificables, rastreables y modificables. El cliente parte de la suposición de que la utilización de un sistema donde están presentes las visiones parciales de múltiples personas involucradas en un proyecto con distintos roles satisface las siete características de un requerimiento antes mencionadas.

Aprovechar la estructura de los requerimientos para generar elementos nativos de GeneXus

Dado que la gramática Gherkin puede ser interpretada por un *parser,* el cliente espera que un requerimiento escrito en dicha gramática apoyándose en las tablas de ejemplo, identificadas en la gramática Gherkin con la *keyword* Ejemplos o *Examples* puedan generarse las vinculaciones que se mencionan en el segundo punto y además se puedan crear objetos nativos GeneXus, como transacciones, procedimientos, pruebas unitarias, entre otros. Para aquellos elementos que no se pueden generar automáticamente, se debe poder establecer la relación de forma manual.

3 Solución

En esta sección se presenta la solución propuesta por el equipo, se exponen las principales funcionalidades y el alcance del proyecto.

3.1 Solución propuesta

La solución presentada, llamada GxReq, se trata una prueba de concepto que tiene por un lado un entregable de desarrollo para el cliente, que cuenta de tres partes separadas en su despliegue, pero unidas en cuanto a funcionalidad.

Entendiendo por prueba de concepto, *proof of concept* en inglés, o PoC a la presentación de un producto, programa o método propuesto, generalmente con características limitadas, que permite probar su potencial viabilidad [20].

3.1.1 Extension GeneXus

Respecto de los entregables de desarrollo, por un lado se encuentra la Extensión. La misma es un *plugin* integrado al entorno de desarrollo de GeneXus. La Extensión cuenta con un nuevo objeto GeneXus, el cual es llamado *Feature*, este objeto contiene un requerimiento del sistema que está siendo construido, escrito en gramática Gherkin. La *Feature* cuenta con un editor de texto Monaco[3] embebido. Siendo Monaco el mismo editor que utiliza Visual Studio Code[4]. También hace uso del *parser* de Cucumber[5] para la gramática Gherkin. Cucumber es una herramienta orientada a Behavior-Driven Development ampliamente utilizada en la industria.

La Extensión usa el *parser* de Cucumber para la generación automática de los objetos nativos de GeneXus, como son la transacción y el procedimiento. Al generarse estos objetos partiendo

[3] https://microsoft.github.io/monaco-editor/
[4] https://code.visualstudio.com/
[5] https://cucumber.io/

desde la *Feature*, se generan referencias entre estos, esto permite mantener la trazabilidad entre el requerimiento y el incremento resultante.

También soporta una integración con GxTest[6] que permite generar un conjunto de casos de pruebas unitarias y ejecutarlas. Las pruebas unitarias se asocian a un procedimiento y constan de un conjunto de tres tipos de objetos GeneXus, primero tenemos el SDT, este objeto establece los datos de entrada y de salida de la prueba, luego el objeto Data Provider, el cual establece, con un listado de objetos con el formato establecido por el SDT, un listado de casos de pruebas y por último el objeto que establece la prueba unitaria en sí misma. El objeto de prueba consume datos del Data Provider, ejecutando la prueba con cada uno de los casos establecidos en este. El Data Provider, que como se dijo antes contiene los casos de prueba, es construido por la Extensión de forma manual en el código, usando los datos de la tabla de ejemplo detallada en la definición del requerimiento en Gherkin. El SDT y el objeto de prueba son creados por la extensión GxTest.

Además tiene una extensión de la gramática Gherkin que permite definir tipos de datos de manera explícita en las tablas de la gramática y una *keyword* especial para indicar parámetros de salida de un procedimiento. Si los tipos de datos no están indicados de manera explícita, la Extensión puede inferir los tipos en función de los datos presentes en las tablas. En la sección 7.3 Requerimientos funcionales se explica en detalle cómo funciona la inferencia de tipos de datos y el tipado explícito.

Existen casos en los cuales los requerimientos, por cómo están especificados, no pueden generar objetos GeneXus, o, lo que es un caso similar, el requerimiento necesita para su implementación algo más que una transacción o un procedimiento. En estos casos entonces se generan objetos de forma manual, con el fin de completar la implementación de este requerimiento. En estos casos no se generaba la trazabilidad entre el objeto *Feature* y el objeto creado manualmente para implementar este, por lo que la extensión también permite agregar, de forma manual, referencias entre una *Feature* y N objetos creados para satisfacer el requerimiento en dicha

[6] https://www.genexus.com/es/productos/gxtest

Feature. Con esta característica se permite mantener trazabilidad entre el requerimiento y los objetos generados a partir de este, más allá de los generados automáticamente.

Por último, la extensión GeneXus permite obtener las *Features* o requerimientos desde un proyecto específico del Cliente de Requerimientos. Para esto fue agregado un menú al IDE de GeneXus. En la siguiente sección se detalla el Cliente de Requerimientos.

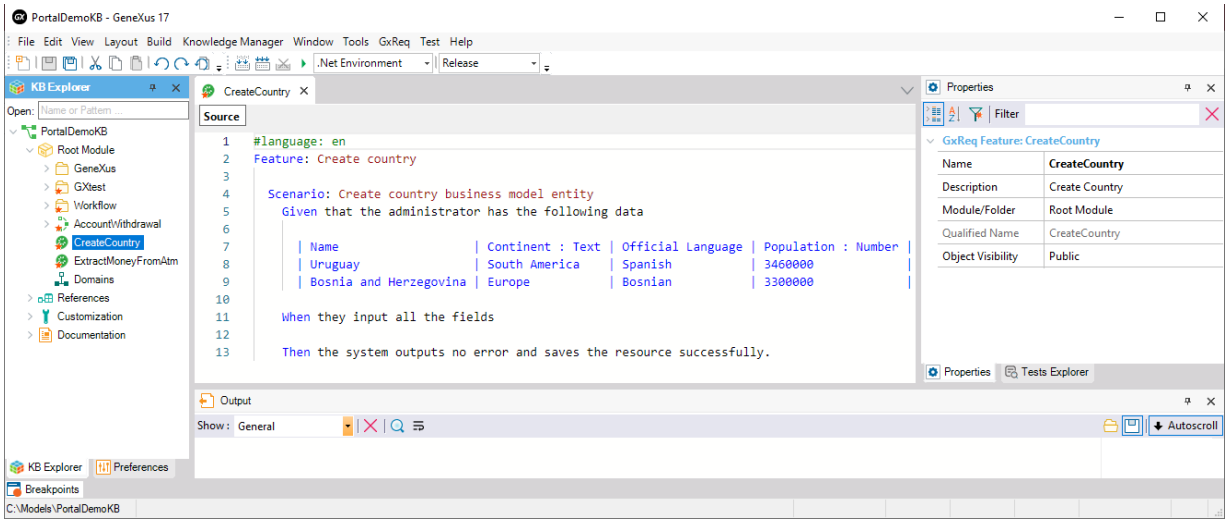


Figura 2: Vista objeto *Feature* en IDE GeneXus

3.1.2 Cliente de requerimientos

El Cliente de Requerimientos es la otra gran parte de la solución. Consta de una aplicación que permite la escritura, edición y revisión de los requerimientos de *software*.

El Cliente de Requerimientos permite a cada usuario registrarse y realizar el inicio de sesión con su respectiva cuenta. En él se pueden crear proyectos, los cuales corresponden con una aplicación en particular, y que para el caso de GeneXus, mapea con una *Knowledge Base* previamente configurada. Los proyectos tienen múltiples colaboradores, siendo cada colaborador un usuario registrado. En los proyectos es donde se encuentra cada *feature* o requerimiento definido para la solución.

Como se mencionó en el párrafo anterior, cada requerimiento está contenido en un proyecto particular, a su vez, este requerimiento fue escrito por cierta persona, que diremos que es su

dueño. Una vez escrito cada requerimiento, estos quedan visibles para todos los colaboradores del proyecto, los cuales podrán comentar estos requerimientos, por ejemplo sugiriendo un cambio, y también podrán aprobarlos cuando lo consideren correcto. Este flujo es muy similar al flujo normal realizado en los *Pull Request* en aplicaciones como Github o Azure DevOps, entre otras. El dueño del proyecto podrá hacer *commit* del requerimiento, cuando él lo considere apropiado. En este momento el requerimiento no se puede editar ni comentar, y está listo para que desde la extensión sea descargado e implementado.

La escritura de un requerimiento se da en un editor Monaco integrado en la aplicación web. Este editor tiene soporte de la gramática Gherkin, dado que los requerimientos deben ser escritos en esta gramática. El editor tiene *snippets* configurados, por ejemplo para la creación de tablas, para facilitar la escritura, también permite realizar un formateo del requerimiento, para que el formato sea el más prolijo posible y por sobre todo *syntax highlighting* de las diferentes *keyboards* de Gherkin.

Figura 3: Vista *feature* en Cliente de Requerimientos

3.2 Alcance del proyecto

El alcance del proyecto consiste en el desarrollo de un conjunto de entregables de *software* y una breve documentación del estado del arte de distintos enfoques para la especificación de requerimientos, el uso de Gherkin para especificar los mismos y metodologías que utilizan esta gramática, entre otras cosas.

A continuación se listan el conjunto de entregables de *software* desarrollados y el documento con el estado del arte:

- **Cliente de requerimientos:** Aplicación web con su *frontend* desarrollado en React, haciendo uso del *framework* React Admin que incorpora un editor de código Monaco Editor, que es una versión de Visual Studio Code en la web. La incorporación de este editor de código le permitió al equipo el rápido desarrollo de un *syntax highlighting* para la gramática Gherkin, un *snippet* para autocompletar las estructuras más comunes de un requerimiento escrito en Gherkin y un formateador de código. Su *backend* está desarrollado en C# .NET 5 haciendo uso del *framework* JSON API .NET[7].

- **Extensión de GeneXus:** Mecanismo de extensión para el IDE GeneXus permite la inferencia de tipos de datos y generación de procedimientos y transacciones, ambos objetos nativos de GeneXus, a partir de un requerimiento escrito en la gramática Gherkin. Esta extensión también incorpora un editor de código Monaco Editor al igual que el Cliente de Requerimientos. También tiene una integración con GxTest, para generar casos de prueba en función de un conjunto definido de datos. Para procesar la gramática Gherkin, hace uso del *Nuget Package* de Cucumber para C#.

- **Estado del arte:** Documento que contiene el estado del arte de los siguientes temas de interés para el cliente: uso de enfoques estructurados/semiestructurados al especificar requerimientos, uso de Gherkin para especificar requerimientos, atributos de calidad de un requerimiento funcional, *Specification by Example,* ¿Qué tan viable es que gente de

[7] https://www.jsonapi.net/

negocio escriba Gherkin vs. que lea, revise y corrija Gherkin?, revisiones de a pares y pares múltiples.

3.3 Entregables

Se acordaron con el cliente los siguientes entregables, producto de la culminación del proyecto:

- Cliente de Requerimientos:
    - *Backend*:
        - Acceso al repositorio con el código del mismo GxReq.Portal.Api[8].
        - Enlace al Swagger[9] del *deploy* de la aplicación en Heroku.
        - Acceso al *deploy* del proyecto en la plataforma Heroku
        - Documentación:
            - Diagrama de arquitectura.
            - Manual de *deploy.*
            - Manual de preparación de entorno de desarrollo local.
    - *Frontend*:
        - Acceso al repositorio con el código del mismo GxReq.Portal[10].
        - Enlace[11] al *deploy* del sitio.
        - Acceso al *deploy* del proyecto en la plataforma Firebase.
        - Documentación:
            - Manual de *deploy*
            - Manual de preparación de entorno de desarrollo local
- Extensión de GeneXus:
    - Acceso al repositorio con el código de la extensión GxReq[12]
    - Documentación:

[8] https://github.com/GxReqOrt/GxReq.Portal.Api

[9] https://gxreq-portal.herokuapp.com/swagger/index.html

[10] https://github.com/GxReqOrt/GxReq.Portal

[11] https://gxreq-portal.web.app

[12] https://github.com/GxReqOrt/GxReq

        - Manual de preparación del entorno de desarrollo local.
        - Configuración de la extensión la *Knowledge Base* o base del conocimiento en español.
        - Diagrama de arquitectura.
- Estado del arte:
    - Documento con el estado del arte de:
        - Uso de enfoques estructurados/semiestructurados al especificar requerimientos.
        - Uso de Gherkin para especificar requerimientos.
        - Atributos de calidad de un requerimiento funcional.
        - Specification by Example.
        - ¿Qué tan viable es que gente de negocio escriba Gherkin vs. que lea, revise y corrija Gherkin?
- Repositorio con *boilerplate* para el desarrollo de una extensión GeneXus

3.4 Ejemplo práctico de la solución propuesta

Con el fin de facilitar al lector el entendimiento de la solución propuesta, se agrega esta sección del documento. Donde se muestra a grandes rasgos el flujo que los distintos actores hacen sobre el sistema. Para simplificar el flujo, se deja de lado todo lo referente al registro, configuración, manejo de proyectos y configuraciones relacionadas al IDE GeneXus. En caso de ser necesario conocer el pormenor de las configuraciones, en el anexo 14.3 Manual de usuario se detallan los procesos de registro y configuración.

Todo comienza en el Cliente de Requerimientos, un usuario perteneciente a un determinado proyecto crea un requerimiento para ese proyecto. Elige un nombre para el mismo y escribe el contenido del requerimiento en el editor de texto Monaco que se abre dentro del navegador.

El requerimiento debe ser una gramática Gherkin válida. Para saber cuàl es la gramática Gherkin válida dirigirse *link* en el pie de página[13]. A continuación se muestra un ejemplo con las *keywords* más importantes de la gramática.

Suponiendo que el usuario escribe la siguiente *feature*:

```
#language: en
Feature: Extract money from ATM

 Scenario: Create account withdrawal procedure
  Given the account balance is <starting_balance>
   And the card is valid
   And the machine contains enough money
  When the Account Holder requests <amount>
  Then the ATM should dispense <amount>
   And the account balance should be <ending_balance>
   And the card should be returned

  Examples:
  | starting_balance | amount | ending_balance   |
  | 100              | 20     | 80               |
  | 200              | 30     | 170              |
  | 300              | 20     | 280              |
```

En la *feature* escrita anteriormente podemos identificar las siguientes *keywords* de la gramática Gherkin:

[13] https://cucumber.io/docs/gherkin/reference/

- *Feature:* El propósito de esta *keyword* es proveer una descripción de alto nivel de la característica del *software* y agrupar los escenarios relacionados a la misma. Debe ser siempre la primera *keyword* para que la gramática sea válida.

- *Scenario:* Esta *keyword* se utiliza para ejecutar el mismo escenario varias veces, pero con distintos valores.

- *Given:* Se utiliza para describir el contexto inicial del sistema. Su propósito es poner el sistema en un estado conocido, antes de que el usuario comience a interactuar con el mismo.

- *When:* Se usa para describir un evento o una acción. Puede ser una interacción del usuario o un evento disparado por otro sistema externo.

- *Then:* Se usa para describir una salida esperada o un resultado.

- *Examples:* Luego de esta *keyword* le sigue una tabla con datos. En el caso de La Extensión desarrollada por el equipo, será esta la tabla que se procesará para la creación del objeto nativo GeneXus procedimiento. Para entender las funcionalidades que se pueden lograr con esta tabla, referirse a la sección 7.3 Requerimientos funcionales.

Una vez el requerimiento es escrito, queda marcado en estado de revisión, es decir, otros usuarios pertenecientes al proyecto pueden realizar comentarios sobre el mismo o aprobarlo si consideran que el mismo es correcto.

A este punto, el flujo puede tomar dos caminos. El usuario que escribió el requerimiento puede cerrarlo, y en ese caso el mismo quedará en un estado en el que ya no se pueden hacer más cambios. Por otro lado, si algún usuario hace un comentario sobre el requerimiento el usuario que inició el requerimiento es notificado vía mail y puede hacer mejoras sobre el requerimiento si lo considera oportuno.

A continuación se muestra un diagrama simplificado que ilustra el flujo anteriormente descrito:

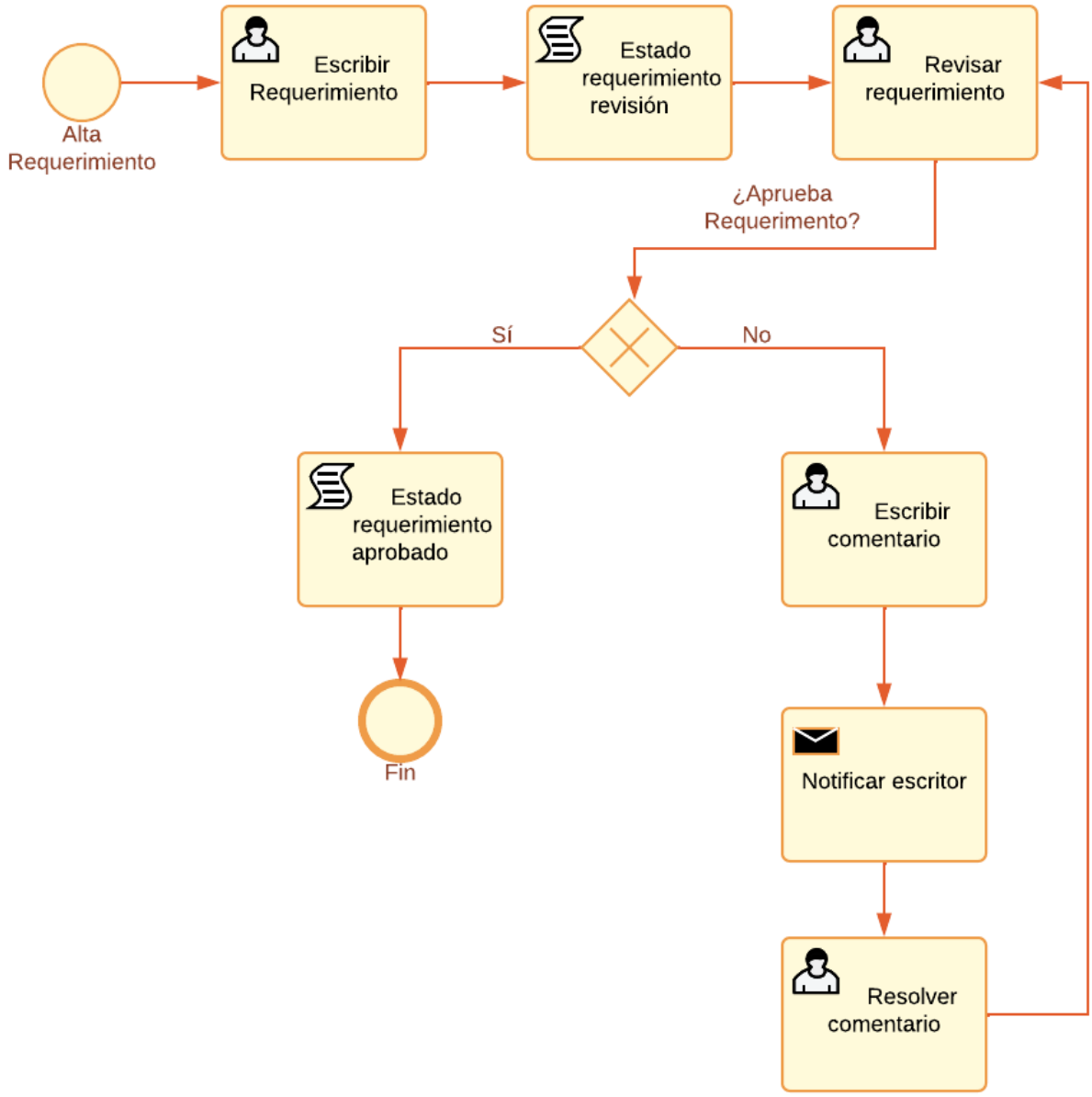


Figura 4: Diagrama BPMN[14] simplificado de un requerimiento

Una vez el flujo llega a su fin, desde el IDE de GeneXus el desarrollador puede descargarlo, siempre y cuando tenga su Knowledge Base configurada apuntando a dicho proyecto y sea miembro del mismo. Para ver esta configuración en detalle referirse a 14.3 Manual de usuario

[14] https://www.bpmn.org/

La siguiente figura muestra los pasos que el usuario debe hacer dentro del IDE de GeneXus para descargar los requerimientos:

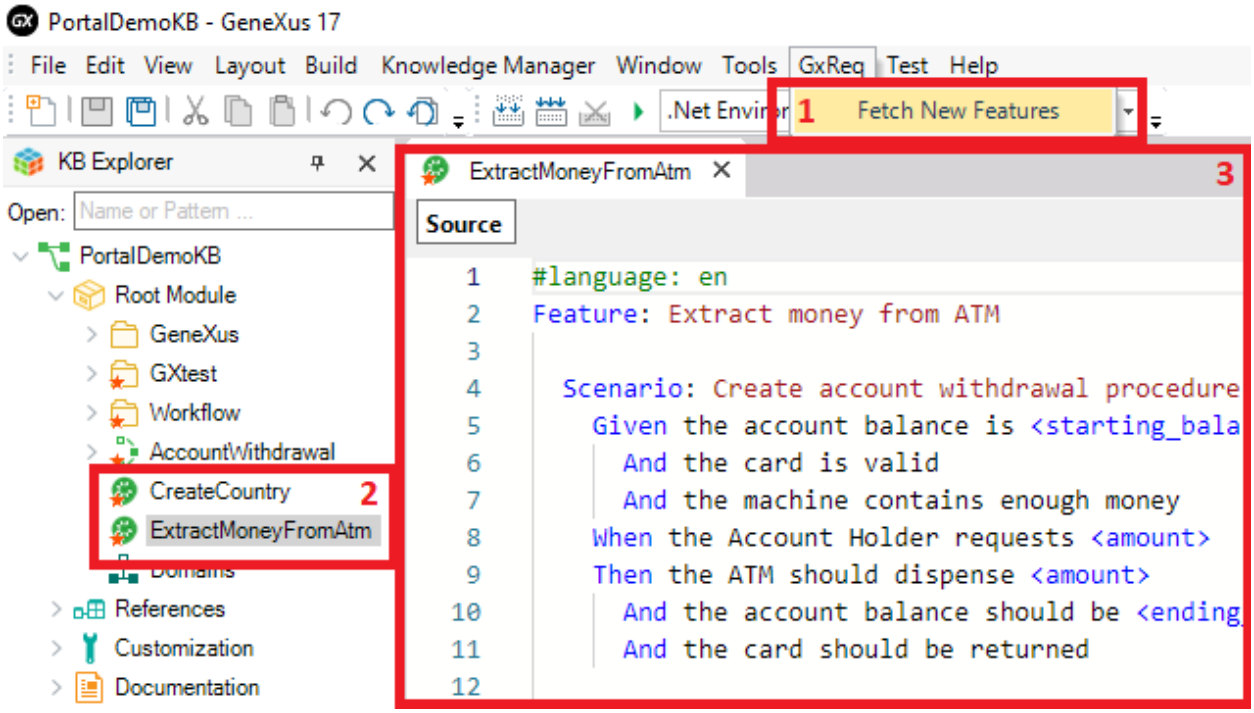


Figura 5: Proceso de sincronización de *features* en el IDE GeneXus

Primero el desarrollador debe desplegar el menú *"GxReq"* y dar clic sobre "*Fetch New Features*". Al hacer esto, en el KB Explorer se sincronizarán las nuevas *features* disponibles que no habían sido sincronizadas previamente. Al hacer doble clic sobre la feature sincronizada, en la parte derecha del IDE se desplegará el código perteneciente a la misma.

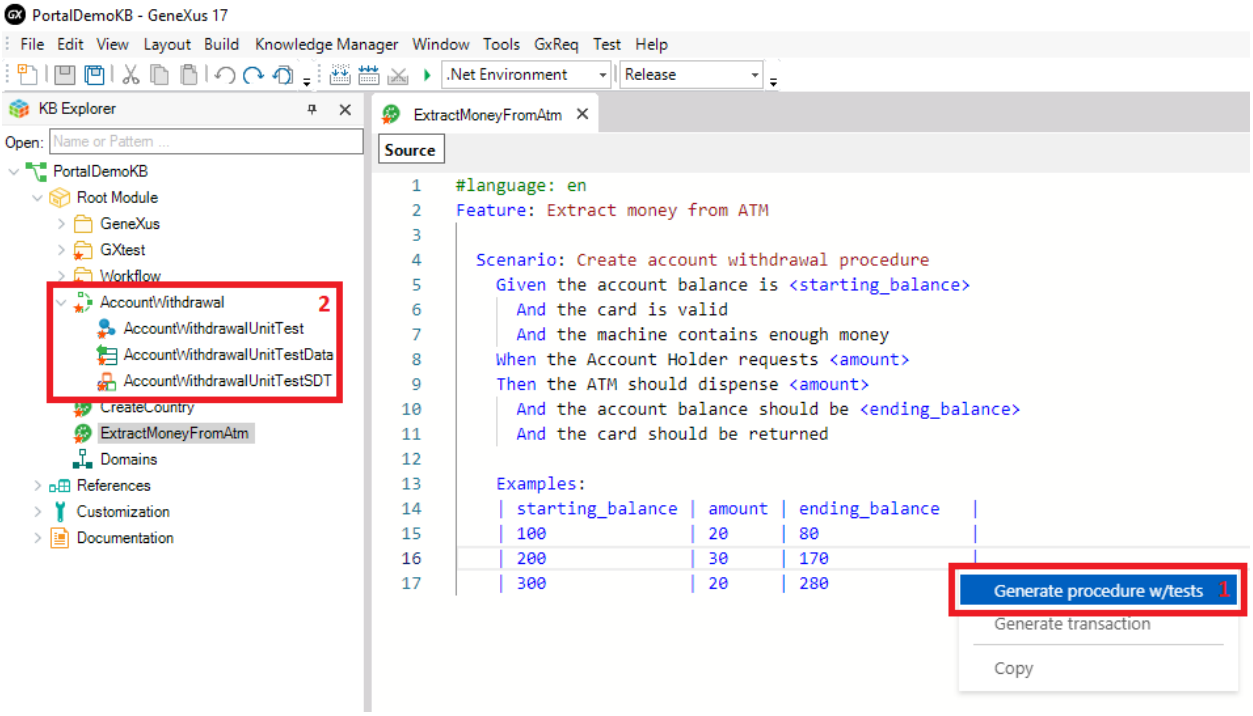


Figura 6: Menú para la creación de objetos nativos GeneXus

Al dar clic derecho sobre el editor Monaco que contiene el código de la *feature* se listan los objetos que el sistema soporta generar. Al seleccionar "*Generate procedure w/tests*" si la feature tiene los datos necesarios para hacerlo, se generarán las estructuras necesarias del objeto GeneXus procedimiento y todo lo necesario para ejecutar las pruebas con sus respectivos casos de prueba.

4 Estado del arte de Gherkin

En este capítulo se presenta una breve discusión del estado del arte de la gramática Gherkin para la redacción de requerimientos de software de manera estructurada o semiestructurada.

Comenzando por una descripción más genérica sobre los enfoques que existen para la especificación de requerimientos estructurados o semiestructurados.

Seguido de esto, se explica puntualmente el uso de Gherkin como gramática para lograr este cometido, y como él mismo asegura los atributos de calidad que un requerimiento funcional debe cumplir.

Por otro lado se discute la especificación de requerimientos basada en ejemplos (SbE) como forma colaborativa de definir requerimientos y pruebas orientadas al negocio utilizando ejemplos reales.

Se analiza la viabilidad de que personas del negocio puedan redactar requerimientos empleando la gramática Gherkin, en contraposición a que solo puedan leerlos, revisarlos o corregirlos.

Por último, se discuten las revisiones de a pares y pares múltiples, y cómo éstas influyen sobre los requerimientos del negocio.

4.1 ¿Qué es Gherkin?

Gherkin es un lenguaje diseñado para ser no técnico y legible por personas, generalmente empleado para describir casos de uso en un sistema de *software* [21]. Tiene una gramática sencilla, pero suficientemente estructurada como para ser entendida por un parser, comúnmente Cucumber. Escenarios de casos de uso de un sistema de *software* pueden ser escritos fácilmente con Gherkin.

Gherkin sirve para múltiples propósitos, como por ejemplo: generar especificaciones ejecutables sin ambigüedades, tests automatizados utilizando Cucumber y documentos de cómo el sistema se debería comportar.

Por lo general, cada requerimiento escrito en Gherkin se escribe en un archivo de texto plano denominado *feature file*. A continuación se muestra un ejemplo del mismo:

```
Feature: Guess the word

  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

4.2 Uso de enfoques estructurados/semiestructurados al especificar requerimientos

Existen diversas metodologías para la especificación de requerimientos de *software* que proponen la redacción de requerimientos de una manera estructurada o semiestructurada. Una de ellas es behavior-driven development (BDD) [22], metodología introducida con el objetivo de proveer un proceso semántico que soporte un vocabulario común y un entendimiento compartido de los requerimientos entre todos los actores involucrados (clientes, ingenieros de requerimientos, desarrolladores, testers).

BDD se apoya en la gramática Gherkin para la redacción de requerimientos de forma estructurada o semiestructurada, basándose en la premisa de que este DSL, con sus características de redacción y estructura particular, satisface el entendimiento común de los requerimientos entre todos los involucrados.

De esta estructura, en la que BDD se basa para redactar un requerimiento, se desprenden dos partes. Por un lado tenemos la narrativa o el comportamiento y por otro lado, el criterio de aceptación. De este último está la posibilidad de aplicar acceptance test-driven development (ATDD) para disminuir la brecha entre el negocio y los aspectos técnicos del *software* [23]. De este modo, tenemos los criterios de aceptación desde antes de la fase de desarrollo, y los mismos deberán ser codificados como pruebas unitarias en la fase de desarrollo con la ayuda de algún *framework*.

De acuerdo a estudios realizados [24] este último punto es visto como una ventaja, ya que los criterios de aceptación han sido previamente acordados entre los interesados, antes de comenzar con el desarrollo.

4.3 Uso de Gherkin para especificar requerimientos

El proceso de elicitación de requerimientos es una de las etapas más complicadas previas al desarrollo. El negocio suele tener problemas intentando expresar sus necesidades. Los clientes tienden a mencionar sus necesidades en su propio lenguaje específico al dominio de su negocio [25]. Lenguaje de dominio que el desarrollador no necesariamente entiende.

A este punto es importante pensar los requerimientos de una forma que asegure los siguientes tres criterios: explícito y sin ambigüedades, entendible por todos los interesados, actualizado y documentado [26]. A continuación se muestra como Gherkin asegura estos tres criterios:

- **Explícito y sin ambigüedades**: Si bien es un lenguaje muy cercano al natural, tiene una estructura que lo hace explícito y sin ambigüedades.
- **Entendible por todos los interesados**: Dado que es un lenguaje natural, no debería representar mayores problemas para ser entendido por los diferentes interesados en el requerimiento.
- **Actualizado y documentado**: Dependiendo de la forma en la que esté implementado, cada requerimiento escrito en Gherkin constituye un artefacto de documentación. Gherkin por sí solo no asegura que el mismo esté actualizado, pero dado que es entendido

por todos los interesados si alguno de estos artefactos queda desactualizado será más notorio, dado que más personas son conscientes de su existencia.

4.4 Atributos de calidad de un requerimiento funcional

Utilizando la gramática Gherkin para la redacción de requerimientos, se pueden escribir los mismos en un lenguaje natural, como ya fue mencionado en secciones previas, entendido por todos los interesados. Los requerimientos se escriben siguiendo lineamientos que satisfacen los atributos de calidad de un requerimiento bien definido [19]. Estos atributos de calidad son: completitud, no ambigüedad, correctitud, verificabilidad, rastreabilidad, modificabilidad, consistencia y clasificación acorde a la importancia y/o estabilidad.

Entendiendo por:

Correctitud: Un requerimiento es correcto si y sólo si cada requisito establecido en el mismo satisface lo que el *software* debe cumplir. Alternativamente el cliente puede determinar si el requerimiento refleja sus necesidades.

No ambigüedad: Un requerimiento es no ambiguo si y sólo si todo requisito expresado en él tiene una única interpretación. Como mínimo esto requiere que dicha característica en el producto final sea descrita usando un único término único.

Completitud: Un requerimiento se considera completo si y sólo si cumple con que todo requisito relacionado con funcionalidad, rendimiento, limitaciones de diseño o interfaces externas es reconocido y tratado. El requerimiento especifica todas las condiciones de entrada y salida que pueden ocurrir. Debe incluir todas las etiquetas y referencias a todas las figuras, tablas, y diagramas en el requerimiento y unidades de medida.

Consistencia: Hace referencia a la consistencia interna de un requerimiento. Si el requerimiento se contradice con algún documento de alto nivel, como por ejemplo una especificación de requerimientos, no es correcto. Existen tres tipos de consistencia interna de requerimientos: las características de un objeto de la realidad pueden entrar en conflicto, existe un conflicto lógico

o temporal entre dos acciones especificadas o dos requerimientos describen un objeto de la realidad pero con términos diferentes.

Clasificación acorde a la importancia y/o estabilidad: Este atributo busca que los requerimientos puedan ser clasificados u ordenados en base a la necesidad y a la estabilidad. Un requerimiento puede ser esencial, es decir, que el software no es aceptable si no se encuentra este desarrollado, puede ser condicional, estos requerimientos mejoran el producto, pero no son necesarios para que el software sea aceptable, y por último, un requerimiento puede ser opcional, estos requerimientos pueden o no dar valor adicional. Por otro lado está la estabilidad, esta se define como la cantidad de cambios que se estima que va a sufrir el requerimiento en un futuro.

Verificabilidad: Un requerimiento es verificable si y sólo si todos sus requisitos son verificables. A su vez, un requisito es verificable si existe un método con el que una persona o máquina pueda que el software cumple con el requisito. En general decimos que cualquier requisito ambiguo no es verificable.

Modificabilidad: Un requerimiento es modificable si y sólo si consta de una estructura que permite realizar cualquier cambio de manera fácil, completa y consistente. La redundancia en sí no es un error, pero nos puede llevar a errores de modificabilidad.

Rastreabilidad: Un requerimiento es rastreable si el origen de sus requisitos es claro. Son recomendados dos tipos de trazabilidad, hacia atrás, es decir, a etapas anteriores del desarrollo. Esto depende de que cada requisito haga referencia explícita a su fuente en documentos anteriores. Trazabilidad directa, esto es tener trazabilidad a todos los documentos generados por la especificación del requerimiento de *software*. Para esto es necesario que cada requisito de la especificación sea identificado de forma única.

4.5 Specification by Example

En esta sección definiremos lo que es Specification by Example (SbE), veremos sus aplicaciones, ventajas y cómo aplica a Gherkin, la gramática usada para la especificación de requerimientos en nuestro sistema.

Según el autor Gojko Adzic, Specification by Example es un conjunto de patrones de proceso que facilitan el cambio en los productos de software para garantizar que el producto correcto se entregue de manera eficiente [27]. Bajando esto un poco a tierra, Specification by Example es un enfoque colaborativo para definir requerimientos y pruebas funcionales, que tengan una orientación al negocio. Es empleado habitualmente en enfoques ágiles y en particular en BDD (Behavior-Driven Development, n.d.). Para definir requerimientos usando esta técnica se usan DSL (Domain Specific Language) [28] o Ubiquitous Languages [29], definidos por DDD [30]. Estos dos tipos de lenguajes antes mencionados tienen la ventaja de que pueden ser comprendidos tanto por personas expertas en el negocio, como por personas técnicas o desarrolladores. Apoyados por esto último, estos requerimientos se escriben en conjunto con las personas de negocio (cliente).

El propósito de este enfoque es crear el producto correcto y crear el producto correctamente [31]. Se centra en la comprensión compartida entre los usuarios y los desarrolladores, estableciendo así una única fuente de la verdad. A modo de ejemplo, podríamos describir un requerimiento como un conjunto de ejemplos, con determinados datos de entrada y salida, una vez se cumpla con el 100% de estos ejemplos, entonces el requerimiento está terminado al 100%.

Poniendo como ejemplo BDD, los ejemplos (o criterios de aceptación) son definidos con un lenguaje semi-estructurado, y claro también comprensible tanto por la gente de negocio como por los desarrolladores. El hecho de que se especifiquen teniendo cierta estructura permite la automatización de pruebas. Esto, en conjunto con la comprensión compartida, nos dan grandes ventajas. Los requerimientos especificados con este enfoque nos sirven también como documentación sobre el sistema.

4.6 ¿Qué tan viable es que gente de negocio escriba Gherkin vs. que lea, revise y corrija Gherkin?

Gherkin se presenta como un lenguaje diseñado para ser no técnico y legible por cualquier tipo de usuario. Su propósito es que pueda ser entendido por todas las personas involucradas dentro

del proceso de desarrollo de *software*. Hasta aquí es un lenguaje que satisface el ser entendido por todos los interesados, dado que emplea un lenguaje natural, pero a la vez casi sin ambigüedades por su estructura. Este último punto hace posible que pueda ser procesado por un *parser* y que tenga utilidad para la generación de *tests* automáticos entre otras cosas.

Hasta ahora, tenemos la certeza de que el lenguaje es legible por los interesados, y esto implica que pueda ser revisado. Pero esto no asegura que el lenguaje pueda ser escrito o corregido por personas del negocio.

John Ferguson afirma en su cuenta de Twitter que es falso que los requerimientos con el estilo BDD haciendo uso de Gherkin puedan ser escritos por cualquier persona y agrega que si bien los requerimientos son fáciles de leer por cualquiera no asegura que sean fáciles de escribir por cualquiera [32].

Alineado a esto y en un sentido más abstracto se encuentra la opinión de Martin Fowler quien plantea que la gracia de utilizar un DSL no es la posibilidad de que el negocio pueda escribir los requerimientos por sí mismo sino que al entenderlos se abre un canal de comunicación más profundo entre todos los interesados. Si el negocio puede entender el código entonces puede sugerir cambios, pero dejando la implementación a los desarrolladores [28].

Por otro lado, Fowler menciona que pueden existir beneficios de un DSL que pueda ser escrito por el negocio. El esfuerzo de crear un DSL con estas características también requiere la creación de un entorno de edición, mensajes de error útiles y herramientas de *debugging* y testeo.

En esta línea de pensamiento también se encuentra Michael Hunger quien plantea que para escribir un DSL las herramientas del mismo también deben estar adaptadas al tipo de usuario que va a emplear el DSL. Si va a ser usado por desarrolladores, entonces las herramientas deben estar pensadas para desarrolladores, en cambio si va a ser usado por usuarios de negocio entonces las herramientas deben ser pensadas para este tipo de usuario [33].

Si se piensa en las herramientas existentes para escribir requerimientos con la gramática Gherkin generalmente se hace a través de un editor de código o IDE, que soporte o tenga los *plugins*

instalados para dicha sintaxis. Un ambiente muy diferente al que una persona del negocio puede estar acostumbrada. Esto hace que la tarea de escribir un requerimiento funcional desde el punto de vista de una persona de negocio se dificulte aún más.

Bajo este escenario, la posibilidad real de que un usuario de negocio escriba un requerimiento en Gherkin es mucho más amplia que manejar la estructura del lenguaje. Se necesitan otros aspectos técnicos en consideración como la utilización de las herramientas y entorno de desarrollo adecuados al rol del usuario en cuestión.

4.7 Revisiones de pares y pares múltiples

La detección de defectos es una etapa crucial en el desarrollo de *software*. Para esta etapa se pueden realizar diferentes tipos de técnicas. Tenemos técnicas llamadas Dinámicas [31], las cuales implican la ejecución del artefacto a revisar, ingresando datos de entrada y validando los datos de salida. Por otro lado, existen las técnicas Estáticas [34], las cuales no implican la ejecución del artefacto, sino revisar el cómo está construido, a diferentes niveles. Nuestro sistema se centra en la etapa de especificación de requerimientos, por lo que nos centraremos en las técnicas estáticas.

Dentro de las técnicas estáticas se encuentran las revisiones. Una revisión es definida por la IEEE como un proceso o reunión durante la cual un producto o proceso de *software* es presentado al personal involucrado en el proyecto para examinación, retroalimentación y aprobación. Existen diversas técnicas de revisión, las cuales se suelen diferenciar en su grado de formalidad. El autor Karl E. Wiegers define siete técnicas de revisión y las ordena según su formalidad. Las técnicas antes mencionadas son (en orden de formalidad), *Ad Hoc*, Revisión de Pares, Revisión de Pares Múltiples, Programación en Pares, Tutorial o Presentación, Revisión en Equipo e Inspección, siendo esta última la más formal de las técnicas mencionadas [35].

Antes de pasar a definir las dos técnicas en la cual esta sección hace énfasis, veamos por qué es beneficioso realizar revisiones.

Una de las principales razones por lo cual es beneficioso realizar revisiones de código es el costo de mitigar los defectos. Corregir un defecto en una etapa tardía del ciclo de vida del desarrollo es mucho más costoso que hacerlo en una etapa temprana [36].

El otro gran beneficio es el hecho de que el conocimiento se comparte con el equipo, es decir, un miembro de un equipo realiza cierto artefacto y luego N miembros del equipo revisan este artefacto, estos revisores sabrán cómo está hecho, el avance conseguido. Esto permite definir metas y objetivos comunes dentro del equipo [37].

4.7.1 Revisión de pares

En una revisión de pares, solo una persona, aparte del autor, examina el artefacto creado. El autor puede no saber cómo se realiza esta revisión. Esta revisión se puede realizar haciendo uso de *checklists*, formularios específicos o métodos de análisis por parte del revisor. Este modo de revisar depende completamente del conocimiento que tenga el revisor. Al finalizar la revisión lo esperado es una lista de defectos a corregir o sugerencias a aplicar por parte del autor del artefacto.

Este enfoque de revisión es económico, solo conlleva el tiempo de un miembro del equipo aparte del autor. También puede ser menos intimidante para el autor el saber que solo otro miembro va a revisar su trabajo, y no todo un equipo entero. Por otra parte, los errores detectados se verán limitados a los que encuentre el único revisor. Al momento de la revisión, el autor no estará presente para responder dudas que le puedan surgir al revisor.

El conjunto de aspectos antes mencionados hace que este enfoque de revisión sea ideal para sistemas de bajo riesgo y/o, para comenzar a inculcar una cultura de revisiones en una empresa [38].

4.7.2 Revisión de pares múltiples

Una revisión de pares múltiples es una revisión de pares que se realiza de forma concurrente por varios revisores (comúnmente de 3 a 15 revisores). Esta se puede realizar de dos formas, o se le

entrega una copia del artefacto a cada revisor, y cada uno de estos hace su lista de defectos/sugerencia de mejoras por separado, o, por otra parte, se comparte el artefacto entre los revisores en un ambiente compartido, en donde cada revisor ve los comentarios de los demás. Esta segunda forma de entregar el artefacto a revisar puede favorecer la eliminación de la redundancia entre los revisores, al ver los aspectos señalados por los demás, no se repiten. A su vez, esta forma de realizar el proceso de revisión, puede llevar a largos debates entre los revisores.

Este tipo de revisión mitiga el principal riesgo de la revisión de pares, el hecho de que el único revisor no haga bien su trabajo, por diversas razones. Si bien mitiga este riesgo, puede seguir ocurriendo que los múltiples revisores no encuentren defectos, los cuales quizás sí se encuentren en discusiones cara a cara. Este último sigue siendo el gran problema de este tipo de revisiones, carecen de la estimulación mental de una reunión en la cual se encuentren los miembros del equipo cara a cara.

Este enfoque es útil para transferir el artefacto a más revisores de los cuales podría juntar en una reunión al mismo tiempo [35].

5 Gherkin y los atributos de calidad de un requerimiento funcional

En esta sección se establece una relación entre Gherkin y los atributos de calidad de un requerimiento funcional. Cabe destacar que esto representa la opinión casi exclusiva de los integrantes del equipo y el tutor de esta tesis. La bibliografía existente sobre Gherkin y los atributos de calidad de un requerimiento funcional es aún muy escasa aún.

En 4 Estado del arte de Gherkin se describieron los atributos de calidad de un requerimiento funcional, más específicamente en la sección 4.4 Atributos de calidad de un requerimiento funcional.

Gherkin, que es un caso de Specification By Example descrito en la sección 4.5 Specification by Example y en conjunto con diferentes visiones del código, o, Revisiones de pares, nos permite cumplir con cada uno de los atributos de calidad de un requerimiento.

5.1 Gherkin y Specification by Example

Como se detalló en la sección 4.1 ¿Qué es Gherkin?, esta gramática consta con una estructura en donde se especifica una *Feature* (Característica) y para cada una de estas un conjunto de Escenarios, que detallan los criterios de aceptación. Estos escenarios son ejemplos de cómo debe funcionar la *Feature*, se dan datos de entradas (de diversas formas) y se detalla la respuesta del sistema, es decir, una salida. Con esto vemos cómo usar Gherkin es una forma de especificar requerimientos mediante ejemplos.

5.2 Specification by Example y los atributos de calidad de un requerimiento

La especificación de requerimientos mediante ejemplos apoya los siguientes atributos de calidad de un buen requerimiento:

- **No ambigüedad:** Dado que los ejemplos se detallan haciendo uso de datos, tanto de entrada, como de salida, entonces se mitiga la ambigüedad con respecto a cómo debe funcionar el requerimiento. Mientras más ejemplos diferentes haya, entonces menos ambigüedad existirá.

- **Verificabilidad:** Parecido al anterior, al existir datos de entradas y de salida para cada ejemplo, entonces se pueden usar estos para verificar que el requerimiento funciona correctamente. Si los ejemplos tienen cierta estructura (como es el caso de Gherkin), entonces se pueden crear pruebas de forma automática que también permiten verificar el requerimiento. Mientras más ejemplos haya, más exhaustiva será la verificación.

- **Correctitud:** Se menciona en esta sección que el cliente participa de manera activa en la escritura del requerimiento, en conjunto con usuarios técnicos, por lo que esto apoya el atributo de correctitud.

5.3 Revisiones y atributos de calidad de un requerimiento

Similar a lo hecho en la sección anterior, podemos relacionar la práctica de realizar revisiones con atributos de calidad de un buen requerimiento. En este caso, el requerimiento escrito (en gramática Gherkin) será nuestro artefacto a revisar.

Las revisiones entonces apoyan los siguientes atributos de calidad de un requerimiento:

- **Completitud:** Como mencionamos antes, el conjunto de visiones parciales, siendo estas visiones parciales cada usuario que revisa y da su punto de vista sobre el requerimiento, nos apoya en que el requerimiento sea completo y consistente.

- **Correctitud:** En el momento que la revisión se completa, entonces el artefacto, y en nuestro caso el requerimiento, es “correcto” para todos los usuarios que participaron. En este caso se cumple el atributo de correctitud en conjunto para todos los usuarios que participaron, no solo para los escritores.

- **No ambigüedad:** Aunque este atributo se ve mayormente apoyado por la estructura de Gherkin a la hora de especificar un requerimiento, el hecho de contar con las visiones parciales de varios usuarios nos ayuda a mitigar el problema de la ambigüedad.

6 Metodologías

El objetivo de este capítulo es detallar la metodología aplicada en el proyecto y la justificación de la misma. Para esto, se describe el ciclo de vida elegido, se detalla la incertidumbre existente en el proyecto y, por último, se listan y describen cada etapa del proyecto.

6.1 Ciclo de Vida

El ciclo de vida elegido para el proyecto fue evolutivo. Para determinar el ciclo de vida a aplicar el equipo realizó un análisis comparando los diferentes ciclos de vida aprendidos durante la carrera, en los cuales se encuentra el ciclo de vida cascada, incremental iterativo y evolutivo.

El ciclo de vida cascada es un proceso lineal, se caracteriza por dividir el proceso de desarrollo en etapas bien definidas y sucesivas. Cada etapa se ejecuta solo una vez. Este fue el primero en ser descartado, dado que el proyecto a desarrollar es una prueba de concepto, los requerimientos, que si bien contaban con una idea general desde el inicio, tenían mucha incertidumbre. Por esto no era viable establecer un plan inicial rígido y no tener resultados hasta el final.

Por otro lado tenemos el ciclo de vida incremental iterativo, en este, se realiza el relevamiento y definición de requerimientos al principio y, luego, se itera en las etapas de diseño, desarrollo y pruebas. Por su propia definición, y similar al caso del ciclo de vida cascada, este ciclo de vida no fue el elegido por el equipo. Los requerimientos no podían ser definidos en su totalidad al principio del proyecto, existía incertidumbre sobre cada uno de ellos, tanto si el requerimiento aportaba valor, como si era viable técnicamente.

Con el ciclo de vida en cascada y el incremental iterativo descartados, nos queda el ciclo de vida evolutivo, el cual fue el elegido. En este se itera por cada una de las etapas de desarrollo de *software,* desde el relevamiento y definición de requerimientos, hasta la etapa de pruebas. Por esto, este ciclo de vida es el que más se adecua a nuestra prueba de concepto. Al principio de cada iteración se analizan y definen los requerimientos a desarrollar en el *sprint*, y en caso de que del análisis se obtenga que no aporta el suficiente valor o es inviable técnicamente el

requerimiento es cambiado, de forma que cumpla con lo anterior. Por otro lado, este ciclo de vida permite entregas parciales del producto, lo cual nos brinda retroalimentación sobre el o los requerimientos implementados, pudiendo así encontrar defectos de forma temprana, o cambios en ideas iniciales que luego de la implementación cambian de parecer, adaptando así nuestro producto en las siguientes iteraciones. También por cómo se define este, se adapta mejor que los anteriores al agilismo.

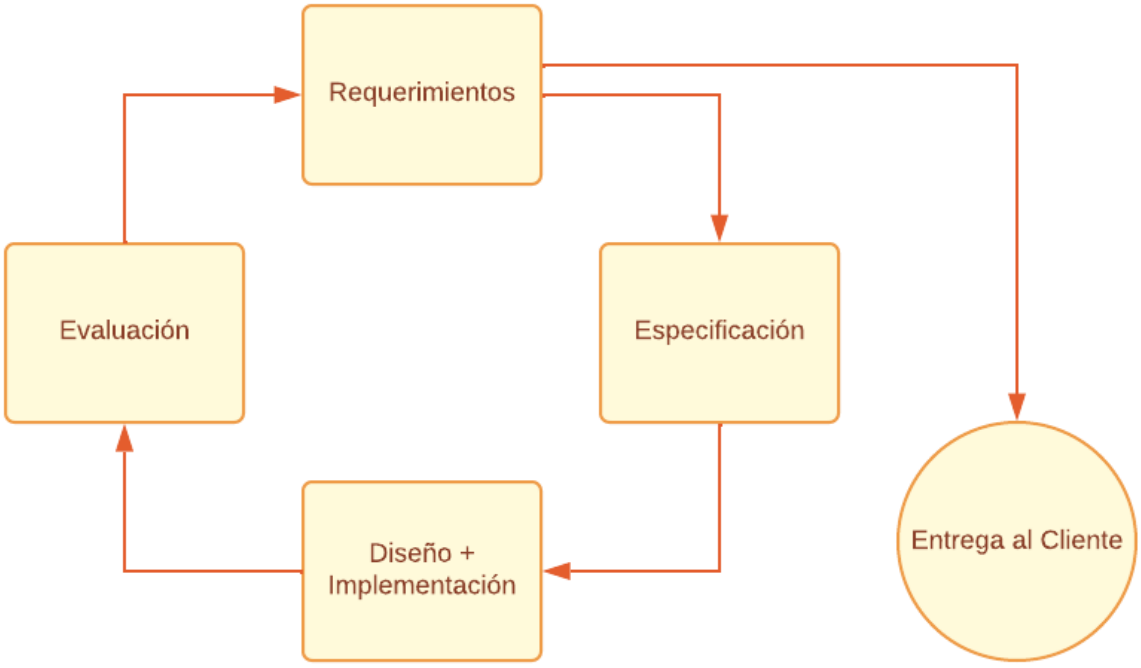


Figura 7: Ciclo de vida

6.2 Incertidumbre

Al diseñar el plan de *releases* del proyecto se tuvo en cuenta la incertidumbre que se manejaría en cada momento y la dificultad esperada a la hora de lidiar con dicha incertidumbre.

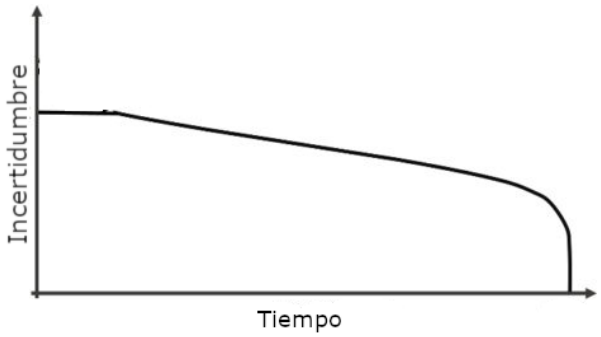


Figura 8: Gráfica de incertidumbre en función del tiempo

El primer *release* fue el de máxima incertidumbre y a medida que avanzaba el proyecto se esperaba un decremento en el nivel de incertidumbre. Esto es así puesto que se espera que la gente de GeneXus pueda ayudar a solucionar dudas técnicas en meses del año en los cuales no es común que estén de licencia. Se buscó minimizar el riesgo de aquello que se fuera a hacer en las épocas de fiestas. Apoyando esta decisión también se encuentra el hecho de que la empresa GeneXus es la creadora de la aplicación de Coronavirus UY [39], lo que les demandó una gran cantidad de recursos en la época de pandemia, y existía una gran incertidumbre con respecto a qué pasaría con una hipotética suba de casos, esto podría provocar que no tuviéramos su ayuda, o al menos no de forma rápida.

6.3 Etapas del proyecto

En esta sección se presentan las diferentes etapas del proyecto.

6.3.1 Capacitación

Al inicio no se sabía nada acerca de desarrollo *low-code/no-code* por ende era imposible comenzar a desarrollar una herramienta para GeneXus en este estado. La primera etapa fue una de capacitación destinada 100% a solucionar este problema. La empresa brindó al equipo acceso sin costo a un curso oficial, al final del cuál todo el equipo obtuvo un diploma de Analista GeneXus for Students v17.

6.3.2 Investigación

Una vez que ya se tuvo una idea sobre la industria *low-code/no-code* el objetivo pasó a ser aquel de poblar el *backlog* inicial de desarrollo. Para ello se debió investigar para entender más sobre el problema, los potenciales usuarios y más.

Esta etapa fue gestionada usando el *framework* Popcorn Flow. El mismo tiene como objetivo estructurar el proceso de descubrimiento de soluciones a problemas que se van presentando. La salida de este proceso terminó siendo el *backlog* de desarrollo para la próxima etapa [40].

Esta estructura definida por Popcorn Flow fue traducida por el equipo a un *board* en la herramienta Trello.

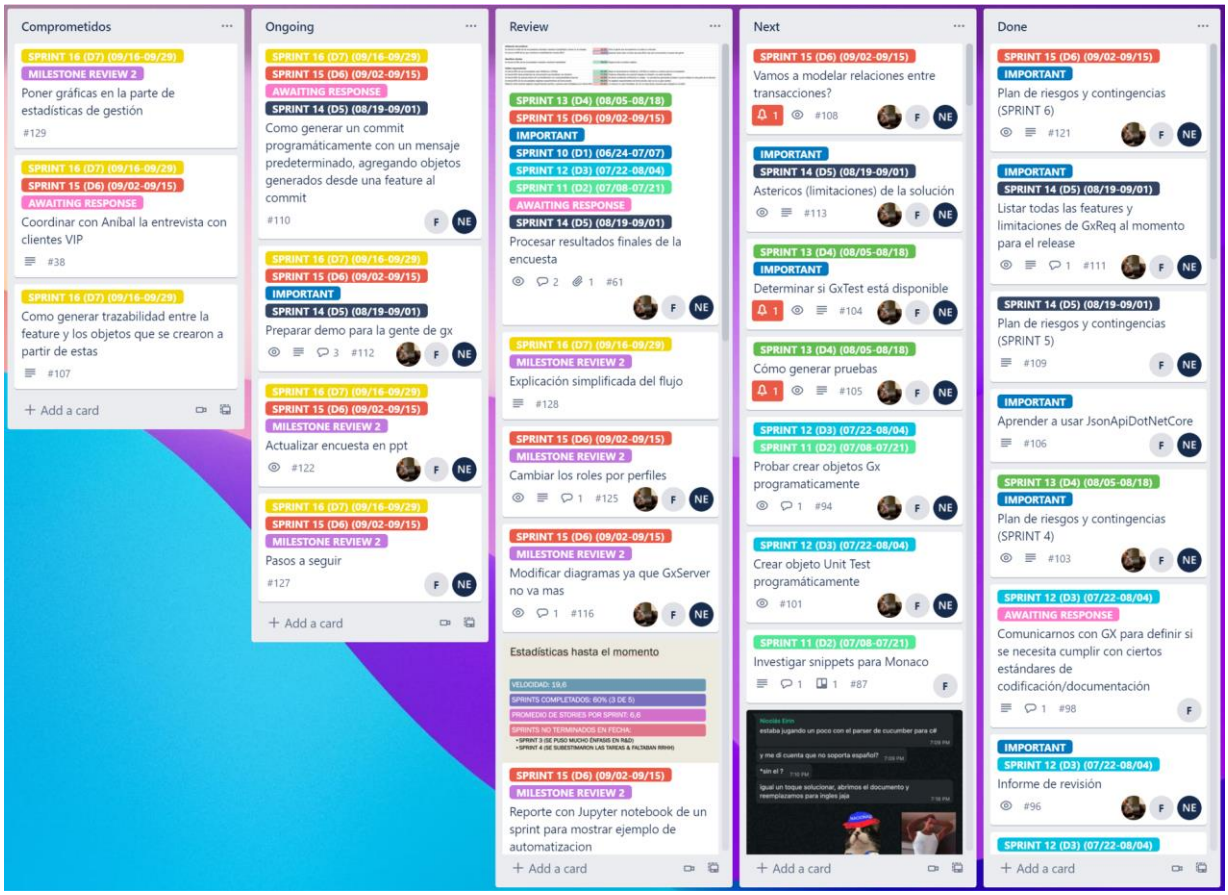


Figura 9: Tablero de investigación con columnas establecidas por Popcorn Flow

En el *board* utilizado por el equipo, se crearon cinco columnas: comprometidos, *ongoing, review* y *next.* La nomenclatura de las columnas es autodescriptiva. En la columna comprometidos se encuentran los experimentos que el equipo se comprometió a hacer. En *ongoing* los experimentos que están en curso. En la columna *review* el equipo evalúa los resultados logrados o no con el experimento terminado. Por último las columnas *next* y *done*. En *next* el equipo identifica los próximos pasos y finalmente se pasan a la columna *done* aquellos experimentos que terminaron. Una explicación más detallada sobre cómo fue aplicado este *framework* de trabajo se encuentra en la sección 11.2.1 Ciclo de vida de un problema - Investigación.

6.3.3 Desarrollo + Investigación

En esta etapa se entrelazaron el desarrollo y la investigación. Ya se había llegado a un punto en el que se podía empezar a trabajar en el desarrollo de la prueba de concepto pero aún quedaban cosas por investigar.

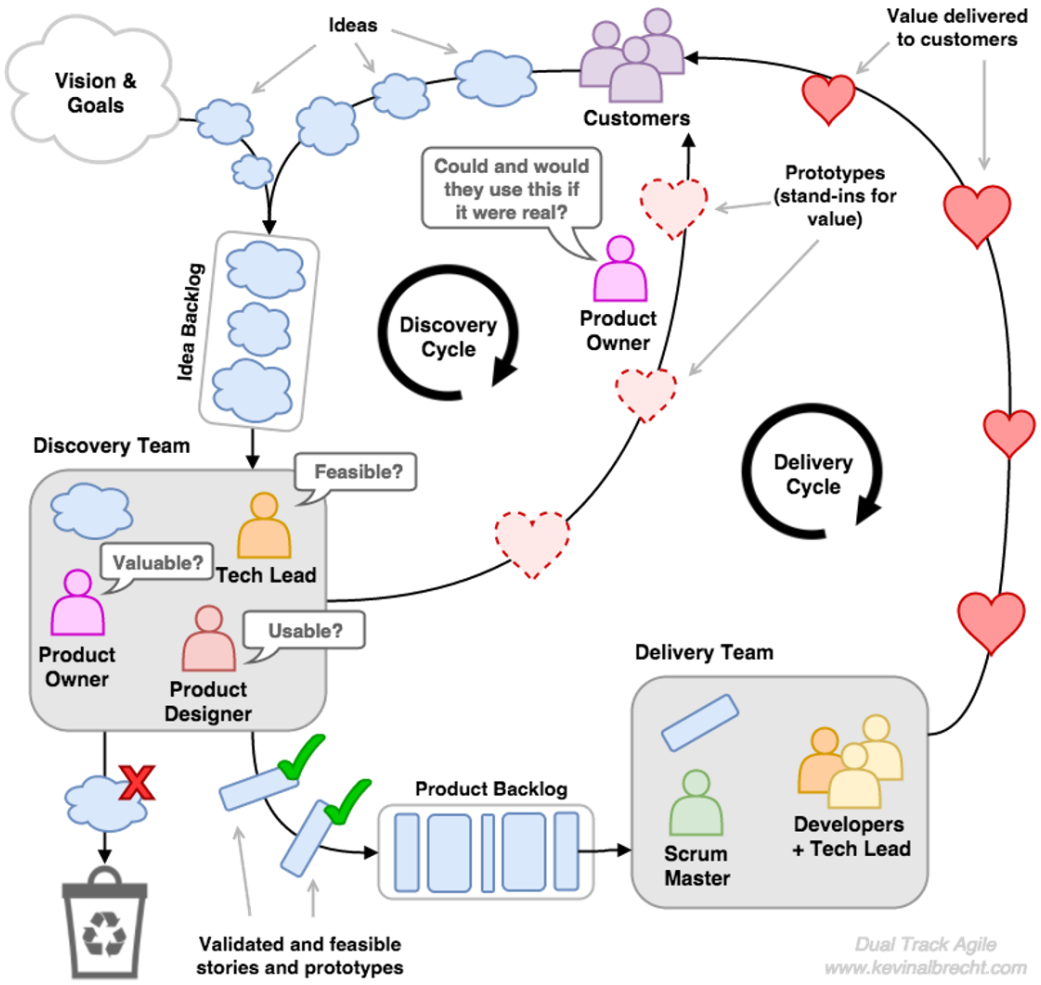


Figura 10: Flujo *Dual Track Scrum*

La gestión de dicha etapa se realizó utilizando un Dual Track Scrum. Donde del equipo principal se bifurca en uno de *Discovery* que es el encargado de llevar a cabo la experimentación con prototipos, descubrir el problema, investigar y experimentar con las tecnologías, entre otras

cosas, y por otro lado, el equipo de *Delivery* que recibe las historias de usuarios construidas por el equipo *Discovery.* Se encarga de priorizar las tareas en *Sprints* y entregar incrementos de producto [41]. Para este caso los equipos de *Discovery* y *Delivery* se conforman por las mismas personas. Uno de los *tracks* utilizados era el Popcorn Flow de la etapa anterior y el otro es un Scrum que fue introducido para manejar el desarrollo.

6.3.4 Desarrollo

Llegó un momento en el cual se estancó el proceso de la investigación. Esto fue considerado como el punto en el cual se podía cortar la investigación de manera formal.

En este momento se siguió trabajando únicamente con un *board* de *Scrum* y el *board* de investigación pasó a ser un Kanban 100% dedicado a tareas de gestión y documentación.

6.3.5 Documentación

Se cortó el desarrollo de la prueba de concepto en el mes de febrero del 2022 para pasar a trabajar 100% en la documentación. La gestión de esta parte del proyecto se hizo de manera informal, se armó el *outline* del documento y se trabajó para completar las secciones una a una.

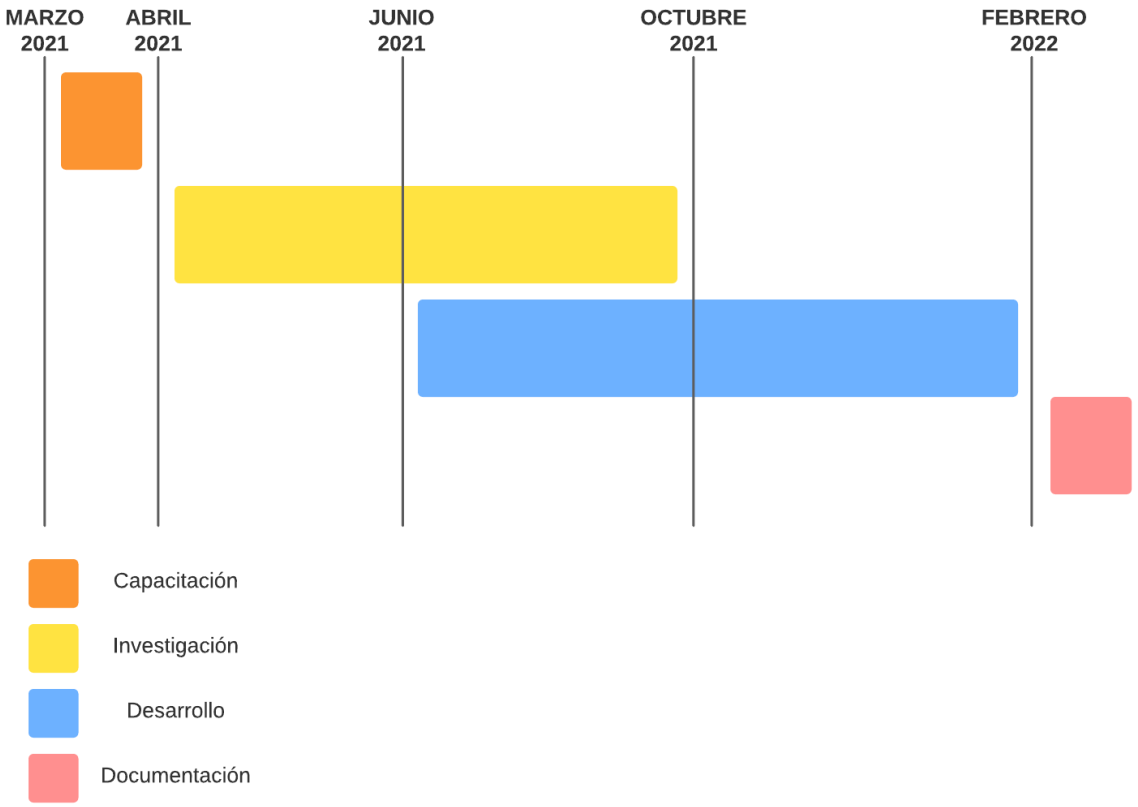


Figura 11: Esquema de etapas

El diagrama anterior muestra las etapas antes comentadas. De marzo a abril del año 2021 se realizaron exclusivamente las actividades de capacitación. A partir de abril y hasta junio del mismo año se realizaron actividades de investigación. Entre junio y octubre en paralelo a la investigación, comenzó con el desarrollo. Entre octubre y febrero del año 2022 el equipo dio por culminada la investigación y continuó solamente con el desarrollo. A comienzos de febrero se cerró el desarrollo y se trabajó en la documentación.

7 Ingeniería de requerimientos

En este capítulo se detalla y fundamenta el proceso de ingeniería de requerimientos realizado por el equipo en el proyecto. Se detalla el proceso realizado, la validación del problema, los interesados en el proyecto y los requerimientos funcionales y no funcionales del sistema.

7.1 Proceso

El proceso de requerimientos tuvo varias etapas. La primera consistió en bajar a tierra los requerimientos que fueron presentados en la propuesta original del cliente. Esta etapa simplemente consistió en leer el documento que el cliente presentó a la feria de proyectos de la Universidad ORT Uruguay y bajar cada uno de los requerimientos a un documento de texto, para entenderlos y analizarlos de manera individual.

Por otro lado se investigaron productos similares a GeneXus, es decir productos para el desarrollo *low code/no code*. La idea original de hacer esta investigación era conocer si estos productos tenían algún mecanismo para resolver los problemas que el cliente detectó como punto débil en su herramienta.

Paralelo a esto se analizó la bibliografía que ya existía del tema, una de ellas fue la tesis GX DevOps Extension [42] y documentación escrita por el tutor de esta tesis Darío Macchi, quién también tuvo un rol de experto de negocio especialmente en el proceso de requerimientos, dada la experiencia que tiene en el campo y las investigaciones que ha llevado a cabo del mismo.

Por otra parte se realizó un análisis del funcionamiento de herramientas como Cucumber, SpecFlow[15] y Concordeon[16], dado que el *parser* de Cucumber fue el elegido para utilizar en la Extensión, el equipo debía saber si los requerimientos planteados originalmente por el cliente

[15] https://specflow.org/
[16] https://concordion.org/

podrían ser satisfechos por dicho *parser.* La lista completa de las herramientas analizadas se puede encontrar en el anexo 14.16 Listado de herramientas similares analizadas.

Luego de las actividades previamente mencionadas el equipo fijó una reunión con Anibal Gonda y Gastón Milano para discutir los requerimientos iniciales que el sistema tendría. El objetivo de la reunión no era tener una lista inmutable de requerimientos durante todo el proyecto, si no acordar una lista de requerimientos viables de implementar para comenzar con el desarrollo de la prueba de concepto, he ir iterando sobre esta lista a medida que avanzaba el proyecto. El cliente siempre se mostró flexible respecto de los requerimientos.

7.1.1 Relevamiento

Con un panorama más claro, dado por las actividades mencionadas en la sección 7.1 Proceso el equipo comenzó a definir los requerimientos del sistema en forma de épicas. Es decir, las grandes funcionalidades que el sistema tendría. Estas épicas fueron inicialmente escritas en un board.

7.1.1.1 Encuesta

Se realizó una encuesta a usuarios de GeneXus, la misma se encuentra en el anexo 14.11 Encuesta y los detalles de la difusión se mencionan en 7.1.3 Validación. Las preguntas de dicha encuesta fueron escritas utilizando la técnica GQM *"goal, question, metric",* enfoque establecido orientado a objetivos de métricas de *software* para mejorar y medir la calidad del *software* [43]. En este caso puntual, esta técnica fue utilizada para crear preguntas que permitieran llegar a un objetivo definido correctamente y a su vez poder medir de manera cuantitativa las respuestas a dichas preguntas.

La siguiente figura ilustra la técnica GQM:

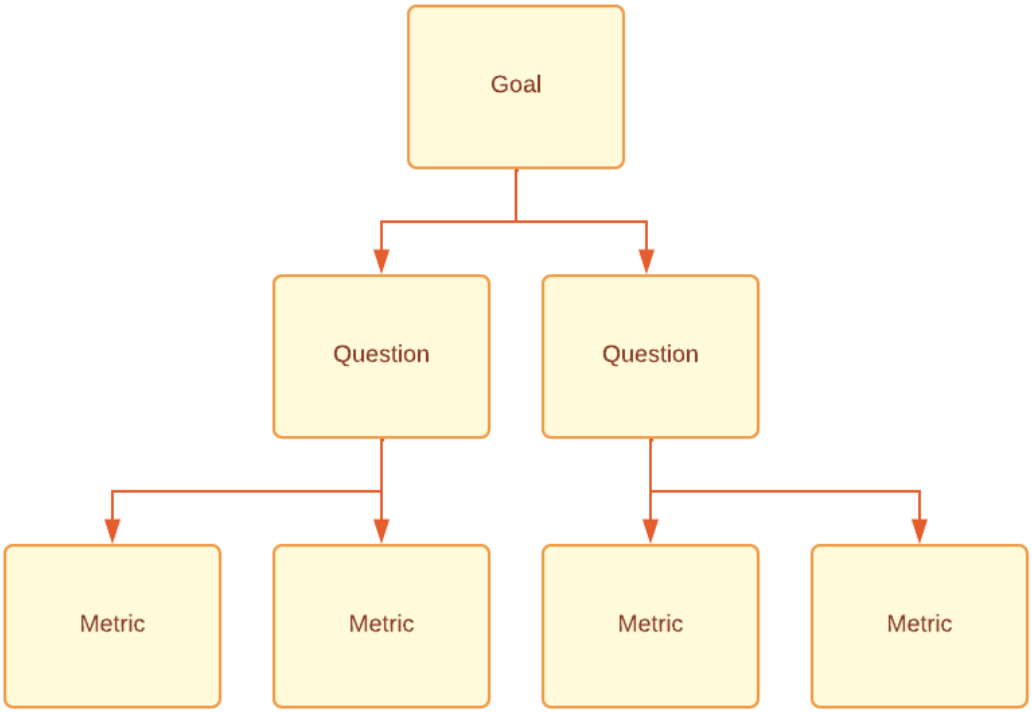


Figura 12: Diagrama GQM

En este punto, las preguntas fueron enfocadas a validar el problema existente, identificar usuarios y validar requerimientos. El procesamiento de los resultados de la encuesta realizada se encuentra en el anexo 14.11.2 Resultados.

7.1.2 Especificación

La especificación de los requerimientos se encuentra en las secciones 7.3 Requerimientos funcionales y 7.4 Requerimientos no funcionales. Éstas fueron escritas en forma de historias de usuario, que en sí mismas no son especificaciones, pero para el caso de este proyecto actúan como tal. Las historias de usuario se enfocan en la experiencia del usuario, en lo que el usuario quiere poder hacer. Mientras que los requerimientos tienden a ser más detallados y considerar aspectos técnicos de cómo el *software* debería funcionar [44]. El equipo entendió que para el

alcance de este proyecto, no era necesario hacer la separación porque todo lo que se construyó tuvo un impacto sobre el usuario final.

7.1.3 Validación

Con el objetivo de entender el problema detectado por el cliente en mayor profundidad, comenzaron a realizarse actividades de validación del problema.

La primera, fueron entrevistas mantenidas con Gastón Milano CTO de GeneXus y Anibal Gonda evangelizador técnico de GeneXus. Se mantuvieron dos entrevistas, las mismas no fueron grabadas ni tuvieron una estructura formal, pero sí había una serie de preguntas preparadas para ser el puntapié y discutir algunos temas que el equipo necesitaba aclarar.

La segunda fue investigar brevemente otras herramientas *low code/no code* existentes para entender si las mismas contaban con algún mecanismo para resolver problemas de trazabilidad. Esta fase simplemente consistió en buscar información en internet sobre dichas herramientas y leer opiniones en foros. Dado la complejidad que suponía probar cada una de las herramientas, en muchos casos pagar el costo de licencias o membresías y que las mismas eran alternativas a GeneXus y no directamente al problema presentado, se decidió no continuar por este camino.

Por otro lado, se realizó una encuesta a usuarios de GeneXus mencionada en 7.1.1.1 Encuesta. Esta encuesta realizada en Google Forms fue distribuida por GeneXus a clientes con distintos perfiles. La encuesta consistió en un conjunto de diez preguntas cerradas y dos preguntas abiertas. Durante el tiempo que la misma estuvo abierta, se obtuvieron treinta y una respuestas. Las preguntas y el resultado de la misma se puede ver en el anexo 14.11 Encuesta.

Como última actividad se realizaron posibles prototipos que potencialmente podrían solucionar el problema que planteó el cliente. Dichos prototipos fueron *mockups* realizados con la herramienta Balsamiq Wireframes[17]. Los mismos se pueden ver en el anexo 14.10 Mockups.

En las secciones siguientes se detallan las actividades que el equipo llevó a cabo.

[17] https://balsamiq.com/

7.1.3.1 Reuniones con el cliente

Como se explicó al comienzo de esta sección, lo primero que hizo el equipo fue realizar una lista de requerimientos, de forma informal en un editor de texto y esta se presentó al cliente. Con el cliente se discutió cada uno de ellos. Esta fue la primera validación de los requerimientos.

La validación de los requerimientos con el cliente se hizo de manera iterativa. A medida que se implementan requerimientos, diversos puntos, por alguna complicación, no se implementan en el momento. A estos puntos pendientes el equipo los llamó asteriscos. Estos asteriscos, puntos pendientes o posibles mejoras también fueron discutidos y priorizados en conjunto con el cliente.

7.1.3.2 Prototipación

El equipo realizó *mockups*, tanto del Cliente de Requerimientos, como del IDE. Los *mockups* realizados fueron de baja fidelidad. Estos *mockups* fueron presentados al tutor y a los primeros revisores. No se realizó una validación de requerimientos tradicional haciendo uso de *mockups*, ya que no fueron presentados a usuarios o al cliente. Esto último fue debido a tres razones:

- El flujo de revisión de requerimientos propuesto para el Cliente de Requerimientos no es un flujo novedoso. Este flujo es similar a lo que ya sucede en aplicaciones como Github o Azure DevOps en la creación de *Pull Requests.*
- Los *mockups,* como se mencionó anteriormente, son de baja fidelidad, esto conlleva a que no se pueda validar el *look and feel*.
- Por último, el desarrollo de una demo funcional alfa, que funcione con usuarios reales, no conlleva demasiado esfuerzo.

Los *mockups* se pueden ver en el anexo 14.10 Mockups.

7.1.3.3 Viabilidad técnica

Existían puntos de diversos requerimientos para los cuales existía incertidumbre en su viabilidad técnica, para eliminar esta incertidumbre se realizaron pequeñas pruebas de concepto. Las principales pruebas de concepto realizadas fueron las siguientes:

- Antes de enfocarse al cien por ciento, se dedicó tiempo a realizar diferentes pruebas con respecto a la integración con GeneXus. Para esto fue usado un repositorio de ejemplos que tiene GeneXus disponible. Se realizaron diferentes pruebas de concepto con este repositorio. Por ejemplo en el agregado de referencias entre dos objetos, en la creación de nuevos objetos o en el agregado de ítems al menú, entre otras pruebas.

- Fue realizada una prueba de concepto, la cual tenía el foco en validar la integración de un editor Monaco a un navegador. Esta prueba de concepto fue importante ya que su resultado tenía un gran impacto en el Cliente de Requerimientos.

- Por último, también sobre el editor Monaco, fueron realizadas diferentes pruebas de concepto, haciendo foco en las características del agregado de soporte para una nueva gramática. Se investigó la posibilidad de hacer esto, y una vez el resultado fue positivo, se hicieron pruebas. El soporte de una nueva gramática implica el *syntax highlighting, snippets* y sugerencias en general.

7.1.4 Prueba de concepto

Tal como fue mencionado en 3.1 Solución propuesta para validar el problema y la viabilidad del mismo se llevó a cabo una gran prueba concepto.

En base a la bibliografía consultada, una prueba de concepto debería satisfacer los siguientes puntos [20]:

¿Puede el producto ser construido o no?

Este punto significó un riesgo desde el principio, dado que el equipo no tenía asegurado que fuera posible integrar a GeneXus un conjunto de sistemas interconectados con las características planteadas. Para minimizar este riesgo, el equipo mantuvo reuniones con el área de *research and development* de GeneXus, más específicamente con Federico Azzato, y comenzó con el desarrollo desde el primer momento. Este enfoque permitió satisfacer esta pregunta. Se logró la construcción de una versión mínima de la Extensión que cumple con los requerimientos con más riesgo planteados con el cliente. Siendo la Extensión la parte que implicó un mayor riesgo, dado que es la más cercana a GeneXus y en la que fue necesario consultar con el área previamente mencionada de GeneXus como integrarse en determinadas partes para lograr satisfacer los requerimientos planteados.

Su principal objetivo es probar una idea o suposición en la vida real.

La idea que se intenta demostrar con la prueba de concepto construida, es que a través del mecanismo que se propone, se puede cumplir con los objetivos iniciales que planteó el cliente. Los objetivos iniciales que el cliente pretende satisfacer son:

- Especificar requerimientos de un modo estructurado.
- Mantener trazabilidad entre el requerimiento y su implementación.
- Realizar revisiones que involucren distintos actores del proyecto y aseguren la calidad.
- Aprovechar la estructura de dichos requerimientos para generar elementos nativos de GeneXus.

Describe una idea, funcionalidad y viabilidad de un producto.

Durante todo el proceso de este proyecto se describe la idea y sus principales funcionalidades que en algunos casos van cambiando de acuerdo a la viabilidad técnica que tienen y los resultados obtenidos en los procesos de validación.

Es una demostración teórica de que la solución es factible.

El resultado final de este proyecto, será un desarrollo que demuestra que la solución es factible y puede ser construida. Dado que es una prueba de concepto, el desarrollo no pretende salir a

un contexto de producción, ni satisfacer todo potencial que podría tener, es decir, el desarrollo está acotado en requerimientos dado que es una prueba de concepto para validación de la idea.

7.1.4.1 Proceso de prueba de concepto

Para la construcción del proceso de la prueba de concepto, el equipo tuvo en cuenta los siguientes cinco pasos:

Demostrar la necesidad del producto

En este paso los líderes de proyecto deben establecer cuál es la necesidad del producto mencionando al mercado objetivo y cuáles son los puntos débiles de dicha solución.

Para encontrar los puntos débiles se necesitan encontrar respuestas reales y verificadas. Estas respuestas se pueden obtener al entrevistar una muestra representativa de clientes.

Dado que la solución fue presentada por el cliente, en este punto el equipo asume que el cliente tiene un mercado objetivo para el producto, o al menos supone que tiene un mercado objetivo y conoce los puntos débiles de la solución.

Idear la solución correcta

A partir de las respuestas obtenidas en el punto anterior, el equipo debe comenzar a buscar distintas soluciones a los puntos débiles de los clientes. Teniendo en cuenta la factibilidad y la capacidad del cliente para poder implementarlas.

En este punto el equipo elaboró dos soluciones que intentan satisfacer las necesidades planteadas por el cliente y validó con el mismo que existe la viabilidad técnica para llevarlas a cabo. Las soluciones elaboradas, consistieron en dos *mockups* acompañados de dos documentos que explican las principales funcionalidades del sistema, y cómo éstas satisfacen las necesidades del cliente.

Crear un prototipo y testearlo

Una vez que se tiene una idea factible el equipo debe crear un prototipo basado en las características determinadas en el paso anterior. El prototipo debe ser funcional, de tal manera que las personas de la muestra puedan probar la funcionalidad por completo. Se deben documentar los comentarios de las personas de la muestra.

En este punto se desarrollaron todos los componentes necesarios para que el usuario pueda realizar una prueba *End-To-End* de todas las funcionalidades acordadas y mencionadas en 7.3 Requerimientos funcionales. Esto implica el desarrollo del Cliente de Requerimientos y la Extensión del entorno de desarrollo de GeneXus. El prototipo funcional fue presentado a un conjunto selecto de usuarios.

Recolectar *feedback* y documentar

Durante las pruebas de prototipo, el equipo debe recolectar y documentar *feedback* acerca de las experiencias, reacciones y otros detalles que aporten valor al grupo de muestra.

En este punto el equipo realizó entrevistas vía videoconferencia con distintos usuarios del grupo de muestra que fueron grabadas. Las técnicas utilizadas con los usuarios del grupo de muestra fueron *lab usability testing,* donde lo que se quiso medir fue la habilidad que los usuarios tenían de completar las tareas en el sistema [45] y una demo de las principales funcionalidades. Uno de los integrantes del equipo tuvo el rol de moderador en esta fase, para guiar al usuario en el flujo, pero no intervino en el proceso.

Presentar la prueba de concepto para aprobación

Con el concepto probado y mejorado en base al *feedback* obtenido el equipo de proyecto puede preparar la presentación a los interesados.

Este punto no fue cumplido estrictamente, ya que la presentación preparada fue anterior al punto “Recolectar *feedback* y documentar”. El equipo presentó a Armin Bachmann, *Product Manager* de GeneXus la idea desarrollada hasta el momento.

7.2 Interesados

En función de las actividades realizadas en 7.1.3 Validación, más específicamente en las primeras entrevistas realizadas con el cliente se comenzaron a determinar los interesados en el proyecto.

Se identificaron los siguientes interesados:

- **Desarrollador GeneXus:** Personas que convivirán con la herramienta que el equipo está construyendo. Son usuarios con suficiente nivel técnico para manejar una computadora, pero, basado en lo observado por el equipo en las personas que formaron parte del curso de capacitación de Analista GeneXus for Students v17 no tienen demasiados conocimientos sobre programación a más bajo nivel.

- **Desarrolladores con mucha experiencia:** Desarrolladores que gracias a GeneXus han podido trabajar con las últimas tecnologías sin necesidad de aprender nuevos flujos de trabajo, ni tecnologías. Dado que GeneXus se encarga de generar todo lo necesario.

- **Desarrolladores cuya primera experiencia con el desarrollo es GeneXus**

- **Usuario de negocio:** Es aquel que debe estar convencido que invertir en el uso de la herramienta a crear va a pagar dividendos, es decir, va a cumplir con el objetivo sin requerir un tiempo de capacitación excesivo. Lo esperable para éste será que la herramienta sea lo suficientemente simple de usar como para justificar que los desarrolladores a su cargo la utilicen y maximicen su productividad agregando la trazabilidad esperada a las historias de usuario.

- **Revisores y quienes tomen el rol de ingenieros de requerimientos:** Son quienes escribirán potencialmente los requerimientos y los revisarán hasta que estén en un estado lo suficientemente maduro para ser enviados a los desarrolladores quienes, dentro de GeneXus, tendrán un mecanismo para cargarlos. A priori no tienen licencia GeneXus, no son usuarios técnicos y querrán focalizar sus energías en producir requerimientos de calidad, y no en aprender a utilizar la herramienta.

- **Empresa GeneXus:** Es a quién el equipo cederá la herramienta.

- **Universidad ORT Uruguay:** Si bien el proyecto es cedido al cliente, en nombre de la Universidad ORT quedará publicado. Por lo que se espera que el proyecto sea de buena calidad y cumpla con las expectativas para un proyecto académico.

- **Darío Macchi:** Tutor de esta tesis, interesado en que el equipo apruebe la misma.

- **El equipo de tesis:** Quien llevará a cabo este proyecto y espera que el mismo satisfaga al cliente y a la Universidad ORT, dado que este proyecto es requisito para la obtención del título Ingeniero en Sistemas.

7.3 Requerimientos funcionales

A continuación, se listan los principales requerimientos funcionales discutidos y priorizados con el cliente en las primeras etapas del proyecto. Donde primero aparecen los requerimientos funcionales para la Extensión y luego los requerimientos funcionales para el Cliente de Requerimientos.

Los requerimientos se detallan en una forma rígida, sin respetar el formato de historia, esto debido a que estos fueron, en su gran mayoría, establecidos por GeneXus, el cómo se presenta su arquitectura y la idea y visión del cliente. El definir los requerimientos de esta forma nos deja sin la posibilidad de cumplir el atributo de Negociación de INVEST.

Cada requerimiento funcional se identifica con las iniciales RF, seguidas de un número de dos cifras, rellenado con 0 a la izquierda si corresponde y luego una letra que identifica si el requerimiento pertenece a la Extensión (E) o al Cliente de Requerimientos (C).

7.3.1 La Extensión

7.3.1.1 RF01E - Descarga de features

El sistema debe permitir a todo usuario que tenga configurada la *Knowledge Base* de su proyecto con su usuario del Cliente de Requerimientos y asociado el ID de proyecto, descargar todas las

features que fueron cerradas *(commit)* para ese proyecto, a través de un botón en el menú de la aplicación.

7.3.1.2 RF02E - Abrir feature en modo readonly

Al dar doble clic sobre una *feature* descargada en el explorador de la *Knowledge Base* se debe abrir un editor Monaco en modo *readonly* con el contenido de la *feature* con *syntax highlighting.*

7.3.1.3 RF03E - Menú para crear objetos

Al hacer clic derecho sobre el editor Monaco con el contenido de la *feature* debe aparecer un menú con las opciones de creación de un procedimiento, creación de una transacción y una opción para copiar el contenido.

7.3.1.4 RF04E - Soporte para la gramática Gherkin

La Extensión debe poder ser capaz de procesar la gramática Gherkin. Para esto debe incluir un *parser* de la misma.

7.3.1.5 RF05E - Soporte múltiple idioma

La *feature* debe poder ser procesada por lo menos en Español o Inglés.

7.3.1.6 RF06E - Creación de una transacción - Extracción del nombre

Al hacer clic sobre "*Generate transaction*" en el menú que se despliega sobre el editor Monaco, el sistema debe procesar la gramática. Si al escenario de la *feature,* identificado con la *keyword Scenario* o Escenario en español, le sigue una frase que cumple con la siguiente expresión `Create * business model entity` o en español `Crear entidad de modelo de negocios *` el sistema debe extraer el valor de *, pasarlo a *PascalCase* y crear una transacción con dicho nombre.

7.3.1.7 RF07E - Creación de una transacción - Inferencia de tipos

Al hacer clic sobre "*Generate transaction*" en el menú que se despliega sobre el editor Monaco. El sistema debe tomar y procesar la tabla que aparece debajo de la *keyword Given* o Dado en

español. Por cada columna de dicha tabla la Extensión debe inferir el tipo de dato menos genérico que abarque a todos los tipos de datos de la columna.

Por ejemplo, dada la siguiente columna:

Identity Machine
26582546
IBMAS400

Tabla 1: Ejemplo inferencia de tipo *Text*

El sistema debe inferir que el tipo de datos de la columna debe ser *"text"*.

En cambio, para la siguiente:

Population
345000
227410

Tabla 2: Ejemplo inferencia de tipo *Number*

El sistema debe inferir que el tipo de datos de la columna debe ser *"number".*

7.3.1.8 RF08E - Creación de una transacción - Tipos soportados

El sistema debe ser capaz de inferir los siguientes tipos de datos: *"text", "number", "boolean", "identifier", "date"* de una tabla de datos presente luego de la *keyword Given* o Dado en español.

7.3.1.9 RF09E - Creación de una transacción - Tipos nullable

Si alguna celda de la columna de que aparece debajo de la *keyword Given* o Dado en español tiene uno o más datos vacíos, el sistema debe inferir el tipo de dato, pero debe permitir valores `null`.

Por ejemplo, dada la siguiente columna:

Cohabitant Identifier
0a46891f-c72f-412d-8c69-d5768dbe7a61

Tabla 3: Ejemplo tabla con fila vacía

El sistema debe inferir que el tipo de datos de la columna es *"Identifier"* pero debe permitir valores `null`.

7.3.1.10 RF10E - Creación de una transacción - Tipos particulares

- *Number*

En el caso que la columna tenga datos numéricos que empiezan con cero, el sistema debe inferir que el tipo de datos es *"text"*.

Por ejemplo, dada la siguiente tabla:

Telephone Number
095852456
43346674

Tabla 4: Ejemplo inferencia de tipo TextNumber

El sistema debe inferir que el tipo de datos es *"text"* en vez de *"number"* para conservar el cero al principio del número.

Por otro lado si la columna contiene datos de la siguiente forma: 10, 0, 10.2, 10,3 el sistema debe inferir el tipo de datos "*number*".

- *Boolean*

Si la columna contiene true, True, TRUE, tRue, false, False, FALSE, faLse o cualquier combinación de letras mayúsculas y minúsculas que forman la palabra *true o false*, el sistema debe inferir que el tipo de datos es *"boolean".*

- *Identifier*

Si la columna contiene datos que cumplen con alguno de los siguientes formatos: ca761232ed4211cebacd00aa0057b223, CA761232-ED42-11CE-BACD-00AA0057B223, {CA761232-ED42-11CE-BACD-00AA0057B223}, (CA761232-ED42-11CE-BACD-00AA0057B223) el sistema debe inferir que el tipo de datos es *"Identifier".*

- *Date*

Si la columna contiene datos que cumplen con alguno de los siguientes formatos: 02-22-2021, 02/22/2021, 07.27.2021, 7/27/2021, 7-27-2021, 7.27.2021, 1-1/2021, 01.1-2021, 09/22.2021 el sistema debe inferir que el tipo de datos es "*date*".

7.3.1.11 RF11E - Creación de transacción - Autogeneración de Id

El sistema debe autogenerar la clave primaria de la transacción o Id, si el mismo no está presente. Para ello debe generar un atributo en la transacción de tipo *"number"* con el nombre de la transacción concatenado con "Id". Por ejemplo, dada la transacción *Person* generada a partir de la siguiente tabla:

Name	Continent	Official Language	Population
Uruguay	South America	Spanish	3500000
Czechia	Eastern Europe	Czech	10700000

Tabla 5: Ejemplo autogeneración de Id

El sistema debe agregar un atributo llamado PersonId de tipo "*number*" que será clave primaria de la transacción.

En cambio, si la tabla a partir de la que se genera la transacción ya tiene una columna Id, el sistema deberá usar esta columna como clave primaria de la transacción.

7.3.1.12 RF12E - Creación de transacción - Relaciones One-To-Many

El sistema debe soportar relaciones *one-to-many* entre distintas transacciones. Si en una tabla existe un campo que tiene el nombre de una transacción existente seguido de *"Id"* entonces debe establecer una relación entre ambas transacciones.

Por ejemplo, si se tiene la siguiente transacción *Country*, generada a partir de la siguiente tabla:

Name	Continent	Official Language	Population
Uruguay	South America	Spanish	3500000
Czechia	Eastern Europe	Czech	10700000

Tabla 6: Ejemplo relación *One-To-Many*

Y por otro lado se tiene la transacción *Person* generada a partir de la siguiente tabla:

First Name	Last Name	Born Date	Country Id
Juana	Fernández	03/08/1892	1
Antonín	Panenka	12/02/1948	2

Tabla 7: Ejemplo relación *One-To-Many*

Queda una relación establecida por el atributo *CountryId* entre *Person* y *Country.*

7.3.1.13 RF13E - Creación de transacción - Estandarización en nombre de atributos

Cuando se crea una transacción partiendo de una tabla, los atributos deben seguir la nomenclatura estándar de atributos de GeneXus. Es decir dada la transacción *Country* con el siguiente encabezado:

Name	Continent	Official Language	Population

Tabla 8: Ejemplo nombrado propiedades transacción

El sistema concatena el nombre de la transacción al nombre del atributo en *uppercamelcase,* también concatenado, es decir, para este caso creará una transacción con los siguientes atributos: `CountryName`, `CountryContinent`, `CountryOfficialLanguage`, `CountryPopulation` y agregará `CountryId` como clave primaria.

7.3.1.14 RF14E - Creación de transacción - Actualización de atributos

Si se ejecuta nuevamente la creación de una transacción que tiene el mismo nombre que una transacción ya existente, el sistema debe actualizar dicha transacción:

- Si se agregan nuevas columnas a la tabla, el sistema debe inferir el tipo de datos y crear los atributos correspondientes en la transacción existente. Por ejemplo, existe una transacción *Country* en el sistema generada a partir de la siguiente tabla:

Name	Continent	Official Language	Population
Uruguay	South America	Spanish	3500000
Czechia	Eastern Europe	Czech	10700000

Tabla 9: Ejemplo edición transacción

Y luego se ejecuta la creación de una transacción con el mismo nombre pero generada a partir de la siguiente tabla:

Name	Continent	Official Language	Population	Capital
Uruguay	South America	Spanish	3500000	Montevideo
Czechia	Eastern Europe	Czech	10700000	Prague

Tabla 10: Ejemplo edición transacción

El sistema debe agregar a la transacción existente el atributo *CountryCapital*.

- Si se cambia el tipo de datos en una columna de la tabla, el sistema debe actualizar el tipo de dato correspondiente en el atributo de la transacción. Dada la transacción *Person* creada a partir de la siguiente tabla:

Identity Document	First Name	Last Name	Born Date
79019286	Juana	Fernández	03/08/1892
52740951	Antonín	Panenka	12/02/1948

Tabla 11: Ejemplo edición transacción

Donde el tipo de datos correspondiente al atributo de la transacción Person*IdentityDocument* es inferido como *"number"*. Si luego se ejecuta la creación de una transacción con el mismo nombre, pero generada a partir de la siguiente tabla:

Identity Document	First Name	Last Name	Born Date
P31331823	Juana	Fernández	03/08/1892
P99990253	Antonín	Panenka	12/02/1948

Tabla 12: Ejemplo edición transacción

El tipo de dato del atributo *"PersonIdentityDocument"* debe ser actualizado a *"text"* en la transacción existente, para soportar los nuevos tipos de datos.

7.3.1.15 RF15E - Creación de transacción - Tipado explícito

Se debe poder indicar el tipo explícito de datos al crear la transacción. Para esto es necesaria una extensión de la gramática Gherkin. Para indicar el tipo de datos, en el encabezado de las columnas de la tabla, se debe poder luego de escribir el nombre de la columna indicar el tipo, cumpliendo con el siguiente formato `<column_name> : <data_type>`.

Por ejemplo, dada la siguiente tabla que creará la transacción *Country*:

Name	Continent	Official Language	Population	EU Member : Boolean
Uruguay	South America	Spanish	3500000	0
Czechia	Eastern Europe	Czech	10700000	1

Tabla 13: Ejemplo tipado explícito

Al inferir los tipos cuando se crea la transacción correspondiente a la tabla anterior, el sistema determina que el atributo *"CountryEUMember"* es de tipo *"number",* pero como se indica específicamente que es booleano, el sistema debe crearlo como tal.

7.3.1.16 RF16E - Creación de transacción - Tipos explícitos soportados

Los tipos explícitos soportados deben ser: *"date", "number", "text", "boolean"* e *"identifier".*

7.3.1.17 RF17E - Creación de transacción - Lenguaje tipos explícitos

El sistema debe soportar tipados explícitos en lenguaje Inglés y Español. Las *keywords* soportadas deben ser las siguientes:

Keywords EN	*Keywords* ES	Tipo de dato
Date	Fecha	date
Number	Número	number
Text	Texto	text
Boolean	Booleano	boolean
Identifier	Identificador	identifier

Tabla 14: Tipos de datos en español e inglés

7.3.1.18 RF18E - Creación de un procedimiento - Extracción del nombre

Al hacer clic derecho sobre el editor Monaco y seleccionar la opción *"Generate procedure w/tests"* el sistema debe procesar la gramática. Si al escenario de la *feature* identificado con la *keyword Scenario* o Escenario en español, le sigue una frase que cumple con la siguiente expresión `Create * procedure` o en español `Crear procedimiento *` el sistema debe extraer el valor de *, pasarlo a *uppercase* y crear un procedimiento con dicho nombre.

7.3.1.19 RF19E - Creación de un procedimiento - Creación de variables

El sistema debe tomar el encabezado de las columnas de la tabla que están luego de la *Keyword Examples* o Ejemplos en español para cada una de ellas inferir el tipo, si no se encuentra especificado específicamente al igual que se hace para la transacción[18] y crear dichas variables en el procedimiento.

[18] La inferencia de tipos de datos y el tipado explícito son comunes a los objetos nativos de GeneXus, como transacción y procedimiento.

7.3.1.20 RF20E - Creación de un procedimiento - Declaración de reglas

El sistema debe generar las reglas, con las variables de entrada y salida. Por ejemplo, dada la siguiente tabla:

starting_balance	amount	ending_balance
100	20	80
200	30	170
300	20	280

Tabla 15: Ejemplo creación de procedimiento

Al ejecutar la creación de un procedimiento el sistema debe tomar la última columna como parámetro de salida y el resto de las columnas como parámetros de entrada. Generando una regla de la siguiente manera: `Parm(in:&Starting_Balance, in:&Amount, out:&Ending_Balance);`

7.3.1.21 RF21E - Creación de un procedimiento - Declaración explícita de variable de salida

El sistema debe permitir la declaración explícita de variables de salida a través de una extensión de la sintaxis de Gherkin. Por ejemplo, tomando la tabla anterior, si se quiere indicar que *amount y ending_balance* sean variables de salida se debe hacer de la siguiente manera `@result(amount, ending_balance)`

7.3.1.22 RF22E - Creación de un procedimiento - Generación de Tests GxTest

El sistema debe generar la estructura necesaria para correr una colección de datos de prueba con GxTest. Entendiendo por estructura necesaria, a la creación de los objetos GeneXus *Unit Test, Data Provider y Structured Data Provider.*

7.3.1.23 RF23E - Creación de un procedimiento - Generación de colección de pruebas

Partiendo de los datos presentes en la tabla debajo de la *keyword Examples* o Ejemplos en español, el sistema debe generar una colección de datos de prueba en el *Data Provider* creado, que puedan ser ejecutados por GxTest.

7.3.1.24 RF24E - Agregado de referencias entre objetos GeneXus y objeto Feature

La extensión GeneXus permite agregar referencias entre las *features* existentes en la KB y los objetos creados manualmente a partir de esta. Esto es fácil de realizar, basta con dar clic derecho en la *feature*, elegir la opción "*Add references*" y luego elegir los objetos que queremos referenciar. Esto permite mantener la trazabilidad entre las *features* y los objetos generados de forma manual en la KB.

7.3.1.25 RF25E - Feedback del estado del sistema

El sistema debe proveer feedback del estado del mismo por la consola del IDE de GeneXus cuando:

- Se sincronizan las features del Cliente de Requerimientos con el proyecto.
- Se inicia la creación de una transacción.
- Se finaliza la creación de una transacción.
- Se finaliza la creación de una transacción con excepción.
- Se inicia la creación de un procedimiento.
- Se finaliza la creación de un procedimiento.
- Se finaliza la creación de un procedimiento con excepción.

7.3.2 Cliente de requerimientos

7.3.2.1 RF26C - Manejo de proyectos

El sistema debe listar los proyectos en los que el usuario logueado aparece como colaborador o es dueño de proyecto. Se deben mostrar los campos de ID correlativo y nombre de proyecto. Al seleccionar un proyecto de la lista de proyectos, el sistema debe mostrar una pestaña con:

- El ID correlativo, nombre y dueño del proyecto.
- Los colaboradores del proyecto donde se muestre el ID correlativo, nombre a mostrar y correo electrónico.
- Una lista con las features pertenecientes al proyecto, donde se muestre el título, el estado, el nombre a mostrar del usuario que la creó y la fecha de última actualización.

7.3.2.2 RF27C - Manejo de features

El sistema debe listar las *features* pertenecientes a los proyectos en los que el usuario logueado es dueño o colaborador. Se deben mostrar los campos de título de la feature, proyecto al que pertenece, estado, nombre a mostrar del usuario que la creó y fecha de última actualización.

7.3.2.3 RF28C - Crear un proyecto

El sistema, dado un usuario con sesión iniciada, debe permitir que este cree uno o más proyectos. Estos proyectos podrán tener colaboradores, que se indican en la creación, también tendrá un dueño, quien será quien que cree el proyecto, un título y una descripción. Un usuario colaborador de un proyecto deberá ver este en su lista de proyectos.

7.3.2.4 RF29C - Crear una feature

El sistema debe permitir a los usuarios crear *features.* Estás *features* deben estar en el contexto de un proyecto en particular. Las *features* tienen, en un principio, un título y un contenido.

7.3.2.5 RF30C - Editar contenido de una feature

El sistema debe permitir al usuario creador de una *feature* editar su contenido. Al editar el contenido, el sistema deberá reiniciar las aprobaciones de la *feature*. La edición se permite siempre que la *feature* no esté cerrada.

7.3.2.6 RF31C - Aprobar una feature

El sistema debe permitir a un usuario, perteneciente al proyecto de la *feature*, aprobar una *feature.* La *feature* no deberá poder ser aprobada por el usuario creador. Si una *feature* está cerrada, entonces no podrá ser aprobada por un usuario.

7.3.2.7 RF32C - Comentar una feature

Las *features* podrán ser comentadas por todos los miembros del proyecto en el que se encuentra la *feature*. Se podrá comentar siempre y cuando la *feature* no esté cerrada.

7.3.2.8 RF33C - Cerrar una feature

El sistema debe permitir, al dueño del proyecto, cerrar una *feature.*

7.3.2.9 RF34C - Eliminar una feature

El sistema debe permitir, al escritor de la *feature*, eliminar esta.

7.3.2.10 RF35C - Idioma del Cliente de Requerimientos

El Cliente de Requerimientos debe soportar inglés y español. El idioma debe ser tomado del navegador. Si el navegador está en otro idioma, el sitio se debe visualizar en inglés.

7.3.2.11 RF36C - Envío de notificaciones

El Cliente de Requerimientos realiza envío de notificaciones vía *email.* Estas notificaciones se envían en dos ocasiones, en caso de que se agregue un miembro a un proyecto, entonces se le notifica a este y, en el caso de que se agregue un comentario a una *feature*, se le envía un *email* al dueño de la *feature* correspondiente.

7.4 Requerimientos no funcionales

Para la determinación de los requerimientos no funcionales el equipo utilizó el modelo de calidad de *software* [46] que determina las características que se deben tener en cuenta para evaluar la calidad de un producto determinado.

Dicho modelo está compuesto por los factores presentados en la siguiente figura:

Figura 13: Modelo de calidad de *software*

En las siguientes secciones se describen las características del modelo que el equipo tomó en cuenta para la construcción del *software*. Los atributos que no estén presentes en el desglose simplemente no fueron tenidos en cuenta. Por ejemplo, no fue especialmente tenida en cuenta la eficiencia pues los *frameworks* y/o tecnologías que fueron utilizados ya hacen todo el trabajo necesario al respecto para este proyecto.

7.4.1 Mantenibilidad

Se recalca la importancia de la mantenibilidad en este producto dada su naturaleza de prueba de concepto. La empresa GeneXus debe poder tomar el *software* construido como punto de partida para futuros emprendimientos.

Este atributo comprende los siguientes puntos:

- **Modularidad**

- Se propone que los componentes definidos en la solución sean lo más independiente y cohesivos que se pueda, de modo que cambios futuros tengan impacto solo en las partes que realmente tengan sentido.

- **Reusabilidad**

  - Se propone que las partes más genéricas del *software*, como podría ser el editor, sean de código abierto y publicadas como tal para su reutilización por parte de la comunidad.

  - Se propone que los datos manejados por el sistema sean accesibles para futuros desarrollos con objetivos similares.

- Analizabilidad (no fue tenida en cuenta durante el desarrollo)

- **Capacidad de ser modificado**

  - Se propone la utilización de principios informales que maneja la comunidad de desarrollo como *KIS (Keep it simple)* y *DRY (Don't repeat yourself)* para mejorar la legibilidad del código en general.

  - Se propone la utilización de mecanismos provistos por las tecnologías para ofrecer máxima extensibilidad. Por ejemplo: *reflection*.

  - Se propone la utilización de patrones de diseño de *Gang of Four* para maximizar el nivel de cumplimiento de los principios SOLID.

  - Se propone tener en cuenta los principios de división de paquetes para maximizar cohesión y minimizar acoplamiento en los mismos.

- **Capacidad de ser probado**

  - Se considera que la aplicación de principios de diseño presentados en la sección de "Capacidad de ser modificado" aportan directamente a que el sistema pueda ser probado.

7.4.2 Usabilidad

Este atributo comprende los siguientes puntos:

- **Capacidad para reconocer su adecuación**
    - Se propone disponer de documentación sencilla que presente al usuario la información que puede esperar obtener del sistema. Por ejemplo: Swagger[19] de una REST API.
    - Se propone la escritura de manuales que permitan a los usuarios saber qué hace el sistema y cómo hacer uso de estas funcionalidades.
- **Capacidad de aprendizaje**
    - Se propone facilitar el aprendizaje con los manuales del punto anterior.
    - Se propone la utilización de *frameworks* de diseño para apelar al conocimiento previo que pueda tener el usuario de otras aplicaciones. Este conocimiento debería agilizar el proceso de aprendizaje.
- **Capacidad para ser usado**
    - Se propone la utilización de *Syntax highlighting* para el editor de código.
    - Se propone el uso de *snippets* para diferentes bloques comunes de código (*boilerplate*).
    - Se propone que la UI del sistema se adapte al idioma del sistema operativo del usuario.
- **Protección contra errores de usuario**
    - Se propone validar todo formulario que exista en aplicaciones cliente.

[19] https://swagger.io/

- Se propone validar toda entrada al sistema desde las API que fueran desarrolladas.
- Se propone minimizar las instancias de entrada libre de información para minimizar la cantidad de errores que el usuario pueda cometer.

- **Estética de la interfaz de usuario**
  - Se espera que el uso propuesto de *frameworks* de diseño permita el desarrollo de un sistema placentero a la vista sin necesidad de poner mucho esfuerzo en el diseño.

- **Accesibilidad**
  - Se propone el uso de *linters* con reglas que ayuden a cumplir con parte de las buenas prácticas de accesibilidad.

7.4.3 Portabilidad

Este atributo comprende los siguientes puntos:

- **Adaptabilidad**
  - Se propone asegurar que cualquier aplicación cliente sea compatible con las plataformas en las que se espera los usuarios las utilicen. Es decir, se espera que una aplicación web funcione en las últimas versiones de los navegadores más populares.
  - Se propone que la extensión del IDE funcione con la última versión y se descarta la necesidad de probar con versiones anteriores.

- **Facilidad de instalación**
  - Se propone que el Cliente de Requerimientos sea un SaaS, por lo que no requeriría instalación.

- **Capacidad de ser reemplazado**

- Se propone que toda información expuesta por las APIs desarrolladas pueda ser accedida desde cualquier cliente HTTP.

7.4.4 Seguridad

Este atributo comprende los siguientes puntos:

- **Confidencialidad**
  - Un usuario solo puede acceder a *features* asociadas a proyectos creados por él mismo o a los cuales haya sido expresamente invitado por el creador.
- **Integridad**
  - Un usuario solo puede modificar *features* creadas por él mismo.
  - Un usuario solo puede comentar *features* a las cuales tiene acceso según la lógica presentada en el punto de confidencialidad.
  - Un usuario solo puede aprobar *features* asociadas a proyectos creados por él mismo.
- **No repudio**
  - Toda entidad del sistema debe estar asociada con su creador. Esta información no puede ser modificada a mano por ningún usuario.
- **Autenticidad**
  - Todo usuario debe poder proveer ciertas credenciales al sistema, como un par usuario-contraseña, a lo que el sistema responda con un código que él pueda utilizar para identificarse en futuras solicitudes.
- Responsabilidad (no fue tenida en cuenta durante el desarrollo)

8 Arquitectura y diseño

La arquitectura utilizada, a grandes rasgos, está dividida en dos partes - la extensión de GeneXus (bordó) y el Cliente de Requerimientos (azul).

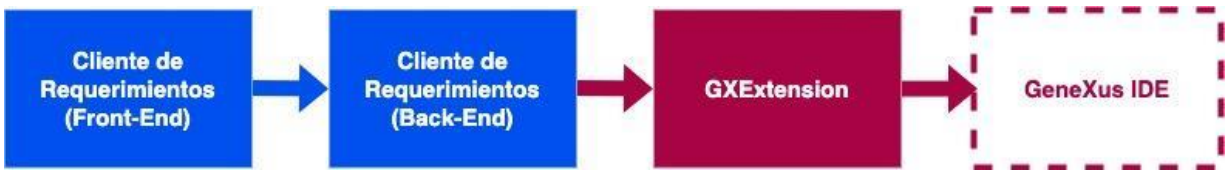


Figura 14: Diagrama básico de arquitectura

El Cliente de Requerimientos está compuesto a su vez por su aplicación cliente, la cuál se referencia como *frontend*, y su API correspondiente, también referenciada como *backend*.

La extensión es una dll que es utilizada desde el IDE, el cuál fue agregado al diagrama de cajas con líneas punteadas para representar que no fue hecho por el equipo, pero de igual modo es un componente clave de la solución.

La conexión entre ambas partes se da entre la API y la extensión.

8.1 Descripción

La arquitectura de alto nivel presentada en la introducción de esta sección se traduce al siguiente diagrama de componentes y conectores.

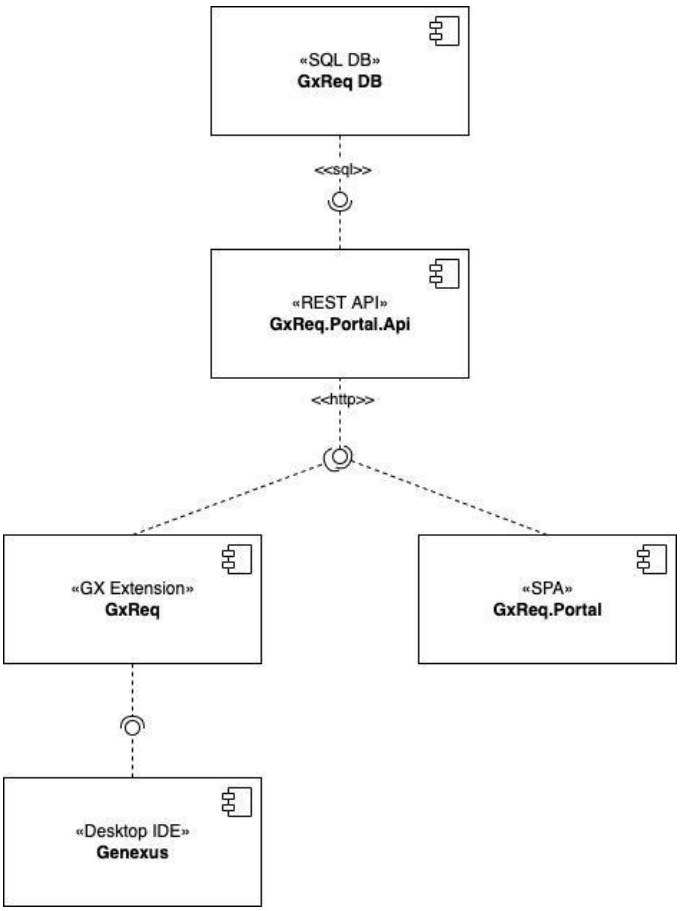


Figura 15: Diagrama de componentes y conectores

8.1.1 Catálogo de elementos

- GxReq.Portal: Aplicación web para usuarios que escriben requerimientos. Ofrece el manejo de proyectos, usuarios y, por supuesto, *features* o requerimientos. Refiriéndose a esta como *frontend*.

- GxReq.Portal.Api: REST API encargada del manejo de las entidades del sistema como lo son los requerimientos, los usuarios y los proyectos. Permite que esta información pueda ser accedida desde tanto GxReq.Portal como desde GxReq (Extensión).

- GeneXus (IDE): Entorno de desarrollo de GeneXus. Aplicación de escritorio utilizada por desarrolladores.

- GxReq: Es la extensión para el IDE que fue construida. Se conecta con GxReq.Portal.Api para obtener las *features* del proyecto que se está trabajando.

- GxReq DB: Base de datos relacional que almacena toda la información del sistema.

8.2 Atributos de calidad

Se toman en cuenta los atributos de calidad especificados en la normativa ISO 25010 [46] y se presenta una justificación por la cual el software resultante de este proyecto cumple con dichos atributos.

La propuesta original que se hizo está en la sección 1.1.4 Ingeniería de requerimientos. Además, se presenta el proceso de ingeniería de requerimientos en la sección anterior con una sección dedicada únicamente a Requerimientos no Funcionales, los mismos que se discuten en la presente sección pero vistos desde dicha perspectiva en 7.3 Requerimientos funcionales.

Se presentan únicamente los atributos que el equipo consideró a la hora de desarrollar el sistema y que lo llevaron a ser lo que es.

8.2.1 Mantenibilidad

Este atributo comprende los siguientes puntos:

- **Modularidad**

    - Ambos componentes ricos en lógica de negocio, en este caso el *backend* y la Extensión de GeneXus, están armados como la conjunción de paquetes altamente cohesivos y de baja dependencia entre unos y otros. Se puede perfectamente extender la funcionalidad de estos paquetes y se puede agregar nuevos. Para ver cómo esto se logró, ver las secciones de 8.4.2 Diseño de la Extensión de GeneXus y de 8.4.3 Diseño del backend del Cliente de Requerimientos.

- **Reusabilidad**

- El editor Monaco que se configuró para uso del sistema está en un repositorio público en GitHub y puede ser utilizado en aplicaciones que utilicen el *stack* web del mismo modo que el equipo utilizó este mismo editor para tanto el Cliente de Requerimientos como la Extensión de GeneXus.

- La primera prueba de concepto que se hizo para integrar el editor Monaco con la librería React Admin está publicada[20] y quienquiera que lo desee puede utilizar el componente resultante de la prueba en su aplicación.

- La información que expone el *backend*, al ser el mismo una REST API, es accesible por múltiples sistemas al mismo tiempo. Nuevas aplicaciones pueden desarrollarse para consumir estos datos. Además, al haber usado el estándar JSON:API se reduce el costo de desarrollo a la hora de comunicarse con la *API* pues existen múltiples clientes para muchos lenguajes de programación que abstraen toda la lógica necesaria para esta comunicación.

- Analizabilidad (no fue tenida en cuenta durante el desarrollo)

- **Capacidad de ser modificado**

  - Las áreas consideradas críticas, definidas en la sección 8.4.6 Áreas críticas del sistema, de la lógica de negocios, tanto del *backend* como de la Extensión de GeneXus, cuentan con *tests* unitarios y/o *End-To-End* (ver 10.3 Métricas) por lo que al introducir cambios existe la posibilidad de saber a grandes rasgos que estos no introdujeron fallas al sistema. También cuentan con *tests* unitarios y/o *End-To-End* el *frontend* y el *backend* del Cliente de requerimientos en todas las secciones del sistema que requirieron implementación y no fueron resueltas por los *frameworks* utilizados.

  - Se presentan en la sección de desarrollo varios módulos del sistema que hacen uso de Reflection como mecanismo de extensibilidad, es decir, se pueden agregar

[20] https://github.com/GxReqOrt/monaco-react-admin

nuevas funcionalidades específicas creando nuevas clases (implementando ciertas interfaces o extendiendo ciertas clases abstractas) y las mismas serán utilizables sin necesidad de que el desarrollador haga ningún mayor esfuerzo.

- **Capacidad de ser probado**
    - Los módulos con mayor carga de lógica de negocios son fácilmente testeables en aislamiento, es decir, tienen sus pruebas unitarias y se pueden crear nuevas de ser necesario.
    - La lógica de negocios de la API o *backend* fue testeada en su mayoría en un marco *End-To-End.* Se armó un sistema inspirado en el mismo que utiliza la librería para su propio *testing*, dado que integrantes del equipo ya habían contribuido a la misma y lo conocían relativamente bien, y así se simplificó el modo de hacer *testing*. Se eliminó la necesidad de hacer *mocks* en este caso, usamos una base de datos SQLite[21] independiente para cada corrida de pruebas, etc.
    - Las pruebas son extremadamente rápidas y sencillas de ejecutar dentro del IDE Visual Studio[22]. En el caso particular del *backend*, al utilizar .NET Core se pueden incluso ejecutar sin el IDE utilizando el CLI de .NET Core que incluso es multiplataforma. Esto permite continuar el desarrollo en cualquier computadora que cuente con sistema operativo Windows, Linux o MacOS.

8.2.2 Usabilidad

Este atributo de calidad fue especialmente tenido en cuenta durante el desarrollo del Cliente de Requerimientos, del *frontend* en particular, y durante el desarrollo del editor. Esto se debe a que es la parte del sistema que interactúa de forma directa con los usuarios finales. De todos modos, el *backend*, por más que sea a través del uso de un *framework*, mantiene altos niveles de usabilidad que serán justificados a continuación.

[21] https://www.sqlite.org/index.html
[22] https://visualstudio.microsoft.com/vs/

Este atributo comprende los siguientes puntos:

- **Capacidad para reconocer su adecuación**
    - El *backend* del Cliente de Requerimientos ofrece documentación de toda la información que expone a través de un Swagger público. Es sencillo para un desarrollador que quiera construir una aplicación que consuma esta API saber qué información puede esperar obtener de ella.
    - Se escribieron manuales, presentes en los anexos, tanto para el uso de la Extensión de GeneXus y para el Cliente de Requerimientos, que permiten a los usuarios saber qué hace el sistema y cómo hacer uso de estas funcionalidades.
- **Capacidad de aprendizaje**
    - Ídem al segundo punto del subatributo anterior (manuales escritos).
    - Se llevaron a cabo instancias de validación varias en las que se identificaron los puntos fuertes y aquellos no tanto en cuanto a la facilidad de aprendizaje del sistema para trabajar en mejorarlos.
    - Se utiliza el *framework* React Admin que, a su vez, hace uso de Material Design. Material Design es un conjunto de pautas, componentes y herramientas para el diseño de interfaz de usuario desarrollado por Google [47]. Esto significa que la UI del Cliente de Requerimientos cumple con patrones a los que los usuarios en general están acostumbrados y apela a ese conocimiento a la hora de aprender a utilizar el sistema.
- **Capacidad para ser usado**
    - La UI soporta múltiples idiomas (inglés y español están soportados en la prueba de concepto). Basta con agregar la definición de otro idioma para que este sea soportado. La definición consta de un archivo Javascript, el cual está compuesto por una propiedad que contiene clave-valor, en donde la clave es la cadena

utilizada en el código y el valor su traducción. A modo de ejemplo, la clave "not_authorized", en Inglés tiene un valor de *"You're not authorized to access this resource."*

- **Protección contra errores de usuario**
    - Todos los formularios del Cliente de Requerimientos cuentan con validación del lado del cliente y del lado de la API por lo que los usuarios no pueden enviar información con formato incorrecto a la base de datos.
    - El Cliente de Requerimientos muestra mensajes de error siempre que hay algún fallo, sea local como en la comunicación con la API.
    - El Cliente de Requerimientos permite al usuario deshacer cualquier acción de escritura contra la base de datos en los primeros 5 segundos con un *snackbar* como propone Material Design. Véase la aplicación de celular de Gmail[23] que permite deshacer la acción de archivar un email, por ejemplo.
    - La extensión de GeneXus solo requiere que el usuario escriba un archivo JSON de configuración y que haga clic en un botón que hace la sincronización, no se espera que haga nada más, minimizando la cantidad de instancias en las que el usuario podría llegar a equivocarse.
- **Estética de la interfaz de usuario**
    - El uso de React Admin con Material Design aporta a que la interfaz sea familiar y placentera a la vista del usuario.
    - El editor Monaco, al ser el mismo que utiliza el editor de código Visual Studio Code, es uno ya conocido por desarrolladores y también considerado, en general, como uno simple y cómodo de usar.
- **Accesibilidad**

[23] https://play.google.com/store/apps/details?id=com.google.android.gm&hl=en&gl=US

- Se hace uso de ESLint[24] durante el desarrollo del *frontend* del Cliente de Requerimientos con un *preset* que avisa al desarrollador cuando comete un error en cuanto a accesibilidad.

- Se hace el *test* de Lighthouse[25] ofrecido por el navegador Chrome que contiene un apartado de accesibilidad para verificar el cumplimiento con los estándares de la industria en este aspecto.

8.2.3 Portabilidad

Este atributo comprende los siguientes puntos:

- **Adaptabilidad**

  - Se prueba y asegura el funcionamiento del Cliente de Requerimientos en navegadores modernos.

    - Google Chrome V. ^49

    - Edge V. ^14

    - Mozilla Firefox V. ^50

  - La Extensión de GeneXus es compatible con la versión 17 del IDE dada la versión del SDK que se utilizó. No fue probado en versiones anteriores. Funciona tanto en GeneXus 17 Trial como en la versión *Full*.

- **Facilidad de instalación**

  - Se puede utilizar el Cliente de Requerimientos en cualquier computadora que tenga un navegador web moderno (ver punto anterior para navegadores probados) y acceso a internet. **Es un SaaS**.

[24] https://eslint.org/

[25] https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk?hl=es

- Se puede instalar la *dll* de la Extensión de GeneXus en el IDE con relativa facilidad desde el menú de manejo de extensiones que provee la herramienta. Para este punto podemos ver 14.6 Configuración de la extensión con la Knowledge Base, agregado como anexo.

- **Capacidad de ser reemplazado**
  - La información expuesta por el *backend* puede ser accedida desde cualquier cliente HTTP. Esto abre la puerta para que se exporte toda la información para un proyecto o cliente dado y se la importe en otra herramienta de ser requerido.

8.2.4 Seguridad

Este atributo comprende los siguientes puntos:

- **Confidencialidad**
  - Sin elementos notorios de arquitectura
- **Integridad**
  - Sin elementos notorios de arquitectura
- **No repudio**
  - Sin elementos notorios de arquitectura
- **Autenticidad**
  - Todo usuario debe, para ingresar al sistema, proveer un *token* de tipo JWT. Para obtener este *token* el usuario debe solicitarlo al sistema a cambio de su par email-contraseña. Las contraseñas no pueden obtenerse con ninguna consulta a la API, ni encriptadas ni en texto plano. De vulnerarse la base de datos, las contraseñas se encuentran hasheadas por medio del algoritmo bcrypt [48], por lo que no hay

modo de aplicarles un proceso inverso que resulte en las contraseñas en texto plano.

- Responsabilidad (no fue tenida en cuenta durante el desarrollo)

8.3 Tecnologías

A la hora de elegir las tecnologías a utilizar en el proyecto se realizó una tabla comparativa entre las opciones que se manejaban. Una tabla para definir lo que se utilizaría en el *frontend* y otra para definir lo que se utilizaría en el *backend.*

La **restricción de tecnología** definida por el proyecto es que la extensión GxReq debe ser programada en .NET Framework 4.7.2 con C#.

8.3.1 Tecnologías de backend

	.NET Core	node
Experiencia del equipo	Alta	Media
Consistente con las demás tecnologías*	Consistente	No consistente
Performance*	Muy alta	Media
Curva de aprendizaje	Alta	Media
Satisfacción de los desarrolladores*	Alta	Muy alta
Tamaño de comunidad*	Medio-bajo	Grande
Calidad de documentación*	Media	Alta

Tabla 16: Tecnologias *backend*

- “Consistente con las demás tecnologías” significa que, dado que existe una restricción en cuanto al lenguaje a usar para la extensión de GeneXus, hacer el *backend* en el mismo lenguaje era deseable.

- Para la *performance* los resultados de *benchmarks* ofrecidos por techempower[26] fueron tenidos en cuenta.
- La satisfacción de desarrolladores fue medida según lo expuesto en Slant[27].
- Para tamaño de comunidad se tuvo en cuenta de stackoverflow[28] la cantidad de usos de sus *tags*.
- Calidad de documentación es opinión del equipo

El balance final pone a **.NET Core** como la tecnología más apropiada para el proyecto por lo que fue la tecnología elegida para el *backend.*

8.3.2 Tecnologías de frontend

			
Experiencia del equipo	Media	Alta	Media
Curva de aprendizaje	Baja	Alta	Media
Satisfacción de los desarrolladores*	Muy alta	Media-baja	Muy alta
Tamaño de comunidad*	Grande y creciendo	Medio y bajando	Muy reducida
Calidad de documentación*	Muy buena	Muy buena	Muy buena

Tabla 17: Tecnologias *frontend*

- Para la satisfacción de los desarrolladores se tuvo en cuenta el enlace al pie de página[29].

[26] https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=fortune&l=zik0sd-sf&p=zik0zj-zijocf-zijocf-zik0zj-1ekf&f=zik0z3-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-zik0zj-hra0hr-zih7un-zik0zj-zik0zj-zik0zj-cn3

[27] https://www.slant.co/topics/1397/~best-web-frameworks-to-create-a-web-rest-api

[28] https://insights.stackoverflow.com/trends?tags=node.js%2Casp.net-core

[29] https://i0.wp.com/programmingwithmosh.com/wp-content/uploads/2021/02/Screen-Shot-2021-02-02-at-3.37.06-PM.png?ssl=1

- Para tamaño de comunidad se tomaron en cuenta los *tags* más representativos en stackoverflow y se observó la gráfica al pie de página[30].
- Se leyó el artículo Top Frontend Frameworks To Learn in 2021 [49].
- Calidad de documentación es según opinión del equipo.

El balance final pone a **React** como la tecnología más apropiada para el proyecto por lo que fue la tecnología elegida para el *frontend.*

8.3.3 Diagrama de componentes + tecnologías

Se presenta un diagrama con los componentes de la solución asociados a la tecnología que fue usada para su construcción.

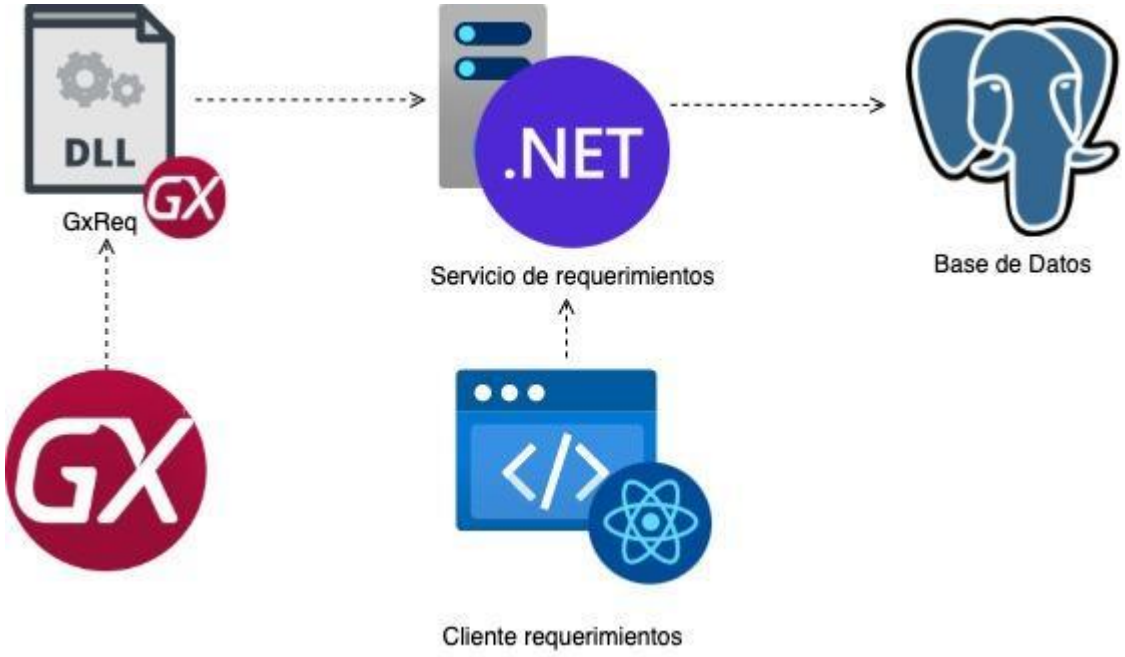


Figura 16: Diagrama de arquitectura con tecnologías seleccionadas

[30] https://insights.stackoverflow.com/trends?tags=reactjs%2Cangular%2Cvuex

8.4 Desarrollo

A lo largo del desarrollo de esta prueba de concepto se enfrentaron múltiples desafíos técnicos. De cada uno de ellos se aprendieron lecciones valiosas.

Es en esta sección que se intentará presentar lo más notable de esto.

8.4.1 Punto de partida

La premisa que se manejó desde un principio para el desarrollo de esta prueba de concepto fue la de "desarrollar la mayor cantidad de funcionalidad con la menor cantidad de código posible". Para ello era imprescindible no reinventar la rueda y seleccionar librerías y *frameworks* que apoyaran al equipo a la hora de cumplir con este objetivo.

Con respecto al *backend* Cliente de Requerimientos se hizo uso del estándar JSON:API[31] por medio del *framework* JsonAPiDotNetCore[32]. Esto permitió no escribir absolutamente nada del código repetitivo, comúnmente conocido como *boilerplate* en la industria, ya que es el *framework* el que provee todo eso; se desarrolló únicamente la lógica de negocios.

El *frontend* del Cliente de Requerimientos logró el objetivo planteado por medio del uso de la librería React Admin[33]. Para poder utilizarla se hizo una contribución a la librería ra-jsonapi-client[34]. La misma no fue tenida en cuenta por el autor así que se trabajó con el *fork*.

La extensión de GeneXus no se pudo simplificar por medio de la utilización de librerías al mismo nivel que el Cliente de Requerimientos por lo que tiene una complejidad mayor. Se utilizaron librerías para el hacer el *parsing* de *features* escritas en Gherkin pero no se pudo hacer mucho más en este sentido.

[31] https://jsonapi.org/
[32] https://www.jsonapi.net/
[33] https://marmelab.com/react-admin/
[34] https://github.com/GxReqOrt/ra-jsonapi-client

8.4.2 Diseño de la Extensión de GeneXus

Dada la naturaleza del producto a construir, que es una prueba de concepto, la extensibilidad y mantenibilidad del sistema a futuro toma un rol de suma importancia. Fue el mismo CTO de GeneXus, Gastón Milano, quién comunicó su intención de utilizar no solo la información que se obtenga como resultado de este proyecto sino también el código para seguir desarrollando esta idea.

8.4.2.1 Diagrama de paquetes de la Extensión de GeneXus

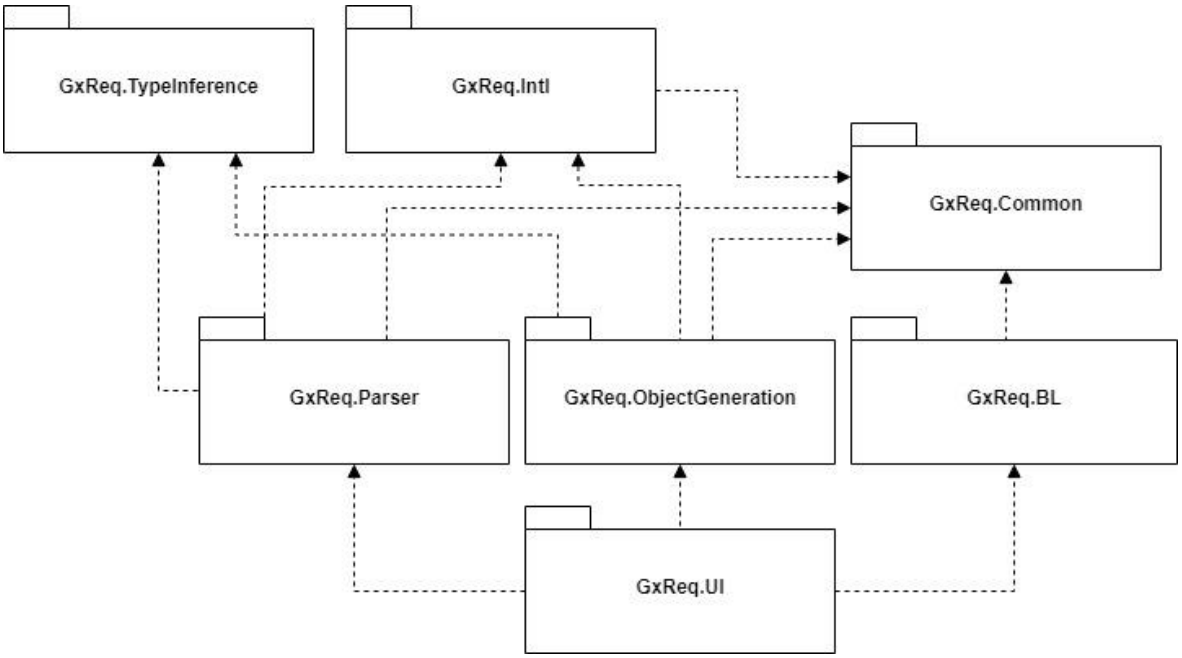


Figura 17: Diagrama de paquetes de la extensión GeneXus

8.4.2.2 Catálogo de elementos

GxReq.TypeInference

En este paquete se encuentra la clase que permite inferir el tipo, dado un string, que es más genérico. Esta clase "levanta" mediante Reflection aquellas clases que implementen la interfaz ITypeInferrer. Esta interfaz define tres propiedades:

- Priority: La prioridad define el orden en el que se van a evaluar los tipos, para inferir los tipos se evalúan todos en cierta prioridad, el primero que haga *Match*, entonces, es el tipo inferido. A modo de ejemplo el que más prioridad tiene es el string (el último que se evalúa) y el que menos tiene es el StringNumber (el primero que se evalúa). StringNumber es aquel número que se debe guardar como un string, como puede ser un número de celular, para no perder el "0" inicial.

- Type: Esta propiedad es de tipo eDBType, que es el tipo de las propiedades definidas por GeneXus.

- IsMatch: Función que permite saber si un *string* hace *Match* con el tipo.

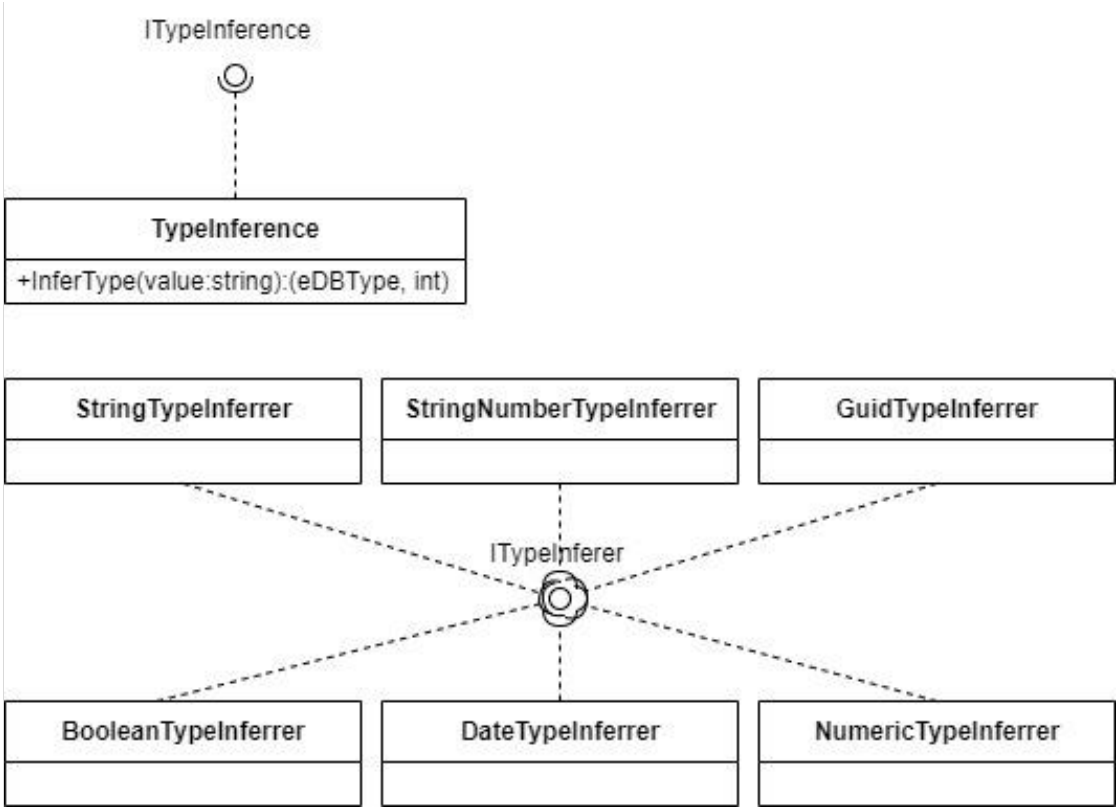


Figura 18: Diagrama de clases de la inferencia de tipos

Los tipos provistos por defecto son:

- String

- StringNumber
- Boolean
- Date
- Guid
- Numeric

Para definir un nuevo tipo, basta con agregar una clase que implemente esta interfaz.

GxReq.Intl

Este paquete provee el LanguageProvider, el cual dado el código del lenguaje definido en Gherkin, provee las traducciones para los diferentes tipos definidos, los cuales se pueden usar para definir de forma explícita el tipo de una columna de una tabla de Gherkin.

Aparte de las traducciones para los tipos, provee las traducciones para los patrones definidos para los títulos de los escenarios.

Este LanguageProvider reconoce los diferentes lenguajes definidos mediante Reflection. Si queremos definir un nuevo lenguaje bastaría con implementar la interfaz ILanguage. Actualmente hay dos lenguajes definidos:

- Español
- Inglés

GxReq.Parser

Este paquete contiene una única clase, la cual se encarga de, como su nombre indica, parsear el contenido de una *Feature*. Dado el contenido de esta, en formato string, nos brinda un documento Gherkin, este es un objeto provisto por la librería Gherkin[35].

GxReq.ObjectGeneration

Este paquete es uno de los principales de la solución, contiene la lógica que permite generar los objetos nativos de GeneXus programáticamente.

[35] https://www.nuget.org/packages/Gherkin/20.0.1

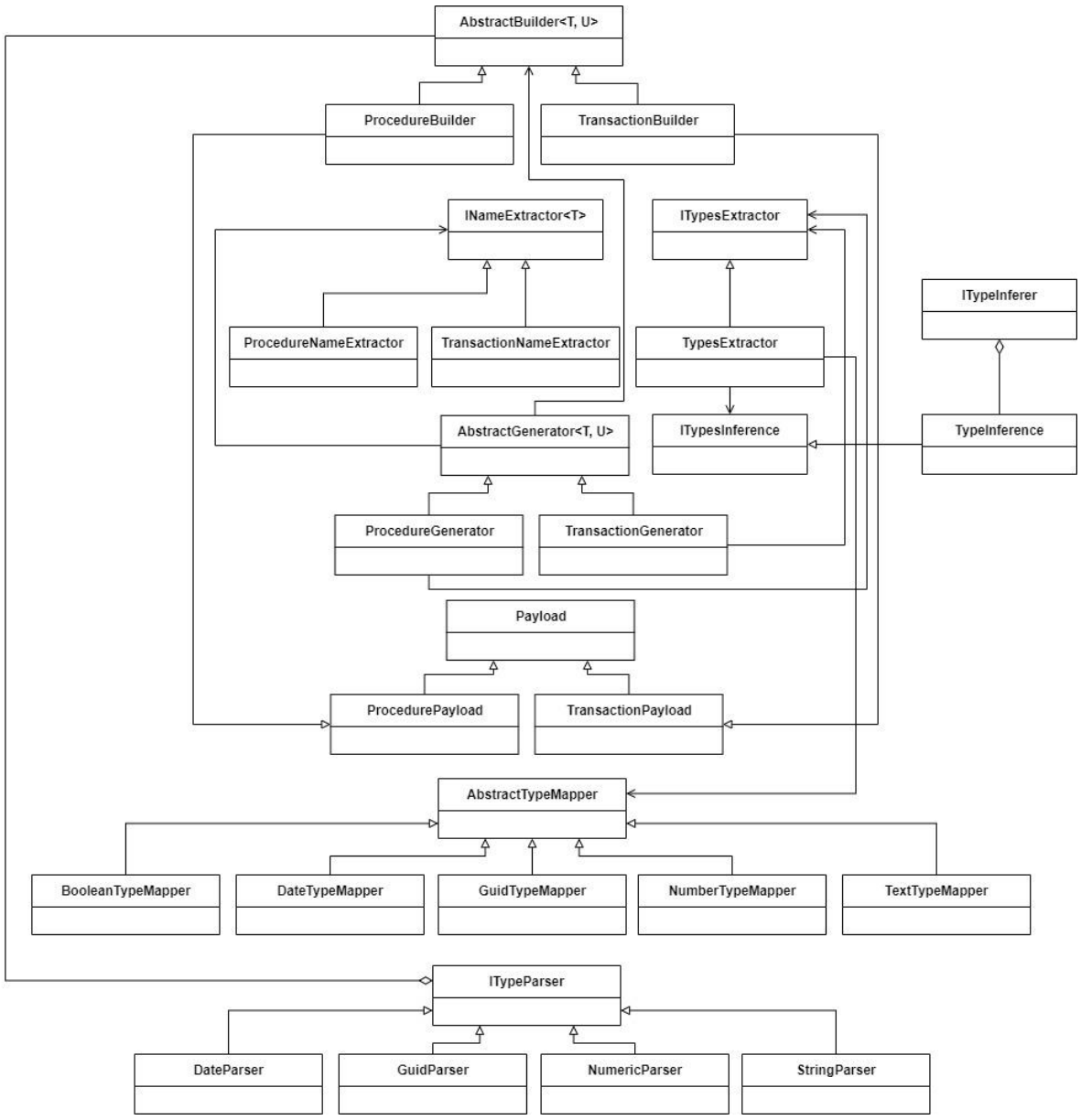


Figura 19: Diagrama de clases de parseo y mapeo de tipos

- Builders: son los encargados de la generación de los objetos nativos GeneXus. Actualmente existen dos, TransactionBuilder y ProcedureBuilder. Ambos extienden al AbstractBuilder, el cual define, de forma abstracta, el método Build. Es una aplicación del patrón de diseño definido por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (popularmente conocidos como *Gang of Four*) en su libro (*Design Patterns*, 1995) *Template Method*.

- Extractors: define la clase que dada una tabla definida en un escenario devuelve los elementos que luego se convertirán en las propiedades de las transacciones y procedimientos.

- Generators: contiene las clases TransactionGenerator y ProcedureGenerator, las cuales se encargan de armar los Payloads. Estos objetos son los que luego se van a utilizar en el *builder* para construir el objeto.

- Payloads: contiene los Payloads de cada objeto que, como se menciona antes, servirá para construir cada uno de éstos.

- TypeMappers: define los mapeos entre los tipos de datos de GeneXus y el tipo escrito por el usuario. A modo de ejemplo, el tipo GUID de GeneXus es definido por el usuario como Identificador o Identifier. Esto es así para que sea más usable por usuarios no técnicos.

- TypeParsers: define el parseo entre los tipos de datos y el formato requerido por la extensión GxTest para los casos de prueba. A modo de ejemplo, el valor de la fecha es #{valor_fecha}#.

GxReq.BL

En este paquete se encuentra la lógica referente a la interacción entre la *API* del Cliente de Requerimientos y la extensión. Además se encuentra la definición del objeto feature de GeneXus.

GxReq.Common

En este paquete se encuentran los elementos comunes y transversales a todos los otros paquetes. En este están las interfaces, modelos, clases abstractas, helpers, atributos y el logger.

GxReq.UI

En este paquete se define la interacción con el editor Monaco integrado al IDE y también los comandos definidos. Elementos de la UI en general.

8.4.3 Diseño del backend del Cliente de Requerimientos

8.4.3.1 Diagrama de backend del Cliente de Requerimientos

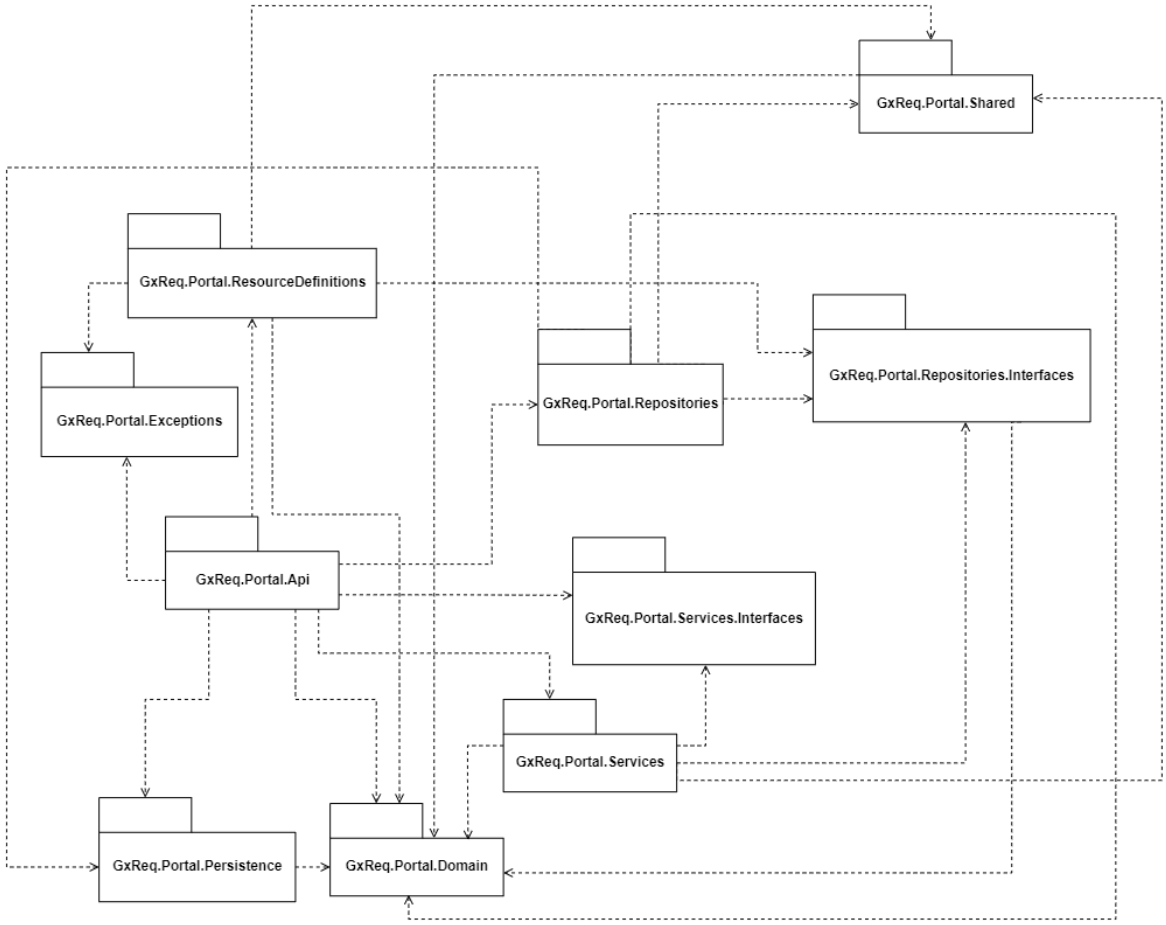


Figura 20: Diagrama de paquetes *backend* Cliente de Requerimientos

8.4.3.2 Catálogo de elementos

NOTA: Los paquetes `<Nombre>.Interfaces` se crearon para las interfaces de los servicios y repositorios. Permitiendo así no depender directamente de los detalles de implementación, sino que de la abstracción, esto brinda una mayor modificabilidad.

GxReq.Portal.Api

En este proyecto se definen los diferentes controllers de la API. Al definir los *controllers*, *JsonApiDotnetCore* autogenera *endpoints* básicos para cada entidad. En este proyecto se definen también los *endpoints* personalizados, DTOs y *Exception Handlers*.

GxReq.Portal.Exceptions

En este paquete se definen las excepciones personalizadas que se usan en el sistema. Ejemplo de estas son: `BadRequestException`, que es usada en caso de que una solicitud esté mal formulada, entre otras varias.

GxReq.Portal.Persistence

Se define en este proyecto el contexto de la base de datos para el ORM Entity Framework Core, *Object-Relational Mapper* de código abierto y multiplataforma basada en Entity Framework [50], tanto el de producción como el de *testing*. También se guardan las diferentes migraciones existentes en la solución.

GxReq.Portal.Domain

En este proyecto se definen las entidades del dominio. Como pueden ser User o Project, entre otras.

GxReq.Portal.Repositories.Interfaces

En este proyecto se definen las interfaces para los repositorios.

GxReq.Portal.Repositories

En este proyecto se definen los repositorios, en éstos se definen operaciones personalizadas sobre la base de datos puesto que la funcionalidad básica de los mismos viene dada por el *framework JsonApiDotNetCore*.

GxReq.Portal.Shared

Este es el proyecto compartido en donde se define lógica común necesaria en varios proyectos.

GxReq.Portal.Services.Interfaces

Se definen las interfaces para el proyecto de servicios.

GxReq.Portal.Services

Se definen los servicios personalizados, los que no vienen definidos por JsonApiDotnetCore, actualmente se encuentra el servicio de Login.

GxReq.Portal.ResourcesDefinition

Resource definition[36] es un concepto definido por el *framework* JsonApiDotNetCore y son clases que definen lógica de negocios específica de un modo orientado a recursos. En este paquete se encuentran todas estas clases.

8.4.4 Trabajando con el editor Monaco

En primer lugar hay que responder la pregunta: ¿Por qué Monaco? - Es un editor sumamente versátil, aquel que utiliza Visual Studio Code, es aquel que la empresa GeneXus está utilizando para todas sus extensiones desarrolladas *in-house* y por ende el que recomiendan utilizar. Además, se puede hacer uso de extensiones de Visual Studio Code para obtener nueva funcionalidad para integrar al editor de manera relativamente sencilla.

Otro motivo por el cuál Monaco es una elección muy acertada es que, al ser un editor 100% web, no solo funciona dentro del IDE de GeneXus sino que también se puede usar en el navegador. Esto fue, de hecho, lo que hicimos. El Cliente de Requerimientos ofrece un Monaco para editar los requerimientos con *syntax highlighting*, *snippets*, etc.

Los objetivos al comenzar a trabajar en el editor fueron los siguientes:

[36] https://www.jsonapi.net/usage/extensibility/resource-definitions.html

- Se necesita tener *syntax highlighting* para el lenguaje Gherkin.
    - No se quiere escribir la sintaxis del lenguaje en Monarch, que es una tecnología para especificar sintaxis de un lenguaje en JSON. En su lugar se quiere utilizar la definición de la sintaxis presente en la extensión Cucumber (Gherkin) Full Support[37] la cual está especificada usando TextMate Grammars.
- Se necesita tener *snippets* para que los usuarios del editor tengan una mejor experiencia a la hora de editar texto.
- Se necesita modificar las acciones del menú contextual que aparece al hacer "clic derecho" con el *mouse*.
    - Se quiere exponer aquí la funcionalidad de crear procedimiento con pruebas unitarias.
    - Se quiere exponer aquí la funcionalidad de crear transacciones.
- Se necesita tener *dark* y *light theme* porque la estética del IDE GeneXus requiere el *light theme* pero aún así se quiere que el Cliente de Requerimientos tenga un *dark theme*.
- Se necesita que el editor pueda ser de solo lectura para que los desarrolladores dentro del IDE de GeneXus no tengan la posibilidad de modificar los requerimientos.

Como punto de partida se hizo un *fork* del repositorio monaco-™[38]. El mismo es una prueba de concepto de la integración de TextMate Grammars con Monaco, en particular utilizando la gramática del lenguaje Python.

El primer paso fue cambiar el lenguaje a Gherkin. Se obtuvo la definición de la gramática de la extensión Cucumber (Gherkin) Full Support[39].

[37] https://marketplace.visualstudio.com/items?itemName=alexkrechik.cucumberautocomplete
[38] https://github.com/GxReqOrt/monaco-™
[39] https://marketplace.visualstudio.com/items?itemName=alexkrechik.cucumberautocomplete

Los demás requerimientos se pudieron cumplir simplemente configurando el editor con el JSON que recibe al instanciarse.

Para utilizar el editor en GeneXus hubo que hacer un *build* y copiar los archivos a la carpeta de instalación de GeneXus para referenciarlos desde la Extensión. Para más información ver el repositorio presentado en la siguiente sección de lecciones aprendidas.

Para utilizar el editor en el navegador se adaptó el código de esta demo para funcionar como un Input de React Admin y se lo utilizó como cualquier otro *form control*. Se hizo una prueba de concepto de esto antes de usarlo en el producto final que se puede encontrar en el repositorio público monaco-react-admin[40].

8.4.5 Ingeniería inversa

En la etapa de desarrollo, el equipo, ante dudas en el funcionamiento de la aplicación GeneXus, realizó ingeniería inversa. La ingeniería inversa implica el análisis de un producto con el objetivo de obtener conocimiento sobre su funcionamiento [51]. Un ejemplo de ingeniería inversa aplicada al ámbito del software es la reversión de código de bajo nivel, por ejemplo código intermedio compilado, a código de alto nivel, con el que fue escrito, este es el caso aplicado en este proyecto. El equipo aplicó la descompilación de dlls para investigar ciertas funcionalidades de la aplicación GeneXus, para luego poder consumirlas desde la extensión. Para esto fue usada la herramienta DotPeek. Esta es una herramienta gratuita que permite descompilar ensamblados de .NET, obteniendo código C# [52].

Previo a la realización de la ingeniería inversa, el equipo se comunicó con GeneXus para avisar si había algún problema en aplicar esta técnica. Desde GeneXus se informó que no había ningún problema en hacerlo.

[40] https://github.com/GxReqOrt/monaco-react-admin

8.4.6 Áreas críticas del sistema

En la solución se identifican áreas críticas. Estas áreas críticas son definidas según su relevancia y su riesgo, teniendo en cuenta los requerimientos solicitados por el cliente.

En general, la parte más crítica de la solución es la extensión GeneXus, esta es la que se integra con el IDE GeneXus y realiza la transformación desde una *feature* escrita en Gherkin a los objetos nativos de GeneXus. También tuvo una considerable cantidad de riesgo durante el desarrollo, ya que el equipo no tenía claro si cada punto a desarrollar se podía realizar de la manera pensada en un inicio. Ejemplo de esto fue el mecanismo de autenticación, en un inicio se pensó hacer este mecanismo mediante OAuth[41], pero debido a una restricción de GeneXus se tuvo que realizar de otra manera.

A continuación se presenta un listado de las áreas críticas de la extensión.

- **Generación de objetos nativos GeneXus**: Esta funcionalidad es la que se encarga de generar, dada una *feature,* los objetos nativos de GeneXus. Esta es sin dudas una de las partes más importantes de nuestra solución.

- **Inferencia de tipos:** Esta funcionalidad es la encargada de, dada una columna de una tabla, si no tiene el tipo definido de forma explícita, de inferir el tipo de está, haciendo uso de los datos de cada fila. Esta funcionalidad es importante, ya que, si funciona incorrectamente, impacta directamente en las propiedades de los objetos creados. Esta funcionalidad es extensible, se usó Reflection y se crearon múltiples pruebas unitarias.

- ***Parsing* de tablas de la gramática Gherkin**: Esta funcionalidad, y haciendo uso de la anterior, permite, dada una tabla, definir las propiedades de un objeto GeneXus, ya sea una transacción o un procedimiento. Para esta característica también se agregaron pruebas unitarias, haciéndola lo más mantenible posible.

[41] https://es.wikipedia.org/wiki/OAuth

8.4.7 Lecciones aprendidas

Fue entregado a GeneXus, junto con otros documentos, un repositorio[42] cuyo *README* presenta toda la información que hubiera sido útil tener desde el principio del proceso de desarrollo, es decir, contiene las lecciones aprendidas desde el lado técnico.

Este repositorio, además, contiene los ejemplos que la empresa provee de extensiones para su plataforma actualizados para que funcionen con versiones modernas de .NET Framework y MSBuild.

[42] https://github.com/GxReqOrt/Samples

9 Gestión del proyecto

Para el proyecto se utilizó un **marco de gestión ágil**. La razón es que se necesitaba tener una buena capacidad de reacción al cambio para proveer al cliente con el máximo valor posible y era completamente imposible tener todos los requerimientos de antemano, imposibilitando la adopción de metodologías tradicionales.

9.1 Herramientas de gestión

Para la gestión del proyecto se hizo uso de distintos *boards* en la herramienta Trello[43] para tener registro de las tareas hechas y por hacer en las distintas etapas del proyecto.

Se utilizó a su vez una herramienta desarrollada por el equipo para extraer información desde estos *boards* y generar reportes. Dichos reportes son la velocidad del equipo, listados de tareas hechas en un sprint y fecha de culminación para cada tarea terminada.

Se usó también un calendario compartido en Google Calendar[44] para llevar de forma automática el control de las ceremonias correspondientes a los *frameworks* utilizados y para manejar las reuniones en Google Meet[45] de igual manera.

La gestión de configuración, llamada SCM por sus siglas en inglés (*Software Configuration Management)* es discutida en la sección 1.1.8 Gestión de la configuración.

Para el control de horas invertidas en el proyecto por parte del equipo fue utilizada la herramienta Toggl. Esta es una herramienta de registro de esfuerzo, que permite generar diferentes reportes y es gratuita. Esta herramienta tiene tres grandes componentes, Toggl Track, Toggl Plan y Toggl Hire, el equipo utilizó solamente Toggl Track [53].

[43] https://trello.com/
[44] https://calendar.google.com/
[45] https://meet.google.com/

9.2 Etapas y principales hitos del proyecto

Las etapas que se manejaron se mencionan ya en la sección 6.3 Etapas del proyecto.

Los hitos que se manejaron fueron los siguientes:

- Al finalizar la etapa de investigación se consideró que el equipo estaba preparado para comenzar con el desarrollo de la prueba de concepto.
- Durante el desarrollo cada *release* se consideró como un hito.
- Al dar por terminado el desarrollo el próximo hito fue la entrega de los documentos y artefactos para culminar con la aceptación de la empresa GeneXus.
- El último hito fue la compleción del presente documento.

9.3 Aplicación de frameworks de gestión

En esta sección se describe cómo se utilizaron los *frameworks* de gestión con los que el equipo acordó trabajar en las diferentes fases del proyecto. Se explica cómo se aplicó cada *framework* y en qué fase fue utilizado.

9.3.1 Popcorn Flow

Para la etapa de investigación inicial que se hizo, se utilizó este *framework*. La idea era tener un marco de trabajo ordenado pero muy simple, que permitiera al equipo concentrarse en las tareas que había que hacer sin necesidad de aprender una modalidad compleja y burocrática de trabajo.

El resultado de este Popcorn Flow pasó a ser el *product backlog* del Scrum presentado en la siguiente sección. En particular, se arma el plan de *releases* con lo que se aprendió de esto y el desglose de dicho plan pasó a ser el *backlog*.

Se hizo un *board* en Trello para aplicar esta metodología en el que se ponían todas las ideas y problemas a investigar de modo informal y se hacía una puesta a punto por semana en la reunión

que llamaremos *weekly catch-up*. Esta reunión se mantuvo en toda fase del proyecto y se la utilizó para todas las metodologías.

9.3.2 Scrum

La aplicación de Scrum viene de la mano con el comienzo del trabajo en el desarrollo de la prueba de concepto. Es decir, no se aplicó Scrum en la primera etapa del proyecto ni en la última. Para ver en detalle las etapas véase 6.3 Etapas del proyecto.

El *framework* fue adaptado al proyecto del siguiente modo en cuanto a sus ceremonias propuestas:

- No se hicieron *daily meetings*, se hicieron *weekly catch-up meetings* a mitad de cada *sprint* para destrabar todo lo que fuera necesario destrabar con tiempo antes de llegar a la fecha de cierre.
- Las ceremonias de *sprint planning* y *sprint review* fueron hechas en conjunto ya que la parte de *review* era lo suficientemente corta y concisa como para juntar ambas en una sola.
- Solo se hizo la *sprint retrospective* el primer *sprint*. Dada la conformidad del equipo con la metodología (tanto su sencillez como su efectividad) se decidió no hacer más estas reuniones.

El modo en el que se armó el *sprint backlog* fue progresivo. En cada ceremonia de *sprint planning* que se llevaba a cabo el proceso era el siguiente:

- Dado el plan de *releases*, se sabía el objetivo del *release* en el que se estaba trabajando.
- Dado el objetivo, se seleccionaban las tareas que el equipo estaba en posición de atacar.
- Una vez seleccionadas las tareas se las escribía como historias de usuario.
    - Para escribir cada historia se siguió el *template* propuesto por Atlassian en su documentación [54]. **Como [actor], Quiero [funcionalidad], Para [beneficio].**

    - Para cada historia de usuario se definían **criterios de aceptación**.
        - Estos criterios estaban escritos en Gherkin. Se hicieron excepciones solo en casos puntuales y excepcionales.
        - Una tarea podía solo pasar a revisión posterior a la compleción de todos estos criterios según su encargado.
        - La compleción de los criterios venía marcada en un *checklist* presente en la *card.*

- Dadas las historias de usuario ya especificadas y discutidas se procedía a estimar las mismas.
    - La estimación se llevó siempre a cabo con la metodología Planning Poker. Dicha metodología ayuda a los equipos ágiles a estimar el tiempo y esfuerzo necesario para completar cada iniciativa del *product backlog* [55]. La secuencia que se utilizó es 0, 1, 2, 3, 5, 8, 13, 20, 40 y 100, secuencia basada en Fibonacci, que la recomendada por Planning Poker.
    - Para hacer uso de la herramienta automática de extracción de información cada *card* tenía que tener un título que cumpliese con el siguiente formato: **[XXSP] Título.**
        - De este modo se sabe que toda *card* tiene estimación.
        - Al estar la estimación siempre en el mismo lugar y con el mismo formato se la podía extraer con la herramienta y realizar cálculos automáticos de velocidad y otros.

El estado de cada tarea venía dado por la columna en la que se encontraba en el *board*. Existían las columnas *TODO (*léase *to do* en inglés) con las tareas del *sprint backlog*, *In Progress* con las tareas que estaban siendo hechas, *Awaiting Feedback* con las tareas que estaban listas según

quien estaba encargado de las mismas pero aún debían pasar por revisión de pares (en el *Pull Request* en *GitHub*) y *DONE* con las tareas ya terminadas.

Para ver todas las tareas que hicieron parte del *sprint backlog* referirse a la sección 14.1 Reportes de cierre de sprints, los mismos contienen una lista de todas las historias de usuario realizadas en cada iteración, su estimación y fecha de culminación.

9.3.3 Kanban

Kanban entró en juego cuando el *board* de investigación pasó a ser manejado de un modo que asemejaba más a Kanban que a Popcorn Flow. Es decir, no se estaba llevando a cabo ninguna investigación sino que simplemente se buscaba llevar registro ordenado de todo lo que se hacía en cuanto a tareas misceláneas que no estaban relacionadas directamente con el desarrollo del producto.

Esto pasó cuando el desarrollo estuvo lo suficientemente avanzado y la investigación había ya llegado a un punto en el que se tenía claro lo que se necesitaba hacer para seguir y terminar con el desarrollo.

La implementación de este *framework,* en relación a sus ceremonias, se hizo así:

- Las *daily standup meetings* no se llevaron a cabo, se usaba la misma *weekly catch-up* que ya existía para *Scrum* y desde el principio del proyecto antes de la adopción de cualquiera de estas dos metodologías.

Kanban no tuvo en ningún momento relación con Scrum. Eran dos partes completamente separadas del proyecto.

9.3.4 Planificación de las iteraciones

Se buscó que las iteraciones empezaran y terminaran a mitad de semana porque se sabía que los fines de semana el equipo tendría su máximo nivel de productividad. Se quería que ese nivel

máximo de productividad no estuviera sobre el final del *sprint* porque de estarlo se aumentaba la posibilidad de terminar las iteraciones de forma apresurada.

Esta premisa se utilizó a lo largo de todo el proyecto sin importar la metodología.

El largo de las iteraciones empezó siendo de 1 semana para las etapas iniciales de investigación. Al comenzar con el desarrollo las iteraciones fueron extendidas a un largo de 2 semanas. El día de inicio siempre fue el jueves, siendo el miércoles el día de finalización de las iteraciones.

Las reuniones que llamamos *weekly catch-up* siempre fueron llevadas a cabo los días sábado a la tarde. Se sumaron reuniones extra a conveniencia pero solo si había un motivo puntual por el que fuese necesario.

La planificación de las tareas a nivel de cada iteración fue discutida en detalle en las secciones anteriores que discuten cómo los distintos marcos de trabajo fueron aplicados.

NOTA: La numeración de las iteraciones no se mantuvo de manera contínua desde que se comenzó con la etapa de investigación. Por ejemplo: el *sprint* 1 de desarrollo se llevó a cabo al mismo tiempo que el *sprint* 10 de investigación.

9.4 Gestión de la documentación académica

A lo largo del proyecto se fueron documentando aspectos puntuales que el equipo consideraba importantes en documentos independientes. Estos documentos fueron guardados en su totalidad en un directorio compartido en OneDrive para que todos los integrantes pudieran acceder a todo con facilidad.

Al llegar al final del proyecto y tener que armar el documento final se hizo una gestión informal como mencionado en la sección 6.3.5 Documentación. Basado en otras tésis y un *template* que fue facilitado al equipo por el tutor se listó todo el contenido que debía contener la documentación y se la fue completando poco a poco sin mayor esfuerzo desde el lado de la gestión.

9.5 Plan de releases

Previo al comienzo del desarrollo se trabajó en un plan de *releases* basado en la incertidumbre esperada según el momento del proyecto. Ver la sección 6.2 Incertidumbre para más detalles.

El plan de *releases* en su versión final es el siguiente:

Figura 21: Plan de *releases*

9.6 Métricas de gestión

A partir de cada *sprint* de desarrollo se construyó un pequeño reporte con la herramienta de procesamiento de datos que se armó al empezar el proyecto.

De cada *sprint* se pudo obtener fácilmente la totalidad de SP planeados vs la cantidad de SP hechos, lo cual se presenta en el siguiente gráfico.

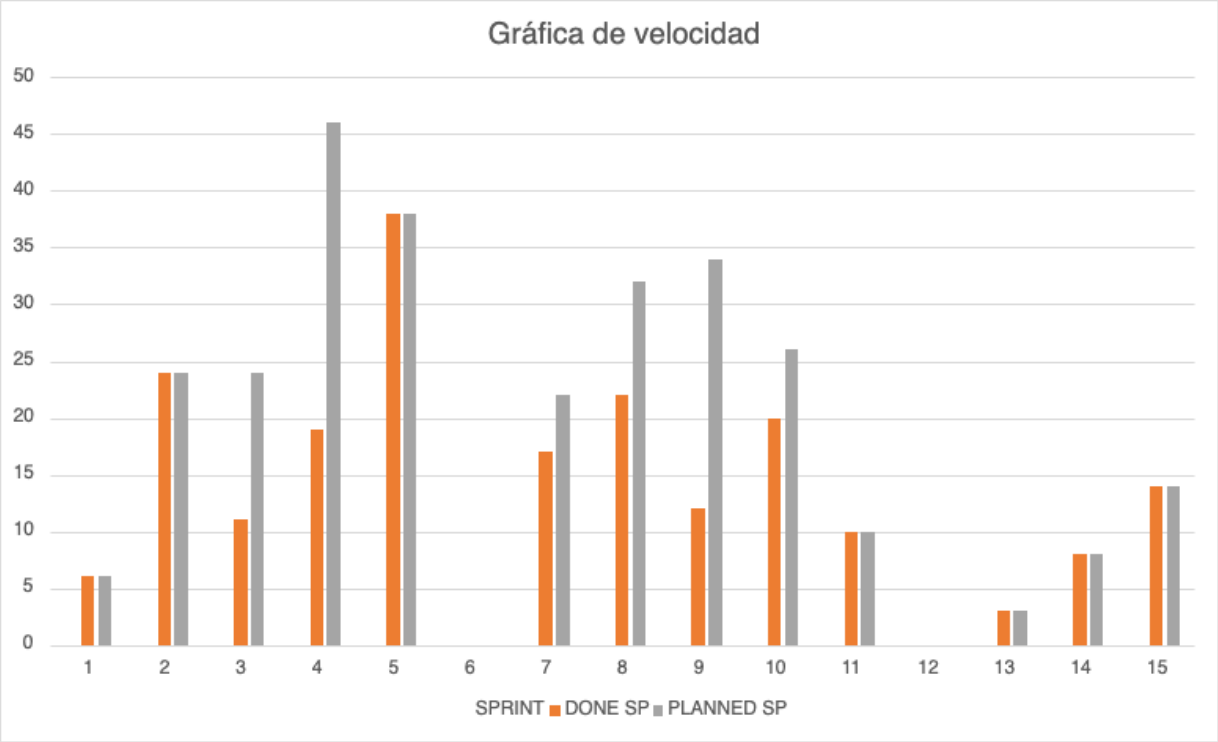


Figura 22: Gráfica de velocidad

La **velocidad promedio** del equipo al terminar el proyecto es de **15,69 SP**.

Como se puede ver, los *sprints* 6 y 12 no tienen tareas asociadas. Esto es debido a que el *sprint* 6 fue dedicado pura y exclusivamente a investigación y gestión y que el *sprint* 12 fue tomado como descanso para el equipo por navidad y año nuevo.

Además se generó un *burndown chart* para cada uno lo cual se presenta en la sección 14.1 Reportes de cierre de sprints.

9.7 Gestión de la comunicación

La gestión de la comunicación fue parte importante del proyecto desarrollado. Para la comunicación entre los distintos involucrados en el proyecto se determinaron canales de comunicación específicos. Dada la situación sanitaria de la pandemia de COVID-19 y de que producto de esto, se popularizó e instaló el trabajo remoto, el equipo se apoyó en medios de comunicación remotos. Las instancias de comunicación presencial se redujeron exclusivamente para los miembros del equipo.

9.7.1 Comunicación interna

Se dice comunicación interna a la comunicación entre dos o más miembros del equipo y a aquella entre miembros del equipo y el tutor.

Esta comunicación se lleva a cabo utilizando la herramienta WhatsApp: se hicieron dos grupos para este propósito. Uno donde se encontraban todos los miembros del equipo y el tutor. En este grupo se pasaban mensajes de carácter administrativo, como fechas de revisiones y donde se coordinaban las reuniones con el tutor. Además se hacían preguntas sobre dudas al tutor de la tesis. En el otro grupo, solo estaban presentes los integrantes del equipo de tesis, en éste, se coordinaron reuniones internas y se hacían preguntas sobre dudas respecto a tecnologías, entre otras cosas.

Las ceremonias correspondientes a la aplicación de los *frameworks* de gestión 9.3.2 Scrum se hicieron con videollamadas a través de Google Meet. Estas estaban fijadas en Google Calendar o simplemente se pasaba el *link* por el grupo de Whatsapp, en el caso de que fuera necesario hacer fuera de un horario que no estuviese en el calendario.

9.7.2 Comunicación con GeneXus

El contacto principal del equipo dentro de la empresa fue, desde el principio, Aníbal Gonda. Su rol dentro de la empresa es descrito como Evangelizador de la plataforma.

Ante cualquier necesidad o duda se enviaba un correo a él. Su responsabilidad era la de delegar la consulta a quien correspondiese.

Como ya fue mencionado, el equipo se comunicó también de forma recurrente con Federico Azzato, quien es *Developer* de la empresa GeneXus y brindó mucha ayuda en aspectos técnicos. En algunas ocasiones el equipo tuvo que hacer sesiones colaborativas de codificación, para esto utilizó la herramienta provista por Visual Studio llamada Live Share que permite compartir instantáneamente el proyecto a través de un *link* [56].

En casos puntuales también existió comunicación con Gaston Milano, CTO de GeneXus y Armin Bachmann, Product Manager de GeneXus. En estas reuniones puntuales se hicieron *reviews* de los requerimientos ya implementados.

La comunicación con los nombrados miembros de GeneXus siempre comenzaba vía *email*. En estos *emails* se coordinaban, en caso de ser necesarias, reuniones virtuales para discutir el tema a tratar, o simplemente se discutía el tema correspondiente vía *email* si el alcance del mismo era pequeño y no era necesario hacer una llamada. Estas reuniones o videollamadas se realizaban con las herramientas Google Meet o Zoom. También se hizo uso del calendario Outlook para agendar las llamadas.

9.7.3 Comunicación con otras empresas

El equipo mantuvo comunicación con una empresa externa a GeneXus, esta empresa es Abstracta[46]. Esta empresa es la encargada del desarrollo de GXtest. La comunicación fue con German Gonzalez, quien es desarrollador de dicha empresa. La comunicación fue vía *email*, el

[46] https://abstracta.us/

objetivo de esta comunicación fue eliminar dudas técnicas en lo que respecta a la generación de pruebas automáticas de GXtest.

9.7.4 Comunicación con la Universidad ORT Uruguay

En las instancias de revisión, la comunicación entre los miembros del proyecto, el tutor y el revisor asignado, se hizo mediante la plataforma Microsoft Teams. Las reuniones fueron grabadas con el fin de que el equipo más tarde pudiera procesar el *feedback* recibido y autoevaluar su propia exposición.

9.8 Gestión de riesgos

Se hizo una gestión de riesgos pensada para un proyecto ágil, es decir, pensada para trabajar con solo aquello de lo que se tiene certeza absoluta. Esto fue propuesto por Álvaro Ortas, docente de la Universidad ORT Uruguay, quién fue revisor del equipo en la primera instancia de esta índole.

En cada *sprint planning* se construye un nuevo plan de riesgos que solo aplica para el *sprint* en cuestión, definiendo así los riesgos pertinentes al *scope* del *sprint.* Muchos riesgos son compartidos entre *sprints*.

La evolución del plan de riesgos está presente en el anexo 14.2 Planes de riesgos.

10 Gestión de la calidad

En este capítulo se presentan las distintas actividades, estándares utilizados y buenas prácticas que el equipo incorporó con el fin de asegurar la calidad del proyecto. Se detalla la definición de calidad de *software* manejada por el equipo, las actividades de aseguramiento de la calidad llevadas a cabo, las métricas obtenidas producto de las actividades realizadas.

10.1 Definición de calidad

El equipo entiende por calidad la definición de la norma ISO/IEC 25010 que dice: *"La calidad del producto software se puede interpretar como el grado en que dicho producto satisface los requisitos de sus usuarios aportando de esta manera un valor. Son precisamente estos requisitos (funcionalidad, rendimiento, seguridad, mantenibilidad, etc.)"* [46]

Sobre esta norma se hizo referencia previamente en 7.4 Requerimientos no funcionales y en 8.2 Atributos de calidad donde se utilizó el modelo de calidad del producto definido por la ISO/IEC 25010 compuesto por las ocho características de calidad, para justificar los aspectos que fueron tenidos en cuenta en los requerimientos no funcionales y en los atributos de calidad.

10.2 Aseguramiento de la calidad

Se presentan las acciones que se llevaron a cabo por parte del equipo para asegurar la calidad.

10.2.1 Aplicación de estándares

La herramienta principal que se utilizó para asegurar el cumplimiento de los estándares propuestos en todo el código JavaScript es ESLint. El código JavaScript de la solución está presente en el editor y en la totalidad del Cliente de Requerimientos o *frontend*.

Los estándares que el equipo decidió seguir son los estándares por defecto de create-react-app, que es el CLI de *React*, con las modificaciones que fueron aplicadas en el *template* publicado por

la *software factory* uruguaya SpaceDev[47]. El mismo fue desarrollado por uno de los miembros del equipo a partir de experiencias previas trabajando con React.

En cuanto a lo que no tiene tanto que ver con estándares sino con estética del código JavaScript se hizo uso de la herramienta Prettier en conjunto con ESLint. Para ésta se usaron, al igual que para la configuración de ESLint, las mismas opciones que usa SpaceDev.

Para el código C# de la Extensión y del *backend* se hizo uso de las herramientas provistas por el IDE Visual Studio que ayuda de manera automática con el formateo del código y con el mejoramiento del mismo. Para todo lo que no se puede automatizar se tuvieron en cuenta las C# Coding Conventions publicadas por Microsoft [57].

10.2.2 Revisiones

Se implementó como parte del flujo básico de una tarea una instancia de revisión de pares del código.

Al implementar una nueva funcionalidad, *bug fix*, o cualquier tipo de contribución al código del proyecto la misma entra al repositorio en una *branch* específica para dicha contribución y se hace un *Pull Request* para, luego, cuando este *Pull Request* esté aprobado, entonces, hacer el correspondiente *merge* del código a la rama *dev*. Para más información sobre el manejo de branches en el repositorio ver la sección 11 Gestión de la configuración.

Cada *Pull Request* era revisado por los demás miembros del equipo, quienes podían sugerir mejoras. Esto con el objetivo de que la lectura de varias personas con distintas perspectivas aumentara la calidad del código resultante.

10.2.3 Pruebas

Al planear las tareas para cada iteración, como se presenta en la sección 9.3.4 Planificación de las iteraciones, se definía un *checklist* con los criterios de aceptación para cada una.

[47] https://spacedev.uy/

- Para toda tarea, todos los criterios de aceptación debían verse cumplidos en una prueba funcional.

- Si los integrantes del equipo así lo definían, se podía definir una meta de cobertura de pruebas automáticas para la funcionalidad en cuestión como un criterio de aceptación extra.

- El criterio que el equipo tomó para definir la meta de cobertura de las pruebas fue la siguiente:

    - Si la funcionalidad era core, debía tener la mayor cobertura posible, definiendo un mínimo de 80% de cobertura.

    - Si la funcionalidad no era provista por algunos de los *frameworks* utilizados para la construcción de los entregables de *software* entonces aplica el punto anterior, por lo que la cobertura debe ser la mayor posible, con un mínimo de 80%.

    - En el caso que la funcionalidad fuera prevista por alguno de los *frameworks* utilizados, los puntos anteriores no aplican. Por lo que no es requerido hacer tests de estas funcionalidades.

- De requerir pruebas automáticas, las mismas podían ser unitarias o *End-To-End*, lo cual también debía ser definido por los miembros del equipo.

Al cumplir todos los criterios de aceptación se hace el *Pull Request* y se pasa a la revisión de pares para aseguramiento de calidad descrita en el punto anterior.

10.2.3.1 Deploy del ambiente de pruebas

Al hacer cada *release* se quería que tanto el experto como el cliente pudiesen probar el *software* para aumentar la entrada de *feedback* y encontrar errores más prontamente. Para esto se hizo un *deploy* de la parte web de la solución, es decir, el Cliente de Requerimientos.

Dicho *deploy* se hizo según el siguiente diagrama de despliegue UML:

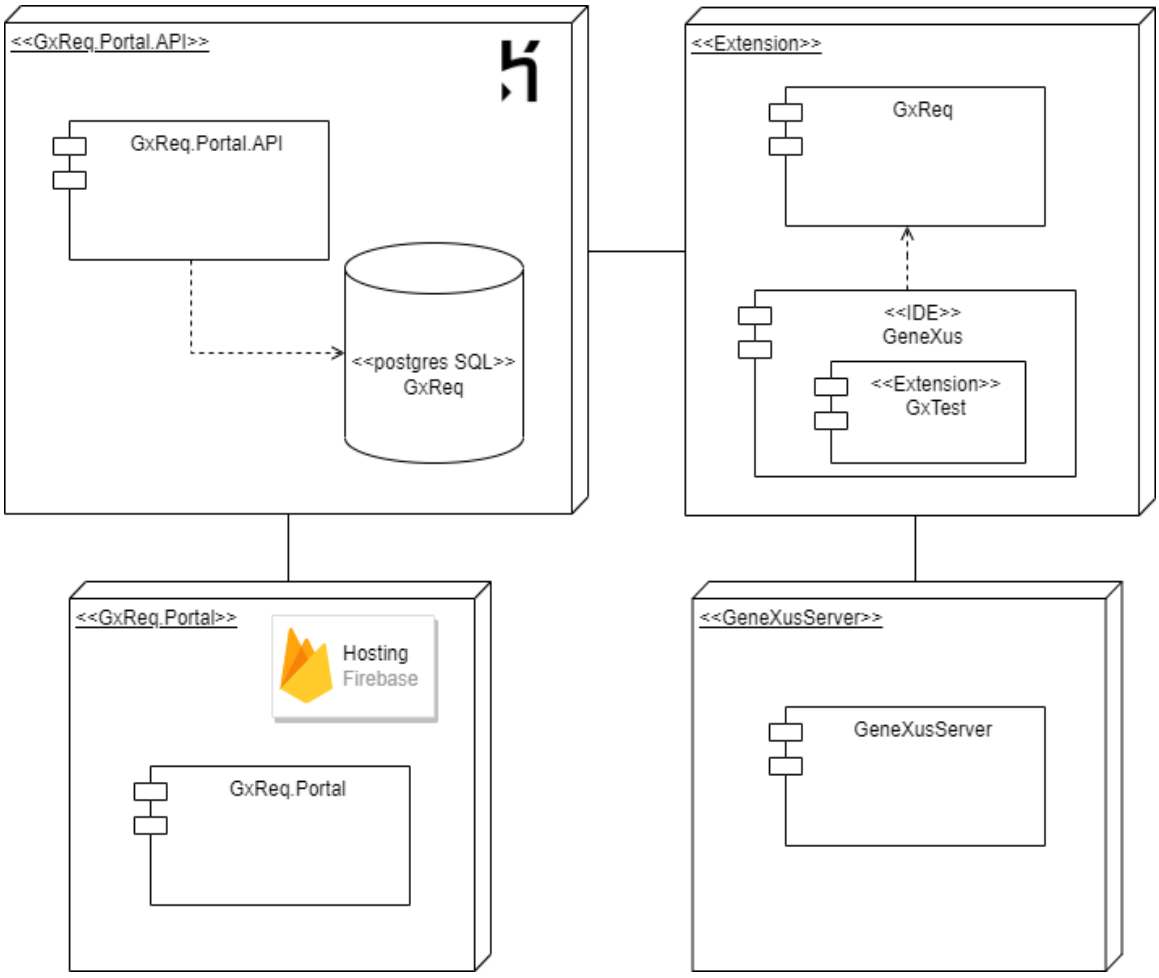


Figura 23: Diagrama de despliegue

A continuación se describen los elementos que aparecen en el diagrama:

- **GxReq.Portal.API:** *Backend* del Cliente de Requerimientos de la aplicación deployado en *Heroku* con su respectiva base de datos Postgres también en Heroku. Accesible a través de HTTPS, expone una REST API que es consumida por el *frontend* GxReq.Portal.

- **GxReq.Portal:** *Frontend* del Cliente de Requerimientos Web deployado en Firebase. Consume la REST API que expone GxReq.Portal.API.

- **Extension:** Extensión de GeneXus que se encuentra dentro del IDE que también hace uso de GxTest. La misma se conecta por HTTPS a GxReq.Portal.API (*backend* del Cliente de

Requerimientos) para obtener las *Features* que fueron cerradas (*Commited*). La Extensión cuenta con un *parser* de Gherkin y es la que realiza la autogeneración de objetos nativos de GeneXus.

- **GeneXusServer:** Instancia de GeneXus Server con las dll necesarias para poder serializar los objetos creados por la Extensión (*Features*) y ejecutar *tests* con GxTest.

10.2.4 Definition of Done

El equipo estableció a nivel global que para que una historia de usuario esté terminada debe cumplir con determinados criterios. Los criterios son los siguientes:

- La codificación debe ser acorde a los estándares mencionados en 10.2.1 Aplicación de estándares.
- Al integrar los cambios, se debe hacer un *pull request* que debe haber sido aprobado por los dos miembros del equipo restantes que no lo iniciaron, tal como se mencionó en 10.2.2 Revisiones.
- En el caso de ser una funcionalidad que no es provista por los *frameworks* utilizados, la cobertura de las pruebas unitarias debe ser de un 80% o superior. En la sección 10.2.3 Pruebas se explica en mayor detalle.
- Todas las pruebas unitarias deben ejecutarse y no fallar.
- La funcionalidad debe haber sido probada localmente o en el ambiente de prueba mencionado en 10.2.3.1 Deploy del ambiente de pruebas.
- Debe cumplir con cada uno de los criterios de aceptación definidos en la historia de usuario.

10.2.5 Documentación

Para la documentación se siguieron los estándares desarrollados por la Universidad ORT Uruguay. A continuación se listan los usados:

- Documento 302 - Normas específicas para la presentación de trabajos finales de carrera (TFDC) Facultad de Ingeniería [58].

- Documento 303 - Hoja de verificación de formato de trabajos finales de carreras de la Facultad de Ingeniería [59].

- Documento 306 - Orientación para títulos, resúmenes o *abstracts* e informes de corrección de trabajos finales de carrera [60].

- Instructivo para la entrega final de proyectos [61].

10.2.6 Capacitación

El equipo realizó las capacitaciones necesarias en las tecnologías implicadas en el proyecto, de las cuales no se tenía un conocimiento profundo.

Lo primero y más importante fue la capacitación GeneXus, por parte de GeneXus fue ofrecido de forma gratuita el curso GeneXus For Students v17, el cual el equipo realizó, siendo así cada uno de los integrantes certificados, en el anexo 14.15 Evidencia de certificaciones - GeneXus for Students existe evidencia de esto.

También, dado que no todo el equipo tenía el suficiente conocimiento en las tecnologías y *frameworks* a utilizar, se realizaron tutoriales de cada una de estas, por ejemplo, tutorial de React[48], React Admin[49] y JsonApiDotnetCore[50].

10.3 Métricas

El equipo realizó diferentes evaluaciones del código realizado.

[48] https://es.reactjs.org/tutorial/tutorial.html
[49] https://marmelab.com/react-admin/Tutorial.html
[50] https://www.jsonapi.net/getting-started/install.html

Para el código del *backend* del Cliente de Requerimientos fue usada la herramienta Sonarqube[51]. Esta es una herramienta de análisis estático de código fuente, es de código abierto y una de las más robustas del mercado.

Una de las características interesantes de Sonarqube es que da como resultado, entre otros, la deuda técnica. La deuda técnica es definida por Sonarqube como el esfuerzo necesario para solucionar cada advertencia en el código. Para este caso, y como se ve en la imagen de abajo, la deuda técnica encontrada es de dos horas y veinte minutos, lo cual es un tiempo corto y bueno. La calificación en este punto es de A, lo máximo que brinda Sonarqube, por lo que se considera un resultado satisfactorio. Tener una deuda técnica no demasiado grande es importante dado que este código, al ser una prueba de concepto, se va a seguir trabajando en el futuro, por lo que la mantenibilidad es sumamente importante.

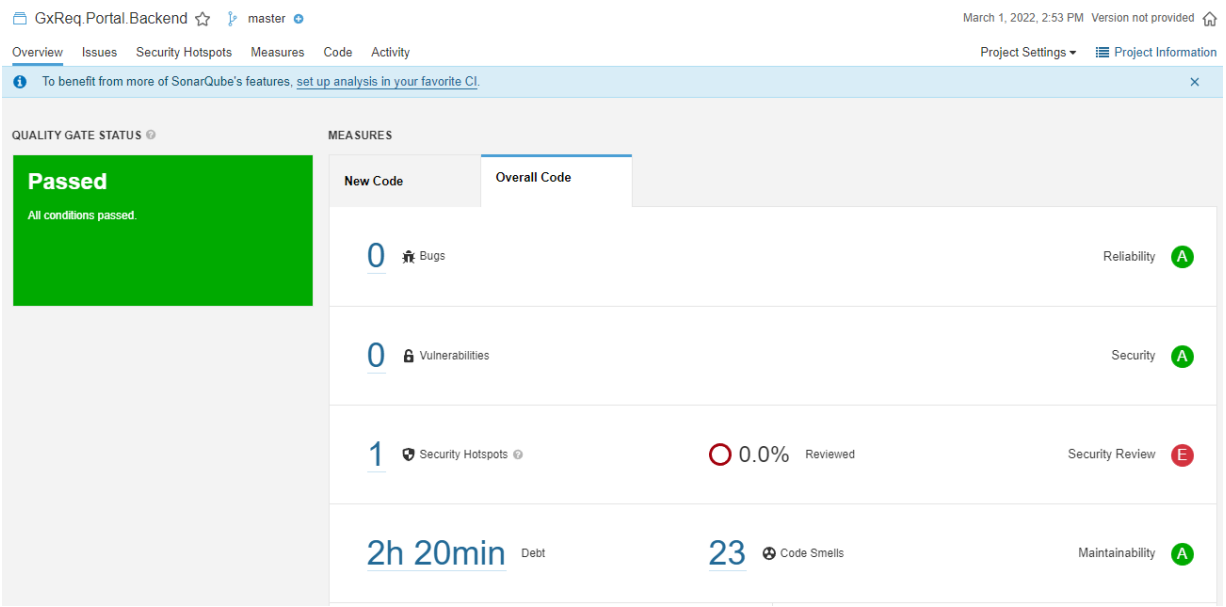


Figura 24: Resultados Sonarqube

[51] https://www.sonarqube.org/

En el punto de *Security Review* se ve una calificación de E, lo cual es lo más bajo que se puede tener. Esto se debe a que existe un punto de acceso inseguro en nuestro código. Esto es algo que el equipo identificó y, teniendo en cuenta que la seguridad no es el *core* de la prueba de concepto, dejó pasar. Se debe a una configuración en el servicio CORS[52], en donde se permite el acceso desde cualquier origen.

Por otro lado, para el *frontend* del Cliente de Requerimientos el equipo utilizó ESLint, como ya se mencionó en la sección de aseguramiento de la calidad. En la siguiente imagen se muestra evidencia de las advertencias que esta librería muestra durante la etapa de desarrollo.

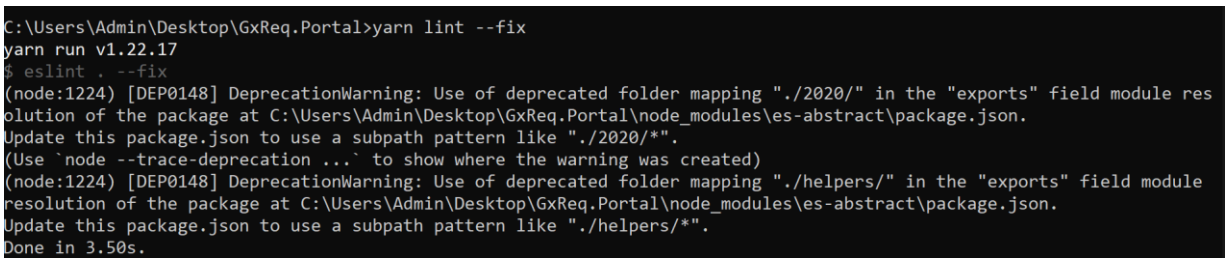


Figura 25: Advertencias ESLint

Para el *frontend* también se crearon pruebas unitarias, estas validan los formularios *customs* que fueron creados, como pueden ser el formulario de registro o de inicio de sesión. Para la creación de estas pruebas fue utilizada la librería Jest[53]. En la siguiente imagen se presenta evidencia de la ejecución de las pruebas unitarias creadas para el *frontend.*

[52] https://developer.mozilla.org/es/docs/Web/HTTP/CORS

[53] https://jestjs.io/

```
C:\Users\Admin\Desktop\GxReq.Portal>yarn test
yarn run v1.22.17
$ craco test
Browserslist: caniuse-lite is outdated. Please run:
npx browserslist@latest --update-db

Why you should do it regularly:
https://github.com/browserslist/browserslist#browsers-data-updating
Browserslist: caniuse-lite is outdated. Please run:
npx browserslist@latest --update-db

Why you should do it regularly:
https://github.com/browserslist/browserslist#browsers-data-updating
 PASS  src/custom/Signin/formSchema.test.js
 PASS  src/custom/Signup/formSchema.test.js

Test Suites: 2 passed, 2 total
Tests:       29 passed, 29 total
Snapshots:   0 total
Time:        2.515 s
Ran all test suites related to changed files.
```

Figura 26: Pruebas *Frontend*

Se hicieron pruebas *end-to-end* del *backend* de Cliente de Requerimientos. Estas pruebas validan el funcionamiento de toda la aplicación, de principio a fin, probando así el flujo de la aplicación y su correcto funcionamiento. En este caso, como fue utilizado el *framework* JsonApiDotnetCore, que ya implementa varias funcionalidades, se probaron aquellas funcionalidades implementadas en su totalidad por el equipo y de forma personalizada. Estas pruebas fueron realizadas con el *framework* para pruebas, *open source,* xUnit[54]. Se deja evidencia de estas pruebas a continuación.

[54] https://xunit.net/

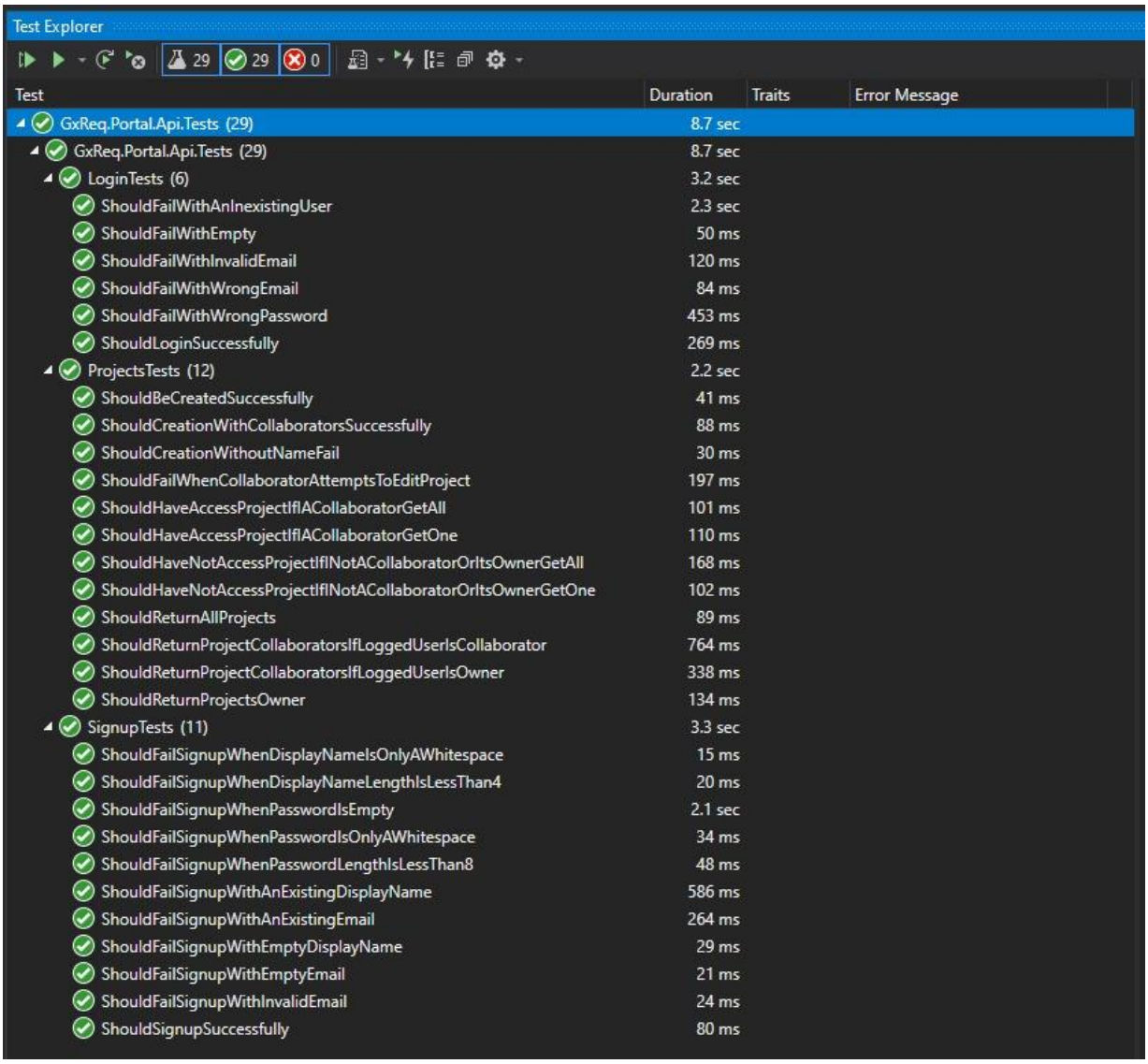


Figura 27: Evidencia pruebas *end-to-end backend* Cliente de Requerimientos

Por último, y no menos importante, se crearon pruebas unitarias para la extensión GeneXus construida. Estas son sumamente importantes, dado que es la parte *core* de la solución y en donde existe la mayor parte de la lógica de negocio. En el anexo 14.14 Cobertura de pruebas unitarias - Extensión GeneXus se presenta la cobertura de pruebas.

11 Gestión de la configuración

En esta sección se detalla el plan de gestión de la configuración (SCM). El objetivo de este es hacer el seguimiento y control de los cambios, con el fin de mantener la integridad y calidad de estos durante el desarrollo. Se identifican los elementos de la configuración del proyecto y posteriormente se detalla la gestión de los cambios y el versionado.

11.1 Elementos de configuración

Los elementos de la configuración es toda información resultante o generada como parte del proceso de ingeniería. Aplican como elementos desde un documento, porciones de código y hasta conjuntos de datos, entre muchos otros. Estos elementos son susceptibles a cambios. A continuación se presentan los elementos identificados para este proyecto:

11.1.1 Código fuente

El código desarrollado para la prueba de concepto es uno de los principales elementos de la configuración. En este están incluidas las dos grandes partes del producto, la extensión GeneXus y el Cliente de Requerimientos. Aparte de estas partes del producto antes mencionadas, el equipo desarrolló un *template* de una extensión GeneXus, con el fin de facilitar a futuros desarrolladores el crear su propia extensión.

Para la gestión de los cambios del código fuente se usó la herramienta Git. Git es un sistema de control de versiones, distribuido y de código abierto. Se eligió esta herramienta dado que todos los miembros del equipo tienen una gran experiencia, en la facultad es el sistema de control de versiones que enseñan, y también dado que es, sin lugar a dudas, la herramienta más usada, sobre TFVC[55] y SVN[56].

[55] TFVC: Team Foundation Version Control, sistema de control de versiones usado por Azure DevOps

[56] SVN: Apache Subversion, sistema de control de versiones.

Para los repositorios de código fuente se usó Github. Github es un proveedor de servidores remotos de Git. El por qué se eligió es similar al punto anterior, la experiencia del equipo y lo más utilizado en el mercado fueron puntos tomados en cuenta. Existen alternativas como Azure DevOps y Bitbucket, entre otras.

El proyecto consta de tres repositorios principales, para la extensión GeneXus, para el *backend* del Cliente de Requerimientos y para el *frontend* del Cliente de requerimientos. También se cuentan con más repositorios, se hizo uno específicamente para entregarle al cliente, el cual consta de un *boilerplate* de una extensión GeneXus, el cual ya está configurado y documentado, con el fin de facilitar a futuros desarrolladores de nuevas extensiones. Existen también diversos repositorios en los cuales se realizaron pruebas de conceptos.

Figura 28: Ilustración de los repositorios en Github

Para los cambios de código el equipo estableció una política de *branching*, esta política se llama GitFlow[57]. GitFlow define dos principales tipos de ramas, ramas principales y ramas de apoyo. Los cambios de código se desarrollan en una rama de apoyo, nunca sobre una de las ramas principales, y luego estas ramas de apoyo se fusionan con las ramas principales. Para realizar este flujo se crean lo que se llaman *pull request* (también conocido como *merge request*). Un *pull*

[57] https://cleventy.com/que-es-git-flow-y-como-funciona/

request es una petición para fusionar nuevo código en una de las ramas principales, esto permite al equipo de desarrollo ver y aprobar (o dejar sugerencias de cambio) el nuevo código que se va a fusionar en las ramas principales.

A continuación se describen las ramas utilizadas en este proyecto.

Como ramas principales se crearon dos:

- ***Master***: Contiene todo el código estable del proyecto. Cada *commit* incluido en esta rama debe estar listo para una subida a producción. Para este proyecto, por cada *release* finalizado, se hace *commit* a la rama *master*.

- ***Dev***: La nuevas características o arreglos desarrollados, al completarse estos, se hace un *commit* sobre la rama *dev*.

Como ramas de apoyo se utilizaron principalmente tres:

- ***Feat***: El desarrollo de una nueva característica.

- ***Fix***: El arreglo de un error.

- ***Chore***: Cambios que no afectan al código estable.

La convención de nombres de las ramas se basó en la convención definida por Angular. Esta convención define también un estándar para los mensajes de *commit* [62].

A continuación se presenta una imagen como ejemplo del flujo GitFlow.

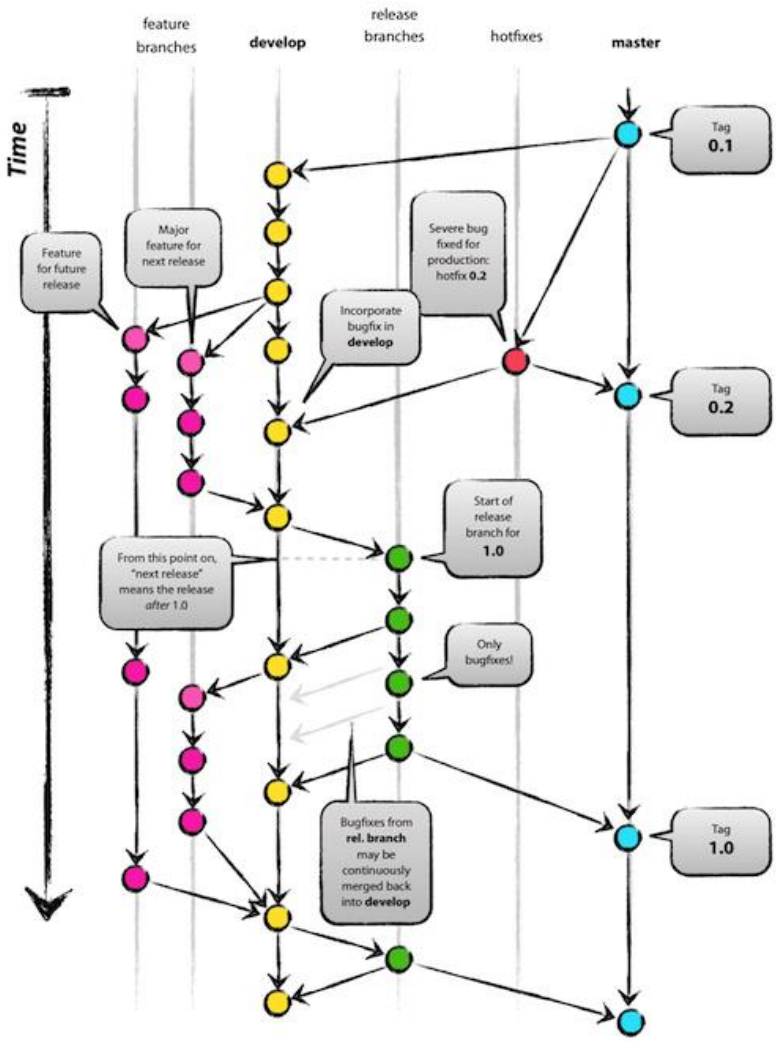


Figura 29: Flujo GitFlow

El equipo usó un formato numérico para identificar cada una de las versiones de la aplicación. Las versiones se forman por tres números, de la forma X.Y.Z. Este proyecto, como ya se mencionó, trata de una prueba de concepto, por lo cual nunca tuvo una versión final y lista para poner en un ambiente productivo, por esto, las versiones siempre fueron 0.0.Z, dado que no existe una versión 1.0.0. El formato de versionado antes mencionado es llamado Semver. Cada versión corresponde con la finalización de un *Release* [63]. Se agrega una imagen ilustrando lo anteriormente explicado, en esta imagen se ve la versión (0.0.2), el número de *release* (2) y los cambios realizados en este.

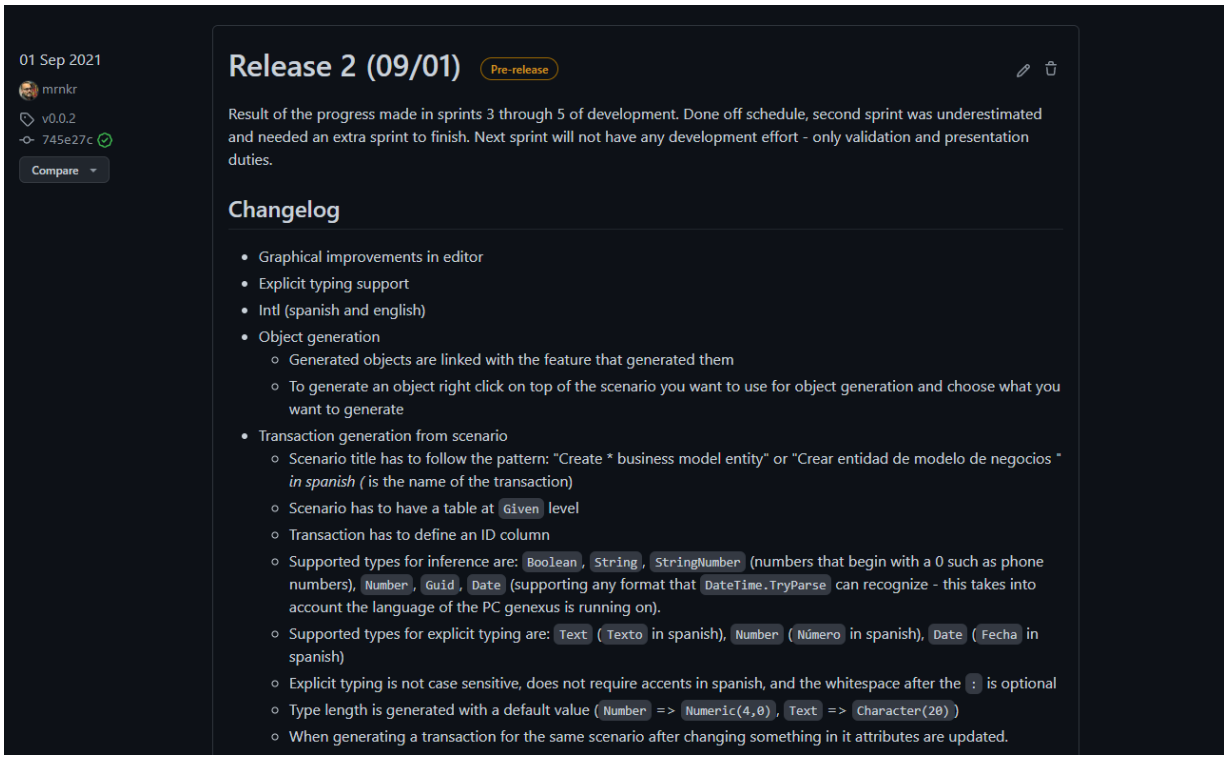


Figura 30: Ejemplo de *Release*

Como complemento, a la X se le llama versión "*Major*", a la Y versión "*Minor*" y, por último, a la Z versión "*Patch*".

Al crearse un *pull request* se ejecuta la integración continua, que está configurada para todos los repositorios. La integración continua evalúa si el código se compila y si las pruebas unitarias construidas son satisfactorias, detectando errores de forma automática. Así mismo, al producirse un *merge* sobre la rama Master, entonces se realiza el *deploy* de forma automática. Con esto último el equipo provee de entrega continua, esta es solo aplicable al Cliente de Requerimientos, dado que la extensión GeneXus no se encuentra desplegada en ningún ambiente. Estos elementos, es decir, entrega continua e integración continua fueron configurados haciendo uso de Github Actions [64]. El Cliente de Requerimientos, al ser un SaaS (*Software As A Service*), como ya se mencionó antes, no se ve afectado.

11.1.2 Documentación

Toda la documentación generada son elementos de la configuración. En esta están incluidos los documentos generados: *readme* de repositorios, manuales de usuarios e instalación, métricas de gestión, el presente documento, entre otros.

La documentación formal generada fue escrita en Microsoft Word. Esta decisión fue tomada debido a que el equipo decidió usar como repositorio remoto OneDrive. La integración de estas dos herramientas es dinámica. Se tomó la decisión de usar OneDrive debido a la experiencia del equipo en esta, y también dado que brinda facilidades a la hora de realizar revisiones, tanto por otros miembros del equipo, como por el tutor. En este repositorio se crean documentos formales y de apoyo, siendo estos últimos una suerte de borradores.

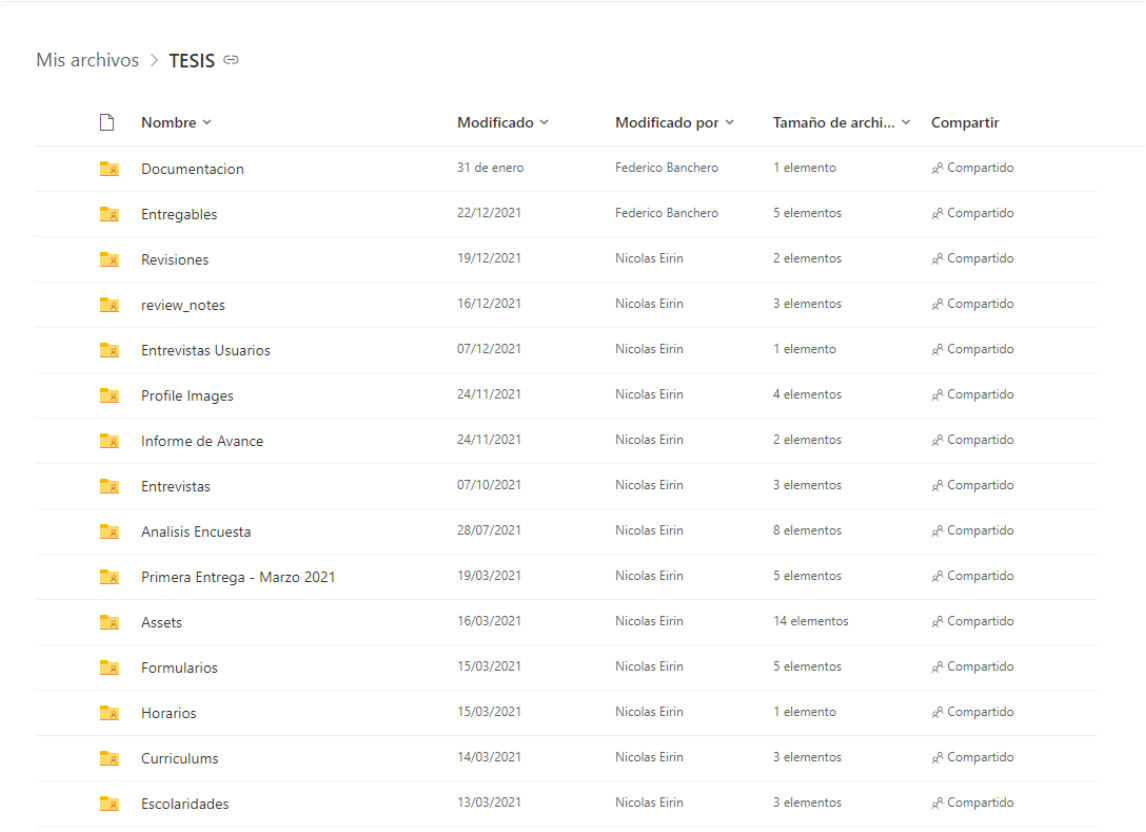


Figura 31: Evidencia de documentación en OneDrive

Para otros tipos de documentos, como pueden ser hojas de cálculos o presentaciones, también se usan el conjunto de herramientas que provee Microsoft.

Para escribir notas rápidas, por ejemplo, de diversas llamadas, el equipo usó Notion. Esta aplicación permite tomar notas de una forma sencilla y clara, permite compartir estas, editarlas de forma concurrente y también es gratuita. Cabe destacar que brinda más funcionalidades, esta fue la usada por el equipo.

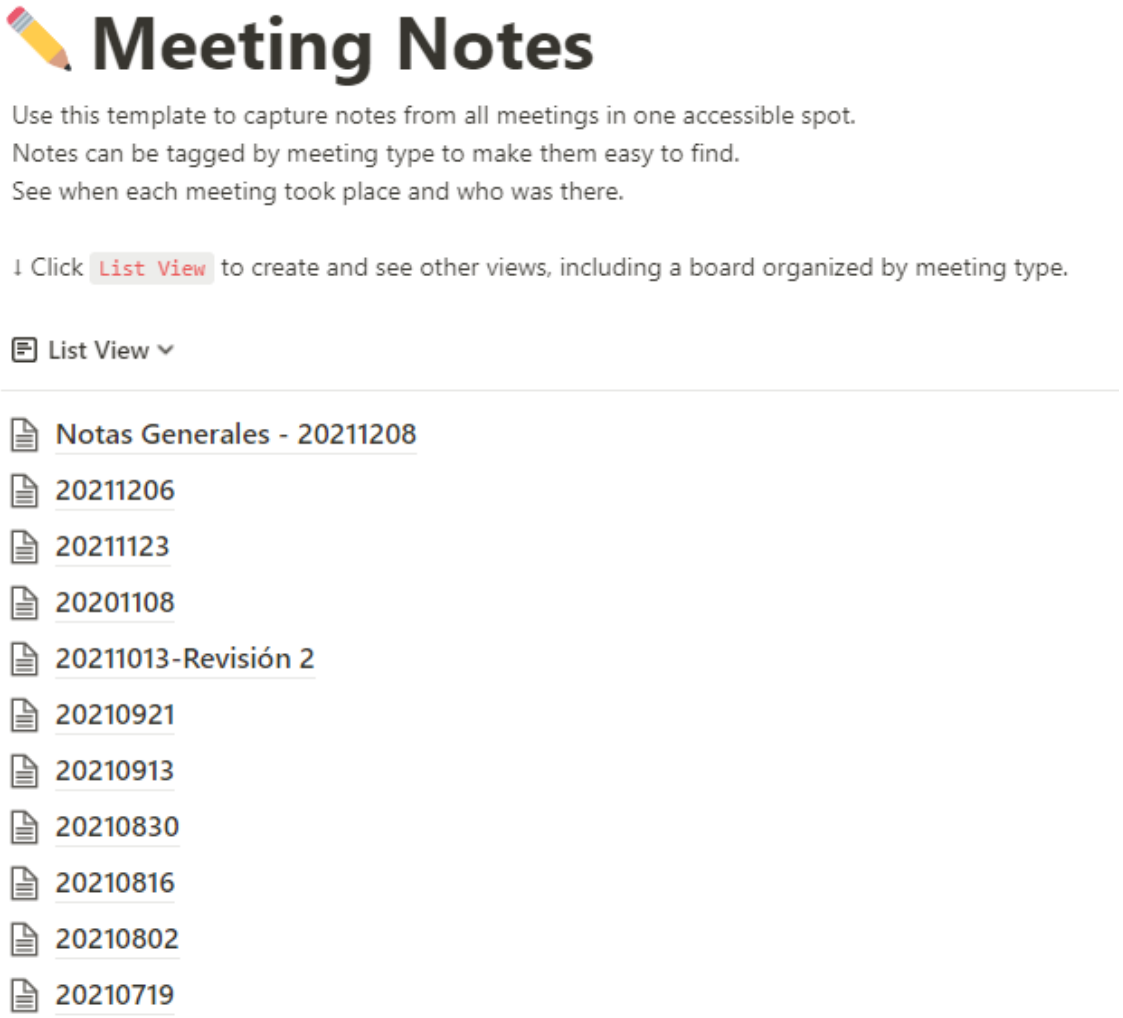


Figura 32: Evidencia de documentación en Notion

Los diagramas de arquitectura generados por el equipo fueron realizados en Draw IO. Esta es una herramienta que permite la creación de diversos diagramas, de forma gratuita y brindando la capacidad de integrarse con otras aplicaciones. Se eligió esta debido a que es gratuita y ya se contaba con experiencia previa.

11.1.3 Historias de usuario

Los *tickets* de los tableros Scrum y Kanban también son elementos de la configuración de nuestros proyectos.

Como se detalló anteriormente, para el proyecto fueron creados dos tableros, ambos en la aplicación Trello. Trello es un *software* de gestión de proyectos, el cual brinda la capacidad de definir tableros, con diversas columnas y tarjetas. Cada tarjeta tiene un título, una clara descripción, la estimación en *Story Points* y un *sprint* y *release* relacionado.

11.2 Flujo de trabajo

En esta sección se detalla el flujo de trabajo llevado a cabo en el proyecto, describiendo el ciclo de vida de las historias de usuario de desarrollo y los problemas identificados en la etapa de investigación.

11.2.1 Ciclo de vida de un problema - Investigación

En Popcorn Flow el punto de partida es un problema que el equipo identifica. Para cada problema se buscan múltiples soluciones y de las mismas se estudia su factibilidad.

Para esto el equipo definió un *board* con un mapeo uno a uno de las etapas de Popcorn Flow a columnas:

- ***Problemas y observaciones:*** Contiene los problemas a nivel más macro. No son tareas accionables ni se las puede describir en un alto nivel de detalle. Para cada uno se definen varias posibles soluciones. **Las *cards* de esta columna nunca se mueven**.

- ***Opciones:*** Son las posibilidades que el equipo identificó para solucionar un problema de la columna anterior. Que una *card* existiese para una opción no quería decir que el equipo se estuviese comprometiendo a llevarla a cabo, actúa solo como registro. **A partir de esta columna las *cards* eran vinculadas con el problema a partir del cuál surgieron**.

- ***Comprometidos:*** Un subconjunto de las opciones encontradas para cada problema pasaba a este estado de compromiso, a partir de este punto se las podía llamar experimento. Una vez que una *card* llegaba a este estado pasaba a ser accionable, todo el equipo debía comprender de qué se trataba por medio de la lectura de la descripción asociada y debía estar comprendida dentro de un período de tiempo, llamado *timebox*, que desde el principio llamamos *sprints* y tenían duración de una semana. **Se guardaba el *sprint* correspondiente a cada experimento en un *tag***.

- ***Review:*** Una vez realizado el experimento se pasaba a un estado de *review*. Esto quiere decir que el equipo sacaba conclusiones del resultado del experimento. Estas conclusiones podían ser que la idea detrás del experimento no era factible y se descartaba, que se podía seguir investigando en un modo más detallado o que simplemente se tomaba la idea y se la guardaba para ser tenida en cuenta durante el desarrollo.

- ***Next:*** Si la conclusión de un experimento era que se podía seguir trabajando para obtener resultados más tangibles o accionables la *card* correspondiente pasaba a esta columna y se creaban nuevas *cards* en ***Opciones*** con experimentos derivados. **Estas nuevas *cards* derivadas eran vinculadas a aquella por la cuál fueron creadas**.

- ***Done:*** Si la conclusión de un experimento era que no había nada más que hacer, sea porque la idea detrás del mismo no era factible o que ya se tenían suficientes elementos para llevar esa idea a cabo durante el desarrollo, la *card* terminaba en esta columna. **Cada experimento descartado llevaba un *tag* que así lo marcaba. Esta columna fue el punto de entrada para el *product backlog* del desarrollo**.

11.2.2 Ciclo de vida de las historias de usuario - Desarrollo

Las historias de usuario se crean, discuten y estiman durante la *Sprint Planning Meeting.* A continuación se describe el flujo de estas en el tablero. Para esto, se definirá cada una de las columnas del tablero de desarrollo.

- ***Backlog*****:** En esta columna se encuentran las Épicas definidas del proyecto, de estas luego se desprende cada historia de usuario.

- ***Release backlog:*** Al momento de crearse, la historia es colocada en esta columna. En la *Sprint Planning Meeting* se crean historias relacionadas a las épicas que correspondieran al *Release* actual según la planificación (aún sin estimarse). De esta columna luego se toman las que se crean convenientes para el *sprint* que empieza. Las historias incluidas en el *sprint* se mueven a la columna *TODO*.

- ***TODO*****:** En esta columna se encuentran las historias que se espera completar durante el *sprint* actual, entiéndase *sprint backlog*. Se las estima y especifica en detalle en la misma ceremonia. En este momento no se realizan asignaciones a ningún miembro del equipo. Cuando uno decide comenzar a trabajar, entonces se asigna la historia y la mueve a la siguiente columna, *In Progress.*

- ***In Progress:*** Esta columna contiene las historias que están en progreso. Cuando el desarrollador termina su trabajo, es decir, considera haber cumplido todos los criterios de aceptación, entonces mueve la tarea a la columna *Awaiting Feedback.*

- ***Awaiting Feedback:*** En esta columna se encuentran las tareas que están pendientes de revisión por los demás miembros del equipo. Esto tiene una relación uno a uno con un *pull request* en Github.

- ***DONE:*** Una vez que el *pull request* estuviere aprobado por todos los miembros del equipo, entonces se puede hacer *merge* ese *pull request* y la historia debe moverse a la columna *DONE***.** Todo el código correspondiente a las historias presentes en la columna *DONE* se encuentra en la rama *dev*.

12 Conclusiones

En este capítulo se presentan las conclusiones del proyecto realizado. Primero se presentan las conclusiones generales y se relacionan con el problema inicial planteado con cliente, luego las lecciones aprendidas por los miembros del equipo y por último los próximos pasos.

12.1 Generales

En la última ceremonia de retrospección que se llevó a cabo se llegó a la conclusión de que el equipo está conforme con el trabajo realizado.

Se cumplieron todos los objetivos propuestos, en cuanto al alcance del proyecto. GeneXus, complementado con GxReq, ofrece herramientas sólidas para trabajar con especificaciones escritas en Gherkin y para mantener su trazabilidad con los cambios hechos a las bases de conocimiento.

Por el lado técnico, se logró hacer todo esto utilizando herramientas de alto nivel que permitieron que el equipo pudiera concentrarse en el desarrollo de las funcionalidades y no en detalles técnicos de implementación. Esto permitió agilizar el desarrollo y lograr más en menos tiempo. Esto, como presentado en la 8 Arquitectura y diseño era uno de los principales objetivos propuestos desde un principio.

El equipo considera que GeneXus podrá seguir sacando valor de esta prueba de concepto en varios sentidos.

Retomando el desglose del problema planteado en la sección 2.2 Resumen se logró cumplir con los puntos allí mencionados.

12.1.1 Especificar requerimientos de un modo estructurado

Se introdujo exitosamente un *parser* de la gramática Gherkin dentro del IDE de GeneXus, el hecho de que el *parser* viniera dado y no fuera desarrollado por el equipo, le permitió al mismo enfocarse en otros aspectos del desarrollo desentendiendose del costo y esfuerzo necesario para

desarrollar y probar un *parser*. Solo fueron agregadas algunas extensiones a la gramática por sobre el *parser* existente.

12.1.2 Mantener trazabilidad entre el requerimiento y su implementación

Respecto de este punto fue donde el equipo tuvo mayor incertidumbre en saber si lo iba a poder cumplir o no, dado que se necesitó ayuda por parte del área de desarrollo de GeneXus y esta ayuda llegó muy cerca de la *deadline* del desarrollo.

12.1.3 Realizar revisiones que involucren a distintos actores del proyecto y aseguren la calidad

El mecanismo del Cliente de Requerimientos cubre las necesidades planteadas en este punto, el desarrollo de éste utilizando *frameworks,* como ya fue mencionado en varias partes de este documento, permitió desarrollarlo de manera rápida, debe recordarse que este proyecto es una prueba de concepto, por lo que no fueron tenidos en cuenta todos los aspectos que deberían tenerse en cuenta en un producto que estuviese en un entorno de producción.

12.1.4 Aprovechar la estructura de los requerimientos para generar elementos nativos de GeneXus

Este punto fue el más interesante y desafiante desde el lado del desarrollo, el equipo tuvo que adentrarse en el funcionamiento de un producto existente desde hace muchos años para poder integrarse y generar objetos dentro del mismo. Producto de esto se aprendió mucho del funcionamiento interno de GeneXus y de todas las cosas interesantes que ocurren detrás del entorno de desarrollo, desconocidas incluso para los propios desarrolladores *low code/no code* que usan GeneXus.

12.2 Cumplimiento de los objetivos

Al culminar la fase de desarrollo el equipo evaluó los objetivos inicialmente definidos. Dichos objetivos, eran objetivos más genéricos que se subdividieron en objetivos académicos, objetivos de producto y objetivos de proyecto. Para cada una de estas categorías se plantearon los objetivos que el equipo esperaba cumplir.

12.2.1 Objetivos académicos

Se plantearon cinco objetivos en esta categoría, el equipo considera haber cumplido con los mismos. A continuación se repasa cada uno de los objetivos con los que el equipo cuenta con evidencia objetiva para concluir.

Poner en práctica conocimientos adquiridos durante la carrera

Sin dudas se utilizó más de un conocimiento aprendido en la carrera para la realización del proyecto en todas las fases del mismo, desde la ingeniería de requerimientos, la arquitectura y diseño, la gestión del proyecto, la gestión de la calidad y gestión de la configuración. Si bien el equipo ha puesto en práctica conocimientos adquiridos en la carrera en su contexto laboral, nunca había llevado a cabo un proyecto desde cero, teniendo que aplicar conocimientos de diversas áreas de la ingeniería de *software*.

Aprender nuevas tecnologías

Si bien al menos un miembro del equipo conocía por lo menos uno de los lenguajes o *frameworks* utilizados para el desarrollo del proyecto, no todos los miembros conocían todas las tecnologías por igual. El hecho de haber aplicado metodologías como revisiones de a pares, hizo que todos los miembros del equipo tuvieran que revisar todas las partes de la solución escritas en las diferentes tecnologías con las que fue construida. Este proceso llevó a cada integrante a aprender sobre todas las tecnologías involucradas en el proceso.

Aprender nuevas herramientas

El equipo logró cumplir con este objetivo, ya que se hizo uso de herramientas que los miembros no conocían. Como ejemplo de herramientas usadas y que los miembros del equipo no conocían tenemos Notion, herramienta que permite tomar notas de manera sencilla y efectiva, Planning Poker, para realizar la estimación de las historias de usuario, dotPeek, para realizar descompilación de dlls, Toggl, para dejar registros de las horas invertidas en el proyecto, entre otras herramientas.

Aprender nuevos procesos

Definimos en los objetivos aprender tres procesos que no fueron aplicados en la carrera por los miembros del equipo. Estos fueron Popcorn Flow, Dual Track Scrum y Kanban. Popcorn Flow y Dual Truck Scrum no fueron vistos en la carrera, mientras que Kanban si, pero no de forma práctica. Este objetivo fue cumplido ya que se usaron estos tres procesos, con un resultado que el equipo considera satisfactorio.

12.2.2 Objetivos de producto

En esta categoría se plantearon dos objetivos, de los cuales si bien el equipo recibió un buen *feedback* en las instancias de validación del proyecto con el cliente, no tiene elementos para afirmar completamente que el objetivo Satisfacción de los interesados fue completamente cumplido.

Calidad interna

Se realizaron todas las actividades planteadas en la medida de éxito del objetivo. Se realizaron revisiones de a pares en cada *pull request*, esto llevó a aumentar la calidad del código sustancialmente y a nutrirse de cada uno de los comentarios que dejaban los miembros del equipo en los comentarios de sugerencias de cambio en el *pull request.* Muchos de los comentarios hacían referencias a artículos en internet o documentación de cómo hacer de una mejor forma lo que estaba hecho en el *pull request*. Esto fue algo muy valioso, porque no solo sirvió para mejorar la calidad del código, sino que también permitió a los integrantes aprender nuevas formas de implementar o resolver determinados problemas de programación.

En cuanto a las pruebas unitarias se llegó a más del 80%, según los criterios mencionados en 10.2.3 Pruebas.

Respecto de las pruebas *End-To-End,* se realizaron pruebas con dichas características para el *backend* del Cliente de Requerimientos.

Por último, las historias de usuario cumplieron con los criterios INVEST.

12.2.3 Objetivos de proyecto

En esta sección se define un único objetivo, este trata sobre la factibilidad técnica del proyecto.

Medir la factibilidad técnica del proyecto

La medida de éxito de este objetivo era la capacidad de implementar la prueba de concepto en el año de proyecto. El equipo considera que cumplió de forma satisfactoria con este objetivo.

12.3 Lecciones aprendidas

- Se aprendió lo que es la industria *low-code/no-code*.
    - Se conocen los tipos de usuarios que tiene este área.
    - Al conocer los conceptos sobre los que se basa se comienza a entender que la simplicidad que muestra tiene por detrás una complejidad inmensa.
- El equipo se conoce mejor a sí mismos como profesionales.
    - Cada día se entienden más los propios perfiles, fortalezas y debilidades.
    - Con el pasar del tiempo se fue logrando una mejor sinergia entre los miembros del equipo.
- Se crece como desarrolladores.
    - Los desafíos propuestos por este proyecto han empujado al equipo a aprender más de las tecnologías que se usaron.

- Se aprendió a encarar este proyecto que tiene poca relación con aquello a lo que se acostumbra en los trabajos de cada uno de los miembros del equipo.

12.4 Próximos pasos

El producto desarrollado fue una **prueba de concepto**, como tal tiene múltiples aspectos a mejorar. Su valor, principalmente, es la información que permitió recopilar a partir de su uso y desarrollo. Se pueden, además, emplear más y mejores técnicas para seguir exprimiéndolo desde este punto de vista.

Se sabe, y se supo desde que se definió el *scope* del proyecto, que hay aspectos que no fueron prioritarios. Discutiblemente, uno de estos puede ser la experiencia de usuario. Este y otros son pilares del producto que la empresa GeneXus podrá pulir como crean conveniente para construir un producto robusto que aporte el valor que necesitan en materia de trazabilidad. Se cree haber contribuido de buena manera a que GeneXus pueda ser usado en aplicaciones de contexto crítico y por mercados más exigentes en este aspecto.

Por lo que se habló en varias instancias con gente de la empresa GeneXus la idea que se maneja es la de utilizar esta prueba de concepto como un punto de partida para un desarrollo más complejo, es decir, se cede a GeneXus el código de esta prueba de concepto en su totalidad para que ellos tomen las riendas del desarrollo a futuro del producto como lo consideren adecuado.

El equipo pretende también que esta tesis sea útil para quien quiera adentrarse en el desarrollo de extensiones para GeneXus, dado que de ésta se desprende documentación técnica relacionada a los mecanismos de integración con los que cuenta GeneXus.

13 Referencias

[1] Atlassian, “Guardar cambios | Atlassian Git Tutorial,” *Atlassian*. https://www.atlassian.com/es/git/tutorials/saving-changes (accessed Mar. 13, 2022).

[2] “What is an Epic User Story?,” *Agile Alliance |*, Aug. 31, 2017. https://www.agilealliance.org/glossary/epic/ (accessed Mar. 13, 2022).

[3] “Gherkin Reference - Cucumber Documentation.” https://cucumber.io/docs/gherkin/reference/ (accessed Mar. 09, 2022).

[4] “Frontend vs Backend,” *GeeksforGeeks*, Jul. 11, 2019. https://www.geeksforgeeks.org/frontend-vs-backend/ (accessed Mar. 13, 2022).

[5] “GeneXus,” *Wikipedia, la enciclopedia libre*. Mar. 11, 2022. Accessed: Mar. 13, 2022. [Online]. Available: https://es.wikipedia.org/w/index.php?title=GeneXus&oldid=142216877

[6] “Gherkin for Business Analysts.” https://modernanalyst.com/Resources/Articles/tabid/115/ID/3810/Gherkin-for-Business-Analysts.aspx (accessed Mar. 13, 2022).

[7] Atlassian, “Historias de usuario | Ejemplos y plantilla,” *Atlassian*. https://www.atlassian.com/es/agile/project-management/user-stories (accessed Mar. 13, 2022).

[8] “Protocolo de transferencia de hipertexto,” *Wikipedia, la enciclopedia libre*. Feb. 19, 2022. Accessed: Mar. 13, 2022. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Protocolo_de_transferencia_de_hipertexto&oldid=141779019

[9] “Low-code development platform,” *Wikipedia*. Feb. 15, 2022. Accessed: Mar. 13, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Low-code_development_platform&oldid=1072008877

[10] “Notion (productivity software),” *Wikipedia*. Mar. 09, 2022. Accessed: Mar. 13, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Notion_(productivity_software)&oldid=1076146055

[11] “Personal Cloud Storage – Microsoft OneDrive.” https://www.microsoft.com/en-ww/microsoft-365/onedrive/online-cloud-storage (accessed Mar. 13, 2022).

[12] “Pipeline (software),” *Wikipedia*. Nov. 24, 2021. Accessed: Mar. 13, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Pipeline_(software)&oldid=1056938880

[13] “Procedure object | Category.” https://wiki.genexus.com/commwiki/servlet/wiki?6293,Category%3AProcedure+object (accessed Mar. 13, 2022).

[14] “Repository (version control),” *Wikipedia*. Feb. 25, 2022. Accessed: Mar. 13, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Repository_(version_control)&oldid=1073850251

[15] “What is REST,” *REST API Tutorial*. https://restfulapi.net/ (accessed Mar. 13, 2022).

[16] “Transaction object | Category.” https://wiki.genexus.com/commwiki/servlet/wiki?1908,Category%3ATransaction+object (accessed Mar. 13, 2022).

[17] “How to write SMART goals (with examples),” *Work Life by Atlassian*, Dec. 26, 2021. https://www.atlassian.com/blog/productivity/how-to-write-smart-goals (accessed Mar. 09, 2022).

[18] “Gherkin Syntax - Cucumber Documentation.” https://cucumber.io/docs/gherkin/ (accessed Mar. 09, 2022).

[19] “IEEE Recommended Practice for Software Requirements Specifications,” IEEE. doi: 10.1109/IEEESTD.1998.88286.

[20] J. Rodela, “How to Create a Proof of Concept in 2022,” *The Blueprint*, Apr. 10, 2020. https://www.fool.com/the-blueprint/proof-of-concept/ (accessed Mar. 09, 2022).

[21] “Cucumber (software),” *Wikipedia*. Nov. 26, 2021. Accessed: Mar. 09, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cucumber_(software)&oldid=1057242422

[22] “Behavior-driven development,” *Wikipedia*. Dec. 27, 2021. Accessed: Mar. 09, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Behavior-driven_development&oldid=1062331685

[23] “Acceptance test–driven development,” *Wikipedia*. Jul. 13, 2021. Accessed: Mar. 09, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Acceptance_test%E2%80%93driven_development&oldid=1033456649

[24] M. Irshad, R. Britto, and K. Petersen, “Adapting Behavior Driven Development (BDD) for large-scale software systems,” *J. Syst. Softw.*, vol. 177, p. 110944, Jul. 2021, doi: 10.1016/j.jss.2021.110944.

[25] K. Henney, Ed., “Your Customers Do Not Mean What They Say,” in *97 things every programmer should know: collective wisdom from the experts*, 1st ed., Sebastopol, Calif: O’Reilly, 2010, p. 194.

[26] “Speak Gherkin And Learn How To Collect Requirements For Your Project | by Neoteric | Medium.” https://medium.com/@NeotericEU/speak-gherkin-and-learn-how-to-collect-requirements-for-your-project-dad171da8fb (accessed Mar. 13, 2022).

[27] G. Adzic, *Specification by example: how successful teams deliver the right software*. Shelter Island, N.Y: Manning, 2011.

[28] “Business Readable DSL.” https://martinfowler.com/bliki/BusinessReadableDSL.html (accessed Mar. 09, 2022).

[29] “UbiquitousLanguage.” https://martinfowler.com/bliki/UbiquitousLanguage.html (accessed Mar. 09, 2022).

[30] “DomainDrivenDesign.” https://martinfowler.com/bliki/DomainDrivenDesign.html (accessed Mar. 09, 2022).

[31] “Dynamic Testing Techniques.” https://www.tutorialride.com/software-testing/dynamic-testing-techniques.htm (accessed Mar. 09, 2022).

[32] John Ferguson Smart, “It’s actually a myth that BDD-style requirements are easy to write and can be written by anyone, any more than a book by Hemingway could be written by anyone. Good Gherkin is easy to read by anyone - that doesn’t make it easy to write. #BDD,” *@wakaleo*, Jun. 06, 2019. https://twitter.com/wakaleo/status/1136702740900585472 (accessed Mar. 09, 2022).

[33] K. Henney, Ed., *97 things every programmer should know: collective wisdom from the experts*, 1st ed. Sebastopol, Calif: O’Reilly, 2010.

[34] “What is Static Testing?” https://searchsoftwarequality.techtarget.com/definition/static-testing (accessed Mar. 09, 2022).

[35] K. E. Wiegers, “Seven Truths About Peer Reviews,” p. 9, 2002.

[36] J. Marasco, “What Is the Cost of a Requirement Error? | StickyMinds,” Jun. 26, 2007. https://www.stickyminds.com/article/what-cost-requirement-error (accessed Mar. 09, 2022).

[37] “Model Reviews: Best Practice or Process Smell?” http://agilemodeling.com/essays/modelReviews.htm (accessed Mar. 09, 2022).

[38] K. E. Wiegers, “When Two Eyes Aren’t Enough,” p. 8, 2001.

[39] “APP Coronavirus UY - App Store - Catálogo de Datos Abiertos.” https://catalogodatos.gub.uy/showcase/app-coronavirus-uy-app-store (accessed Mar. 10, 2022).

[40] “Private Workshop: Continuous Innovation & Change with PopcornFlow,” *Agile Sensei*. https://agilesensei.com/popcornflow/ (accessed Mar. 10, 2022).

[41] S. C. Jiménez, “Por qué nos aventuramos con el marco de trabajo Dual-Track Scrum,” *Medium*, Apr. 13, 2018. https://blog.ilogica.cl/por-qu%C3%A9-nos-aventuramos-con-un-nuevo-marco-de-trabajo-llamado-dual-track-scrum-fc41894a2388 (accessed Mar. 10, 2022).

[42] I. Eirale, M. García, R. Méndez, and G. Wagner, “GX DevOps Extension Integración de herramientas de apoyo al proceso de desarrollo de software en GeneXus,” Universidad ORT Uruguay, 2019. [Online]. Available: https://sisbibliotecas.ort.edu.uy/bib/90545

[43] “GQM,” *Wikipedia*. Dec. 28, 2021. Accessed: Mar. 10, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=GQM&oldid=1062447128

[44] R. Yang, “User Stories vs. Requirements,” Jan. 23, 2018. https://www.aha.io/blog/user-stories-vs-product-requirements (accessed Mar. 10, 2022).

[45] A. Kaushik, “Lab Usability Testing: What, Why, How Much.,” *Occam’s Razor by Avinash Kaushik*, Nov. 02, 2006. https://www.kaushik.net/avinash/lab-usability-testing-what-why-how-much/ (accessed Mar. 10, 2022).

[46] “ISO 25010.” https://iso25000.com/index.php/normas-iso-25000/iso-25010?start=0 (accessed Mar. 09, 2022).

[47] “Material Design,” *Material Design*. https://material.io/design (accessed Mar. 10, 2022).

[48] “bcrypt,” *Wikipedia*. Mar. 02, 2022. Accessed: Mar. 09, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1074878264

[49] A. R. is a S. C. based in K. C. S. is currently working on building apps with React *et al.*,

"Top Frontend Frameworks To Learn in 2021," *Programming with Mosh*, Feb. 08, 2021. https://programmingwithmosh.com/general/top-frontend-frameworks-to-learn-in-2021/ (accessed Mar. 09, 2022).

[50] ajcvickers, "Overview of Entity Framework Core - EF Core." https://docs.microsoft.com/en-us/ef/core/ (accessed Mar. 10, 2022).

[51] "Reverse engineering," *Wikipedia*. Jan. 28, 2022. Accessed: Mar. 13, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Reverse_engineering&oldid=1068377081

[52] "dotPeek: Decompilador .NET gratuito & Navegador Assembly de JetBrains," *JetBrains*. https://www.jetbrains.com/es-es/decompiler/ (accessed Mar. 13, 2022).

[53] "Toggl: Time Tracking, Project Planning and Hiring Tools to Help Teams Work Better." https://toggl.com/, https://toggl.com/ (accessed Mar. 13, 2022).

[54] Atlassian, "User Stories | Examples and Template," *Atlassian*. https://www.atlassian.com/agile/project-management/user-stories (accessed Mar. 09, 2022).

[55] "What is Planning Poker? | Definition and Overview." https://www.productplan.com/glossary/planning-poker/ (accessed Mar. 09, 2022).

[56] "Visual Studio Live Share: Real-Time Code Collaboration Tool." https://visualstudio.microsoft.com/services/live-share/ (accessed Mar. 13, 2022).

[57] B. Wagner, "C# Coding Conventions." https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions (accessed Mar. 09, 2022).

[58] "normas-especificas-para-la-presentacion-de-trabajos-finales-de-carrera-facultad-de-ingenieria-excepto-biotecnologia__documento-302.pdf." Accessed: Mar. 09, 2022. [Online]. Available: https://www.ort.edu.uy/innovaportal/file/95484/1/normas-especificas-para-la-presentacion-de-trabajos-finales-de-carrera-facultad-de-ingenieria-excepto-

biotecnologia__documento-302.pdf

[59] “hoja-de-verificacion-de-pautas-de-presentacion-de-trabajos-finales-de-carreras-excepto-biotecnologia__documento-303.pdf.” Accessed: Mar. 09, 2022. [Online]. Available: https://www.ort.edu.uy/innovaportal/file/95484/1/hoja-de-verificacion-de-pautas-de-presentacion-de-trabajos-finales-de-carreras-excepto-biotecnologia__documento-303.pdf

[60] “orientacion-para-titulos-resumenes-o-abstracts-e-informes-de-correccion-de-trabajos-finales-de-carrera__documento-306.pdf.” Accessed: Mar. 09, 2022. [Online]. Available: https://www.ort.edu.uy/innovaportal/file/95484/1/orientacion-para-titulos-resumenes-o-abstracts-e-informes-de-correccion-de-trabajos-finales-de-carrera__documento-306.pdf

[61] “GuiaEntregaFinal2021.pdf.” Accessed: Mar. 09, 2022. [Online]. Available: https://aulas.ort.edu.uy/pluginfile.php/530080/mod_resource/content/1/GuiaEntregaFinal2021.pdf

[62] “Conventional Commits,” *Conventional Commits*. https://www.conventionalcommits.org/en/v1.0.0-beta.4/ (accessed Mar. 09, 2022).

[63] T. Preston-Werner, “Semantic Versioning 2.0.0,” *Semantic Versioning*. https://semver.org/ (accessed Mar. 09, 2022).

[64] “Features • GitHub Actions,” *GitHub*. https://github.com/features/actions (accessed Mar. 09, 2022).

14 Anexos

14.1 Reportes de cierre de sprints

14.1.1 Sprint 1 (24/06-07/07)

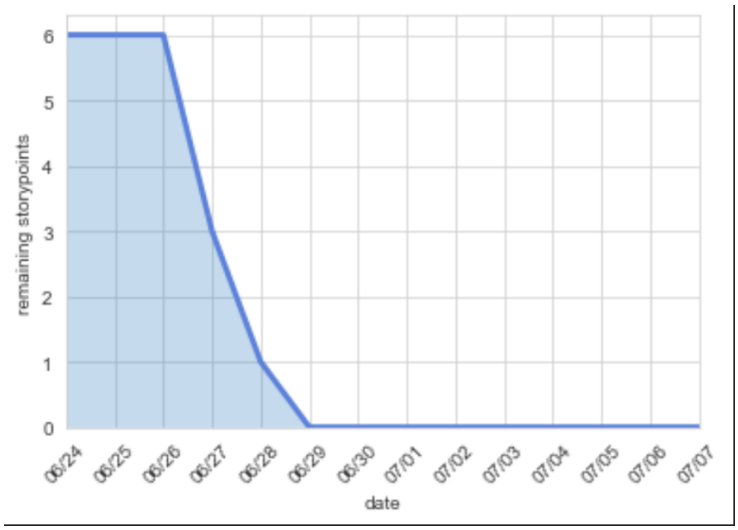


Figura 33: Grafica *sprint* 1

- SP estimados para el *sprint*: 6
- SP hechos en el *sprint*: 6
- Porcentaje de SP planeados que fueron hechos: 100%

	Título	Estimación	Fecha de finalización
1	Editor de texto base *in-browser*	3	26/06/2021
2	Objeto GeneXus "*Feature*"	2	27/06/2021
3	*Syntax highlighting* de *gherkin* para el editor ...	1	28/06/2021

Tabla 18: Tabla tareas *sprint* 1

14.1.2 Sprint 2 (08/07-21/07)

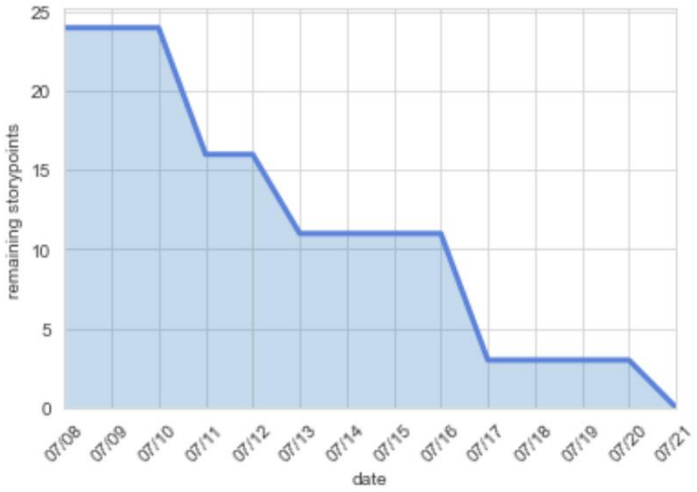


Figura 34 : Grafica *sprint* 2

- SP estimados para el *sprint*: 24
- SP hechos en el *sprint*: 24
- Porcentaje de SP planeados que fueron hechos: 100%

	Título	Estimación	Fecha de finalización
1	Pushear objeto a GxServer	8	16/07/2021
2	Configurar *build* de la extensión para desarrollo	5	12/07/2021
3	Separar proyecto de la extensión en UI y BL	3	20/07/2021
4	Asociación del editor al objeto *Feature*…	8	10/07/2021

Tabla 19: Tabla tareas *sprint* 2

14.1.3 Sprint 3 (22/07-04/08)

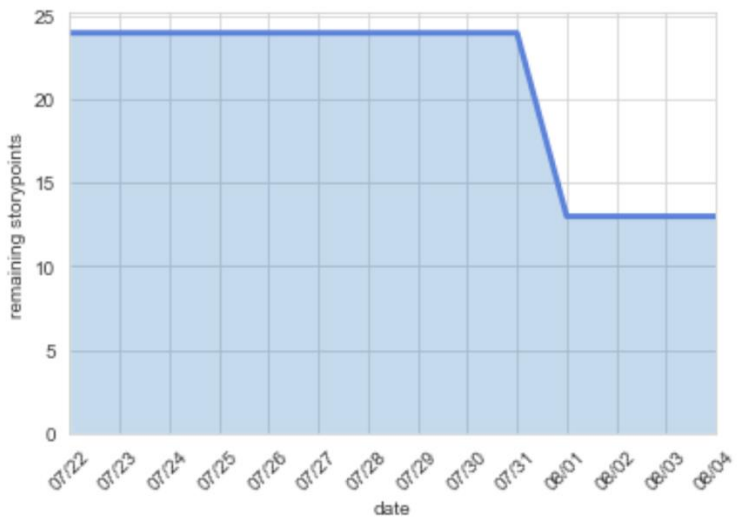


Figura 35: Grafica *sprint* 3

- SP estimados para el *sprint*: 24
- SP hechos en el *sprint*: 11
- Porcentaje de SP planeados que fueron hechos: 45%

	Título	Estimación	Fecha de finalización
1	Vincular transacción a objeto *Feature*	5	No se terminó
2	Generar transacción a partir de escenario	8	No se terminó
3	Parsear contenido de objeto *Feature* en inglés	5	31/07/2021
4	Agregar soporte para tipado explícito	3	31/07/2021
5	Parsear contenido de objeto Feature en español	3	31/07/2021

Tabla 20: Tabla tareas *sprint* 3

14.1.4 Sprint 4 (05/08-18/08)

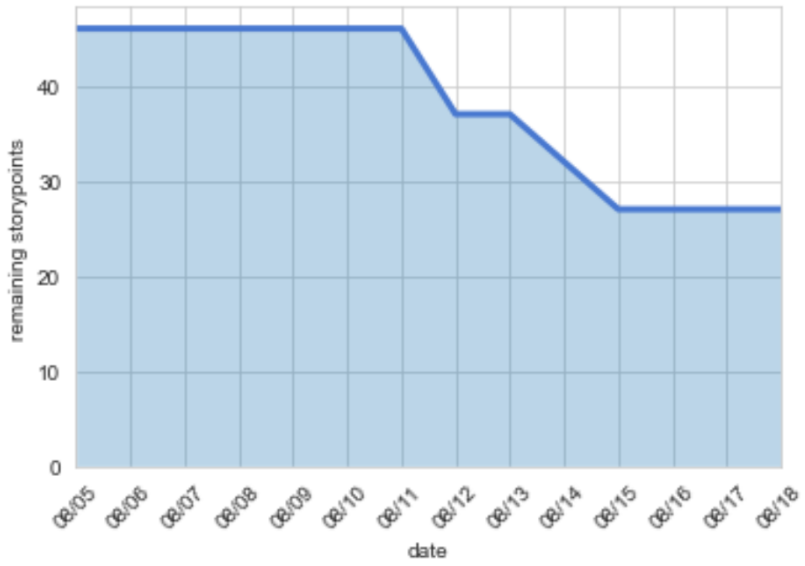


Figura 36: Grafica *sprint* 4

- SP estimados para el *sprint*: 46
- SP hechos en el *sprint*: 19
- Porcentaje de SP planeados que fueron hechos: 41%

	Título	Estimación	Fecha de finalización
1	Agregar soporte para tipado explícito en español	2	No se terminó
2	Extracción de nombre de transacción	1	No se terminó
3	Vincular prueba/*data provider* a objeto *Feature*	3	No se terminó
4	Vincular transacción a objeto *Feature*	5	No se terminó
5	Generar prueba y *data provider* a partir de esc...	8	No se terminó
6	Mecanismo de creación de escenarios específicos	8	No se terminó
7	Inferencia de largo de *strings*	5	14/08/2021
8	Inferencia de tipos numéricos - *double* vs *int*	5	13/08/2021
9	Definición de atributo identificador de transa...	1	11/08/2021
10	Generar transacción a partir de escenario	8	11/08/2021

Tabla 21: Tabla tareas *sprint* 4

14.1.5 Sprint 5 (19/08-01/09)

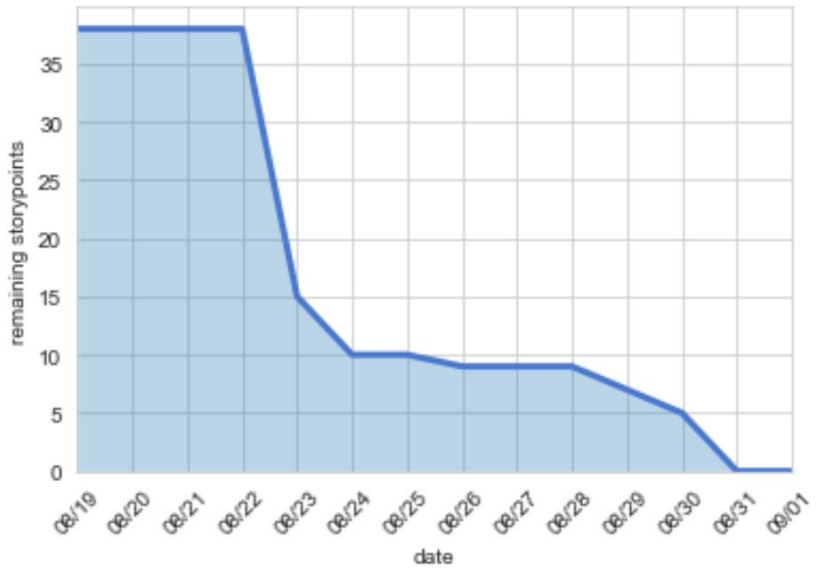


Figura 37: Grafica *sprint* 5

- SP estimados para el *sprint*: 38
- SP hechos en el *sprint*: 38
- Porcentaje de SP planeados que fueron hechos: 100%

	Título	Estimación	Fecha de finalización
1	Mostrar una excepción por consola, si el usuar...	2	30/08/2021
2	Update de los atributos de una transacción (ag...	3	30/08/2021
3	Patrones de títulos para generación en múltipl...	2	29/08/2021
4	Agregar soporte para tipado explícito en español	2	28/08/2021
5	Extracción de nombre de transacción	1	25/08/2021
6	Vincular prueba/*data provider* a objeto *Feature*	3	23/08/2021
7	Agregar *logs*/excepción a la generación de objetos	2	23/08/2021
8	Selección de escenario y generación de procedi…	2	22/08/2021
9	Vincular transacción a objeto *Feature*	5	22/08/2021
10	Generar prueba y *data provider* a partir de esc…	8	22/08/2021

11	Mecanismo de creación de escenarios específicos	8	22/08/2021

Tabla 22: Tabla tareas *sprint* 5

14.1.6 Sprint 7 (16/09-29/09)

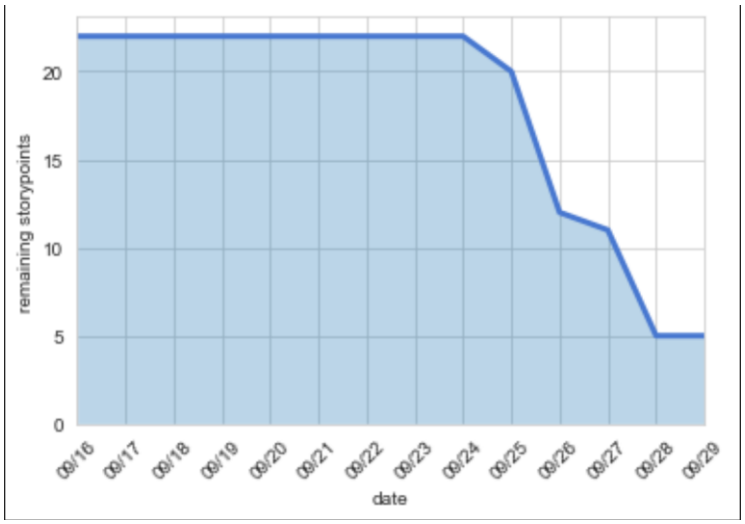


Figura 38: Grafica *sprint* 7

- SP estimados para el *sprint*: 22
- SP hechos en el *sprint*: 17
- Porcentaje de SP planeados que fueron hechos: 77%

	Título	Estimación	Fecha de finalización
1	Autenticación	2	No se terminó
2	Crear usuarios	3	No se terminó
3	Listado de proyectos	2	27/09/2021
4	Configurar proyecto para *frontend* del portal	2	27/09/2021
5	Configurar proyecto para *backend* del portal	2	27/09/2021
6	Permitir valor *nullable* en los atributos de la...	1	26/09/2021
7	Refactor: soporte para múltiples idiomas	3	25/09/2021
8	Agregar soporte para nuevos tipados explícitos	1	25/09/2021
9	Bug: agregar atributo a una entidad que ya exi...	1	25/09/2021

10	Agregar validación a los nombres de atributos...	1	25/09/2021
11	Agregar soporte para relaciones entre transacc...	1	25/09/2021
12	Definir tipo por defecto para una columna	1	25/09/2021
13	Bug: procedimiento creado sin pruebas queda...	1	24/09/2021
14	Agregar ID a transacción especificada sin dich...	1	24/09/2021

Tabla 23: Tabla tareas *sprint* 7

14.1.7 Sprint 8 (30/09-13/10)

Figura 39: Grafica *sprint* 8

- SP estimados para el *sprint*: 32
- SP hechos en el *sprint*: 22
- Porcentaje de SP planeados que fueron hechos: 68%

	Título	Estimación	Fecha de finalización
1	Control de acceso - Proyectos	8	No se terminó
2	DisplayName (*nick*) de usuario debería ser único	1	No se terminó
3	Email de usuario debería ser único	1	No se terminó
4	Config CI/CD Portal API	2	10/10/2021
5	Setup E2E tests	8	10/10/2021

6	El largo de una contraseña debe ser al menos 8...	1	04/10/2021
7	Config CI/CD Portal	2	03/10/2021
8	Config tests portal	3	03/10/2021
9	Autenticación	2	02/10/2021
10	Alta de proyectos	1	02/10/2021
11	Crear usuarios	3	30/09/2021

Tabla 24: Tabla tareas *sprint* 8

14.1.8 Sprint 9 (14/10-28/10)

Figura 40: Grafica *sprint* 9

- SP estimados para el *sprint*: 34
- SP hechos en el *sprint*: 12
- Porcentaje de SP planeados que fueron hechos: 35%

	Título	Estimación	Fecha de finalización
1	Eliminar acceso a proyecto	3	No se terminó
2	Soporte para relaciones en henvo/ra-jsonapi-cl...	8	No se terminó
3	Control de acceso - Proyectos	8	No se terminó
4	Ver miembros de un proyecto	3	No se terminó

5	DisplayName (*nick*) de usuario debería ser único	1	25/10/2021
6	Eliminar código no usado	1	23/10/2021
7	Poner id de usuario en *JWT*	1	23/10/2021
8	Exception filter para la *API*	1	22/10/2021
9	Testear login (*E2E*)	2	22/10/2021
10	Email de usuario debería ser único	1	19/10/2021
11	Configurar ambiente de *staging* (*Frontend*)	3	18/10/2021
12	Configurar ambiente de *staging* (*Backend*)	2	18/10/2021

Tabla 25: Tabla tareas *sprint* 9

14.1.9 Sprint 10 (29/10-12/11)

Figura 41: Grafica *sprint* 10

- SP estimados para el *sprint*: 26
- SP hechos en el *sprint*: 20
- Porcentaje de SP planeados que fueron hechos: 76%

	Título	Estimación	Fecha de finalización
1	Aprobar una *feature*	3	No se terminó
2	Edición de feature	3	No se terminó

3	Ver listado de comentarios en una *feature*	5	03/11/2021
4	Crear comentario en una *feature*	3	03/11/2021
5	Ver listado de todos los requerimientos a los ...	3	30/10/2021
6	Ver listado de *features* en un proyecto	3	30/10/2021
7	Crear nueva *feature* en un proyecto	5	30/10/2021
8	Bug: Creación de proyectos no funciona	1	29/10/2021

Tabla 26: Tabla tareas *sprint* 10

14.1.10 Sprint 11 (13/11-24/11)

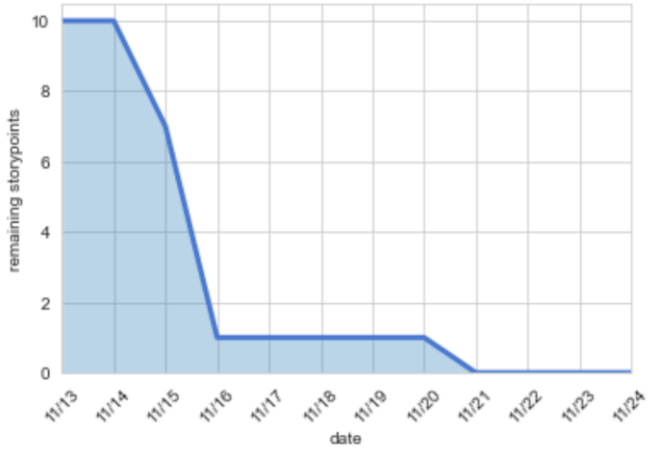


Figura 42: Grafica *sprint* 11

- SP estimados para el *sprint*: 10
- SP hechos en el *sprint*: 10
- Porcentaje de SP planeados que fueron hechos: 100%

	Título	Estimación	Fecha de finalización
1	Desde la vista de features en un proyecto, si...	1	20/11/2021
2	Conectar KB de GeneXus con un proyecto en...	1	15/11/2021
3	Descargar features desde *IDE* GeneXus	5	15/11/2021
4	Eliminar posibilidad de borrar proyectos cuand...	1	14/11/2021

5	La solución del *backend* no tenía las referenci...	1	14/11/2021
6	No se debería permitir editar *features* que est...	1	14/11/2021

Tabla 27: Tabla tareas *sprint* 11

14.1.11 Sprint 13 (09/12-22/12)

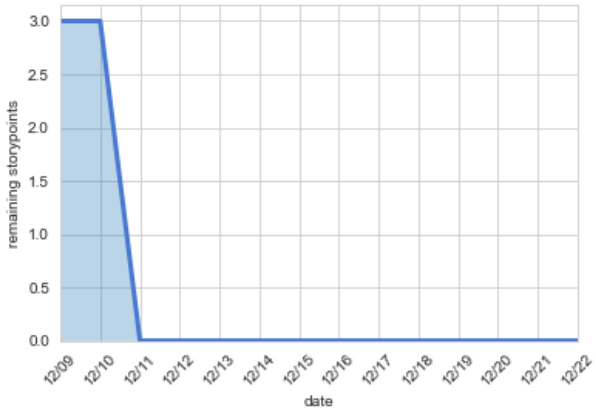


Figura 43: Grafica *sprint* 13

- SP estimados para el *sprint*: 3
- SP hechos en el *sprint*: 3
- Porcentaje de SP planeados que fueron hechos: 100%

	Título	Estimación	Fecha de finalización
1	Redirección en creación de *Features*	3	09/12/2021

Tabla 28: Tabla tareas *sprint* 13

14.1.12 Sprint 14 (23/12-05/01)

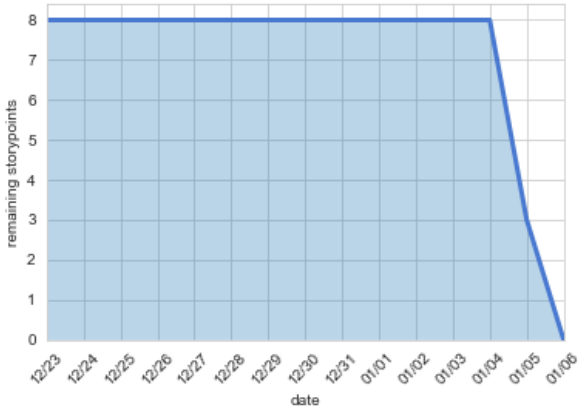


Figura 44: Grafica *sprint* 14

- SP estimados para el *sprint*: 8
- SP hechos en el *sprint*: 8
- Porcentaje de SP planeados que fueron hechos: 100%

	Título	Estimación	Fecha de finalización
1	*Autocomplete*	3	05/01/2022
2	Quitar código no usado de extensión	2	04/01/2022
3	Agregar ícono a objeto *Feature*	3	04/01/2022

Tabla 29: Tabla tareas *sprint* 14

14.1.13 Sprint 15 (06/01-19/01)

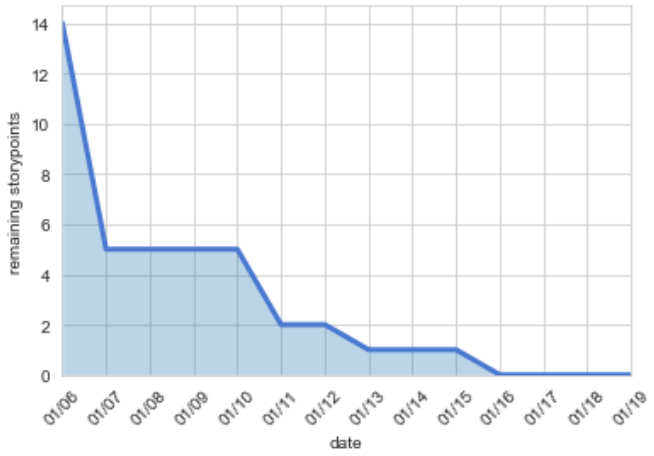


Figura 45: Grafica *sprint* 15

- SP estimados para el *sprint*: 14
- SP hechos en el *sprint*: 14
- Porcentaje de SP planeados que fueron hechos: 100%

	Título	Estimación	Fecha de finalización
1	*Snippets*	5	06/01/2022
2	Usar language del navegador para *i18n*	2	06/01/2022
3	Actualizar traducciones	2	12/01/2022
4	Formateador automático de código	2	10/01/2022
5	Agregar indicador de carga a *login*	1	15/01/2022
6	Eliminar código no usado de la *API*	2	06/01/2022

Tabla 30: Tabla tareas *sprint* 15

14.2 Planes de riesgos

Para cada *sprint* se analizó y, de considerarse necesario, se actualizó el plan de riesgos del proyecto.

Se registra cada riesgo de la siguiente manera en el plan de riesgos inicial:

- Debe tener un identificador único de riesgo.
- Una especificación que detalle de que se trata.
- Estar clasificado dentro de la escala de impacto:
    - Muy Alto
    - Alto
    - Medio
    - Bajo
- Tener una probabilidad de ocurrencia que puede tomar los valores de:
    - Poco probable
    - Probable
    - Muy probable
    - Altamente probable
- Tener una ocurrencia en el tiempo que puede ser:
    - Inmediato
    - Mediano plazo
    - Largo plazo
- Tener un plan de contingencia.

La notación se simplifica en las actualizaciones que se hicieron por *sprint* de la siguiente manera:

- #ID - Título.
    - Contingencia: Qué hacer si se materializa el riesgo.

14.2.1 Plan inicial (previo al sprint 1 de desarrollo)

- **Riesgo 01:**

- **Id:** 01
- **Especificación:** Falta de conocimiento en el ecosistema GeneXus.
- **Impacto:** Muy alto.
- **Probabilidad de ocurrencia:** Muy probable.
- **Ocurrencia en el tiempo:** Corto Plazo.
- **Plan de contingencia:** Realizar los cursos de capacitación ofrecidos por GeneXus.

- **Riesgo 02:**
  - **Id:** 02
  - **Especificación:** Falta de conocimiento del problema a atacar, el cómo afecta este a los usuarios actuales de GeneXus.
  - **Impacto:** Alto.
  - **Probabilidad de ocurrencia:** Probable.
  - **Ocurrencia en el tiempo:** Mediano Plazo.
  - **Plan de contingencia:** Realizar encuestas, charlas con expertos y lectura de documentación existente.

- **Riesgo 03:**
  - **Id:** 03
  - **Especificación:** Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
  - **Impacto:** Alto.
  - **Probabilidad de ocurrencia:** Muy Probable.
  - **Ocurrencia en el tiempo:** Corto Plazo.
  - **Plan de contingencia:** Coordinar entrevistas con expertos en el tema (Gastón Milano y Federico Azzato).

- **Riesgo 04:**

- **Id:** 04
- **Especificación:** Falta del software necesario para poder llevar a cabo el proyecto (GeneXus, GxServer).
- **Impacto:** Alto.
- **Probabilidad de ocurrencia:** Poco Probable.
- **Ocurrencia en el tiempo:** Corto Plazo.
- **Plan de contingencia:** Pedir licencias correspondientes para el desarrollo.

- **Riesgo 05:**
  - **Id:** 05
  - **Especificación:** Que alguno de nosotros se ausente por un tiempo.
  - **Impacto:** Medio.
  - **Probabilidad de ocurrencia:** Poco Probable.
  - **Ocurrencia en el tiempo:** Largo Plazo.
  - **Plan de contingencia**: Acordamos avisar con tiempo de ser posible y prever la situación y trabajar un poco más la semana anterior y la siguiente para compensar.

- **Riesgo 06:**
  - **Id:** 06
  - **Especificación:** Dado que se trabaja remoto que se caiga internet.
  - **Impacto:** Alto.
  - **Probabilidad de ocurrencia:** Poco Probable.
  - **Ocurrencia en el tiempo:** Largo Plazo.
  - **Plan de contingencia:** Se trabajará con este riesgo.

- **Riesgo 07:**
  - **Id:** 07

- **Especificación:** Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo, que nuestro contacto dentro de GeneXus no nos responda o no nos ayude.
- **Impacto:** Muy Alto.
- **Probabilidad de ocurrencia:** Poco Probable.
- **Ocurrencia en el tiempo:** Largo Plazo.
- **Plan de contingencia:** Se trabajará con este riesgo. Mantendremos una buena relación con el cliente para evitar que esto pase.

14.2.2 Plan de riesgos sprint 1 de desarrollo (24/06)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: *Mail* a Federico Azzato pidiendo ayuda al materializarse el riesgo.
- Que alguno de los integrantes del equipo se ausente por un tiempo – es período de parciales y entregas, el riesgo es más alto.
    - Contingencia: se planeó el *sprint* de forma más liviana con menos SP y relativamente poca investigación.
- Problemas de comunicación entre el cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.

14.2.3 Plan de riesgos sprint 2 de desarrollo (08/07)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Mail a Federico Azzato y Maximiliano Barnech pidiendo ayuda al materializarse el riesgo.

- Que alguno de los integrantes del equipo se ausente por un tiempo – es período de parciales y entregas, el riesgo es más alto.
    - Contingencia: quedan menos entregas que en el caso del *sprint* 1 pero sigue habiendo algunas, miembros del equipo quedan libres a mitad de sprint así que se planea casi “normal”.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.
- Problemas configurando IIS para hacer funcionar GXServer local.
    - Contingencia: *Mail* a Federico Azzato que es quien más sabe sobre esto.

14.2.4 Plan de riesgos sprint 3 de desarrollo (22/07)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Mail a Federico Azzato y Maximiliano Barnech pidiendo ayuda al materializarse el riesgo.
    - Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.

14.2.5 Plan de riesgos sprint 4 de desarrollo (05/08)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Mail a Federico Azzato pidiendo ayuda, en caso de materializarse el riesgo.
- Que alguno de nosotros se ausente por un tiempo

- Contingencia: recuperará horas más adelante y trabajará un poco más antes de irse.

- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo, que nuestro contacto dentro de GeneXus no nos responda o no nos ayude.
    - Contingencia: seguiremos insistiendo y usaremos la influencia de Darío con los clientes para intentar que nos contesten.

- Complicaciones en la integración con GxTest - Tenemos que determinar de modo empírico si está disponible – no sabemos si se puede.

- Seguramente lo mejor sea no agregar dependencia directa contra GxTest, es decir, levantar por reflection la dll. ¿Tenemos interfaces? No creemos. Por ello, si hay cambios en la interfaz se rompe todo, incluso si tenemos un wrapper que limite el impacto de cambio sobre la librería hay impacto igual
    - Contingencia: buscar otra alternativa.

14.2.6 Plan de riesgos sprint 5 de desarrollo (19/08)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Mail a Federico Azzato y Maximiliano Barnech pidiendo ayuda al materializarse el riesgo.

- Que alguno de los integrantes del equipo se ausente por un tiempo.

- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.

- Problemas con la compilación del proyecto GeneXus.
    - Contingencia: Reinstalar Gx y en caso de que no funcione, comunicarse con GeneXus.

- Solo se cuenta con una licencia de GxTest (ya que solo hay un GxServer), esto puede causar pérdidas de tiempo, por ejemplo, a la hora de testear.
    - Contingencia: Al momento se hace Pair Programming.
- Hay mucho avance no validado por el cliente o experto.
    - Contingencia: Validar decisiones con el cliente o experto antes de realizar una implementación en adelante.

14.2.7 Plan de riesgos sprint 7 de desarrollo (16/09)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.
- Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.
- Solo se cuenta con una licencia de GxTest (ya que solo hay un GxServer), esto puede causar pérdidas de tiempo, por ejemplo, a la hora de testear.
    - Contingencia: Al momento se hace *Pair Programming*.
- Hay mucho avance no validado por el cliente o experto.
    - Contingencia: Este sprint es 100% de validación y gestión, se pausa el desarrollo para asegurar el rumbo correcto del proyecto.

14.2.8 Plan de riesgos sprint 8 de desarrollo (30/09)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.

    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.

- Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.
- Solo se cuenta con una licencia de GxTest (ya que solo hay un GxServer), esto puede causar pérdidas de tiempo, por ejemplo, a la hora de testear.
    - Contingencia: Al momento se hace *Pair Programming*.
- Hay mucho avance no validado por el cliente o experto.
    - Contingencia: En este *sprint* hay reunión de validación con el cliente, se podrá eliminar dudas.
- El desarrollo de código se puede ver enlentecido debido a la curva de aprendizaje de los *frameworks* usados para el desarrollo y la falta de experiencia en ellos.
    - Contingencia: Estudiar, leer documentación e implementar código, aunque no sea perfecto.

14.2.9 Plan de riesgos sprint 9 de desarrollo (14/10)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.
- Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.
- Hay mucho avance no validado por el cliente o experto.

    - Contingencia: En este sprint hay reunión de validación con el cliente, se podrá eliminar dudas.

- El desarrollo de código se puede ver enlentecido debido a la curva de aprendizaje de los frameworks usados para el desarrollo y la falta de experiencia en ellos.

    - Contingencia: Estudiar, leer documentación e implementar código, aunque no sea perfecto.

14.2.10 Plan de riesgos sprint 10 de desarrollo (29/10)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.

    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.

- Que alguno de los integrantes del equipo se ausente por un tiempo.

- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.

    - Contingencia: insistir con los clientes para intentar que contesten.

- Hay mucho avance no validado por el cliente o experto.

    - Contingencia: En este sprint hay reunión de validación con el cliente, se podrá eliminar dudas.

- El desarrollo de código se puede ver enlentecido debido a la curva de aprendizaje de los frameworks usados para el desarrollo y la falta de experiencia en ellos.

    - Contingencia: Estudiar, leer documentación e implementar código, aunque no sea perfecto.

14.2.11 Plan de riesgos sprint 11 de desarrollo (13/11)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.

    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.

- Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.
- Hay mucho avance no validado por el cliente o experto.
    - Contingencia: En este sprint hay reunión de validación con el cliente, se podrá eliminar dudas.
- El desarrollo de código se puede ver enlentecido debido a la curva de aprendizaje de los frameworks usados para el desarrollo y la falta de experiencia en ellos.
    - Contingencia: Estudiar, leer documentación e implementar código, aunque no sea perfecto.

14.2.12 Plan de riesgos sprint 13 de desarrollo (09/12)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.
- Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.

14.2.13 Plan de riesgos sprint 14 de desarrollo (23/12)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.

    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.

- Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.

14.2.14 Plan de riesgos sprint 15 de desarrollo (06/01)

- Falta de documentación clara sobre la construcción de extensiones sobre GeneXus.
    - Contingencia: Los contactos técnicos del equipo están de vacaciones al momento. Si se materializa el riesgo hay que solucionarlo cuando vuelvan.
- Que alguno de los integrantes del equipo se ausente por un tiempo.
- Problemas de comunicación entre cliente (GeneXus) y los integrantes del grupo.
    - Contingencia: insistir con los clientes para intentar que contesten.

14.3 Manual de usuario

14.3.1 Introducción

En este manual veremos el funcionamiento básico del Cliente de Requerimientos. Incluyendo la creación de un proyecto con sus colaboradores y la creación de *Features* en el mismo. También veremos la sintaxis sugerida para la redacción de un escenario y la necesaria para crear tipos de datos explícitos. Por último, veremos cómo usar los *snippets* provistos por el editor del Cliente de Requerimientos.

Vista general del Cliente de Requerimientos

Una vez ingresadas las credenciales en Cliente de Requerimientos veremos una pantalla similar a la siguiente:

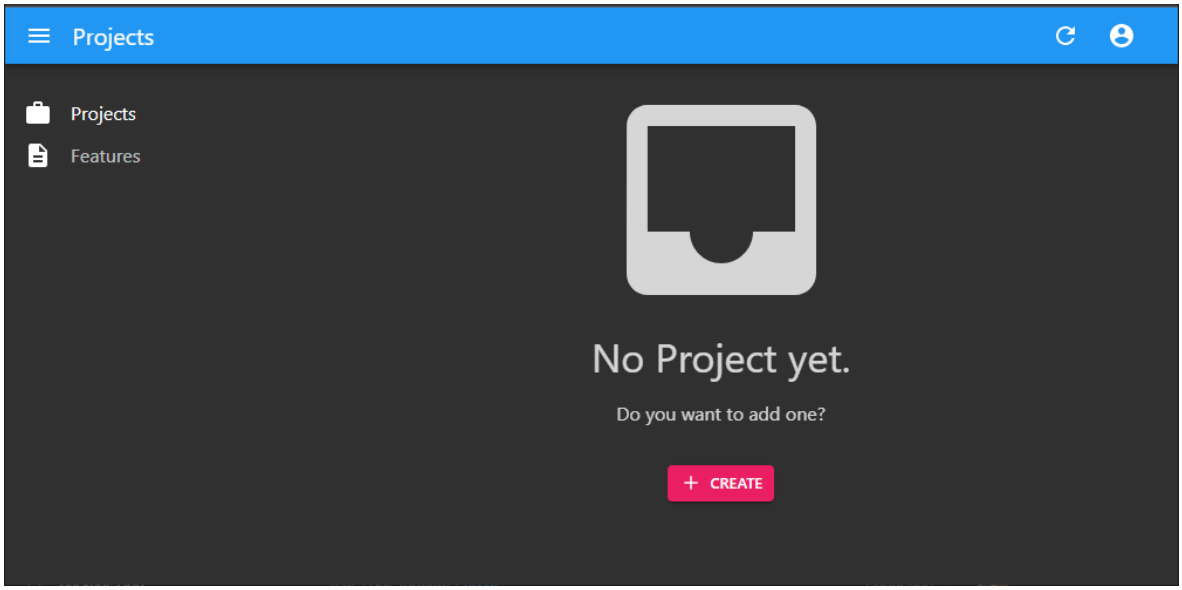


Figura 46: Vista general del Cliente de Requerimientos

En el menú de la izquierda veremos los proyectos a los que pertenece el usuario logueado o en los que es colaborador y en la pestaña *feature*, veremos las *features* que pertenecen a los proyectos en los que el usuario es miembro o colaborador.

Por último a la derecha veremos los botones de cerrar sesión y refrescar la página.

14.3.2 Manejo de proyectos

En esta sección se detalla el funcionamiento de los proyectos en el Cliente de Requerimientos, pero primero, definamos que es un Proyecto. Un proyecto es una de las entidades principales del Cliente de Requerimientos, cada proyecto "mapea" a una *Knowledge Base* de GeneXus. En cada proyecto van a existir N colaboradores, en donde uno de estos es el dueño de este. Cada proyecto también va a contener M *Features*, siendo estas creadas por los colaboradores de cada proyecto.

Creación de un proyecto

Para crear un nuevo proyecto hacemos clic sobre el botón "*create*" o hacemos clic sobre el botón *create* posicionado en la parte superior izquierda sobre la lista de proyectos. Nos pedirá un nombre y una lista de usuarios del sistema que serán los colaboradores de dicho proyecto.

Detalles del proyecto

Una vez creado el proyecto, veremos una vista de detalles del mismo, similar a la que se muestra a continuación.

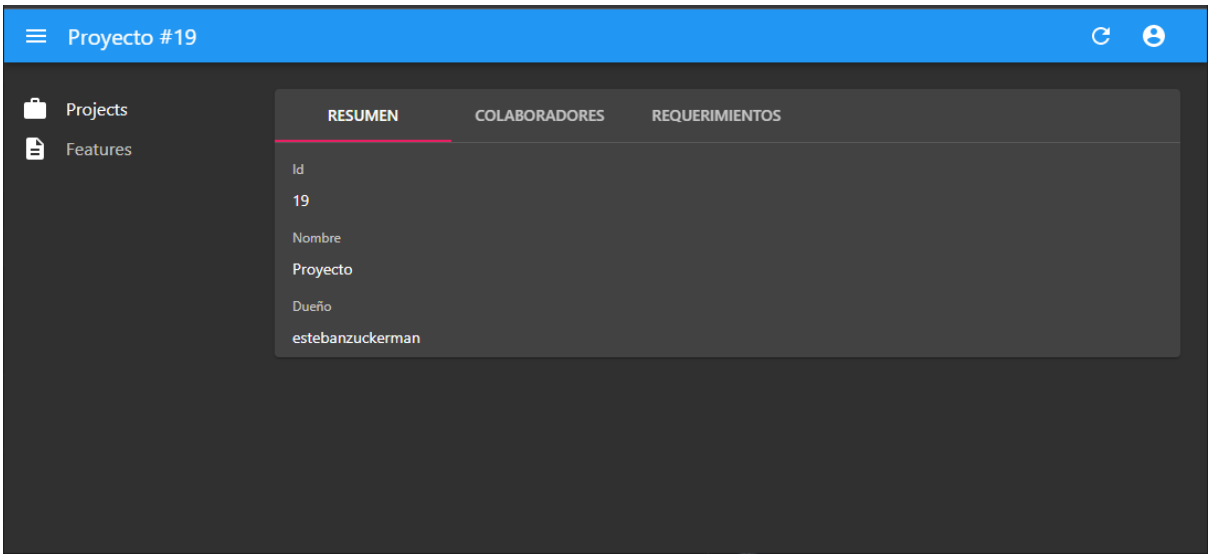


Figura 47: Vista del resumen de un proyecto

En el resumen, podemos ver los detalles del mismo:

- **Id:** Número correlativo que identifica al proyecto (servirá más adelante para vincular el proyecto con una KB de GeneXus).
- **Nombre**: Nombre del proyecto.
- **Dueño**: Nombre del dueño del proyecto.

Como la pestaña lo sugiere, en colaboradores veremos la lista de colaboradores que pueden crear, aprobar y comentar features sobre el proyecto.

Por último en la pestaña *Features*, se ven los *Features* pertenecientes a dicho proyecto.

14.3.3 Manejo de features

La *Feature* es el objeto principal de nuestro sistema, esta representa una necesidad de negocio, es escrita por un usuario con un rol de negocio y luego es usada por el desarrollador para realizar

la implementación de código. Las features tienen un ciclo de vida desde que se crea, hasta que se completa y es descargada por el/los desarrolladores.

Se detalla el manejo de las *Features*, desde la creación hasta su cierre al final del ciclo de vida de esta.

Creación de una Feature

La creación de una *Feature* se puede realizar desde dos pestañas, desde la pestaña de nuestras *Features*, o estando en un proyecto específico.

Para crear la *Feature* debemos seleccionar el proyecto a la que pertenece (si no estamos ya en un proyecto en específico), darle un título y el contenido.

Para saber cómo redactar el contenido de la *Feature* ver la sección "Cómo escribir una *feature*" de este documento.

Comentar una feature

Una vez que un desarrollador escribe una *Feature* en un proyecto, el resto de los integrantes de este pueden verla y comentar al respecto. Esto parece algo trivial, pero es sumamente importante, dado que el escritor de esta pudo haber omitido algo o cometer algún error, la idea de los comentarios es marcar estos puntos, para que el escritor corrija la *Feature* escrita. No solo son errores los que se pueden comentar, sino también sugerencias de mejora o un simple comentario felicitando al escritor por su buen trabajo.

Para comentar una *Feature* existe en esta una pestaña específica, es tan simple como darle clic al botón ¨Agregar comentario¨ y escribir el contenido de este.

En esta sección se mencionó también algo importante, las *Features* pueden ser editadas, siempre y cuando esta no estén en estado *¨Done¨* (*Commit*), esto de los estados lo veremos en los siguientes puntos.

Aprobar una Feature

Como se mencionó antes, la idea es que un usuario escriba una *Feature* y luego el resto de los miembros del proyecto la revisen, dejando comentarios con sugerencia de cambios, si corresponde. Pongámonos en el lugar de revisor de una *Feature*, vemos que esta tiene un detalle y lo comentamos, entonces el escritor lo corrige y nos responde nuevamente, al ver esto nosotros entendemos que la *Feature* está escrita de forma correcta, y ¿qué hacemos? Bueno, en este punto debemos aprobar esta *Feature*, aprobar significa algo así como "la *Feature* es correcta, apruebo que sea cerrada (*Commit*)". Una *Feature* puede ser aprobada por cualquier miembro del proyecto a la que esta pertenece, menos el usuario que la escribió.

Para aprobar una *Feature* existe un botón en la parte superior derecha de esta, que desaparecerá al aprobar esta. También existe una pestaña que muestra las aprobaciones obtenidas hasta el momento. Un detalle no menor es que, si una *Feature* fue aprobada por N usuarios, y luego esta es editada, entonces las aprobaciones serán eliminadas. ¿Te imaginas aprobar una *Feature* y que luego la cambien, cambiando el contenido por uno que no te parece bien?

Commitear una Feature

Una *Feature*, en cualquier momento y solo por el dueño del proyecto, puede ser cerrada (*Commit*). Esto es, pasar la *Feature* de estado *¨Pending¨* a *¨Done¨* (la *Feature* tiene dos estas, *Pending*, en este estado está desde que se crea hasta el momento que es cerrada, en este punto pasa a estado *Done*). Cuando una *Feature* está en estado "Done" puede ser descargada e implementada por los desarrolladores. También cuando está en estado "*Done*" se pierden ciertas características, como el poder agregar comentarios o editarla.

Si bien en la primera frase del párrafo anterior dijimos que la *Feature* puede ser cerrada en cualquier momento, esto no es recomendable. La idea sería (esto es flexible a cada proyecto), definir una cierta cantidad mínima de aprobaciones, antes de que el Dueño del proyecto pueda commitear.

Para cerrar una *Feature* existe un botón en la parte superior derecha que nos permite realizar esta acción (siempre y cuando seas el Dueño del proyecto). Al cerrar, en el Resumen de la *Feature* veremos el cambio de estado.

Eliminar una Feature

Para la eliminación de una *Feature*, la misma no debe estar cerrada. Para eliminarla completamente debemos dirigirnos a la opción editar, dentro de la *Feature* que queremos eliminar, y le damos al botón de borrar en la parte inferior derecha.

Cómo escribir una Feature

En este punto tendremos que escribir nuestro *Feature* de negocio, en nuestro sistema, esto se hace usando la gramática Gherkin. Gherkin es un DSL (*Domain Specific Language*), que nos permite especificar *Features* de una forma semi-estructurada y permitiendo que sea entendido por usuarios técnicos y no técnicos o de negocio. Gherkin cuenta con una serie de *keywords* que definen cada una de las partes de nuestro *Feature* (ver estas en: Gherkin *Keywords*).

Para explicar cómo escribir una *Feature* nos guiaremos con un ejemplo, en este ejemplo veremos cómo escribir una *Feature*, que luego va a resultar en un procedimiento en GeneXus.

El resultado al que queremos llegar es el siguiente:

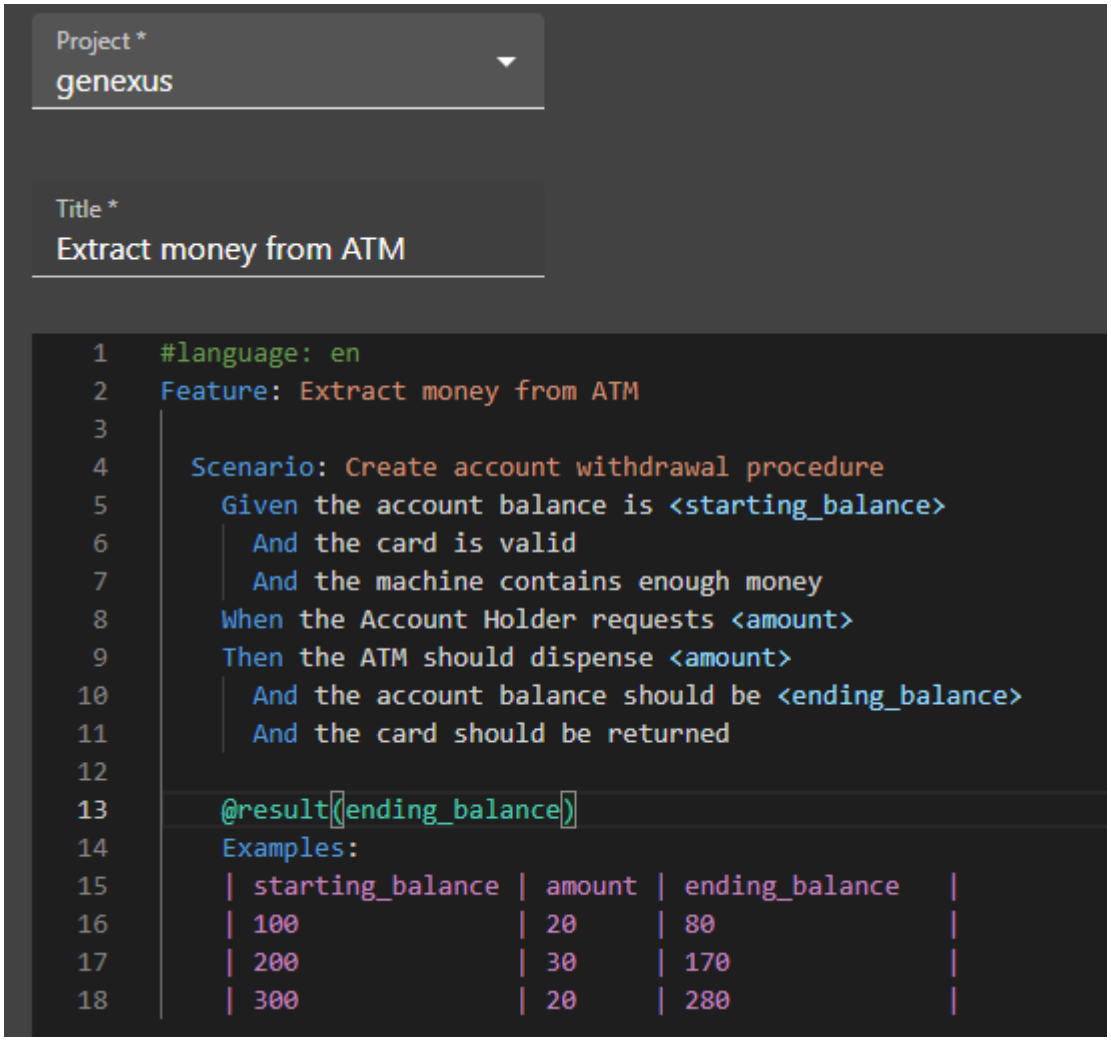


Figura 48: Ejemplo de una *feature*

En este ejemplo se ve el flujo de retirar plata de un cajero automático, en donde se define un escenario, y una tabla de ejemplos para este.

Para escribir una feature nuestro sistema nos brinda ayuda con templates ya generados, para no tener que escribir letra por letra aspectos que son igual para todos los *Features* que vayamos a generar.

Estos templates (técnicamente llamados "*snippets*") se pueden insertar en editar escribiendo su clave. Para generar el template de una *Feature*, tendremos que poner la clave "feature" y dar enter, en la siguiente imagen se ve lo que deberíamos ver al escribir dicha clave.

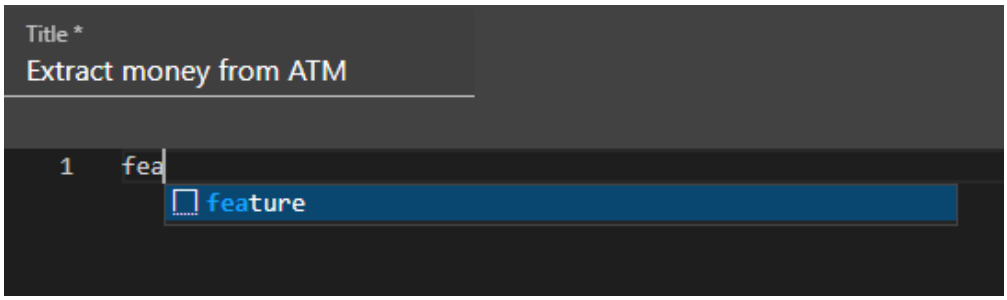


Figura 49: Ejemplo de una sugerencia en el editor del Cliente de Requerimientos

Nota: Nuestro Cliente de Requerimientos está completamente traducido al idioma Español, también así las claves usadas en el editor, por ejemplo para los *snippets*, por lo que, si usa el sistema en Español, algunos aspectos pueden ser diferentes. Por un tema de complejidad, este tutorial se hace en Inglés.

Esto nos dará como resultado, el siguiente *template*.

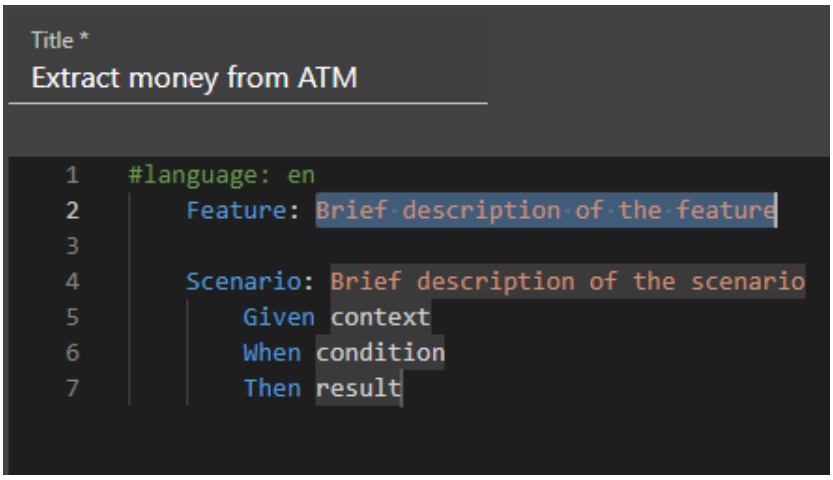


Figura 50: *Template* autogenerado por el editor

Si apretamos la tecla tabulador, podremos navegar por las diferentes secciones y cambiar al texto que queramos.

En este punto existe un detalle, el título de la *Feature* debe seguir cierto patrón, con esto luego se va a poder hacer la autogeneración del procedimiento que mencionamos, en caso de no cumplir con este, entonces este procedimiento no se puede autogenerar y se perderá una característica importante del sistema.

El patrón para el caso de los procedimientos es `"Create <Procedure_Name> procedure"`.

Dejemos esto como el ejemplo de arriba, debería estar de la siguiente forma:

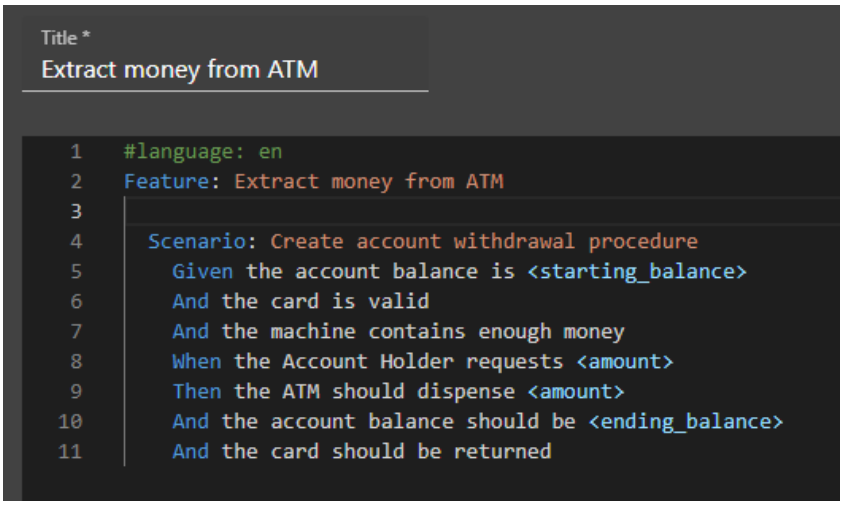


Figura 51: Ejemplo de una *Feature*

Como vemos, aún nos falta una sección, estos son los ejemplos. Para especificar ejemplos, se describen las variables de entrada y salida de un procedimiento. En este caso, tenemos dos variables de entrada, que son, el balance actual de la cuenta y la cantidad de valor que sacamos, y por último la variable de salida (el resultado de nuestro procedimiento), que es el balance resultante.

Para generar los ejemplos, podemos hacer uso de otro "*snippet*", que nos permite agregar tablas.

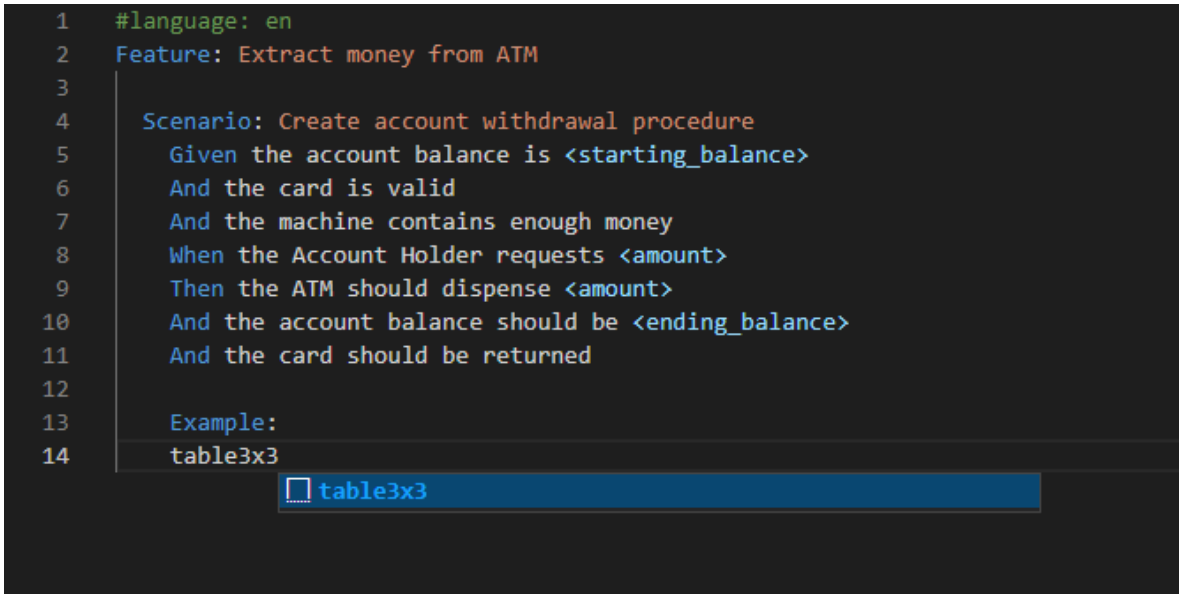


Figura 52: Ejemplo de una sugerencia para la creación de una tabla

El resultado debería quedar:

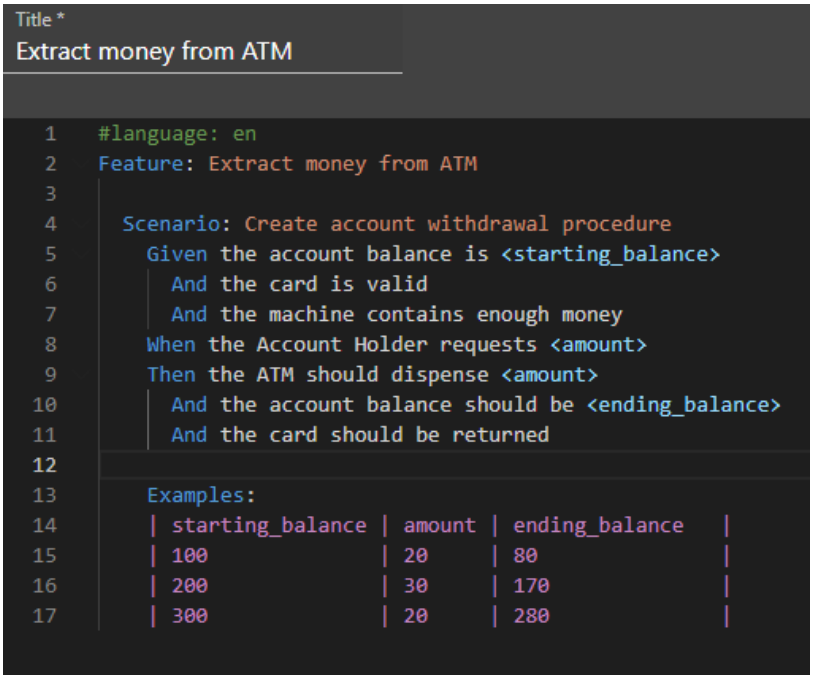


Figura 53: Tabla autogenerada por el editor

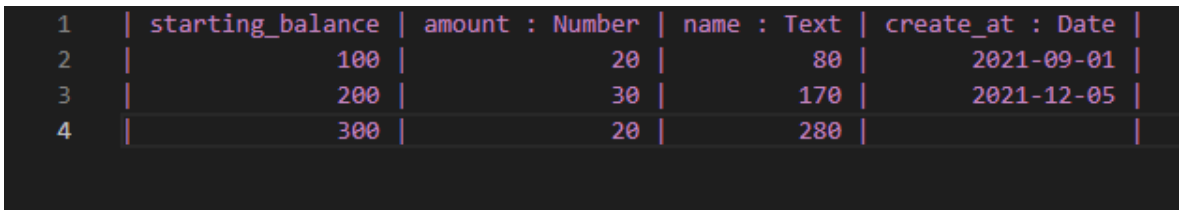
Ya especificamos los ejemplos, ya nos queda poco, solo especificar cual es la variable de salida de nuestro procedimiento. Esto realmente no es necesario, solo que si no lo hacemos se infiere que es la última columna de la tabla. Para definir la variable debemos colocar arriba de la tabla Examples lo siguiente “@result(<Nombre_Variable_Salida>)”.

Con esto tenemos un ejemplo de cómo escribir una *Feature*. En otra sección de este documento se detallan más detalles del editor y de la escritura de *Features*.

14.3.4 Definición de tipos en las tablas

En el Cliente de Requerimientos se pueden definir tablas, las cuales nos permitirán luego en GeneXus autogenerar objetos a partir de las propiedades de esta. Las propiedades en GeneXus deben tener un tipo, por ejemplo, una propiedad edad es de tipo numérica. Para obtener el tipo de las tablas existen dos mecanismos, la inferencia de tipos o la definición explícita de estos.

A modo de ejemplo se presenta la siguiente tabla (las columnas son a modo ejemplar, no tienen un sentido)



starting_balance	amount : Number	name : Text	create_at : Date
100	20	80	2021-09-01
200	30	170	2021-12-05
300	20	280	

Figura 54: Tabla con tipos indicados de manera explícita

En este ejemplo vemos cuatro variables, la primera, starting_balance, no especifica un tipo (el tipo se especifica con los dos puntos luego del nombre), se infiere revisando los datos de la columna, en este caso la columna starting_balance se infiere como un numérico. Mismo para la variable amount, solo que esta tiene el tipo definido de forma explícita. Esto se repite para la variable name y created_at. En la última columna también vemos que una fila está vacía, esto es posible, pero hay que tener en cuenta que afecta al resultado final, ya que la propiedad creada tendrá la característica de ser nullable.

Veamos otro ejemplo:

```
| starting_balance | amount : Number |
|              100 |              20 |
|              200 |              30 |
|             cero |            cero |
```

Figura 55: Tabla con un tipo de dato indicado de manera explícita

En este caso la variable starting_balance será inferida como Text, dado que este tipo es más general, es decir, todos los números coinciden en que son Text, pero los Text, como “cero”, no son de tipo Número. En el caso de la variable amount estamos en el mismo caso, pero al especificar el tipo en el *header*, este será el que termine quedando, por lo cual el tipo de esta propiedad es Number.

Tipos soportados para la inferencia y la definición explícita.

Se detallan los tipos que son soportados para la inferencia, ordenados de más específico a más general. En resumen, los tipos tienen una prioridad, la cual nos permite definir de forma específica el tipo de una columna de una tabla.

- **StringNumber**: Este tipo es el más específico, representa aquellos números que deben ser guardados con tipo string. O en día en el sistema este tipo se corresponde con aquellos números que empiezan con 0.

- **Decimal**: Este tipo representa a los valores números, ya sean negativos o positivos, enteros o decimales. Para definir este tipo de forma explícita debemos usar la clave “Number”.

- **Bool**: Este tipo representa los valores verdaderos o falsos, para definir una columna de forma explícita con este tipo debemos usar la clave “Boolean”.

- **Date**: Este tipo representa las fechas, para usarla para especificar el tipo de una columna se usa este mismo, ya que es explicativo en sí mismo.

- **Guid**: Un GUID es un “Identificador único global” (*Globally Unique Identifier*). Un ejemplo es “ac525197-4483-43ce-93f9-2d5178206b7e”, habitualmente se usa para identificar entidades de forma única. Para especificar una columna de forma explícita como este tipo debemos usar la clave “Identifier”.

- **String**: este es el tipo más genérico, cualquier valor que se ponga en una tabla se puede mapear a este. Para definir una columna de forma explícita con este tipo debemos usar la clave “Text”.

Estos son los tipos soportados, pero, por cómo está hecho el código, agregar la implementación de N lenguajes más es trivial (la explicación de esto no corresponde a esta sección).

14.3.5 Ayudas brindadas por el editor

Existen diversas ayudas brindadas por el editor al momento de escribir una *Feature*, dado que muchos aspectos se van a repetir en todas, no tiene sentido tener que escribir letra por letra siempre.

Snippets

Los *Snippets* son bloques de código pre hechos, estos nos permiten usar un *template* creado para aquel código que se repite en todas las *Features*.

Los existentes son:

- **Table**: Este nos permite ingresar una tabla, la cual puede tener diferentes tamaños

- **Scenario**: El cual nos inserta un escenario por defecto de Gherkin (con sus *Keywords*, *Give*, *When*, *Then*)

- **Feature**: El cual nos inserta una Feature con el formato Gherkin.

Autocompletado

Se brinda autocompletado para los tipos explícitos en las columnas de las tablas y también para los patrones de los títulos de los escenarios (ver siguiente sección).

Para el autocompletado de los títulos de los escenarios deberemos escribir “Scenario: ” y en este momento aparecerán las sugerencias.

Para el autocompletado de tipo explícito, deberemos poner “:” luego del nombre de la columna de la tabla, y en este momento aparecerán las sugerencias.

Formateo del código de la Feature

Si hacemos clic derecho sobre el editor, veremos una opción que nos permite formatear el contenido de la *Feature*. Esto es importante ya que se hace mucho más entendible para los revisores, sobre todo en el caso de las tablas, que al poner diferentes contenidos tiende a perder la forma.

14.3.6 Patrones de nombre de escenarios

El nombre de los objetos que se generan automáticamente se hace en función del título del escenario, siempre y cuando este cumpla un determinado formato.

Los formatos soportados para extraer el nombre del objeto en inglés son:

Transacción: Create <nombre_transacción> business model entity

Procedimiento: Create <nombre_procedimiento> procedure

Los formatos soportados para extraer el nombre del objeto en español son:

Transacción: Crear entidad de modelo de negocios <nombre_transacción>

Procedimiento: Crear procedimiento <nombre_procedimiento>

Donde nombre_procedimiento corresponde al nombre que tomará el objeto procedimiento y nombre_transacción corresponde al nombre que tomará la transacción respectivamente. En el caso que los nombres están compuestos por varias palabras separadas, los mismos se extraerán en *uppercamelcase*.

14.4 Manual de preparación de entorno de desarrollo local - Frontend

14.4.1 Introducción

En este manual explicaremos cómo correr el proyecto del Frontend del Cliente de Requerimientos en nuestra máquina local. A modo de resumen, este *Frontend* es un SPA (*Single Page Application*), desarrollado en React y haciendo uso del *Framework* React Admin.

14.4.2 Requisitos previos necesarios

- Tener instalado un administrador de paquetes, como puede ser npm o YARN, entre otros.
- Tener instalado Git.
- Tener permisos en el repositorio del proyecto.

14.4.3 Pasos a seguir

- Primero debemos bajar el código fuente del *Frontend*, para esto nos dirigimos al repositorio y bajamos el código, ya sea clonando el repositorio o descargando un zip.
- Instalar las dependencias indicadas en el package.json, para esto, basta con ejecutar el comando npm i o yarn
- Debemos agregar una variable de entorno que indique la URL de nuestro Backend:
    - Primero creamos, a raíz del proyecto, un archivo llamado .env.
    - En este archivo creado debemos escribir: REACT_APP_API_URL=<URL_Backend>
- Luego iniciamos la aplicación ejecutando npm start o yarn start y listo. Deberíamos tener nuestro proyecto levantado de forma local.

Nota: En caso de que al abrir el código con un editor este no compile, ejecutar el comando npm run lint --fix o yarn lint --fix

14.5 Manual de preparación de entorno de desarrollo local

14.5.1 Introducción

En este manual se explicará cómo armar un entorno de desarrollo local para la extensión GxReq. El levantar el código de la extensión en un ambiente de desarrollo nos permitirá ver cómo está construido el código, depurarlo, como forma de entenderlo mejor, y también editarlo, esto es sumamente importante, ya que este proyecto es una prueba de concepto que tiene que terminar de ser desarrollada. Para poder armar este ambiente es necesario tener herramientas instaladas, como bien se explican en la siguiente sección, y también, y no menos importante, un conocimiento de las herramientas a usar. Este manual está dirigido a un desarrollador, este deberá tener conocimiento en desarrollo orientado a objetos para entender la solución al completo.

14.5.2 Pasos a seguir

Antes de comenzar es necesario contar con las siguientes herramientas instaladas en la PC:

- Visual Studio 2017 o superior.
- GeneXus17
- GeneXus17PlatformSDK
- .NET Framework 4.7.1 o superior
- GeneXusServer17 (opcional para correr los tests con GxTest)

Una vez comprobada la instalación del stack de tecnologías necesario, proceder a clonar el repositorio https://github.com/GxReqOrt/GxReq.

Luego de esto, los pasos a seguir son los siguientes:

1. Abrir el archivo gx-extensions.bat con el editor de código de preferencia y modificar las variables de entorno:

    a. GX_PROGRAM_DIR: Por la ruta donde se encuentra instalado GeneXus 17

b. GX_SDK_DIR: Por la ruta donde se encuentra el SDK de GeneXus

c. GX_SERVER_DIR: Por la ruta donde se encuentra GeneXusServer17 (si es que este se encuentra en la misma PC, de lo contrario se deberá buscar otra manera de copiar las dll compiladas al directorio de instalación de GeneXus Server si corresponde)

2. Ejecutar la consola de Visual Studio Developer Command Prompt for VS 20XX en modo administrador.

3. Posicionar la consola en el directorio donde se encuentra el script gx-extensions.bat (dentro de la raíz del repositorio)

4. Ejecutar el script gx-extensions.bat.

Si los pasos anteriores fueron seguidos correctamente se lanzará el IDE de Visual Studio. En este momento se está en condiciones de comenzar a desarrollar.

Para debuggear la solución, simplemente elegir GxReq. UI como proyecto de inicio y presionar F5. Si las variables de entorno fueron configuradas correctamente se lanzará el IDE de GeneXus 17 y se copiaran todos los archivos necesarios al directorio de instalación de GeneXus17 y GeneXusServer17 automáticamente durante el proceso de *build*.

14.6 Configuración de la extensión con la Knowledge Base

14.6.1 Introducción

En este manual, se detalla cómo configurar nuestra Knowledge Base (Base de conocimiento), de forma que nos conectemos con nuestro proyecto del Cliente de Requerimientos y sincronicemos las *Features* (Requerimientos) que estén disponible para esto. Se resalta, aunque puede sonar redundante, que se debe contar con GeneXus, versión 17, instalado en nuestra PC, se deberá contar con un mínimo conocimiento de este y también se deben conocer los principales conceptos del Cliente de Requerimientos.

14.6.2 Pasos a seguir

Para la sincronización de las *Features* creadas a través del Cliente de Requerimientos https://gxreq-portal.web.app/ con una Knowledge Base local, se deben seguir los siguientes pasos:

1. Posicionarse en el directorio de la KB a la que se quiere hacer *attach* del proyecto.
2. Crear un archivo con el siguiente nombre y extensión gxreq.config.
3. Abrir el archivo de configuración creado con el editor de preferencia.
4. Copiar el siguiente contenido al archivo:

```
{
  "BaseUrl": "https://gxreq-portal.herokuapp.com",
  "Email": "<someone@example.com>",
  "Password": "<SuperSecretPassword>",
  "ProjectId": "<1>"
}
```

Donde:

- BaseUrl: Apunta al *backend* del Cliente de Requerimientos.
- Email: Es el email del usuario colaborador o dueño del proyecto al que se quiere hacer *attach*.

- Password: Es la contraseña del usuario colaborador o dueño del proyecto al que se quiere hacer *attach*.
- ProjectId: Es el id correlativo del proyecto, que se puede ver en los datos del mismo dentro del Cliente de Requerimientos.

En este momento se está en condiciones de ir al menú GxReq y darle a la opción "*Fetch New Features*", esto creará las *Features* commiteadas en el Cliente de Requerimientos y un archivo gxreq.lock para sincronización.

14.7 Manual de deploy Backend – Heroku

14.7.1 Introducción

En este manual explicaremos como realizar el *deploy* del *Backend* del Cliente de Requerimientos. A modo de resumen, es una REST API desarrollada en DotNet Core y haciendo uso del *Framework* JSON API DotNetCore (https://www.jsonapi.net/).

El resultado de seguir los pasos aquí presentados es que el repositorio queda vinculado con Heroku para hacer CI/CD, es decir, automáticamente al hacer push a la branch configurada se deploya el código nuevo al dyno.

14.7.2 Pasos a seguir

- Crearse una cuenta y una aplicación en Heroku (https://heroku.com). En caso de ser necesario se puede seguir las instrucciones de este videotutorial (https://youtu.be/GDcExsC31hQ?t=225).

- Ir a la pestaña de "*Settings*" dentro de la aplicación y configurar el "*buildpack*" para usar https://buildpack-registry.s3.amazonaws.com/buildpacks/jincod/dotnetcore.tgz.

- Yendo a la pestaña "*Deploy*" buscar la última sección de "*Deployment Method*" y elegir *deploy* manual (https://devcenter.heroku.com/articles/git) o automático (https://devcenter.heroku.com/articles/github-integration).

14.8 Manual de preparación de entorno de desarrollo local – Backend

14.8.1 Introducción

A modo de introducción, esta solución fue construida en .NET 5.0 y haciendo uso del *Framework* JsonApiDotnetCore, que nos facilita el desarrollo. Se usó PostgresSQL como base de datos, y también se usó Sengrid para enviar emails con notificaciones. A continuación, se detallan los requisitos y pasos que permiten levantar la solución en un ambiente local.

14.8.2 Requisitos previos necesarios

- Visual Studio 2017 o superior.
- .Net 5.0
- PostgreSQL

Luego, debemos descargar el código del repositorio (https://github.com/GxReqOrt/GxReq.Portal.Api).

14.8.3 Pasos a seguir

1. Primero debemos crear el archivo appsettings.json, en el proyecto GxReq.Portal.Api, con el siguiente formato

```
{
        "ConnectionStrings": {
        "DefaultConnection": <Database_Connection_String>

},
        "AllowedHosts": "*",
        "Jwt": {
        "SigningKey": <Signing_Key>

},
        "SendGrid": {
        "ApiKey": <Sengrid_Api_Key>,
        "AdminEmail": <Sengrid_Admin_Email>
```

```
        }
}
```

2. Luego debemos aplicar las migraciones de la base de datos, para esto nos dirigimos al menú Tools > NuGet Package Manager > Package Manager Console y definimos como Default Project al proyecto Gxq.Portal.Persistence y ejecutamos el comando Update-Database, esto creará la base de datos, y aplicará las migraciones.

3. Luego nos aseguramos de definir el proyecto GxReq.Portal.Api como el proyecto de inicio de nuestra solución.

4. Por último, presionamos F5 para ejecutar la solución, en este punto, esperamos y debemos ver que se abre Swagger, con las operaciones de nuestra API.

14.9 Manual de deploy Frontend - Firebase

En este manual explicaremos como realizar el *deploy* del Frontend del Cliente de Requerimientos. A modo de resumen, este *Frontend* es una SPA (*Single Page Application*), desarrollado en React y haciendo uso del *Framework* React Admin (https://marmelab.com/react-admin/).

En nuestro día a día, contamos con CI/CD (Integración y despliegue continuo), estos *pipelines* fueron configurados haciendo uso de Github Actions, dado que nuestro repositorio de código es Github. Al momento de realizar un PR (*Pull Request*) se ejecutan el *build* de la solución y las pruebas automatizadas, validando así el nuevo código. A su vez, cuando se realiza un *push* sobre la rama master, entonces se ejecuta de forma automática el *deploy* en Firebase, haciendo uso del servicio Firebase Hosting (https://firebase.google.com/products/hosting).

Elegimos Firebase por dos razones, la primera, que nos resultaba sencillo realizar la configuración para el *deploy* y el *deploy* automático, dado que en la facultad cursamos una materia en donde aprendimos a realizar esto, la segunda razón es que este servicio es gratuito, y a su vez no tiene limitaciones, al menos para lo que nosotros precisamos, por lo que es perfecto. Obviamente, este *Frontend* se podría deployar en cualquier otro servicio de otros proveedores, como puedo ser, a modo de ejemplo, Amazon S3.

14.9.1 Pasos para realizar el deploy en un proyecto Firebase

1. Lo primero que debemos hacer, es tener una cuenta en Firebase, para eso nos dirigimos al siguiente *link* (https://console.firebase.google.com/u/0/) y nos registramos.

2. Una vez tengamos nuestra cuenta de Firebase, entonces debemos crear un proyecto, para esto, ver Anexo 1.

3. Con el proyecto Firebase creado, procederemos a descargar el código de nuestro repositorio (se deberá tener permisos de acceso antes, dado que el repositorio es privado).

4. Una vez descargado el código (de la rama master), debemos instalar las dependencias, que están configuradas en nuestro package.json. Para esto, ejecutar “yarn” o “npm i”.

5. Con las dependencias instaladas, debemos configurar las variables de entorno

6. Luego de configurar las variables de entorno, deberemos realizar el *build* del código, para esto debemos ejecutar el comando “yarn build” (o “npm run build”)

7. Luego, en la raíz del proyecto, veremos el archivo .firebaserc, deberemos editar este, cambiando el proyecto por el que creamos anteriormente.

8. Con el archivo configurado deberemos ejecutar los siguientes comandos, para esto deberemos estar ubicados (en la consola), en la raíz del proyecto

    a. “yarn global add firebase-tools” o “npm install -g firebase-tools”

    b. “firebase login”

    c. “firebase deploy --only hosting”

9. ¡Listo! Nuestro Frontend está deployado. En la pestaña hosting de nuestro proyecto Firebase veremos los deploys realizados.

Tip 1: Si al descargar el código, este no compila, entonces debemos ejecutar el comando “yarn lint --fix” o “npm run lint --fix”. Esto puede pasar en Windows por el carácter de fin de línea.

Tip 2: Cada vez que realizamos un nuevo deploy, es recomendable (y previene de errores) cargar la URL de nuestro proyecto borrando la caché, para esto podemos hacer Ctrl + F5.

14.9.2 Anexos

14.9.2.1 Anexo 1 - Creación de proyecto Firebase

Accedemos al *Link* (https://console.firebase.google.com/), y damos clic en “Agregar proyecto” y le damos un nombre a este

Figura 56: Pantalla de creación de proyecto de Firebase

Definimos las siguientes configuraciones y damos crear proyecto:

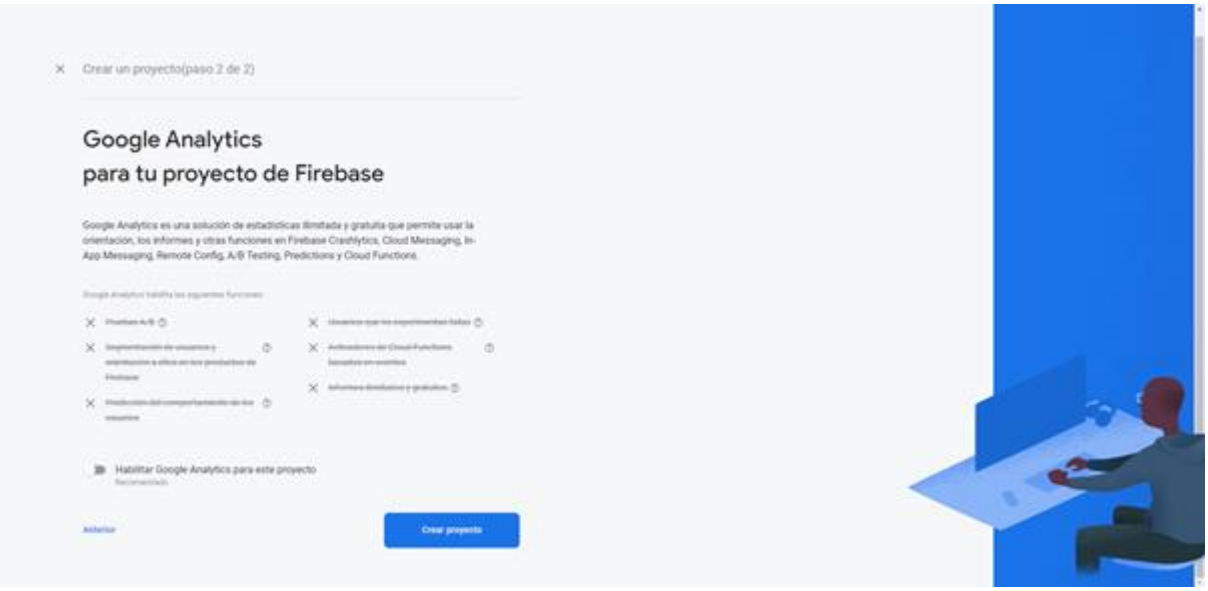


Figura 57: Pantalla de creación de proyecto de Firebase

Esperamos a que la creación de este termine y listo, con eso listo, tenemos nuestro proyecto creado.

Luego debemos registrar nuestra aplicación, haciendo clic en “Agregar App”

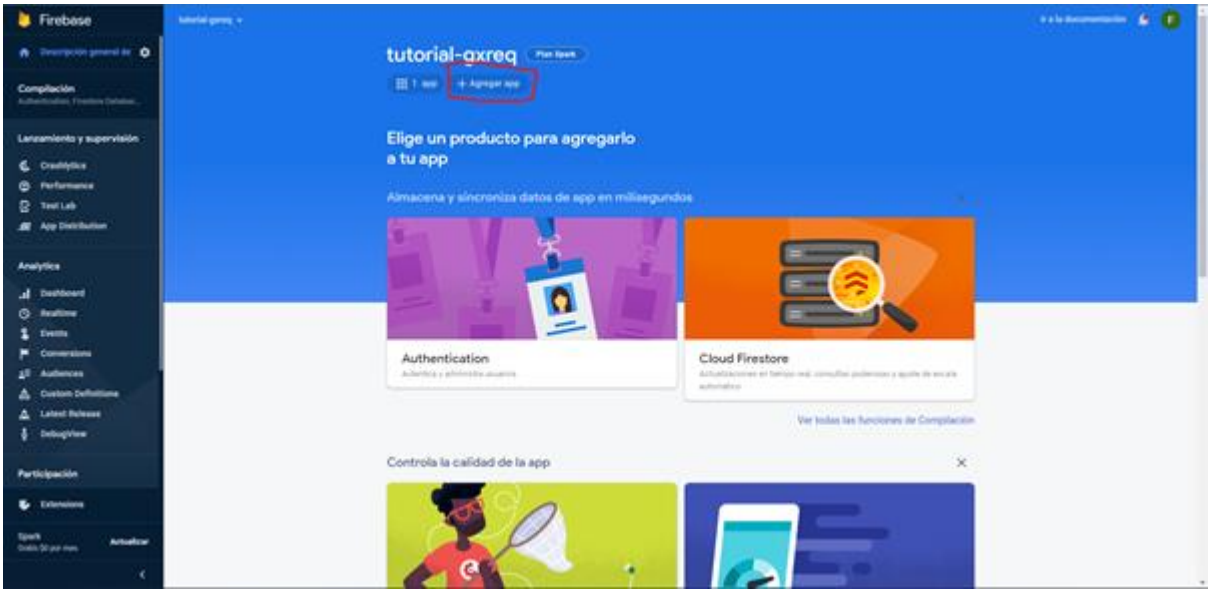


Figura 58: *Dashboard* de Firebase

Y en este punto, le damos a “Ir a la Consola”. Listo, tenemos configurado nuestro proyecto.

14.9.2.2 Anexo 2 - Configuración de variables de entorno

Debemos configurar las variables de entorno de nuestro proyecto, para esto debemos crear un archivo .env en la raíz del proyecto y debemos definir las siguientes variables de entorno:

```
REACT_APP_API_URL=<Url del deploy del Backend>
REACT_APP_FIREBASE_API_KEY=<Api key de nuestro proyecto Firebase>
REACT_APP_FIREBASE_AUTH_DOMAIN=<Auth domain>
REACT_APP_FIREBASE_PROJECT_ID=<Id del proyecto>
REACT_APP_FIREBASE_STORAGE_BUCKET=<Storage bucket>
REACT_APP_FIREBASE_MESSAGING_SENDER_ID=<Messaging sender id>
REACT_APP_FIREBASE_APP_ID=<App Id>
```

Para completar estas variables de entorno, nos dirigimos a nuestras Apps y vemos la configuración, como se muestra a continuación y copiamos estos valores (se tachan por seguridad)

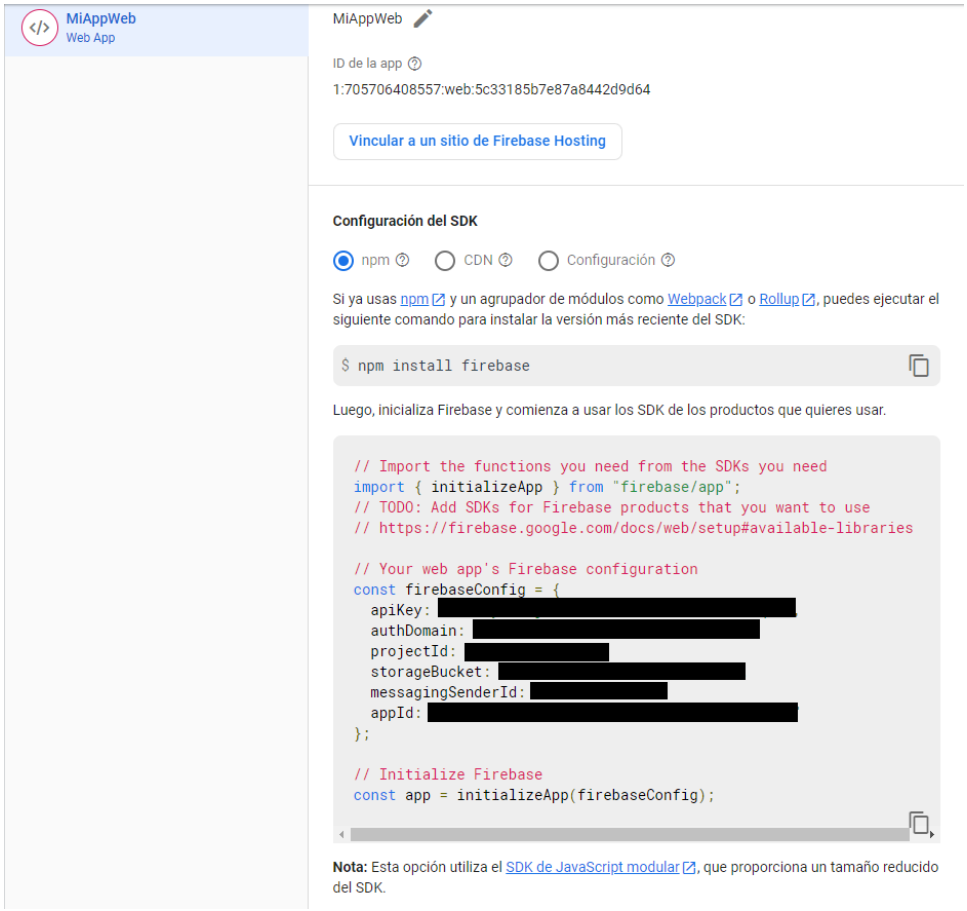


Figura 59: Datos del proyecto Firebase

14.10 Mockups

Se crean prototipos de baja fidelidad para el Cliente de Requerimientos y la extensión GeneXus.

14.10.1 Cliente de Requerimientos

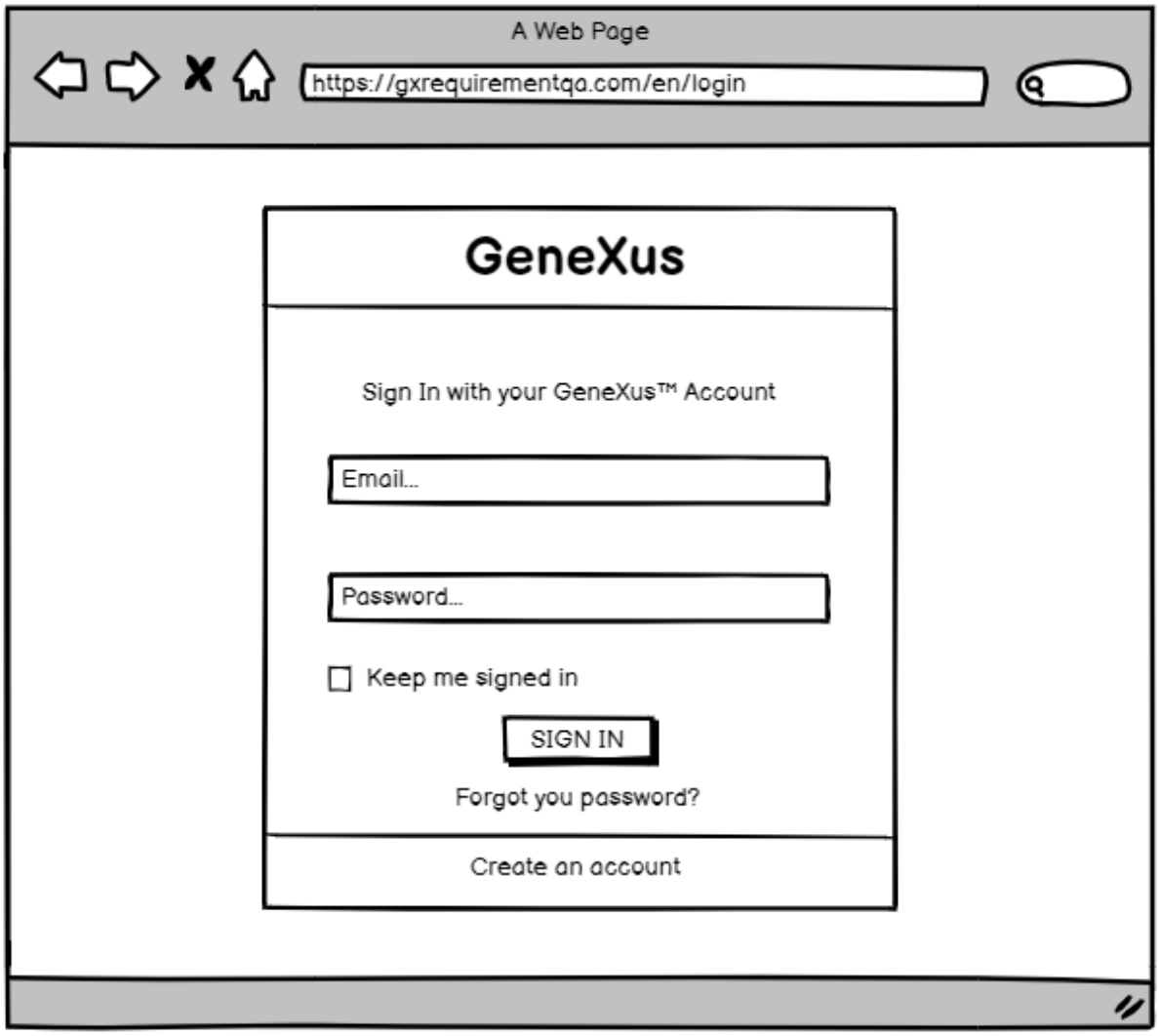


Figura 60: *Mockup Login* extensión GeneXus

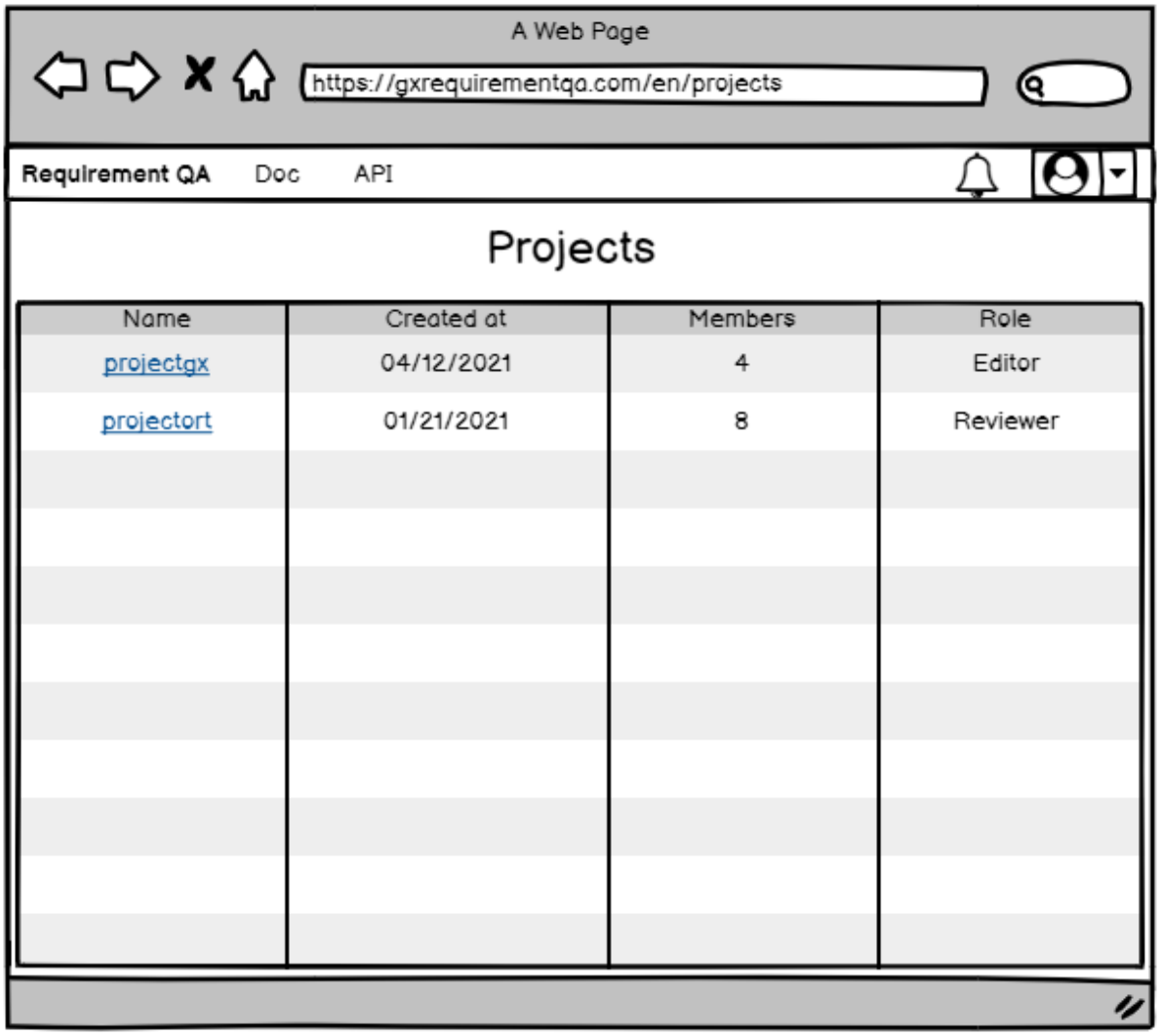


Figura 61: *Mockup* vista de proyectos

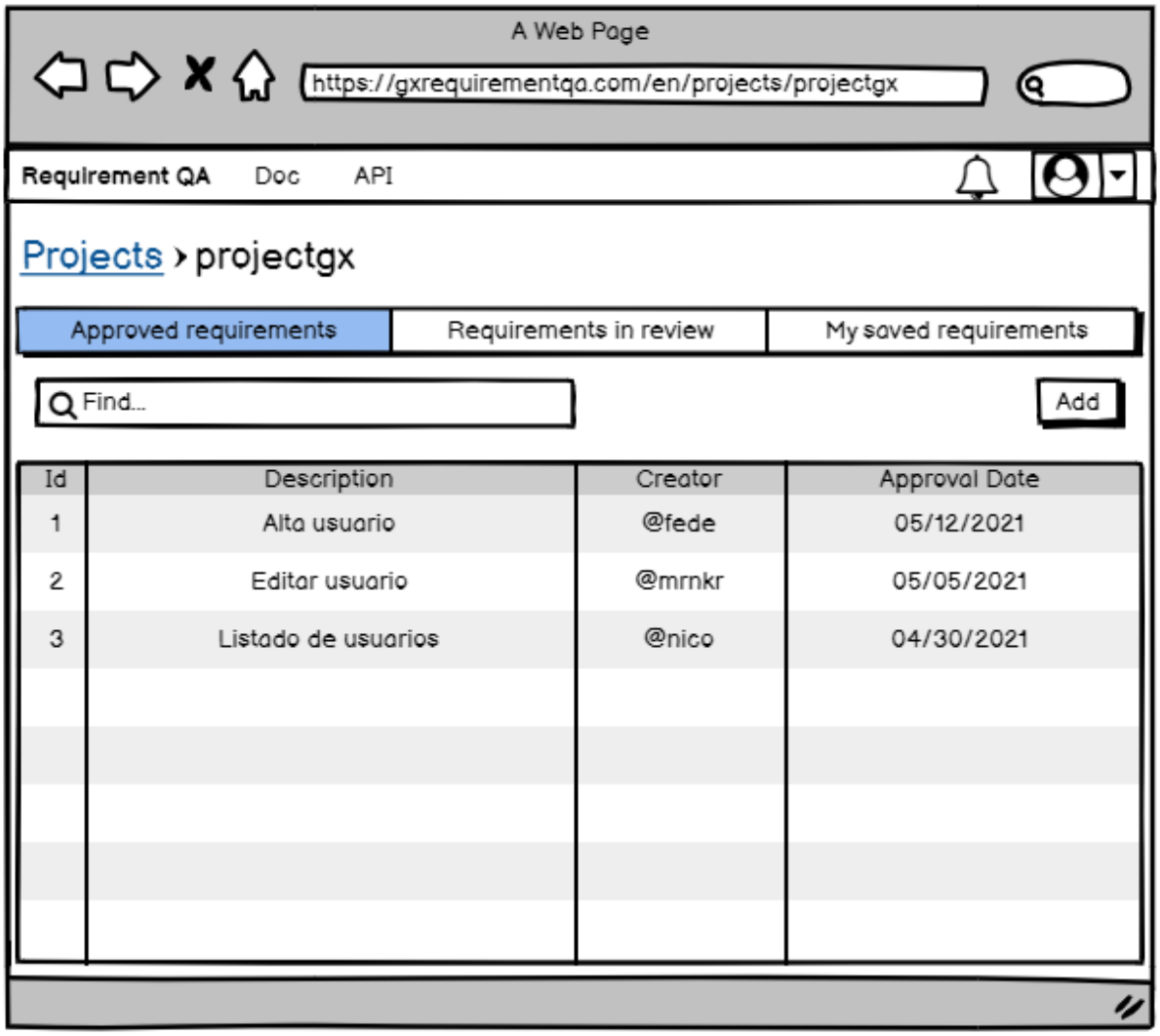


Figura 62: *Mockup* vista de requerimientos aprobados

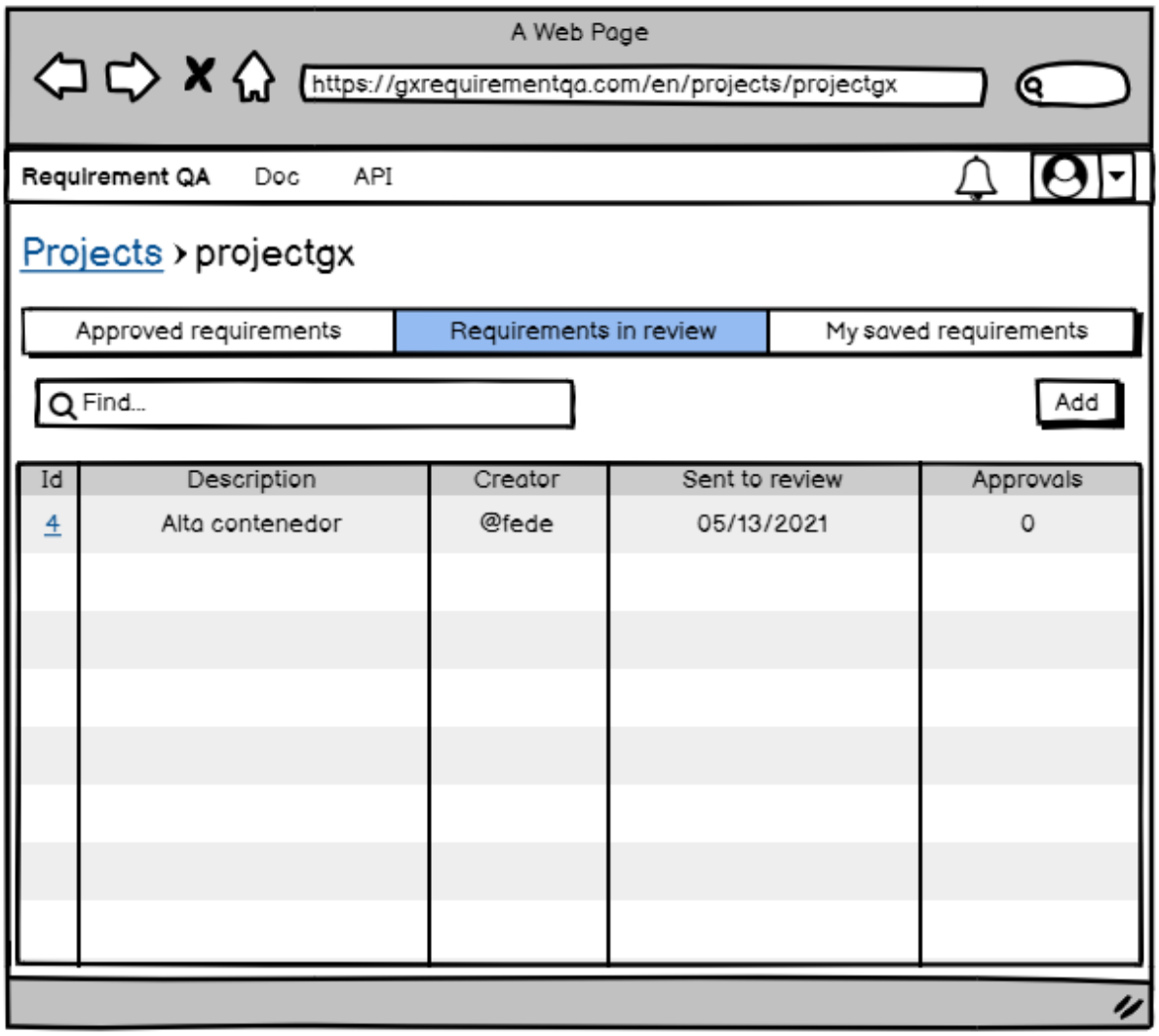


Figura 63: *Mockup* vista de requerimientos en revisión

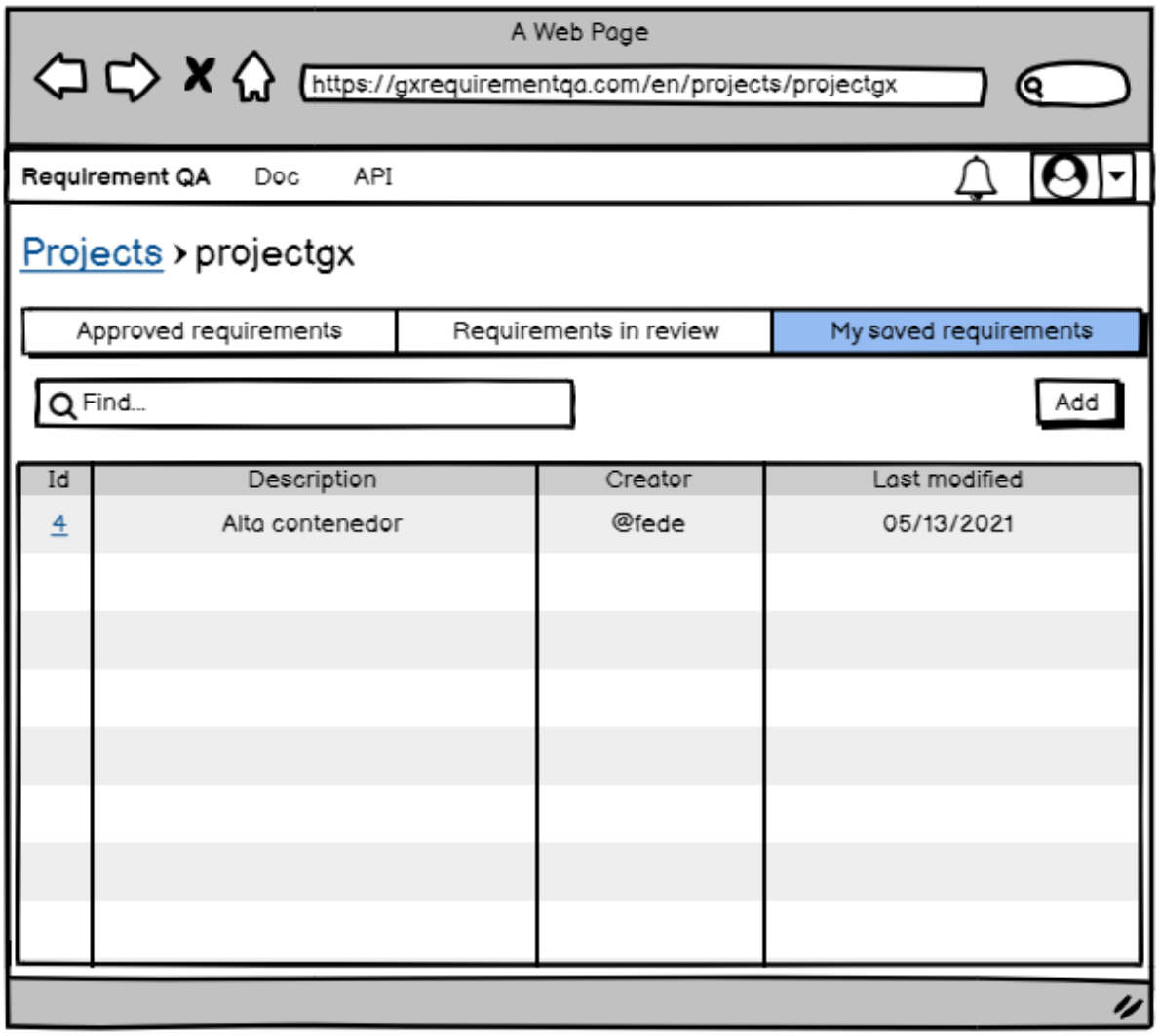


Figura 64: *Mockup* vista de requerimientos del usuario

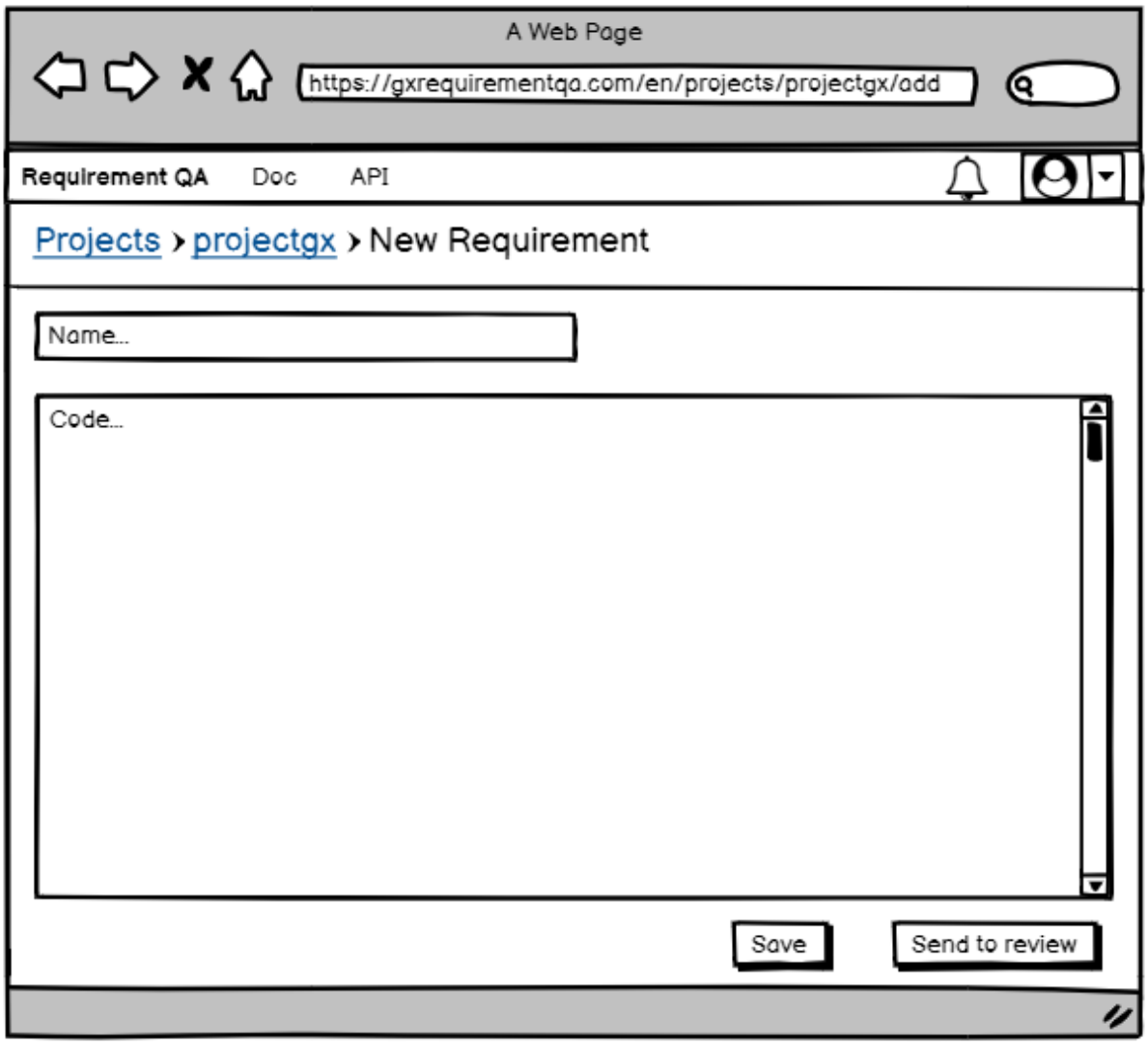


Figura 65: *Mockup* vista de creación de nuevo requerimiento

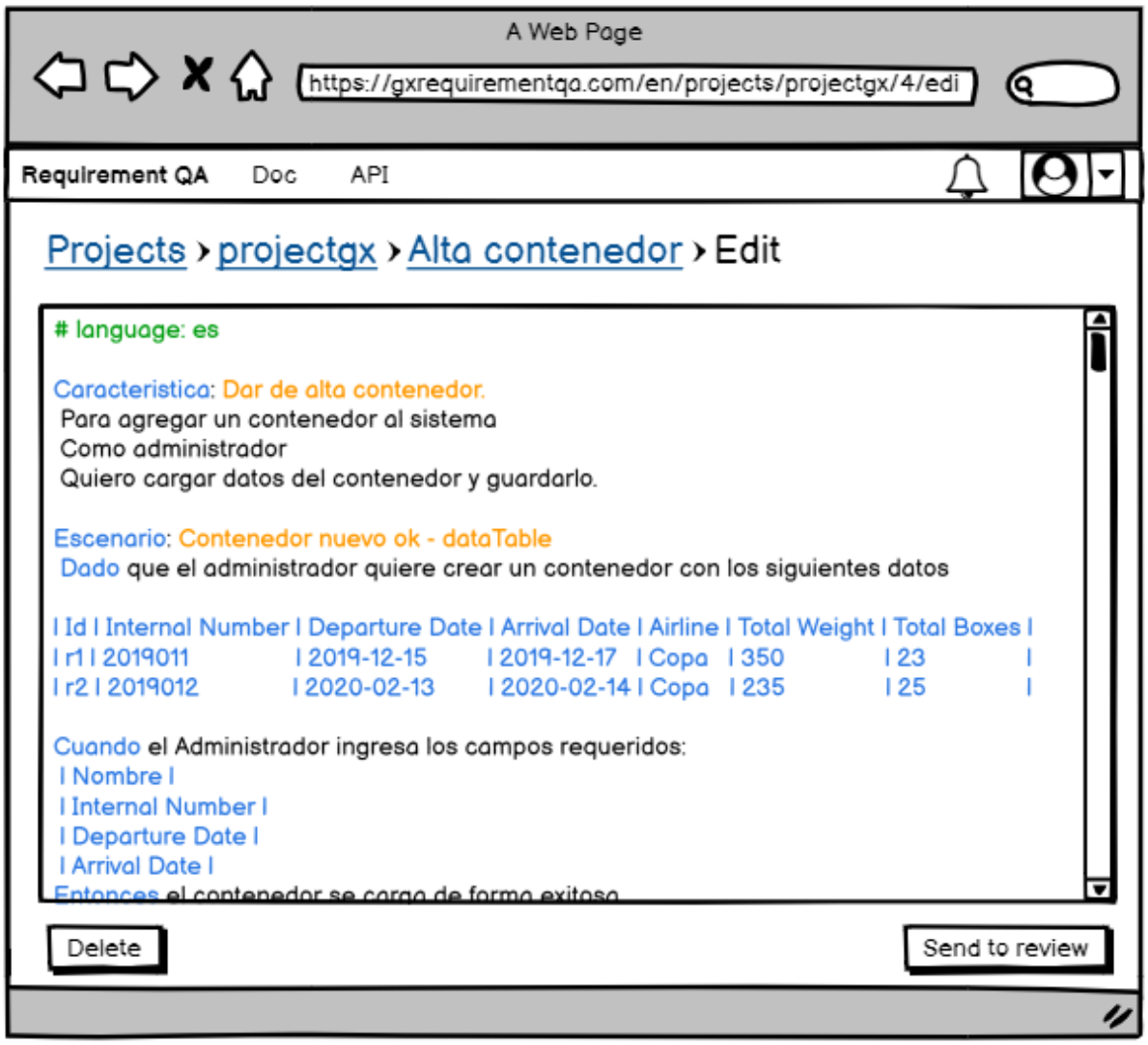


Figura 66: *Mockup* vista edición de requerimiento

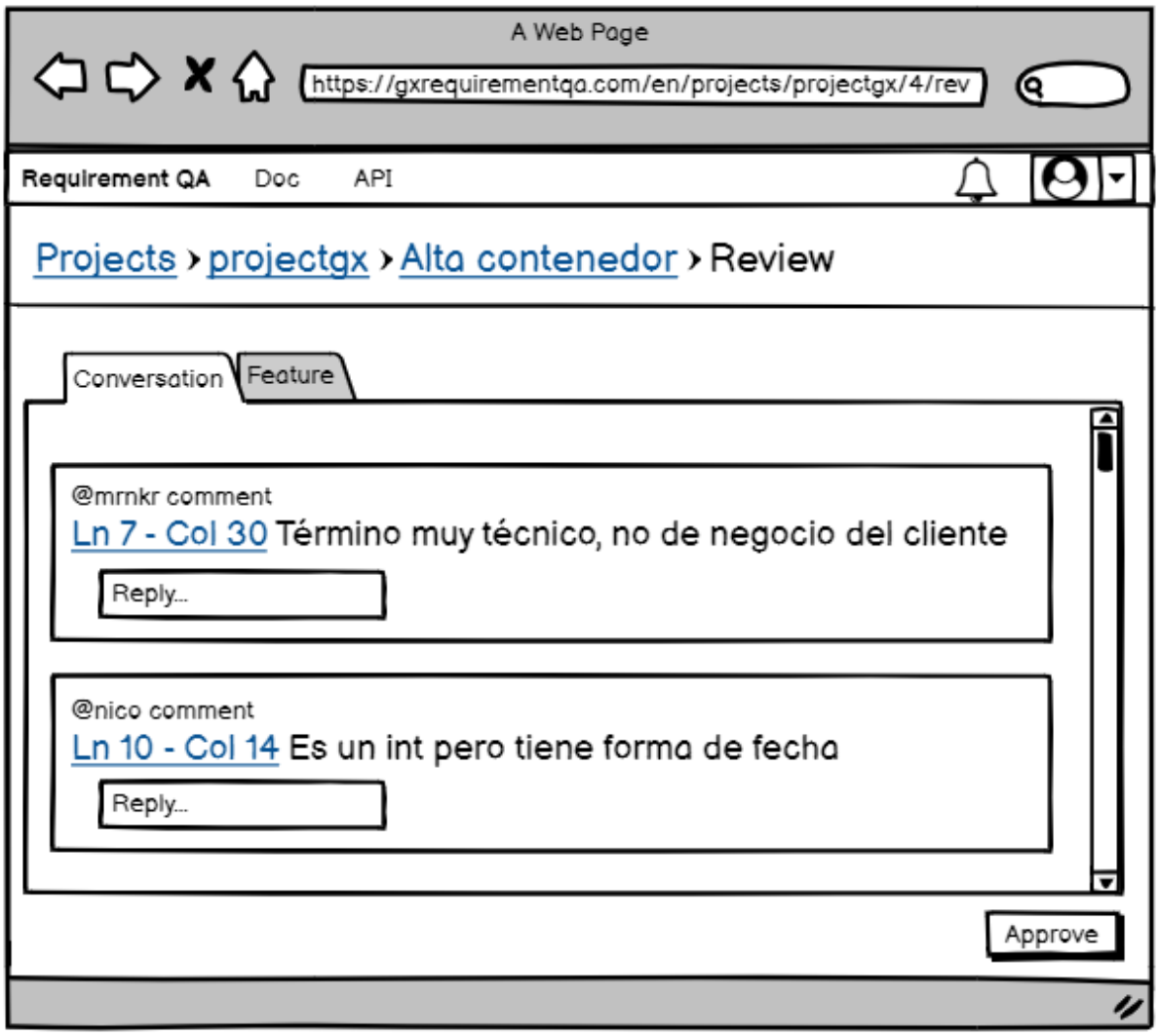


Figura 67: *Mockup* vista comentarios de un requerimiento

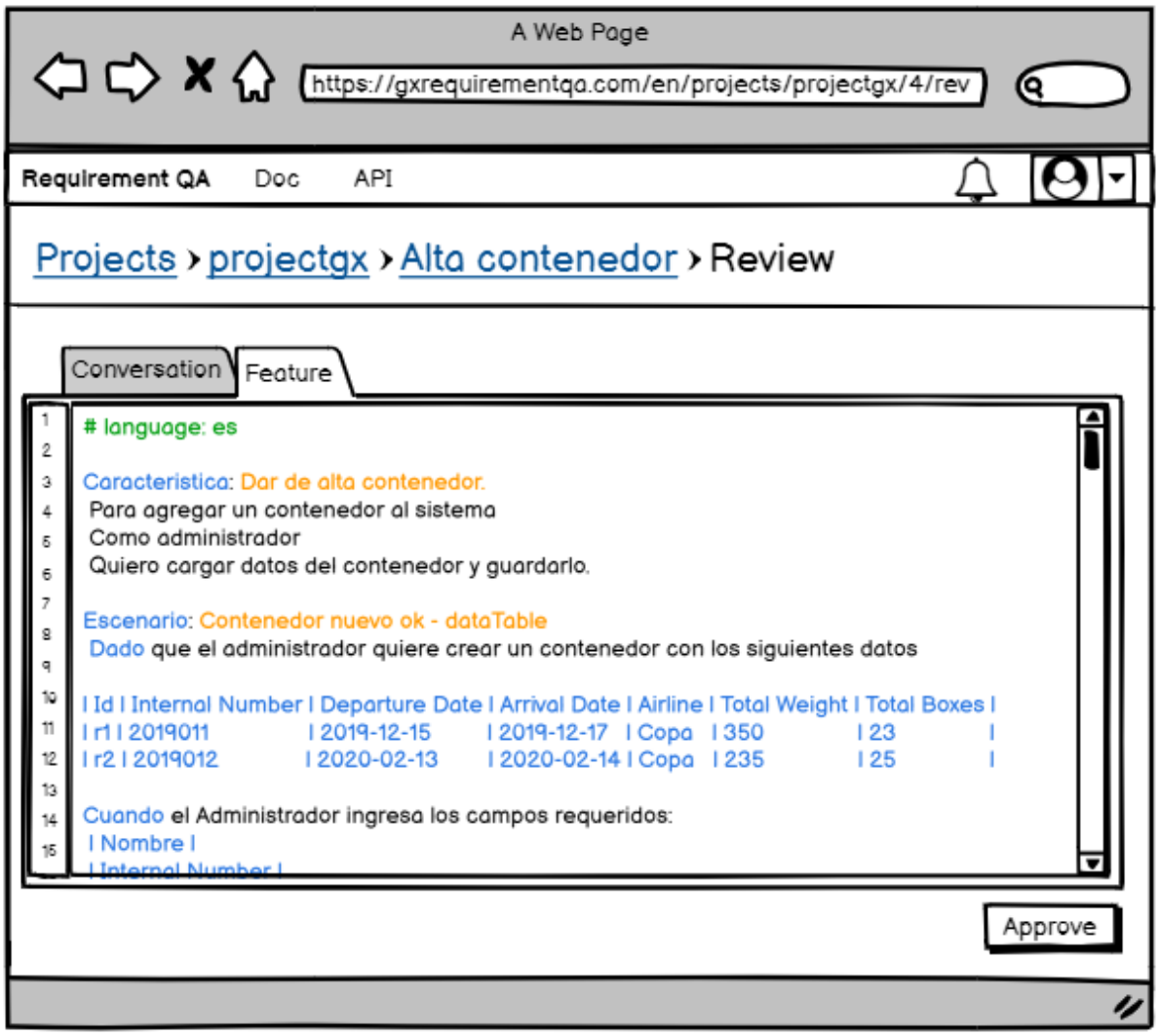


Figura 68: *Mockup* vista de requerimiento en revisión

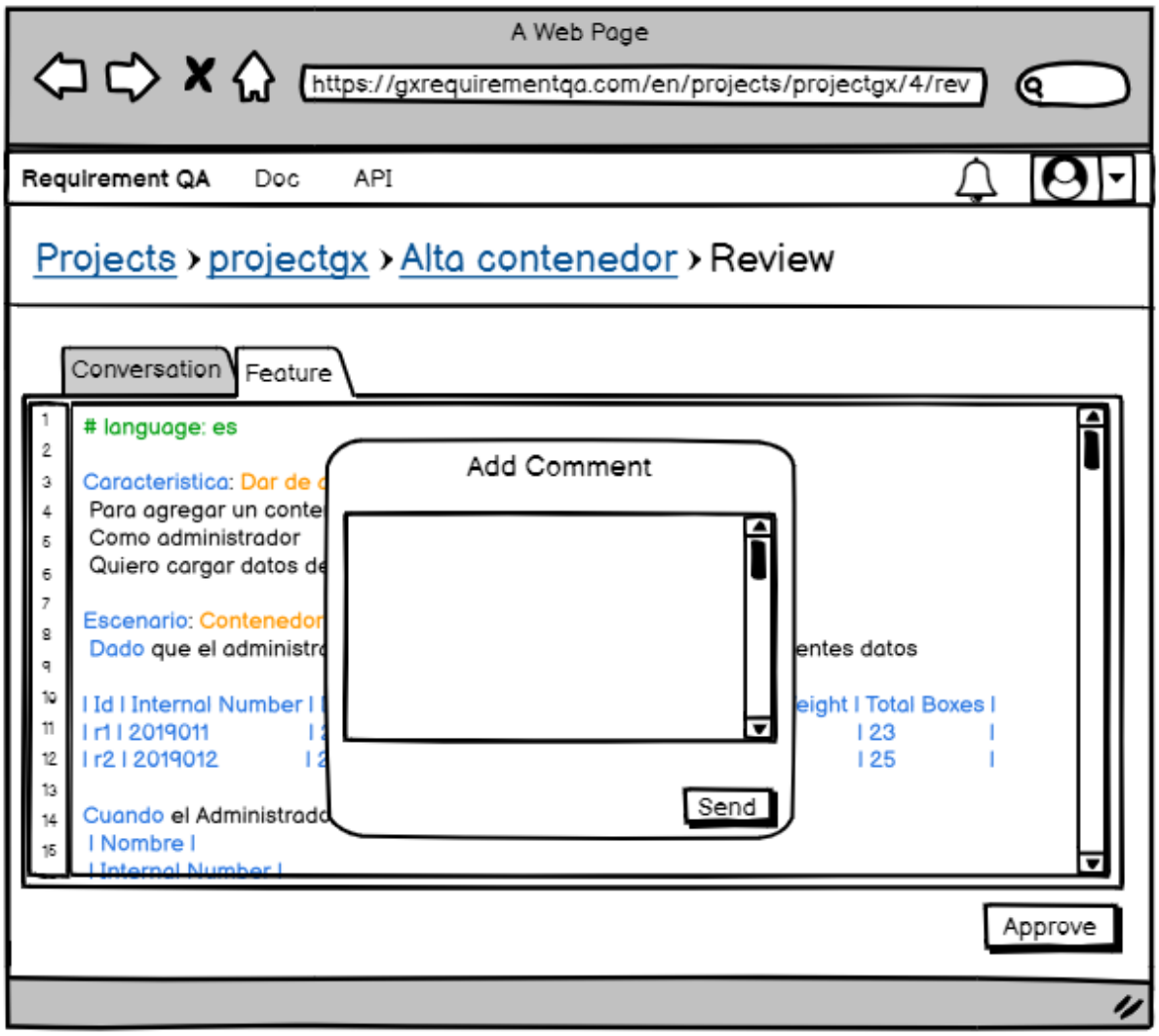


Figura 69: *Mockup* vista de agregado de comentario a requerimiento

14.10.2 Extensión GeneXus

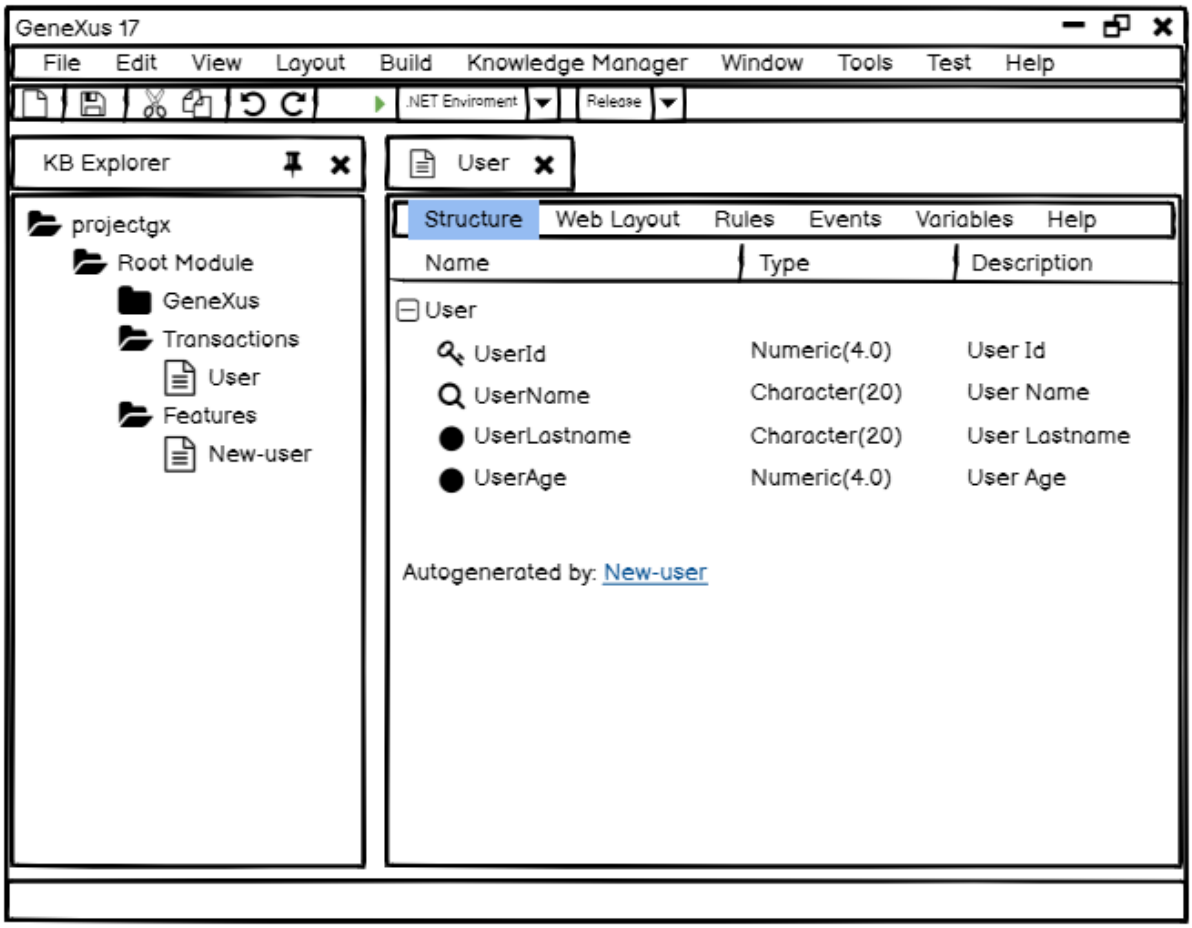


Figura 70: *Mockup* IDE vista de transacción GeneXus

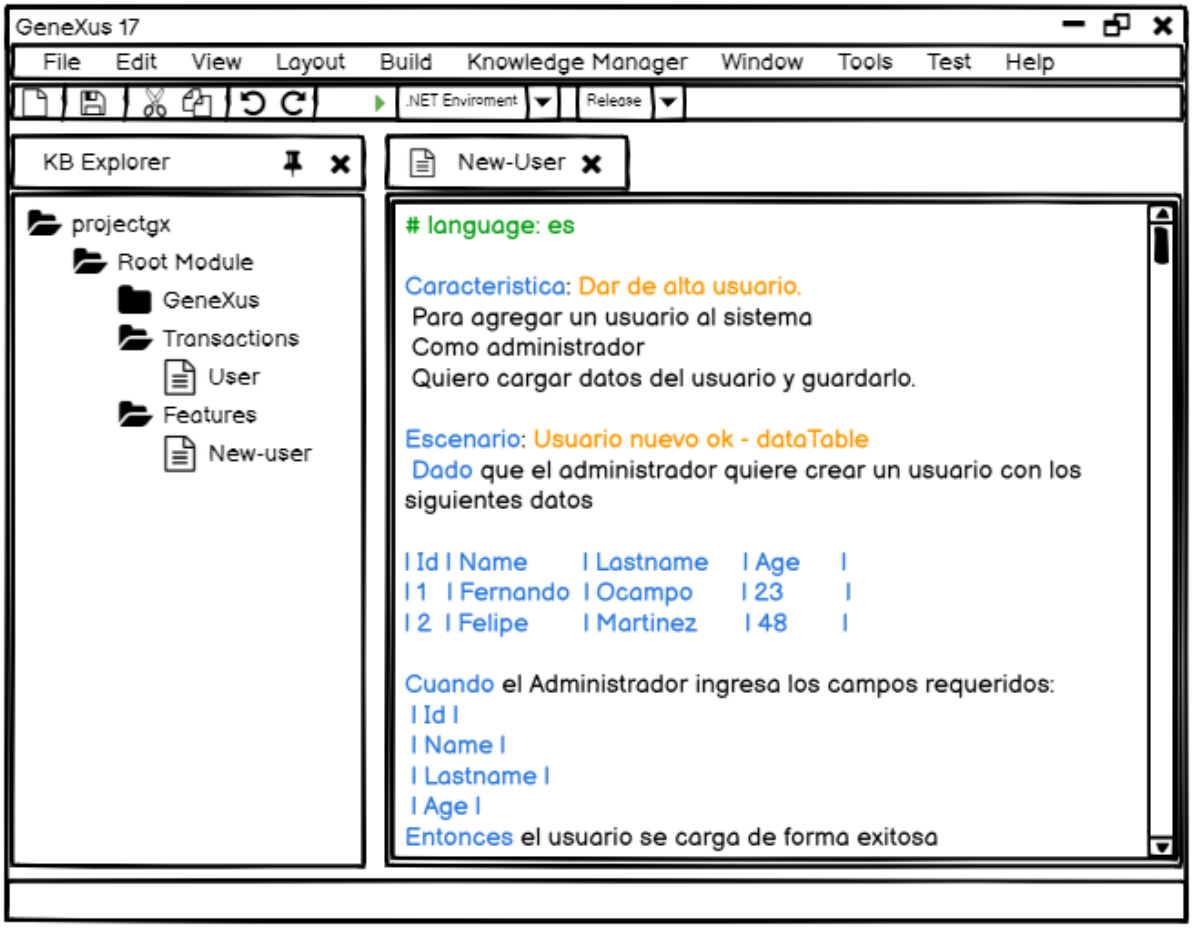


Figura 71: *Mockup* vista de Requerimiento en IDE GeneXus

14.11 Encuesta

14.11.1 GQM

14.11.1.1 Objetivo: Validar que el problema existe

Preguntas

1. ¿Necesita mantener un registro de la trazabilidad de sus requerimientos en GeneXus?
2. ¿Actualmente maneja la trazabilidad de los requerimientos?
3. ¿Requiere de herramientas adicionales a GeneXus para mantener la trazabilidad?
4. En caso de responder que sí a la 3, puntúa de 1-10 que tanto esfuerzo se le dedica a mantener esta trazabilidad

Métricas

1. Esperamos que exista al menos un 50% de encuestados que indican que necesitan mantener un registro de la trazabilidad y actualmente no la manejan
2. La relación entre las respuestas positivas de la 2 y las respuestas mayores a 7 en la pregunta 4. Esperamos una relación mayor a 60%

14.11.1.2 Objetivo: Identificar clientes

Preguntas

1. ¿Necesita mantener un registro de la trazabilidad de sus requerimientos en GeneXus?

Métricas

1. La relación entre la totalidad de encuestados y respondieron de forma positiva la pregunta 1, sea mayor al 60% (Esto porque si esto se cumple, entonces elegimos el público correcto)

14.11.1.3 Objetivo: Validar requerimientos

Preguntas

1. ¿Cómo maneja la trazabilidad de los requerimientos cuando desarrolla?
2. ¿Utiliza las herramientas como GXServer o GXTest?
3. ¿Cómo impacta en la trazabilidad de sus proyectos GXServer o GXTest?
4. ¿Maneja un criterio de aceptación para sus requerimientos?
5. Del 1 al 10, qué tan seguido le pasa que se reabren *tickets* ya que el criterio de aceptación fue mal marcado como cumplido
6. ¿En su día a día experimenta problemas de comunicación entre los desarrolladores y quienes definen los requerimientos?
7. ¿Registra de manera escrita los requerimientos?
8. Sí sí, ¿sigue algún patrón o *template* para escribirlos?

Métricas

1. Esperamos que al menos el 60% de los encuestados use GXServer o GXTest (Esto significa para nosotros que podemos basar nuestra solución en estas herramientas ya usadas)
2. Esperamos que al menos el 60% responda positivamente la pregunta 6 (Esto significa para nosotros que necesitan Gherkin o similar)

3. La relación entre la cantidad de encuestados y las respuestas mayores a 5 en la pregunta número 5 es mayor a 60%. (Nos indica que son gente que se puede beneficiar de que sus criterios de aceptación se puedan comprobar de manera empírica)

4. Relación entre respuestas positivas en la pregunta 4 y respuestas menores a 5 en la pregunta 5 (Nos indica los encuestados que ya manejan criterios de aceptación y que ya tienen buenos resultados en sentido de reapertura)

5. Si un porcentaje mayor a 60% de los encuestados registra de manera escrita los requerimientos quiere decir que no significarán un gran cambio para ellos adoptar esta herramienta, es decir, menor resistencia de los usuarios.

6. Si un buen porcentaje (más de 60%) de los que registran de forma escrita los requerimientos responde que sí utiliza un *template* quiere decir que estarán listos para adaptar Gherkin, si es que no es Gherkin lo que utilizan ya. Puede darnos material para hacer uso de *snippets* en Monaco.

Glosario

Trazabilidad: La trazabilidad está compuesta por procesos prefijados que se llevan a cabo para determinar los diversos pasos que recorre un producto (en este caso requerimiento), desde su nacimiento hasta su ubicación actual en la cadena de abasto.

14.11.2 Resultados

Se presentan los resultados de la encuesta realizada a usuarios GeneXus.

¿Necesita mantener un registro de la trazabilidad de sus requerimientos en GeneXus?
Entendiendo por trazabilidad a la asociación de un...ctos generados durante el desarrollo de software.
31 respuestas

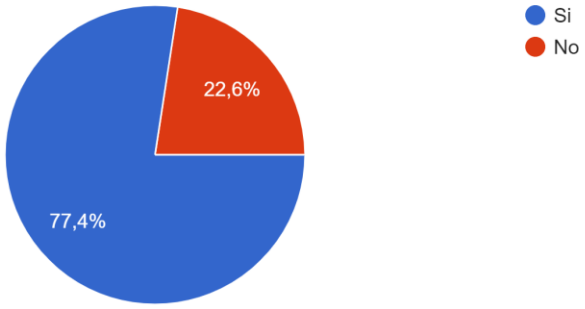


Figura 72: Respuestas pregunta 1 encuesta

¿Actualmente maneja la trazabilidad de los requerimientos?
31 respuestas

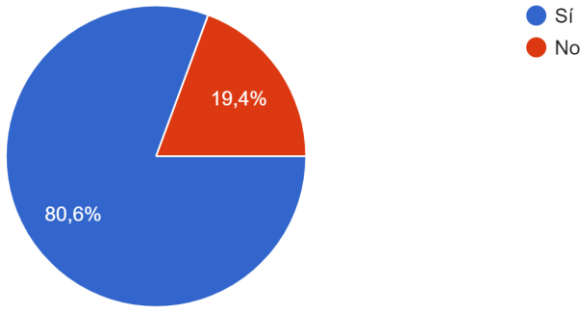


Figura 73: Respuestas pregunta 2 encuesta

En caso de responder que Sí a la pregunta "¿Actualmente maneja la trazabilidad de los requerimientos? ", ¿Requiere de herramientas adicionales a GeneXus para mantener la trazabilidad?

27 respuestas

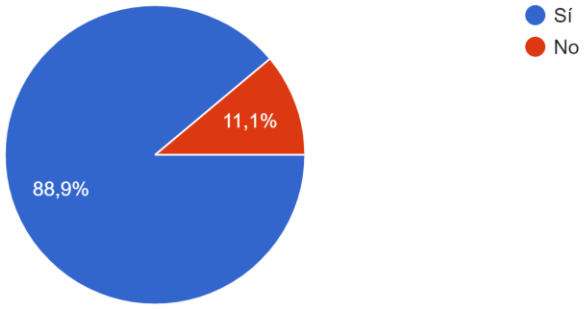


Figura 74: Respuestas pregunta 3 encuesta

En caso de responder que Sí a la pregunta "¿Actualmente maneja la trazabilidad de los requerimientos? ", ¿Cómo maneja la trazabilidad de los requerimientos cuando desarrolla?

23 respuestas

- GXTest, QControl
- Jira, Asana
- Con un sistema interno propio
- Utilizando notas en un txt y trello, mayoritariamente para saber el status de determinado desarrollo, en que etapa se encuentra (desarrollo de x parte, test, para deployar, etc...)
- GxServer
- Trello y una herramienta similar a Mantis (es un desarrollo particular).
- trello, jira
- Por cada versión liberada tenemos un plan de versión, donde se detallan todos los requerimientos asociados a su Mantis o Redmine

Figura 75: Respuestas pregunta 4 encuesta

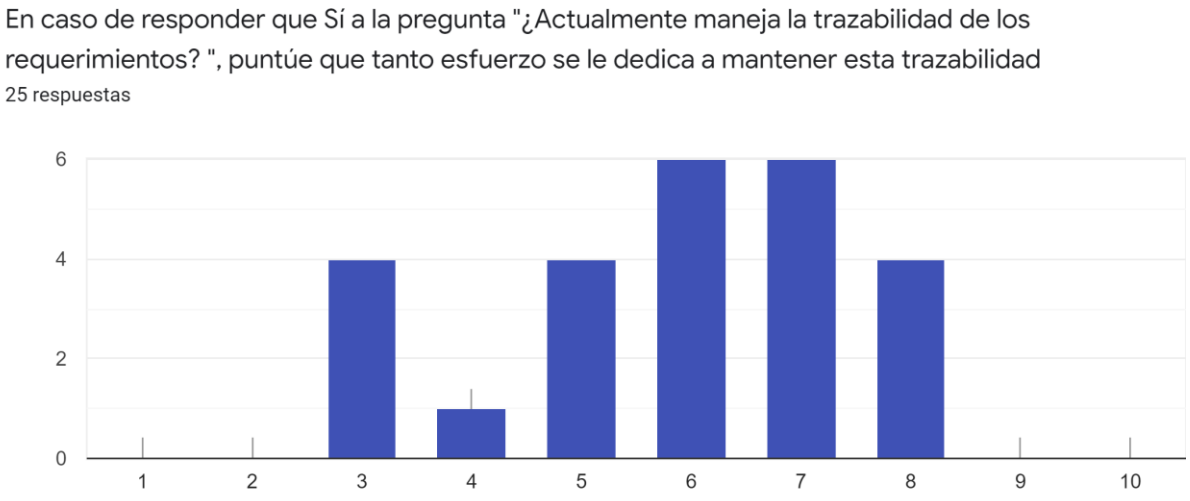


Figura 76: Respuestas pregunta 5 encuesta

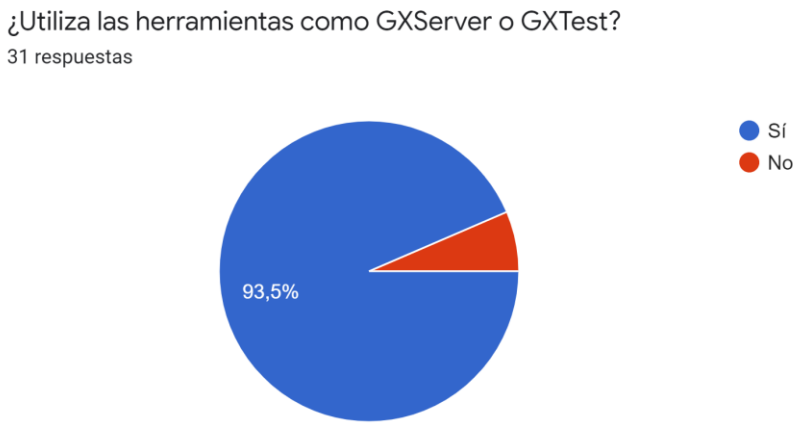


Figura 77: Respuestas pregunta 6 encuesta

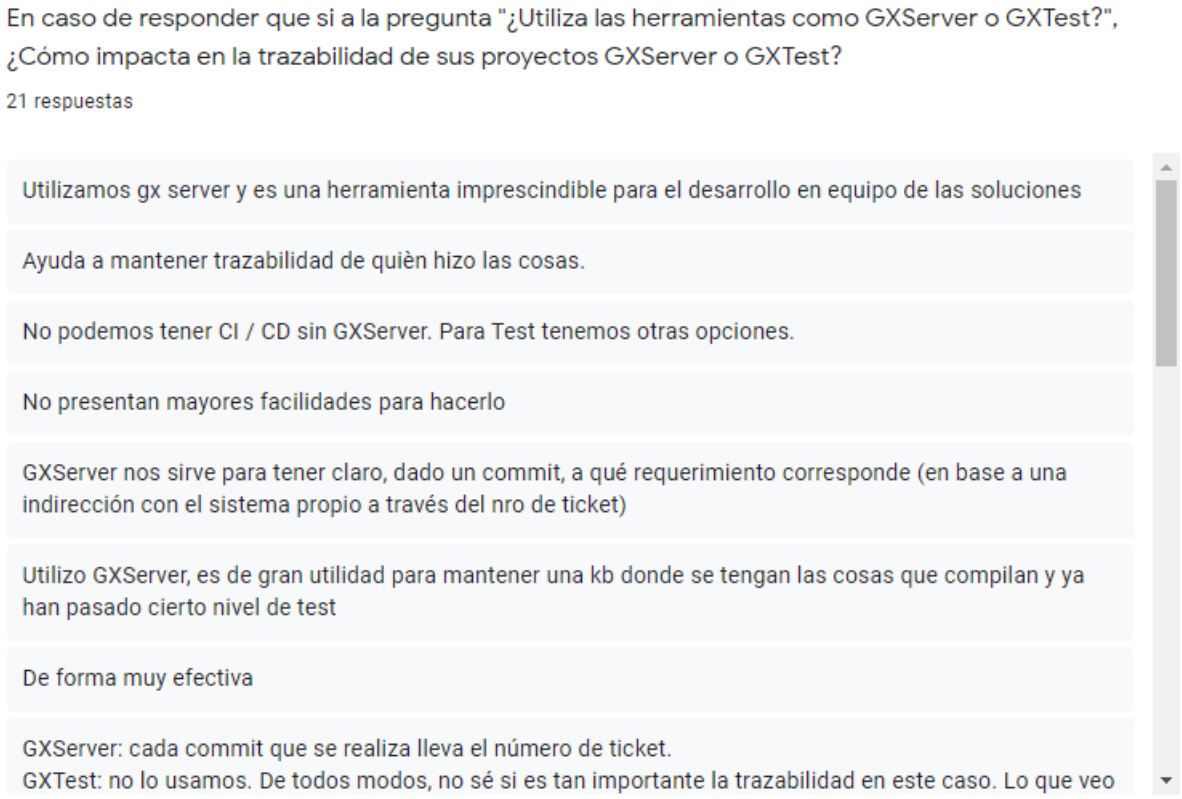


Figura 78: Respuestas pregunta 7 encuesta

Figura 79: Respuestas pregunta 8 encuesta

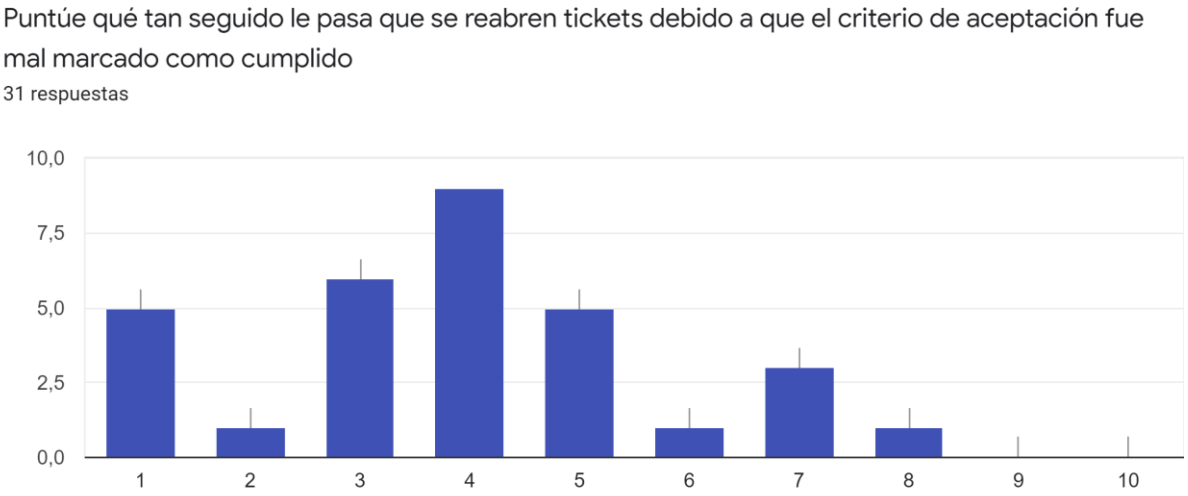


Figura 80: Respuestas pregunta 9 encuesta

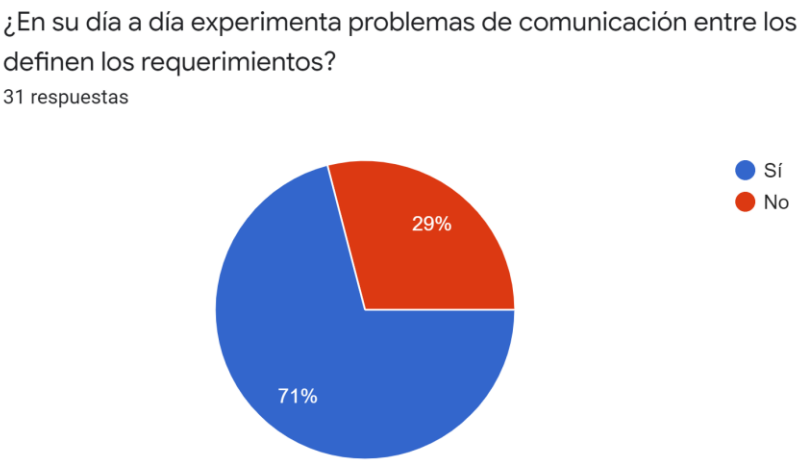


Figura 81: Respuestas pregunta 10 encuesta

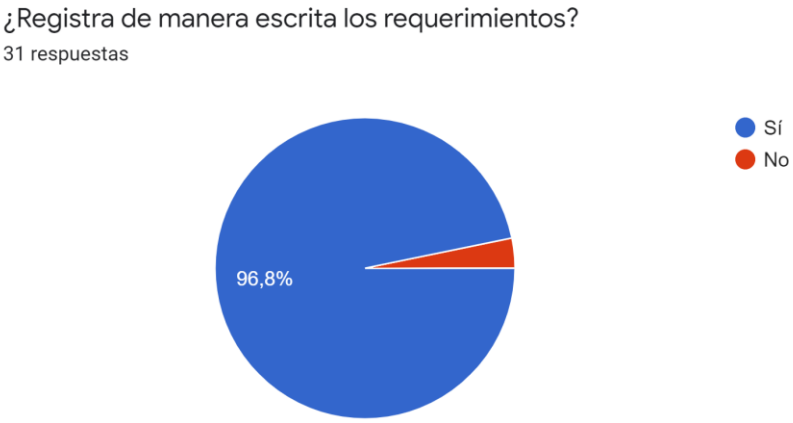


Figura 82: Respuestas pregunta 11 encuesta

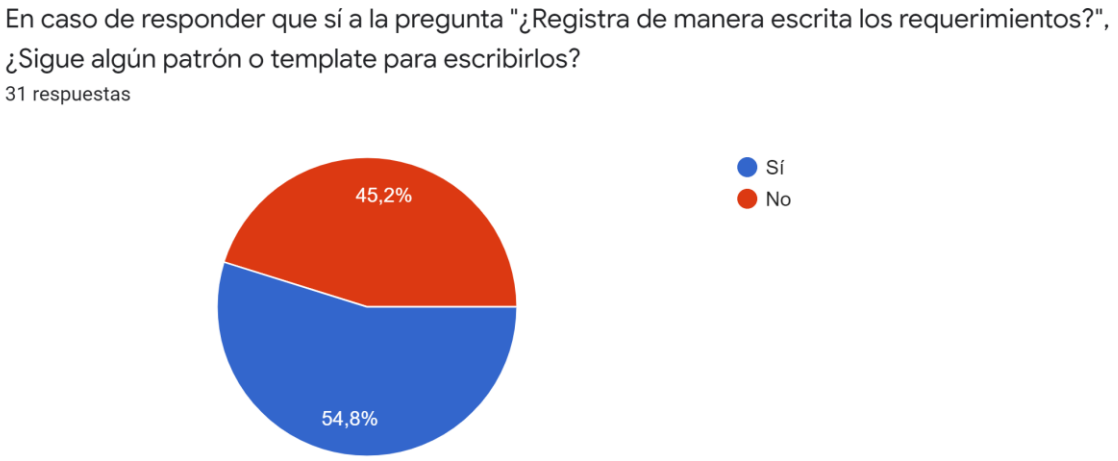


Figura 83: Respuestas pregunta 12 encuesta

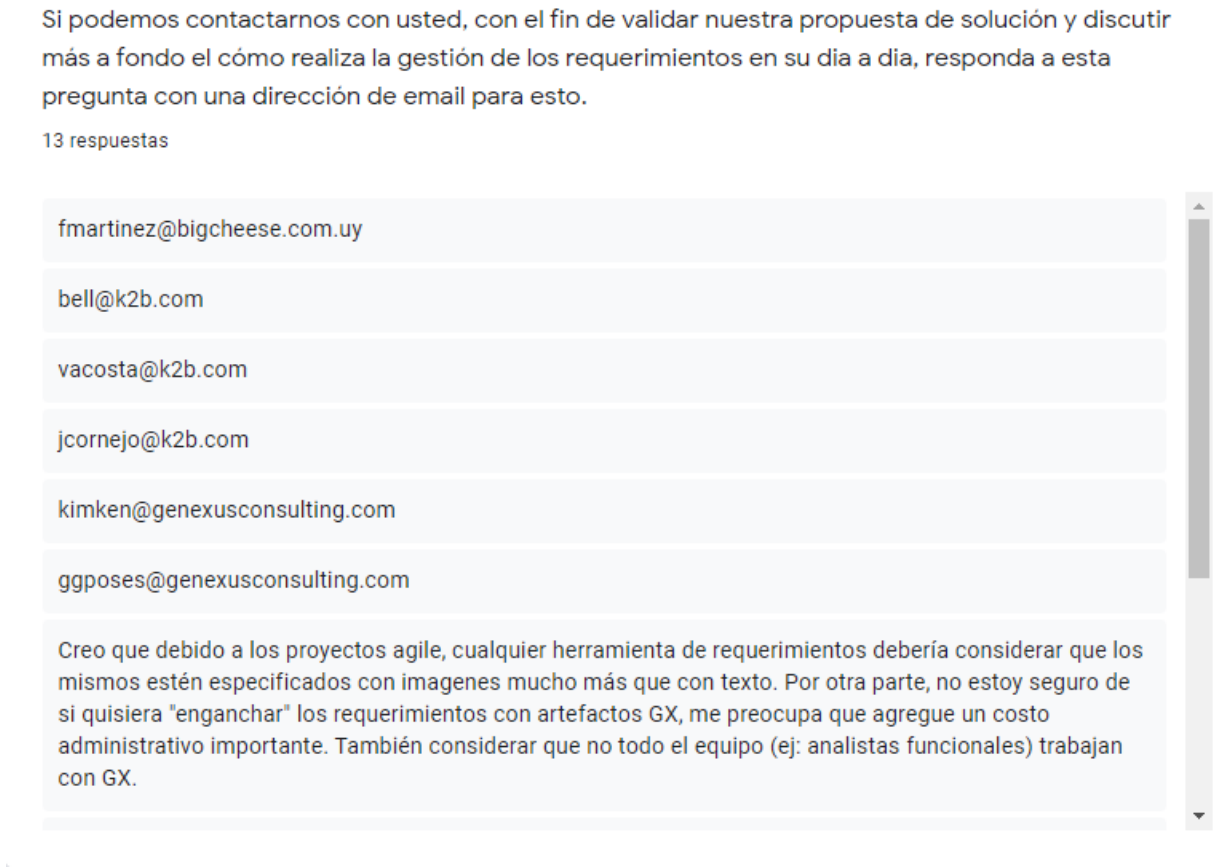


Figura 84: Respuestas pregunta 13 encuesta

14.11.3 Análisis de resultados

El equipo realiza el análisis de las preguntas respondidas en la encuesta, por un lado, se obtienen los datos de las preguntas múltiple opción, y por otro, se analizan las preguntas abiertas.

14.11.3.1 Preguntas con opciones

Recopilando las respuestas de las preguntas con opciones se llega a los siguientes resultados:

- El 25% de los encuestados necesitan mantener trazabilidad entre los requerimientos escritos y los respectivos incrementos, y actualmente no la tienen.
- Al 16% de los encuestados les resulta difícil mantener la trazabilidad entre sus requerimientos y los incrementos resultantes.

- El 77% de los encuestados necesita mantener la trazabilidad.
- El 94% de los encuestados usa GxServer o GxTests.
- El 72% de los encuestados tiene problemas de comunicación entre los desarrolladores y aquellos quienes definen los requerimientos.
- El 96% de los encuestados registran los requerimientos de forma escrita.
- El 53% de los encuestados no usan ningún template para escritura de sus requerimientos.

14.11.3.2 Preguntas abiertas

El análisis de estas preguntas fue dividido en dos, un análisis por herramientas mencionadas en las respuestas y otro análisis por proceso.

Análisis por herramientas

Pregunta: En caso de responder que Sí a la pregunta "¿Actualmente maneja la trazabilidad de los requerimientos? ", ¿Cómo maneja la trazabilidad de los requerimientos cuando desarrolla?

Herramientas mencionadas (ordenadas de más mencionada a menos)

- Mantis (Administrador Proyectos) (Externa)
- Trello (Administrador Proyectos) (Externa)
- Asana (Administrador Proyectos) (Externa)
- Jira (Administrador Proyectos) (Externa)
- GxServer (Versionador Código) (Interna)
- Excel (Planilla de Cálculo) (Externa, con plugins)
- GXTest (Pruebas Unitarias) (Interna)
- QControl (Gestión de Proyectos) (Externa)
- Redmine (Gestión de Proyectos) (Externa)

- Word (Editor Documentos) (Externa, con plugins)
- Google Docs (Editor Documentos) (Externa)
- Txt (Editor Texto) (Externa)
- Balsamiq (Mockups) (Externa)
- Jenkins (CI/CD) (Externa integrada a GXServer)
- Code Pipeline (CI/CD) (Externa integrada a GXServer)

Pregunta: En caso de responder que sí a la pregunta "¿Utiliza las herramientas como GXServer o GXTest?", ¿Cómo impacta en la trazabilidad de sus proyectos GXServer o GXTest?

Con esta pregunta se obtuvo que GxServer tiene mucho impacto en las soluciones de los encuestados, ya que la gran mayoría hace uso de ella. Lo contrario sucede con GxTests, ya que la mayoría de los encuestados no usan esta *feature.*

Análisis por proceso

Pregunta: En caso de responder que Sí a la pregunta "¿Actualmente maneja la trazabilidad de los requerimientos? ", ¿Cómo maneja la trazabilidad de los requerimientos cuando desarrolla?

Resultados:

- Varios encuestados mencionan que manejan los requerimientos sin herramientas especializadas, en donde:
    - Algunos lo hacen con hojas de cálculo (Excel o similar).
    - Algunos con documentos de texto plano.

Pregunta: En caso de responder que sí a la pregunta "¿Utiliza las herramientas como GXServer o GXTest?", ¿Cómo impacta en la trazabilidad de sus proyectos GXServer o GXTest?

Resultados:

- Varios encuestados asocian un número de *ticket* o identificador del requerimiento a GxServer para poder saber de dónde surgió un requerimiento.

- Un encuestado cree que GxServer impacta en la agilidad y el trabajo en equipo.
- Varios encuestados creen que GxServer es imprescindible.
- Varios encuestados creen que GxServer impacta de manera positiva en la trazabilidad de sus proyectos.
- Un encuestado cree que GxTest tiene impacto positivo en la trazabilidad.
- Un encuestado cree que no GxTest no tiene impacto en la trazabilidad.
- Varios encuestados no utilizan GxTest.

14.12 Feedback de revisores academicos

Se presenta, de forma resumida, el *feedback* que recibe el equipo en cada una de las revisiones ocurridas durante la vida del proyecto.

14.12.1 Revision 1

Fortalezas del grupo

- Proyecto bien organizado, los temas más importantes fueron bien pensados y se deja un registro organizado de todo lo que se viene haciendo.
- Las elecciones de tecnología fueron bien fundamentadas y acertadas. No se dejó nada al azar.
- La forma de transmitir el proyecto fue muy clara y consistente. Hay mucho pienso y muchos experimentos que apoyan este proyecto.

Oportunidades de mejora

- Los experimentos en general terminan con una conclusión nuestra de momento, podemos extenderlos y terminarlos con una prueba con algún interesado.
- Con respecto a los requerimientos no funcionales:
    - Podríamos priorizarlos.
    - Podríamos vincularlos con las distintas decisiones que nos llevaron a definir la arquitectura del modo que la definimos.
- No definir los roles tan “a la antigua” – que Álvaro sea PM significa una cosa que no se condice con lo que será la realidad del proyecto, tendríamos que reflejar esto en nuestra documentación y futuras presentaciones.

14.12.2 Revision 2

Fortalezas del grupo

- Venimos bien encaminados, invirtiendo el esfuerzo necesario
- Los tres tenemos un perfil técnico
- Tenemos mucha base hecha, debemos poder aplicarla mejor.

Oportunidades de mejora

- La duración de la presentación fue de 1 hora aproximadamente, debemos poder realizarla en 45 minutos como máximo incluyendo la demo
- Los roles definidos son generales y cada uno de nosotros tenemos impacto en cada parte del proyecto, Amalia nos recomienda que cada uno se especialice en algún aspecto.
- Faltó justificación teórica en la presentación, sobre todo en la parte de calidad. Nos preguntó y no tuvimos herramientas para defendernos.
- La presentación no fue organizada, la introducción no se entendió y luego más adelante se repitió, esto se podría omitir y agregar aspectos que nos faltaron.
- La justificación de las tecnologías elegidas para el desarrollo no las justificamos como lo deberíamos haber justificado, podíamos hacerlo, pero no lo hicimos.
- Llamamos a una parte de nuestro sistema con diferentes nombres, esto confunde.
- Mencionamos los riesgos como algo fuera de la sección “Gestión”, lo cual no es correcto.
- Dudamos cuando hablamos de números, esto le quita rigurosidad a estos números.
- Se mencionó GxConsulting, se menciona que ellos tienen flujos de trabajos armados y se nos pregunta donde entraría nuestra solución. No teníamos conocimiento de esto, deberíamos investigar y quizás charlar con ellos sobre esto.

14.12.3 Revision 3

Fortalezas del grupo

- Considerable mejora en la comunicación con respecto a revisiones anteriores.
- Se destaca la capacidad de adaptación del equipo ante los desafíos técnicos. Esto se ve porque no tuvimos documentación para guiarnos en la construcción de la extensión GeneXus.
- Bien posicionados para dar por cerrado el producto, en cuanto al desarrollo.

Oportunidades de mejora

- No vender Gherkin como una solución mágica.
- Centrar la explicación de Gherkin y la demostración en un ejemplo claro y práctico.
- Explicar de forma detallada qué es un requerimiento.

14.13 Protocolo de toma de decisiones

Template de un caso:

En el caso que _____

El equipo hará _____

En el caso que se esté tomando una decisión y haya mayoría

El equipo tomará la decisión de la mayoría.

En el caso que solo haya una persona presente

El equipo postergará la toma de decisiones hasta que haya al menos dos personas.

En el caso que se esté tomando una decisión y nadie esté de acuerdo con nadie

El equipo hará una reunión para discutir el tema la cual durará máximo 30 minutos – la idea es presentar argumentos.

En el caso que se haya hecho la reunión y continúe el desacuerdo

El equipo se tomará un tiempo para reflexionar – 1 día – de manera individual y se realizará una segunda reunión con el mismo objetivo y duración que la primera.

En el caso que se haya hecho la segunda reunión y el desacuerdo persista

El equipo recurrirá al tutor (Darío Macchi) y le solicitará retroalimentación para resolver el problema. Podrá sugerir nuevas soluciones o lo que considere necesario para ayudar a que el proyecto salga mejor.

En el caso de que habiendo recibido retroalimentación no se llegue a una decisión

El equipo recurrirá al azar – por ejemplo, asociar números a cada opción que se maneja y seleccionar al azar.

En el caso que se tenga que tomar una decisión crítica y no se cuente con alguno de los miembros del equipo en el momento.

El equipo buscará tomar la decisión en el momento siguiendo el protocolo definido en el presente documento prescindiendo de quien no esté.

En el caso que se tenga que tomar una decisión NO crítica y no se cuente con alguno de los miembros del equipo en el momento.

El equipo postergará la toma de la decisión hasta que vuelva a estar disponible quien falta.

14.14 Cobertura de pruebas unitarias - Extensión GeneXus

Se presenta la cobertura de pruebas unitarias de los principales paquetes de la solución de la extensión GeneXus construida.

Paquete	No cubierto	Cubierto
gxreq.common.dll	42,08%	57,92%
gxreq.intl.dll	2,74%	97,26%
gxreq.objectgeneration.dll	48,58%	51,42%
gxreq.parser.dll	0,00%	100,00%
gxreq.typeinference.dll	5,63%	94,37%
Total	19,81%	80,19%

Tabla 31: Cobertura de pruebas extensión GeneXus

14.15 Evidencia de certificaciones - GeneXus for Students

Se presentan imágenes como prueba de la certificación GeneXus For Students obtenida por cada miembro del equipo. También se puede consultar en la página oficial de GeneXus For Students[58].

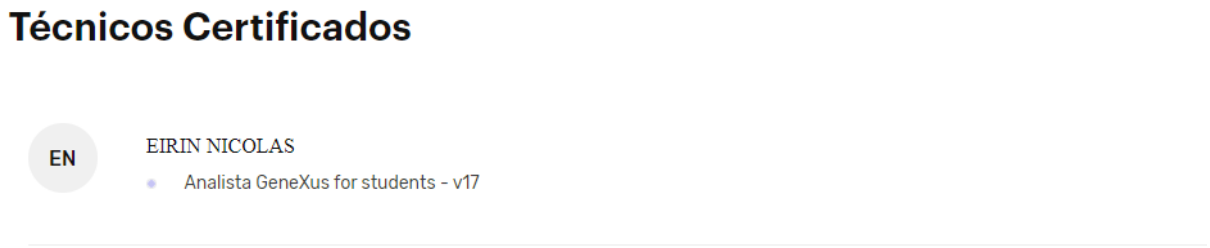


Figura 85: Evidencia de certificación GeneXus For Students

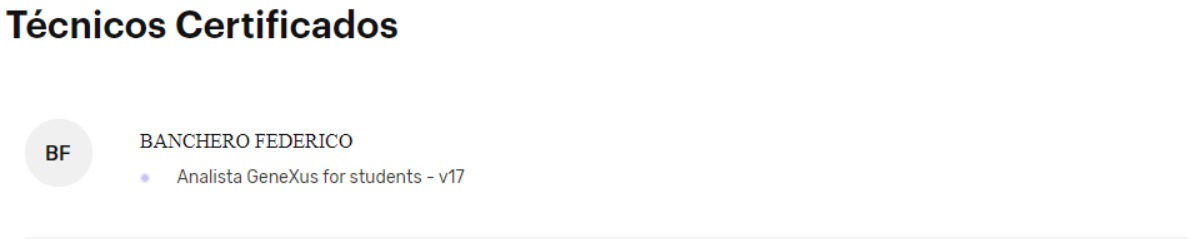


Figura 86: Evidencia de certificación GeneXus For Students

[58] https://training.genexus.com/es/aprendiendo/certificaciones/tecnicos-certificados

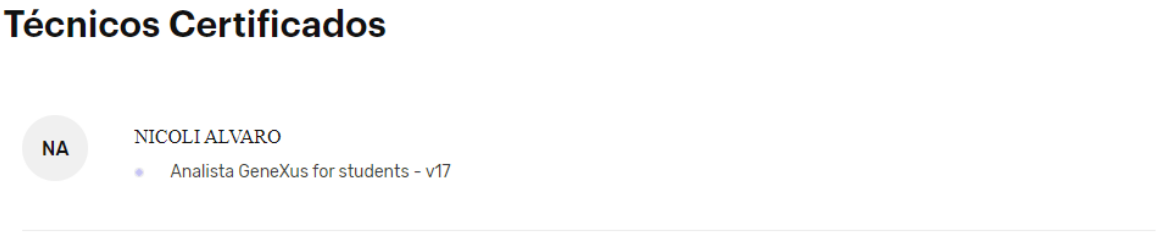


Figura 87: Evidencia de certificación GeneXus For Students

14.16 Listado de herramientas similares analizadas

Se presenta un listado de las herramientas investigadas, las cuales tienen similitud en diferentes aspectos con el producto construido:

- Cucumber
- SpecFlow
- Concordion
- K2BAudit[59]
- GxLog[60]
- jdave [61]

[59] https://marketplace.genexus.com/product.aspx?k2bauditfreeedition,en

[60] https://marketplace.genexus.com/product.aspx?gxlog,en

[61] https://github.com/jdave/JDave

14.17 Entrevistas con usuarios

Se realizaron cuatro entrevistas con usuarios, los cuales tienen dos perfiles, técnicos y no técnicos, siendo los técnicos desarrolladores y los no técnicos orientados a la gestión de proyectos. Estas entrevistas no se hicieron al finalizar el desarrollo, sino antes, por lo que existen comentarios realizados por los entrevistados que se tomaron en cuenta.

14.17.1 Conclusiones

Se realiza un punteo de los principales comentarios de los entrevistados

- La usabilidad del editor no es la mejor. Este fue un comentario recurrente, dado que cuando se realizó la entrevista el editor no tenía soporte para *snippets* y sugerencias. Este comentario fue tomado en cuenta (y en conjunto con las prioridades del cliente) y se aplicó una mejora en la usabilidad del editor.

- Surgió un comentario sobre la utilidad de generar transacciones a partir de requerimientos escritos por usuarios no técnicos, dado que pueden existir múltiples atributos que usuarios con este perfil no saben que deban ir. Por ejemplo en una relación *one-to-many* agregar la clave primaria a un objeto.

- Otro usuario sugirió tener, en el Cliente de Requerimientos, una forma de agrupar las *Features* en una idea más grande. Tener la posibilidad de agrupar estas características según cierto aspecto.

- Uno de los usuarios no encontraba la utilidad de esta solución para proyectos con un tamaño considerable.

- Por último, otro usuario menciono que sería bueno tener la posibilidad de que, al editarse una *Feature* que genere un procedimiento, si esta cambia, entonces que el procedimiento y el *set* de pruebas cambie de forma automática, esto debido a que es algo que ocurre de manera frecuente en el día a día.

14.17.2 Evidencia

Como evidencia de las entrevistas realizadas se agregan enlaces a las mismas:

- Entrevista con Oscar Muñoz, Software Developer en Lean Teach: Entrevista - Oscar Muñoz[62]
- Entrevista con Diana Gelner, Sr Business Analyst en Vindow: Entrevista - Diana Gelner[63]
- Entrevista con Alejandro Bermúdez, QA Automation Engineer en Arbusta: Entrevista Alejandro Bermúdez[64]
- Entrevista con Maximiliano Pascale, Developer en Urudata Software: Video no disponible

[62] https://fi365-my.sharepoint.com/:v:/g/personal/ne200111_fi365_ort_edu_uy/ESRD0-y7IB1FgsHOhQ0LLNMBPZ-3ZQf54Fw83EKxQkRzvw?e=8eJdcz

[63] https://fi365-my.sharepoint.com/:v:/g/personal/ne200111_fi365_ort_edu_uy/EZly7ErVVzFMnEwtnZLUR5MBvKfBG6xic4hShvBk_yGfNw?e=YcGmsA

[64] https://fi365-my.sharepoint.com/:v:/g/personal/ne200111_fi365_ort_edu_uy/Eb_h8OSbirBHqysDygXbDPEBLJvGffbLqqfjgh7nvpjXBA?e=VW7aEF

14.18 Carta de conformidad