E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

Diseño y desarrollo de soluciones DevOps para la automatización de procesos de desarrollo y despliegue de software en sistemas corporativos

Máster en Ingeniería Informática

Trabajo Fin de Máster

Autor: Úrbez Jesús Ramírez Quesada

Director: César Arriaga Egüés

Pamplona, 18 de septiembre de 2023

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen

En la actualidad existen una gran cantidad de empresas de desarrollo de software que presentan la necesidad de transformar sus procesos, técnicas y herramientas de producción. Estos cambios pretenden alcanzar un nuevo nivel de automatización que les permita: reducir los tiempos de desarrollo, disminuir la tasa de defectos y realizar despliegues de forma frecuente, segura y confiable.

Teniendo en cuenta lo anterior, durante el desarrollo de este trabajo se ha diseñado una solución que facilita tanto proceso de desarrollo, como la entrega de aplicaciones web. Dicha solución está diseñada para ser aplicada sobre aplicaciones web genéricas cuyo frontal está desarrollado en Angular TypeScript y su *backend* en C# y .NET. Los principales aspectos sobre los que trata la solución son:

- Calidad de código
- Realización de pruebas de diverso alcance (unitarios, integración, interfaz de usuario, carga y estrés)
- Automatización del proceso de desarrollo, construcción y entrega del producto

Adicionalmente, en la solución se han identificado y propuesto soluciones dos problemas comunes de este tipo de aplicaciones:

- Realización de validaciones en *frontend* y *backend*
- Diseño y aplicación de los entornos de desarrollo para el correcto desempleo de la actividad de desarrollo

Palabras clave

- Continuous Integration, CI
- Continuous Delivery, CD
- Pipelines CI/CD
- Automatización de procesos
- Code Quality
- Aplicaciones Web
- Angular
- TypeScript
- C#
- .NET

Agradecimientos

Agradezco a mi familia, todo el apoyo y cariño recibidos a lo largo de esta etapa universitaria y durante toda mi vida.

Agradezco a la empresa Tracasa Instrumental, por haberme dado la oportunidad de realizar mi TFM en colaboración con ellos y por haber creído en mi desde el primer momento.

Agradezco a mi tutor, César Arriaga Egüés, así como al resto de integrantes del equipo de NF por haberme orientado siempre que lo he necesitado.

Agradezco a mis amigos y compañeros, por haber estado amenizando esta etapa académica.

Índice

Capítulo 1 – Introducción

1.1. Motivación, antecedentes y necesidades

A lo largo de la historia, uno de los objetivos del mundo de la informática y la tecnología ha sido a tomar un conjunto de operaciones y/o tareas realizadas de manera más o menos manual y replicarlas a través de un computador u otros dispositivos electrónicos.

Con este cambio, se busca aumentar el nivel de automatización de las tareas y reducir los procesos manuales (que son, por lo general, más susceptibles al fallo que los automáticos). Tras estas réplicas, se consiguen, para esas mismas tareas, las siguientes mejoras y beneficios:

- Simplificación y automatización de las tareas y procesos.
- Aumento de productividad.
- Aumento de la calidad del producto.
- Reducción de costes.
- Disminución de errores (sobre todo derivados del error humano).

Con el tiempo, muchas empresas de desarrollo software han descubierto que, muchos de los procesos de desarrollo software que aplican, presentan los mismos defectos que las tareas manuales anteriormente reinventadas. Dichas empresas, motivadas por el éxito fruto de la reingeniería de los procesos, han comenzado replicar dicha lógica sobre sus propios procesos de desarrollo, dando comienzo así a la conocida cultura DevOps [1] [2]. Las necesidades que persiguen cubrir estas empresas a través del

alcance de un nuevo nivel de automatización son las siguientes:

- Reducción de los tiempos de desarrollo.
- Facilitación del cumplimiento de estándares definidos.
- Cumplimiento de reglas internas de desarrollo.
- Reducción los costes de producción y mantenimiento.
- Disminución de la tasa de defectos (tanto humanos como técnicos).
- Realización de despliegues y/o entregas con mayor frecuencia, confiabilidad y seguridad.

1.2. Objetivos del trabajo

Considerando todo lo anterior, el objetivo de este trabajo es facilitar y diseñar una solución que pueda ser aplicada a un caso real, pero enfocado desde un punto de vista genérico. Esta generalización tiene como objetivo amplificar su caso de uso, pudiendo ser aplicada en cualquier aplicación que utilice los mismos paradigmas, arquitectura y componentes.

La solución pretenderá **facilitar tanto el desarrollo, como la entrega y despliegue de aplicaciones web**. Entrando en detalles, esta solución va a enfocarse en la definición de los diferentes flujos de trabajo requeridos, la automatización de los procesos de desarrollo y una serie de conceptos útiles para la generación de aplicaciones de calidad.

Adicionalmente, este trabajo también abarcará la búsqueda de soluciones para algunos problemas comunes en este tipo de aplicaciones:

- Realización de validaciones en *frontend* y *backend*
- Diseño y aplicación de los entornos de desarrollo para el correcto desempleo de la actividad de desarrollo

1.2.1. Sistema objetivo

Descripción y funcionamiento del sistema

El caso real sobre el que van a aplicar las diferentes mejoras de procesos consiste en una **aplicación web** cuyo objetivo es establecer un canal de entrada para un servicio informático. En otras palabras: una aplicación web que permita al usuario introducir de

manera sencilla y guiada una serie de información en el sistema.

Este sistema pretende a brindar a la ciudadanía un servicio de carácter crítico que debe ser capaz de cumplir con unos plazos de **presentación, concurrencia y tiempos de respuesta** definidos. Por este mismo motivo, el sistema debe de estar disponible y accesible en todo momento.

Adicionalmente, se va a considerar que la aplicación objetivo trabaja con información sensible y que, por lo tanto, debe de tener el mayor grado de **robustez** posible. Con la finalidad de garantizar la seguridad y privacidad de los datos, el sistema también debe de ser capaz de proteger la diferente información que recoge a través de un sistema de sesiones.

Todos los datos introducidos por el usuario deben ser enviados a través de una estructura lógica que en lo posterior será denominado como **casilla**. Cada casilla tiene un tipo de dato (numérico, alfanumérico...) y un tipo de formato (número de teléfono, correo electrónico, texto libre…).

Dichas casillas pueden tener relaciones entre si y el contenido de unas podría llegar a afectar a las otras (por ejemplo, mostrar una casilla si otra casilla tiene un valor positivo). Para garantizar la validez de la información enviada y tratada por la aplicación, así como mejorar la usabilidad de la aplicación, el sistema debe de ser capaz de validar todos los rasgos aquí descritos de todas las casillas pertinentes.

Tecnologías de desarrollo y arquitectura del sistema

La arquitectura de la solución técnica debe estar debidamente modularizada y se va a encontrar desarrollada tanto en **Angular** [3] y **TypeScript** [4] (parte frontal), como en **C#** [5] y **.NET** [6] (parte *backend*). Como sistema de almacenaje, la aplicación se va a nutrir de una base de datos relacional.

Con la finalidad de reducir la cantidad de pérdidas de datos, evitar caídas del sistema en momentos de alta demanda y mejorar la robustez de del sistema, de manera interna la arquitectura *backend* debe hacer uso de un componente de infraestructura que permita, mediante un **sistema de colas**, el procesamiento asíncrono de los mensajes y facilite su orquestación y encaminamiento. A este componente, de aquí en adelante se le va a denominar orquestador.

El orquestador estará compuesto por un conjunto de procesos automatizados y asíncronos que se van a encargar de tanto el encolamiento, como el procesado de las peticiones cuando se disponga de los recursos necesarios (y su correspondiente desencolado).

En términos arquitectónicos y de flujo, el sistema va a operar de la siguiente manera:

1. La aplicación web (frontal) genera una petición a partir de una acción del usuario y se comunica con el sistema *backend* a través de un servicio web API REST.
2. Los datos recibidos por el API viajarían desde dicho servicio web hacia el orquestador, en donde las peticiones serán encoladas esperarán a que el sistema disponga de la capacidad suficiente para ser procesadas.
3. Tras ello, las peticiones viajarán a otro servicio web el cual es el encargado de realizar las peticiones a la BD y aplicar la lógica de la operación. Esta lógica irá desde el guardado de la información enviada por el usuario hasta la consulta de información entregada con anterioridad.
4. Por último, una vez obtenida la respuesta a la petición, dicho mensaje viajará en sentido inverso hasta llegar al frontal de la aplicación.

Nota: *Todos los servicios backend se encuentran debidamente protegidos y no es posible acceder a los mismos desde una red externa. La única forma de comunicarse con el sistema desde el exterior debe ser a través del servicio API REST habilitado para ello.*

Con todo esto en mente, la arquitectura básica del sistema es la siguiente:

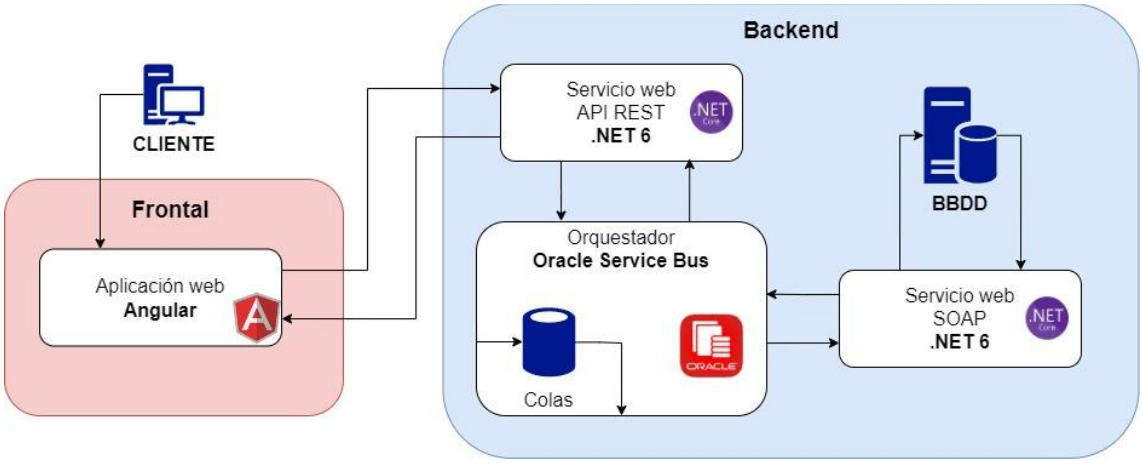


Ilustración 1: Esquema que muestra arquitectónico y el flujo de datos del sistema objetivo.

Proceso de desarrollo y herramientas utilizadas para la creación del sistema

El proceso desarrollo de las diferentes aplicaciones está enfocado al uso de herramientas compatibles con el desarrollo a través de la **plataforma .NET** (tratando de dar prioridad a aquellas que estén desarrolladas por Microsoft).

Como **editores de código** se consideran tanto *Visual Studio* [7], como *Visual Studio Code* [8]. Para la gestión del **sistema de base de datos** relacional, se *emplea Microsoft SQL Management Studio* [9]. Cuando se requiera de un **servidor local** para levantar el

sistema en un entorno local, se va a emplear un servidor IIS Express [10]. El **sistema operativo** de los equipos de los desarrolladores es Windows 10.

Adicionalmente se va a considerar que el equipo de trabajo aplica la metodología ágil **Scrum** [11], junto con la Integración y despliegue continuos, de las que se hablará más adelante.

Dicho equipo está compuesto por unas **10 personas**, entre cuyos miembros hay 9 desarrolladores y un *Scrum Master*. La figura del *Product Owner* es aplicada a través de una persona interna, pero que no pertenece de manera directa al equipo de desarrollo.

Para realizar las gestiones necesarias para la aplicación de la metodología propuesta (tablero de tarjetas, gestión del backlog, comunicaciones internas, etc.) se emplea la herramienta *Microsoft Team Foundation Server* (TFS) [12].

Como herramienta de **control de versiones** de código fuente, dicho equipo emplea un gestor de repositorios basado en la tecnología GIT [13]. Como herramienta de **automatización, integración y despliegue** se va a emplear la herramienta de código abierto Jenkins [14]. También se dispone de un **servidor de dependencias** Nexus [15]. Por último, como **orquestador** se emplea la herramienta *Oracle Service Bus* (OSB) [16].

1.2.2. Requisitos de la solución

La propuesta de solución que se va a realizar en este trabajo, pretende atender a los siguientes **requisitos funcionales**:

- Simplificación de los procesos de desarrollo y mantenimiento
- Disminución de errores (tanto de producto como de desarrollo)
- Permisibilidad de desarrollo en diferentes entornos
- Automatización de los despliegues y entregas de producto
- Definición y seguimiento de estándares de calidad de código
- Desarrollo y aplicación de pruebas
    - Pruebas unitarias
    - Pruebas de integración
    - Pruebas de interfaz de usuario (UI)
    - Pruebas funcionales
    - Pruebas de carga y rendimiento
- Resolución de problemas típicos
- Permisibilidad de aplicación de metodologías ágiles

  - Control de versiones
  - Gestión de tareas
  - Integración y despliegue continuos

***Nota**: Estos requisitos funcionales se encuentran definidos en términos del problema que se está pretendiendo resolver y no del producto concreto tomado como referencia.*

De la misma forma, la solución propuesta debe considerar los siguientes **requisitos no funcionales**:

- Control de versiones mediante GIT
- Aplicabilidad en diferentes formatos de tecnologías web
  - Simple Object Access Protocol (SOAP)
  - API REST (representational state transfer)
- Empleo de tecnologías desarrolladas por Microsoft
  - *Visual Studio*
  - *Visual Studio Code*
  - *Microsoft SQL Management Studio*
  - *Microsoft Team Foundation Server* (TFS)
  - IIS Express
  - Windows 10
- Empleo del *Oracle Service Bus* (OSB)
- Empleo de sistemas de BBDD relacionales
- Empleo del repositorio de artefactos Nexus

Capítulo 2 – *Code Quality*

2.1. Introducción

El objetivo de este capítulo es buscar una fórmula que permita a mejorar en términos de **calidad de código** el caso supuesto práctico estudiado. Como el mundo de la calidad de código es extremadamente amplio, este capítulo se va a centrar en aspectos de calidad de código que sean posibles de automatizar e integrar, a través de una serie de herramientas, dentro de los procesos asociados a la integración continua del producto.

Con estos objetivos en mente, este capítulo considerará los siguientes **requisitos funcionales**:

- Simplificación de los procesos de desarrollo y mantenimiento
- Disminución de errores (tanto de producto como de desarrollo)
- Definición y seguimiento de estándares de calidad de código
- Resolución de problemas típicos
- Permisibilidad de aplicación de metodologías ágiles
    - Integración y despliegue continuos

Asimismo, también se tendrán en cuenta los siguientes **requisitos no funcionales**:

- Aplicabilidad en diferentes formatos de tecnologías web
    - Simple Object Access Protocol (SOAP)
    - API REST (representational state transfer)
- Empleo de tecnologías desarrolladas por Microsoft

- *Visual Studio*
- *Visual Studio Code*
- *Microsoft SQL Management Studio*

- Empleo del *Oracle Service Bus* (OSB)

2.2. Conceptos previos

Definición del concepto calidad de código

Se conoce como **calidad de código** al cumplimiento, por parte de un código fuente, de un conjunto de **estándares y prácticas** que le permiten ser mejor en términos de legibilidad, mantenibilidad, eficiencia y ratio de errores.

Es un concepto que tiene una fuerte relación con el **diseño del producto** (realizar cambios de calidad una vez comenzado el desarrollo puede resultar muy complejo) y afecta de manera directa a la facilidad con la que el código será entendido, probado, modificado y ampliado a lo largo del tiempo.

Aspectos a considerar en la calidad de código

Para medir la calidad de código que tiene un producto, se pueden tener en cuenta los siguientes aspectos (algunos de ellos serán considerados en puntos vistos posteriores en este trabajo):

- **Legibilidad del código**. Se considera que un código es de mayor calidad cuanto más fácil sea de leer y entender para los desarrolladores.
- **Mantenibilidad del código**. Se considera positivo que el código esté debidamente estructurado de tal forma que permita realizar posteriores modificaciones y correcciones con la menor cantidad de esfuerzo posible.
- **Eficiencia del código**. Se considera positivo que el rendimiento código sea lo mejor posible.
- **Robustez del código**. Se considera positivo que el código sea capaz de manejar por si solo la mayor cantidad de situaciones posibles.
- **Reusabilidad del código**. Se considera positivo que el código existente dentro de un producto pueda ser reutilizado en el futuro en diferentes partes tanto del mismo proyecto como de otros proyectos relacionados.
- **Diseño de la arquitectura**. Se considera positivo que el producto esté diseñado de tal forma que sea fácil adaptarlo a posibles futuros cambios de requisitos. En este aspecto cobran especial relevancia el uso de diferentes patrones de código.

- **Pruebas automatizadas**. Cuanto mayor sea la cantidad y calidad de las pruebas automatizadas disponibles para el código, mejor consideración tendrá.
- **Cumplimiento de estándares**. El seguimiento de estándares y convenciones de desarrollo y estilo se considera positivo ya que ayuda a cohesionar el producto y a facilitar la legibilidad del código.
- **Versionado**. Utilizar sistemas de control de versiones de código facilita prácticas de integración continua, así como permite realizar un seguimiento de la evolución del producto a lo largo del tiempo.

Opciones de mejora en términos de calidad de código

Al igual que existen gran cantidad de aspectos que se tienen en cuenta a la hora de evaluar la calidad de código, también existen diversas maneras de llevar a cabo de manera activa acciones para tratar de incluir estos principios durante todo el ciclo de desarrollo del producto:

- **Seguimiento de principios de diseño software y patrones de diseño** (PE: principios SOLID).
- **Refactorización de código**. En el caso de identificar durante el desarrollo componentes con problemas de redundancia, escasa claridad o ineficiencia, estos componentes deben ser rediseñados con la finalidad de solventar dichos problemas.
- **Nombrado de variables, métodos, clases y similares**. El uso de nombres claros y descriptivos ayudan en gran medida a la legibilidad del código.
- **Documentación**. Realizar documentación de todos aquellos aspectos que tengan cierto nivel de complejidad de la aplicación.
- **Diseño y programación de pruebas automatizadas**. Como ya se ha visto en apartados anteriores, el uso de pruebas ayuda a mejorar diversos aspectos del desarrollo software.
- **Uso de una herramienta de control de versiones**.
- **Pair programming y revisiones activas de código.** Involucrar a varios desarrolladores dentro de un mismo desarrollo puede reducir en sobremanera la cantidad de mala praxis introducidas en el código de manera no intencionada.
- **Uso de herramientas de análisis de código.** El uso de herramientas automáticas de análisis de código estático permite identificar problemas de código de diversa índole (*code smells*, seguimiento de principios y patrones, errores estilísticos…).

Formas evaluar la calidad de código

Existen muchas formas distintas de evaluar la calidad de un código. Cada una de ellas tiene un objetivo y forma de aplicación distinto. Las principales son:

- **Verificación estilística del código (*linting*)**. Se realiza directamente sobre el código fuente y su objetivo es buscar errores estilísticos en el código.
- **Pruebas de Seguridad de Aplicaciones Estáticas (SAST)**. Se realiza directamente sobre el código fuente y utiliza técnicas de prueba de caja blanca para detectar errores programáticos.
- **Pruebas de seguridad de aplicaciones dinámicas (DAST)**. Se realiza sobre una versión compilada del código fuente y utiliza técnicas de prueba de caja negra para detectar errores programáticos.
- **Pruebas de seguridad de aplicaciones interactivas (IAST)**. Se realiza a todos los niveles (evalúa tanto el código fuente, como los compilados). Su objetivo principal es la detección de vulnerabilidades.
- **Análisis de composición de software (SCA)**. Se realiza a partir de las referencias del producto que analiza. Su objetivo es revisar dichas dependencias en busca de vulnerabilidades o fallos.

	SAST	DAST	IAST
ESCANEADO REACTIVO			✓
ESCANEADO PROACTIVO	✓	✓	
PRUEBAS DE CAJA NEGRA		✓	
PRUEBAS DE CAJA BRANCA	✓		✓
ANÁLISIS DEL CÓDIGO FUENTE	✓		
PRUEBAS EN TIEMPO DE EJECUCIÓN		✓	✓

Ilustración 2: tabla en la que se muestra una comparativa funcional entre las pruebas SAST, DAST e IAST.

Nota: *Para más información sobre la calidad de código, se puede consultar el libro Clean Code: A Handbook of Agile Software Craftsmanship de Robert C Martin* [17].

2.3. Verificación estilística del código

Se conoce como ***linting*** a un proceso (generalmente automático) de análisis estático

de código cuya finalidad es evaluar si el código desarrollado cumple con una serie de estándares y políticas en términos estilísticos. Se trata de un proceso que aporta valor debido a que **ayuda a mantener la cohesión** del producto, los cumplimientos de estándares y la legibilidad del código.

Las reglas que se comprueban dependen en gran medida del producto que se está evaluando y la tecnología sobre la que está desarrollado. Es habitual que los diferentes equipos de desarrollo sigan los estándares propuestos por la comunidad para cada una de las tecnologías que utilizan.

Pese a ello, dependiendo del producto, puede haber ciertos aspectos que merezca la pena modificar sobre el estándar definido. Para ellos, lo más importante es que entre todo el equipo se llegue a un consenso sobre las reglas a aplicar y que todos los desarrolladores las sigan pie de la letra.

2.3.1. Herramientas

Conceptos sobre las herramientas y su implantación

Como en la mayoría de ocasiones evaluar todo el código de manera manual resulta en una tarea inabordable (además de los posibles errores cometidos mediante este tipo de revisiones), existen un sinfín de **herramientas** que sirven para comprobar las reglas acordadas por el equipo de desarrollo.

A estas herramientas se las conoce como ***linters*** y se encargan de **analizar el código** de manera estática en busca de inconsistencias. En muchas ocasiones, las propias herramientas son capaces de solventar los errores detectados.

Por lo general, los *linters* se suelen **centrar en una tecnología concreta** y disponen de un amplio catálogo de configuraciones posibles para ella. A la hora de utilizar un *linter* resulta importante que previamente a su implantación se hayan definido aquellas reglas que se desean comprobar (en vez de aplicar a ciegas las reglas por defecto). Un cambio en una regla tras una vez comenzado el desarrollo puede requerir de mucho trabajo para su resolución.

Linters para aplicaciones desarrolladas en C# y .NET

En el caso de las **herramientas desarrolladas en .NET** y en *Visual Studio* (en el caso de nuestro sistema, ambos servicios web), aunque existen diversas herramientas como *SonarLint* [18], *StyleCop* [19] o *ReSharper* [20], el propio IDE de ***Visual Studio*** (a partir de su versión de 2019) dispone de una herramienta ya integrada y completa.

La **herramienta integrada**, se alimenta de la configuración establecida en un fichero *.editorconfig* [21] que debe estar alojado en la raíz de la solución. Este fichero puede ser configurado de manera manual, o desde el propio IDE, quien nos ofrece una interfaz gráfica sencilla y eficaz. En la documentación oficial podemos encontrar un listado de todas las opciones que ofrece [22]. Una vez generado dicho fichero de configuración, éste puede ser subido al repositorio de control de versiones para que cualquier persona que edite el proyecto aplique siempre las mismas reglas.

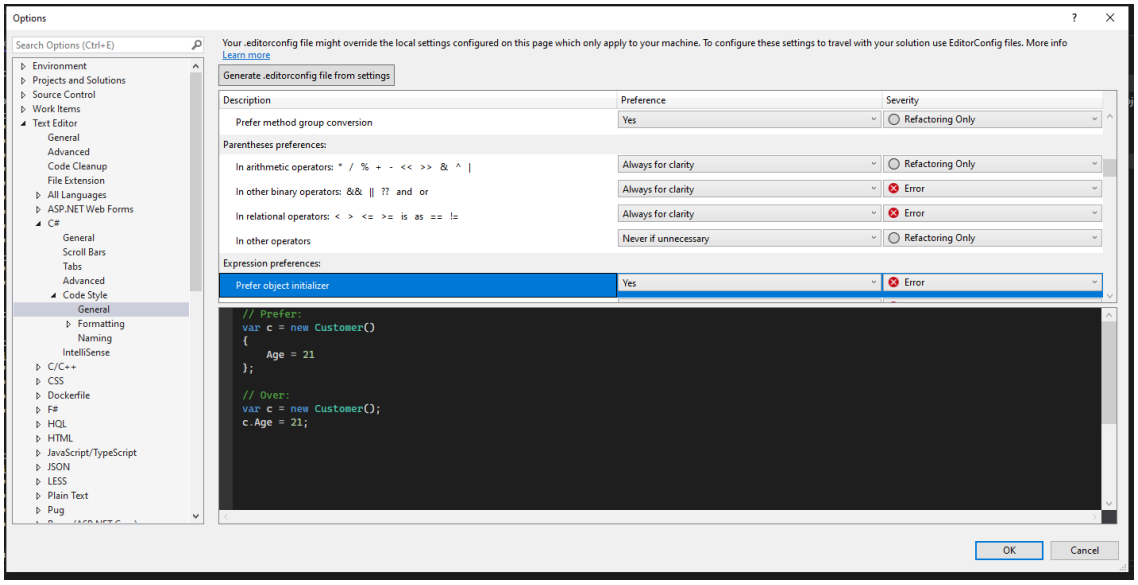


Ilustración 3: Cuadro de configuración de reglas de linteo de Visual Studio.

La opción del ***linter* integrado** dentro del IDE resalta sobre las demás debido a que, además de no requerir ningún software adicional, permite ser configurada desde el propio IDE para que cada vez que se guarden los ficheros o se genere una *build*, se lancen de manera automática todas las reglas y se apliquen las correcciones sobre los fallos encontrados. Las **correcciones automáticas** se aplican sobre aquellas reglas rotas para las cuales su corrección no requiere grandes cambios. Algunos ejemplos de correcciones comunes son: espaciado del código y eliminación de variables e importaciones no utilizadas.

Adicionalmente, durante el desarrollo de código, el motor de *Visual Studio* se adapta a la dicha configuración de reglas. De esta forma, ofrece a través de su bombilla de sugerencias, propuestas de cambios que permiten solucionar los incumplimientos de dichas reglas (por ejemplo, renombrado de variables).

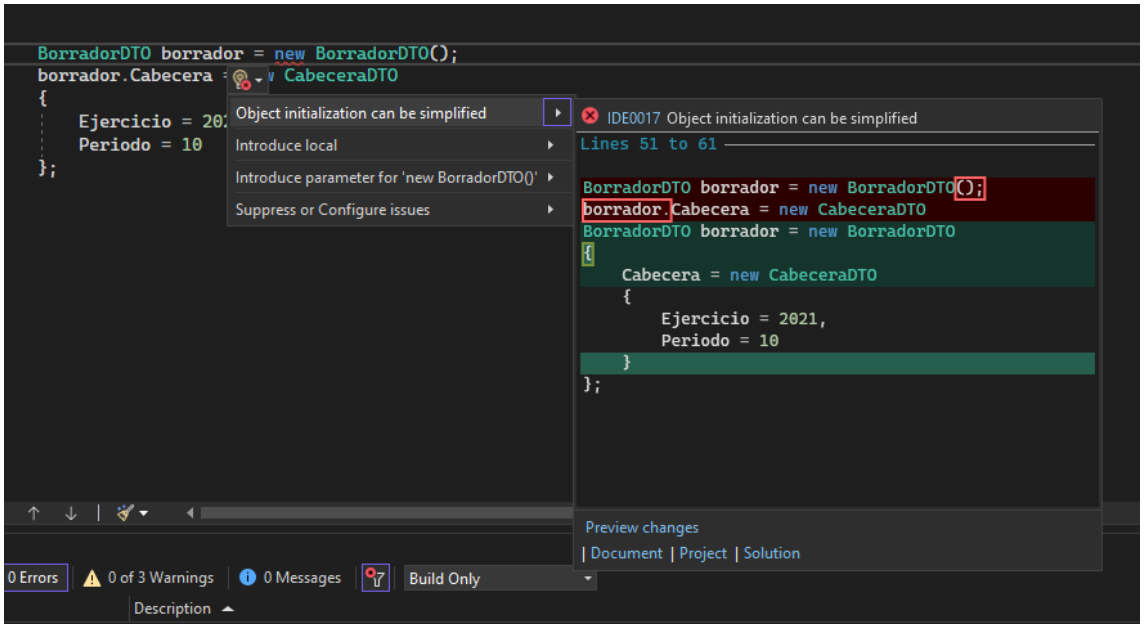


Ilustración 4: Vista en la que se aprecia una sugerencia de linting propuesta por la herramienta integrada dentro de Visual Studio. La corrección se puede aplicar a través de la bombilla de sugerencias.

Por último, además de la verificación o no de una regla, la herramienta permite definir un tipo de severidad para la misma. Según este nivel de severidad, estas reglas rotas aparecerán en diferentes momentos. Las opciones de severidad son:

- **Refactoring Only**. La regla no será aplicada salvo que se le indique al IDE que se está realizando una refactorización de código.
- **Suggestion**. La regla es verificada y puede ser visualizada dentro del IDE al abrir el fichero en el que se encuentra. En la bombilla de sugerencias nos indica cómo solucionar los errores encontrados.
- **Warning**. La regla es verificada y puede ser visualizada dentro del IDE al abrir el fichero en el que se encuentra. Adicionalmente, el cumplimiento de esta regla se verifica durante el proceso de compilación y en caso de romperse, el incumplimiento se marca como *warning* a nivel de compilación. En la bombilla de sugerencias nos indica cómo solucionar los errores encontrados.
- **Error**. La regla es verificada y puede ser visualizada dentro del IDE al abrir el fichero en el que se encuentra. Adicionalmente, el cumplimiento de esta regla se verifica durante el proceso de compilación y en caso de romperse, el incumplimiento se marca como *error* a nivel de compilación y se detiene el proceso de compilación. En la bombilla de sugerencias nos indica cómo solucionar los errores encontrados.

Toda la funcionalidad de la herramienta, además de desde el IDE, también puede ser ejecutado desde la consola de comandos convirtiendo la herramienta en válida para su integración en herramientas de automatización.

Linters para aplicaciones desarrolladas en Angular y TypeScript

En el caso de herramientas desarrolladas en **Angular TypeScript** (en nuestro caso, la aplicación web con la que interactúa el usuario), el linter más destacado es *Prettier* [23].

Se trata de una herramienta de **formateo de código** que soporta muchos lenguajes de programación y se puede integrar de manera nativa con *Node* [24] (servidor local sobre el que se basa Angular). *Prettier*, por defecto se nutre de las reglas que se encuentren configuradas en un fichero *.prettierrc.json (*aunque esto es configurable).

Las posibles reglas a aplicar se pueden encontrar en su documentación [25]. Además de la detección de reglas rotas, *Prettier* también es capaz de solucionar algunas de ellas de manera automática.

La herramienta se puede **integrar** de manera sencilla con ***Visual Studio Code*** a través de un *plugin* [26]. Mediante esta integración, el IDE es capaz de mostrar los errores sobre la marcha mientras se editan los ficheros. Adicionalmente, esta integración permite la posibilidad de configurar la herramienta de tal manera que, cada vez que se guarde un fichero, se realicen todas las correcciones que así se encuentren indicadas en el fichero de configuración. Como es habitual, también es posible lanzar las reglas desde la terminal de comandos, lo que facilita su integración en las herramientas de automatización de procesos.

Linters para BBDD y SQL

Aunque en el caso de la BBDD quizás no tenga tanto sentido utilizar una herramienta de linteo (puesto que la BBDD habitualmente su código no recoge lógica de negocio y su objetivo es plenamente almacenar datos), existen herramientas, que se encargan de aplicar reglas de estilo sobre los diferentes scripts SQL.

Un ejemplo de estas **herramientas** es *SQLFluff* [27]. Esta herramienta de código abierto permite, al igual que todas las herramientas vistas en este apartado, detectar y autocorregir errores estilísticos. Dispone de una **extensión en *Visual Studio Code*** para poder visualizar y corregir los errores en tiempo real, y también se puede utilizar desde consola de comando, estando especialmente preparada para su implementación en procesos de integración continua.

Otra herramienta es *SQL Enlight* [28], una herramienta que permite realizar las mismas acciones e integraciones que *SQLFluff*, y que, además, dispone de una integración mediante extensión para *Microsoft SQL Server Management Studio*. Esta herramienta también permite, como se comentará en los siguientes apartados, la realización de un análisis en busca de errores de en las diferentes BBDD.

Para **decidir cuál de las dos utilizar**, hay que considerar que *SQLFluff* es una herramienta gratuita, mientras que *SQL Enlight* no. En este aspecto, para cada caso implementación concreta, resulta imprescindible evaluar si las mejoras que proporciona *SQL Enlight* sobre *SQLFluff* merecen la pena (principalmente el poder analizar no sólo los scripts, sino además la BD). Como norma general, al ser las BBDD sistemas de almacenaje, con la funcionalidad que aporta SQLFluff debería ser suficiente.

Linters para el orquestador OSB

Por desgracia, actualmente **no existe ninguna herramienta** (ni propia ni externa) que permita controlar los estilos del orquestador (*Oracle Service Bus*).

Para ser capaces de mantener las reglas definidas por el equipo, será necesario que, **los propios desarrolladores revisen de manera manual** el código que desarrollen en la herramienta. Para llevarlo a cabo y mitigar posibles errores humanos, puede ser de utilidad la aplicación de pair programming o de merge request a la hora de realizar los cambios en dicho componente.

2.4. Verificación funcional del código

En este apartado se van a considerar aquellas validaciones fruto de un proceso de análisis estático de código que se centran en la detección de problemas potenciales, búsqueda de defectos, fallos de seguridad incumplimientos de estándares… Principalmente, el apartado se va a centrar en las **pruebas de seguridad de aplicaciones estáticas (SAST) y el análisis de la composición de software (SCA)**. Ambas comprobaciones se hacen directamente sobre el código fuente.

A grandes rasgos, la aplicación de estos procesos permite la detección de errores, aumentar la legibilidad, facilitar la mantenibilidad, aumentar la seguridad y facilitar el desarrollo del código. Para ello, los principales aspectos que se suelen evaluar son:

- **Detección de errores y problemas.** Mediante el análisis estático, en ocasiones es posible detectar errores de sintaxis, accesos incorrectos, problemas de flujo, etc.
- **Cumplimiento de estándares.** Mediante el análisis estático, es posible detectar si el código cumple con los diferentes estándares y principios propuestos por el equipo de desarrollo.
- **Identificación de vulnerabilidades y seguridad.** Este análisis también permite la detección de posibles vulnerabilidades y brechas de seguridad del sistema.

Cuando se realiza a través de herramientas, suele ser habitual la búsqueda de errores conocidos dentro de las herramientas de terceros utilizadas.

- **Optimización y rendimiento.** Mediante un análisis del flujo de datos (generalmente realizado a través de una herramienta), es posible detectar partes de código que ralentizan el producto, consumos de memoria innecesarios, necesidades de reutilización de código…
- **Uso de patrones de diseño.** Mediante el análisis estático es posible evaluar si se están aplicando patrones de diseño donde debería y si la implementación de los mismos es la correcta.

2.4.1. Herramientas

Conceptos sobre las herramientas y su implantación

Al igual que sucedía en el caso del *linting*, evaluar todos los aspectos mencionados con una profundidad suficiente como para sacar provecho del propio análisis resulta prácticamente imposible de abarcar manualmente.

Para simplificar la tarea, existen diversas **herramientas que permiten automatizar estos análisis** para simplificar los procesos de producción y obtener tanto análisis en tiempo real, como análisis posteriores (en un formato de informe). La revisión de estos informes debe de incluirse dentro del proceso de desarrollo.

Cada una de estas herramientas se centra en tecnologías concretas, por lo que en función de la tecnología sobre la que esté construida en componente a probar, se deberán utilizar unas herramientas u otras.

Herramientas para aplicaciones desarrolladas en C# y .NET

En el caso de las **herramientas desarrolladas en .NET** y en *Visual Studio* (en el caso de nuestro sistema, ambos servicios web) es posible emplear, de nuevo, la herramienta de análisis estático que nos ofrece de manera nativa el propio IDE.

Esta herramienta, a través de una configuración realizada de exactamente la misma manera (aunque con diferentes reglas) que su homóloga del apartado de *linting*, nos va a permitir tanto detectar como corregir los diferentes errores del código fuente.

Funciona especialmente bien operando con transformaciones de codificación clásica a métodos más modernos (por ejemplo, el empleo de una función lambda). De nuevo, este análisis puede ser realizado desde la línea de comandos permitiendo así su integridad con las herramientas de automatización.

Una diferencia de la herramienta comparando su análisis funcional de su análisis de estilo es que, en el caso de la verificación de código, aunque el IDE nos muestre y nos proponga cambios para solventar los errores, estos **nunca serán aplicados de manera automática** (cosa que en el caso del *linting* sí que sucedía).

Aunque el uso de la herramienta integrada supone un aumento significativo sobre la no inclusión de herramientas de análisis, el uso de sólo esta herramienta no es suficiente para cumplir los objetivos de este apartado, puesto que se estarían quedando en el tintero comprobaciones como: identificación de vulnerabilidades, correcto empleo de patrones, optimización del código fuente…

Para remediarlo, **su uso se puede combinarse** con el uso de otra herramienta: ***SonarQube*** [29]. *SonarQube* es una plataforma de código abierto de análisis código fuente muy completa. Durante sus procesos, analiza el código en busca de fallos de código, vulnerabilidades, *code smells*, fallos de optimización y posibles duplicidades de código. Adicionalmente, si relaciona junto los *test* también es capaz tanto de ejecutar, como de mostrar los resultados como la cobertura de código (términos de los que se hablará más adelante). Adicionalmente, la herramienta es capaz de almacenar, a través de un histórico, la evolución del producto en todos aspectos que analiza.

Al contrario que las otras herramientas vistas hasta ahora, sonar **no es una herramienta estática** que se lanza en el momento, sino que se trata de una aplicación independiente alojada en su propio servicio web con la que nos vamos a comunicar a través de un api. Para la visualización de los resultados, dispone de una GUI muy completa.

***Visual Studio* 2022** permite, a través de la instalación de un *plugin* [30], la **conexión de manera nativa con SonarQube**. Mediante esta conexión, desde el IDE es posible observar los resultados del análisis realizado por SonarQube en tiempo real, como si de una herramienta integrada se tratase. A través de la bombilla de sugerencias de VS, SonarQube también realiza propuestas de corrección para las violaciones de reglas establecidas. El resultado de esta posibilidad es que combina a la perfección junto con la herramienta por defecto del IDE (que sigue el mismo modus operandi), pareciendo a ojos del usuario que se está utilizando una única herramienta de análisis.

Herramientas para aplicaciones desarrolladas en Angular y TypeScript

En el caso de herramientas desarrolladas en Angular TypeScript (en nuestro caso, la aplicación web con la que interactúa el usuario), el *linter* más destacado es ***ESLint*** [31].

ESLint es una herramienta de código abierto que permite **identificar y reportar errores**

funcionales para código desarrollado en TypeScript. Adicionalmente, la herramienta permite ser integrada de manera nativa con Node.

A través de un *plugin* [32], es posible ver **en tiempo real, desde *Visual Studio Code***, los diferentes errores encontrados por *ESLint*, así como las sugerencias para corregirlos (a través de las sugerencias). Las diferentes configuraciones que permite realizar *ESLint* [33], se almacenan por defecto en un fichero *.eslintrc*, aunque es posible modificar dicho comportamiento. No obstante, esta forma de configuración se está migrando a un nuevo formato que ya se encuentra desarrollado [34], por lo que para nuevos desarrollos utilizar este nuevo formato debe ser prioritario.

***ESLint* combina a la perfección con el *linter* que se ha propuesto** para la aplicación frontal (*Prettier*). Ambos se encuentran, como se ha comentado en sus respectivos apartados, perfectamente preparadas para su integración y visualización a través de *Visual Studio Code* y se pueden lanzar de manera conjunta a través de la **consola de comandos**, facilitando así su puesta en marcha durante los procesos de automatización.

Herramientas para BBDD y SQL

Para la base de datos, si bien es cierto que debido a su naturaleza de almacenamiento (y no de funcionalidad) puede parecer que no tiene tanto sentido el empleo de una herramienta de esta índole, siempre resulta positivo realizar un análisis de tanto las BBDD, como los scripts. Esta necesidad se intensifica sobre todo de cara a una posible ejecución automatizada de los mismos.

Para ello, se puede utilizar la herramienta ***SQL Enlight***. *SQL Enlight* es una herramienta que permite identificar problemas potenciales de diseño de base de datos, permitiendo refactorizar los errores detectados de manera automatizada.

A través de un *plugin* [35], es posible **integrar dicha aplicación junto con *Microsoft SQL Server Management Studio***. A través de esta integración, el IDE es capaz de mostrar los diferentes defectos detectados en tiempo real.

Por último, SQL Enlight también es posible ejecutarla a través de **la línea de comandos**, lo que nos va a permitir insertar esta herramienta dentro del proceso de integración continua.

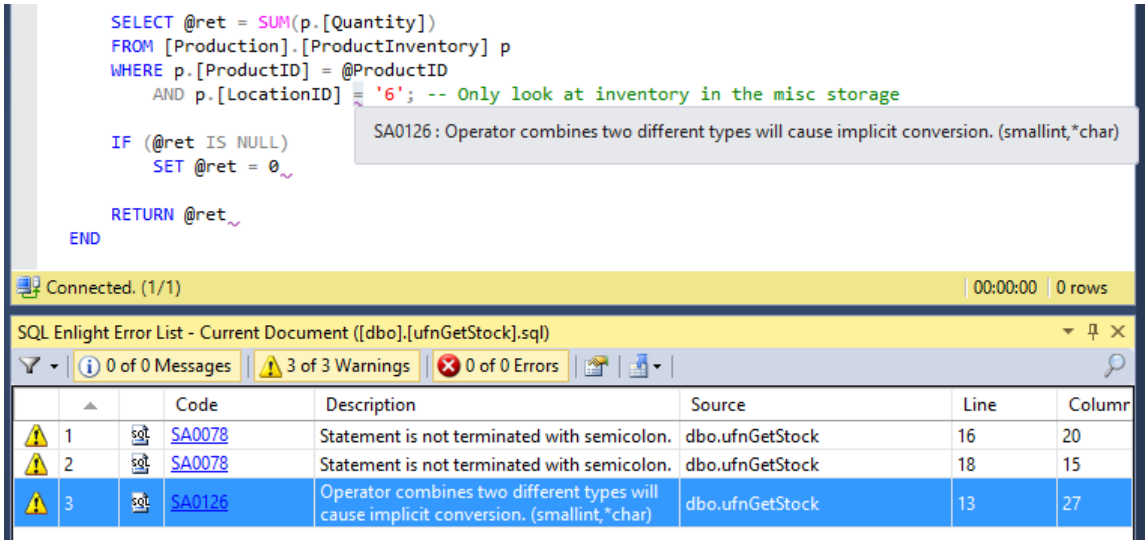


Ilustración 5: Vista de unos errores encontrados a través de SQL Enlight desde la vista del editor SQL Server Management Studio. Imagen tomada de [28].

Nota: *La herramienta SQL Enlight no es gratuita y, dadas las circunstancias del componente que estamos considerando, su uso o no dependerá expresamente de un análisis realizado para el proyecto concreto sobre el que se esté llevando a cabo la solución.*

Herramientas para el orquestador OSB

De nuevo, al igual que sucedía con los *linters*, **no existen herramientas** específicas que nos permitan realizar un análisis estático del código realizado en el OSB. Será **responsabilidad de los desarrolladores analizar y desarrollar de manera correcta** el código de este componente.

La estrategia para llevarlo a cabo y mitigar los posibles errores humanos asociados va a ser la misma que se ha empleado en el caso del *linteo*: aplicar *pair programming* y realizar *merge request* a la hora de completar los cambios en dicho componente.

	FRONTAL (ANGULAR)	**SERVICIOS WEB (C# .NET)**	**OSB**	**BBDD**
VERIFICACIÓN ESTILÍSTICA	***Prettier***	**Integrada *Visual Studio***	**/**	***SQLFluff / SQL Enlight***
VERIFICACIÓN FUNCIONAL	***ESLint***	**Integrada *Visual Studio* + *SonarQube***	**/**	***SQL Enlight***

Ilustración 6: Tabla en la qu se resumen las diferentes herramientas de calidad de código propuestas para la verificación de los diferentes componentes de la solución.

Capítulo 3 – *Testing*

3.1. Introducción

A lo largo de este capítulo se van a definir un conjunto de procesos que van a permitir establecer dentro del proceso de desarrollo de nuestra aplicación, **la validación de su correcto desempleo funcional**. Esta validación se va a realizar a través de un conjunto de pruebas de código.

Adicionalmente, se van a listar las diferentes **herramientas** necesarias para su codificación y posterior replicado en el caso de la solución que estamos evaluando. Mediante estas herramientas, posteriormente se realizará una tarea de automatización que permitirá realizar las pruebas sin necesidad de que el desarrollador intervenga.

La aplicación de estos procesos y herramientas finalmente se traduce en una **mejora de la calidad del sistema** (proveniente de la reducción de errores) y en una **mejora de los procesos de desarrollo** (en los que se tiene un mayor grado de confianza por la realización de las pruebas).

A lo largo del capítulo se van a considerar los siguientes **requisitos funcionales**:

- Simplificación de los procesos de desarrollo y mantenimiento
- Disminución de errores (tanto de producto como de desarrollo)
- Definición y seguimiento de estándares de calidad de código
- Desarrollo y aplicación de pruebas

    - Pruebas unitarias
    - Pruebas de integración
    - Pruebas de Interfaz de usuario (UI)
    - Pruebas funcionales
    - Pruebas de carga y rendimiento
- Permisibilidad de aplicación de metodologías ágiles
    - Integración y despliegue continuos

A su vez, también se van a considerar los siguientes **requisitos no funcionales**:

- Aplicabilidad en diferentes formatos de tecnologías web
    - Simple Object Access Protocol (SOAP)
    - API REST (representational state transfer)
- Empleo de tecnologías desarrolladas por Microsoft
    - *Visual Studio*
    - *Visual Studio Code*
    - Microsoft SQL Management Studio
    - IIS Express
    - Windows 10
- Empleo del *Oracle Service Bus* (OSB)
- Empleo de sistemas de BBDD relacionales

3.2. Conceptos previos

Introducción al testing

En el desarrollo software se conoce como ***testing*** al proceso que tiene como finalidad **verificar el correcto desempeño** de los productos software desarrollados. Esta verificación se realiza a través de unos casos de uso definidos para los cuales se esperan unos resultados concretos. Durante su ejecución, los resultados obtenidos serán comparados con los esperando dando finalmente por válido o no el test en función de esta comparativa.

Mediante la aplicación de este proceso se pretenden obtener los siguientes objetivos:

- **Identificación defectos en el software**. Los *test* permiten encontrar fallos que podrían no identificarse de otra forma.
- **Certificación de la calidad producto entregado**. Los *test* permiten asegurar que el producto funciona debidamente en los casos de uso probados.
- **Definición de límites del producto**. A través del uso de casos de uso

extremos, es posible descubrir dónde están los límites funcionales del software probado.

- **Cumplimiento de normativas relacionadas con la seguridad.** La adhesión de test durante el proceso de desarrollo está recogida dentro de la mayoría de normativas relacionadas con la seguridad.
- **Verificación del cumplimiento de requisitos del producto.** Si se definen unos *test* que prueben los diferentes requisitos funcionales del producto, con la ejecución de dichos test se verifica de manera sencilla el cumplimiento de dichos requisitos.
- **Facilitación del proceso desarrollo de nuevas funcionalidades.** El desarrollo de los *test* libera a los desarrolladores de la necesidad de realizar constantes pruebas manuales durante el proceso de desarrollo.

El *testing* se debe encontrar **presente a lo largo de toda la vida del ciclo de desarrollo** de un producto software. Comienza en las primeras fases, donde se definen los requisitos, continua a lo largo de la implementación (realizando pruebas durante el desarrollo) y sigue desempeñando un papel importante una vez el producto está desplegado en producción usándose para comprobar que toda la operativa sigue funcionando correctamente tras diversos eventos.

Tipos de pruebas

Dependiendo del enfoque que tengan las pruebas, éstas se pueden agrupar en diferentes grupos definidos. Algunos de estos grupos son:

- **Pruebas unitarias**. Son pruebas muy sencillas, pequeñas y rápidas basadas en probar funcionalidades concretas.
- **Pruebas de integración**. Son las pruebas enfocadas al correcto ensamblado de los diferentes componentes que conforman un sistema.
- **Pruebas de interfaz de usuario**. Son pruebas realizadas desde la interfaz de usuario en las que se busca evaluar el comportamiento funcional al completo de la aplicación.
- **Pruebas de carga y estrés**. Son pruebas cuyo objetivo es descubrir los límites de la aplicación en términos de capacidad de procesado.
- **Pruebas de usabilidad**. Son pruebas cuyo objetivo es evaluar el grado de usabilidad de la aplicación.
- **Pruebas de rendimiento**. Son pruebas cuyo objetivo es medir el rendimiento y escalado en términos de velocidad, eficiencia y recursos utilizados.
- **Pruebas de continuidad**. Son pruebas cuyo objetivo es medir la capacidad de recuperación del sistema en caso de fallos o interrupciones inesperadas.

El desarrollo de este trabajo se va a centrar en los 4 primeros tipos principales:

pruebas unitarias, pruebas de integración, pruebas de interfaz de usuario y pruebas de carga y estrés.

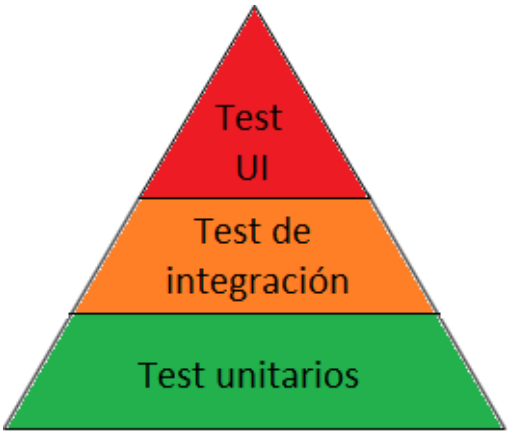


Ilustración 7: Diagrama que muestra la proporcionalidad y el orden de los tipos test unitarios, integración e interfaz de usuario. En él se aprecia que los test unitarios son los más frecuentes y forman la base de testeo seguidos de los test de integración y por último los de interfaz de usuario.

Etapas y codificación de los test

Las **estrategias** y formas de afrontar e implementar el **proceso de pruebas** varían enormemente según la naturaleza del proyecto en cuestión. Por lo general, este proceso está marcado por tres **fases** separadas:

1. Definición de los casos de prueba.
2. Elaboración, preparación y/o implementación de los casos de prueba.
3. Ejecución de las pruebas.

Cada *test*, está compuesto principalmente de tres **etapas** las cuales se deben ejecutar cada vez que se aplica dicha prueba:

1. Preparación de los datos para la prueba.
2. Ejecución de las tareas a probar.
3. Verificación de los resultados obtenidos.

```
[Test]
0 references | 0 changes | 0 authors, 0 changes
public void TestDeEjemplo()
{
    // 1. Preparación de los datos para la prueba
    var datosNecesariosTest = PreraraDatosPrueba();

    // 2. Ejecución de la tarea a probar
    var resultados = FunciónQueSeQuiereProbar(datosNecesariosTest);

    // 3. Verificación de los resultados obtenidos
    Assert.AreEqual(3, resultados.Length);
}
```

Ilustración 8: Ejemplo de test unitario básico desarrollado en C# a través de la librería NUnit, en el que se aprecian las diferentes etapas que componen una prueba.

Es muy habitual el uso de **librerías diseñadas para la realización de test** que permiten configurar las pruebas permitiendo así su reproducibilidad en el futuro. Por lo general, el uso de estas librerías suele requerir de una serie de conocimientos técnicos.

Estos requerimientos se intensifican para aquellas librerías que se alejan del frontal del producto, en donde las pruebas deben realizarse a nivel de código (ejemplos de estas librerías son *JUnit* [36] para JAVA, *NUnit* [37] para C#, *PyTest* [38] para Python…).

/

98.36% Statements 1859/1890 **93.92%** Branches 726/773 **100%** Functions 266/266 **99.73%** Lines 1816/1821 **37 statements, 21 branches** Ignored

File		Statements		Branches		Functions		Lines	
express/		100%	1/1	100%	0/0	100%	0/0	100%	1/1
express/examples/auth/		93.75%	75/80	80.77%	21/26	100%	17/17	100%	71/71
express/examples/content-negotiation/		100%	32/32	100%	2/2	100%	13/13	100%	32/32
express/examples/cookie-sessions/		100%	12/12	100%	4/4	100%	1/1	100%	12/12
express/examples/cookies/		95.83%	23/24	87.5%	7/8	100%	3/3	100%	22/22
express/examples/downloads/		94.12%	16/17	87.5%	7/8	100%	3/3	100%	15/15
express/examples/ejs/		100%	11/11	100%	2/2	100%	1/1	100%	11/11
express/examples/error-pages/		97.14%	34/35	83.33%	10/12	100%	6/6	100%	34/34
express/examples/error/		90%	18/20	66.67%	4/6	100%	4/4	100%	18/18
express/examples/markdown/		95.24%	20/21	66.67%	4/6	100%	5/5	100%	20/20
express/examples/multi-router/		100%	9/9	100%	2/2	100%	1/1	100%	9/9
express/examples/multi-router/controllers/		100%	14/14	100%	0/0	100%	4/4	100%	14/14
express/examples/mvc/		95.45%	42/44	80%	8/10	100%	4/4	100%	42/42
express/examples/mvc/controllers/main/		100%	2/2	100%	0/0	100%	1/1	100%	2/2
express/examples/mvc/controllers/pet/		94.12%	16/17	50%	1/2	100%	4/4	100%	16/16
express/examples/mvc/controllers/user-pet/		92.86%	13/14	50%	1/2	100%	1/1	100%	13/13
express/examples/mvc/controllers/user/		100%	21/21	100%	4/4	100%	6/6	100%	19/19
express/examples/mvc/lib/		97.96%	48/49	80%	20/25	100%	2/2	100%	46/46

Ilustración 9: Ejemplo de reporte de cobertura de código tomado a través de la herramienta Istanbul Code Coverage [39]. La imagen está tomada de [40].

Proceso de definición de los casos de prueba

Para los diferentes casos de prueba, se debe tener en cuenta el momento en el que se encuentra el proyecto. Al comienzo de un desarrollo, se **deben considerar** los **requisitos del proyecto**. Posteriormente, conforme se vayan desarrollando las diferentes **funcionalidades**, estos casos de prueba deben ir actualizándose y ampliándose para que los *test* también recojan las nuevas funcionalidades desarrolladas.

Se debe prestar especial atención sobre el quien evalúa y genera los diferentes casos de prueba. Por lo general, los propios desarrolladores no son quienes deben los casos de uso. Si lo hiciesen, se podrían presentar los siguientes problemas:

- **Falta de perspectiva externa**. Por lo general, los desarrolladores suelen encontrarse demasiado familiarizados con el código que desarrollan y su lógica, pudiendo provocar que se pasen por alto (de manera involuntaria) algunos escenarios de prueba que no sean obvios (casos de error, extremos, etc.).

- **Falta de profundidad**. La confianza y el conocimiento del propio código, pueden provocar una falta de profundidad de las pruebas pudiendo incluso llegarse a omitir casos de prueba difíciles o poco comunes debido a la escasa probabilidad de aparición de los mismos en un caso de uso real.
- **Errores de definición**. Suponiendo el caso de que se haya desarrollado una funcionalidad cuya definición sea poco precisa se puede provocar que sólo se realicen pruebas que abarquen aquellos casos de uso para los que se sabe que el código tiene que operar de una forma concreta, dejando gran cantidad de casos sin probar.

Para evitar este tipo de problemáticas, lo más recomendable es mantener de manera **estanca y separada la definición de las pruebas del código del desarrollo** de los propios *test*. Para ello, es habitual que las pruebas se definan a través de **terceros**, como podrían ser *Product Owners* o directamente desde negocio.

Esta forma de operar encaja muy bien con la metodología SCRUM, en donde los *Product Owners* deben definir, por cada historia a desarrollar, una serie de criterios de aceptación (los cuales, en el fondo marcan los casos de prueba).

Si las dimensiones del equipo lo permiten, es posible que exista un **equipo de control de calidad (QA)** el cual tome dichas pruebas y las amplifiquen. En el caso de que exista dicho equipo, también es recomendable que sean ellos quienes realice la codificación de las pruebas. De esta forma se aíslan todavía más las pruebas del desarrollo del producto.

No obstante, este tipo de enfoques que implican a terceros (equipo QA) en el proceso de desarrollo del producto, aunque aportan ciertas ventajas, suelen ir en **contra de las metodologías ágiles**. Si se quiere seguir una metodología de este tipo, sólo resulta interesante esta posibilidad si se dan ciertas circunstancias concretas tales como: una alta inestabilidad del producto o cierta tendencia a hacia la generación de errores.

Si ninguna de las opciones anteriores es posible, es recomendable que sea un **miembro del equipo diferente quien diseñe y realice los *test* de quien desarrolla la funcionalidad** en cuestión. A partir de la misma especificación funcional, del mismo modo que se desarrolla el código que implementa la funcionalidad, también debe desarrollarse el test que la valide.

Estas técnicas pueden combinarse con **la definición de las pruebas de manera previa al desarrollo** de la funcionalidad, mejorando así la independencia entre el diseño de las pruebas y el desarrollo de la funcionalidad.

Introducción a la cobertura de código y su uso

Para tratar de paliar la falta de profundidad de las pruebas, resulta interesante el concepto de **cobertura de código** (*test coverage* en el argot) [41]. La cobertura de código es una medida que identifica el porcentaje de líneas que están siendo evaluadas a través de *tests*.

Existen muchas herramientas diseñadas para evaluar los casos de prueba de las aplicaciones y detectar **que líneas de código se están probando y cuáles no** a través de los *test*. Este dato suele ayudar a dar una buena perspectiva de la profundidad de los casos de prueba existentes, pudiendo así ajustar los mismos en el caso de que sean insuficientes o excesivos.

La existencia del concepto de cobertura de código y la búsqueda de la calidad del proceso de testeo, muchas veces lleva a pensar en la necesidad de conseguir un 100% de código cubierto. Sin embargo, esta práctica es muy negativa.

En primer lugar, **disponer de una cobertura total del código no garantiza que el código opere correctamente** en todas las situaciones posibles (por lo que se estaría obteniendo una falsa sensación de alta seguridad). Además, mantener todas las casuísticas de un código en evolución requiere de una cantidad de recursos desmesurada, lo que alagaría en sobre manera le producción del código.

La mejor forma de emplear la cobertura de código reside en encontrar un **equilibrio entre el esfuerzo requerido y el porcentaje de código probado**. En este sentido, se debe priorizar aquellas piezas de código que sean más relevantes para el sistema. De esta forma se optimiza la cantidad de recursos invertidos junto con el beneficio obtenido sacándole el máximo partido a las pruebas.

Planes de ejecución

Como algunas de las pruebas pueden resultar tediosas y/o largas de realizar, es habitual tener **planes de pruebas** para que, en función de la acción que se esté realizando con el producto (una nueva funcionalidad, un fallo de código, un despliegue…) se lancen la cantidad de pruebas justa y necesaria para cubrir el cambio en el ámbito en el que se encuentra (por ejemplo, ejecutar sólo *test* unitarios, ejecutar tanto unitarios como integración, etc.).

Un ejemplo de esto sería durante el desarrollo de una funcionalidad que abarque un único módulo. En este caso se lanzarían únicamente los *test* que abarquen dicho módulo. Posteriormente, cuando el módulo se vaya a desplegar junto con el resto de componentes, se lanzarán las pruebas que abarquen al resto de elementos del producto comprobando así el resto de aspectos del sistema.

De manera análoga, no todos los componentes requieren ni la misma profundidad, ni el mismo tipo de pruebas. Los sectores más críticos de las aplicaciones son los que deben abarcan la mayor cantidad de las pruebas, tanto en número de casos de prueba, como en tipos de pruebas a aplicar.

3.3. Pruebas unitarias

Introducción a los test unitarios

Las **pruebas unitarias** son la unidad mínima de testeo de un módulo y forman la base de **confianza funcional** de dicho módulo. Estas pruebas se realizan **directamente sobre el código fuente** el cual se encuentra encapsulado en funciones. Cada uno de ellos debe estar expresamente diseñado para evaluar una única funcionalidad o comportamiento en un escenario concreto (representan la parte más pequeña de un programa que puede ser probada de manera independiente).

Su objetivo principal es asegurar que, para un componente, cada una de sus funcionalidades opera de manera correcta en diferentes escenarios. Estas validaciones se deben realizar de manera individual y estanca (es decir, eliminando cualquier tipo de dependencia ajena a la lógica del propio componente).

Aislamiento modular y mocks

Para conseguir los objetivos de las pruebas, podría ser necesario **aislar ciertas** partes del código y **eliminar las comunicaciones** con el resto componentes y módulos. Esto sucede, por ejemplo, en el caso en el que se quiera probar una funcionalidad que recoja datos de una BD.

En ningún caso durante la realización de una prueba unitaria se debe establecer una conexión con ningún componente diferente al probado. En su lugar, este tipo de interacciones deben ser sustituidas por un conjunto de **componentes falsos** que se limiten sencillamente a devolver respuestas prefabricadas simulando así el comportamiento del módulo global a sustituir.

Estos componentes falsos pueden definirse dentro de los propios *test* y se les conoce como ***mocks***. Existen diversas librerías para que, en función de la tecnología empleada, se puedan llevar a cabo este tipo de acciones. En el caso de C#, destaca la librería Moq [42].

La facilidad con la que se pueden aplicar los *mocks* depende de la forma en la que se haya desarrollado el código. En este sentido es habitual desarrollar el código utilizando

inyección de dependencias. El uso de esta técnica permite que el empleo de *mocks* se realice a través de una mera configuración. Esto convierte al uso de inyección de dependencias en estratégico para facilitar el testeo del software.

Características y ventajas de los tests unitarios

Otra característica de los *tests* unitarios es su **velocidad y posibilidad de automatización**. Para ello, es necesario mantener el tamaño interno de cada una de las pruebas pequeño (manteniendo su lógica de unidad y probando una única funcionalidad por *test*).

Generalmente las pruebas se realizan a través de herramientas que permiten su reproducibilidad de manera sencilla incluso en **herramientas de automatización** ligadas con los procesos de desarrollo.

Algunas de las **ventajas** de este tipo de pruebas son las siguientes:

- Son sencillos, ágiles y requieren de escasos recursos.
- Permiten la detección temprana de errores en un marco de trabajo ágil.
- Permiten aumentar la calidad y robustez del producto, así como el cumplimiento de estándares.
- Facilitan los cambios y la integración de los mismos simplificando el proceso de prueba del sistema.

Para conseguir estas ventajas, es indispensable mantener los test actualizados siempre a la versión funcional del sistema más reciente.

3.3.1. Cómo implementarlos en el sistema

Dadas sus características, los *test* unitarios, deben ser aplicados en **todos los componentes** posibles. Se deberá prestar especial atención en no romper los principios de aislamiento, rapidez y sencillez característicos de este tipo de pruebas.

Durante la implementación y ejecución de los diferentes *test* unitarios se va a evaluar la cantidad de cobertura de código de los diferentes componentes desarrollados. En función de los datos obtenidos podría llegar a ser interesante aumentar la cantidad de datos de prueba de cada uno de los componentes. La forma de aplicar estos test varía para cada uno de los componentes de la solución estudiada.

Implementación de tests en el frontal (aplicación Angular)

De acuerdo al sistema presentado, está operativa de implementación de *test* unitarios

se va a aplicar sobre la aplicación web que conforma el **frontal**, desarrollada sobre el *framework* **Angular**.

La mayor dificultad de los test unitarios de este componente reside en saber **que partes del código son las que se deben probar**. Sólo se deben probar aquellas partes que impliquen funcionalidades propias del frontal, como, por ejemplo, las validaciones de los diferentes campos.

En ningún caso se debe probar el envío ni la recepción de dichos datos a otros componentes (esta prueba formará parte de las pruebas de integración). Tampoco se deben tener en cuenta las diferentes interacciones de interfaz de la aplicación, puesto que este tipo pruebas formarán parte de otro set (pruebas de UI).

Implementación de tests en servicio web API REST

A la hora de considerarlos **test del servicio web API REST**, de nuevo se deben tener muy presente cuáles son los aspectos que se deben incluir en los propios test. La funcionalidad general de este componente va a consistir ser en un punto de entrada que conecta la aplicación frontal de nuestro sistema con la parte *backend*.

Aunque no es habitual, según como se implemente la arquitectura de la aplicación, se podría a llegar a dar el caso de que este componente fuese un mero redireccionador. Si se da esta situación, no se deben realizar pruebas unitarias para el componente (puesto que no se está implementando una lógica per sé en él).

En el caso de que sí que se implementase cualquier tipo de **lógica dentro del servicio** (que es lo más común), es muy importante que las pruebas se hagan llamando de manera interna al código del componente (es decir, sin tener el proyecto web levantado ni activo) y que se utilicen *mocks* cuando así se requiera. En caso contrario, estaríamos perdiendo tanto el principio de aislamiento como el de velocidad de los test unitarios.

Implementación de tests en un orquestador OSB

La realización de pruebas unitarias sobre un **orquestador *Oracle Service Bus* (OSB)** no es trivial debido a que se trata de un componente web de infraestructura autónoma e independiente (no es posible acceder al código fuente del componente, por lo que se depende de las opciones que ofrece nativamente).

Se trata de un componente cuyo objetivo principal es mantener la operatividad del sistema en momentos de alta demanda, pero más allá de esta funcionalidad, **presenta grandes limitaciones** (como su no preparación para la realización de pruebas unitarias o la imposibilidad de desplegarlo de manera automatizada).

Internamente es un componente que se encarga de gestionar una cola de peticiones y de procesar dichas peticiones realizando otras peticiones al servicio web SOAP .NET. En ocasiones puede llegar a incluir parte de **lógica de negocio** (por ejemplo, realizando varias llamadas y trabajando con sus resultados). Son únicamente este último tipo comportamientos los que deberían ser probados mediante pruebas unitarias de este módulo y se debe tener mucho cuidado para mantener su aislamiento.

Dada la operativa de la herramienta (que no está diseñada para este propósito de *testing*) y de la no disposición del código fuente, resulta muy complejo el ser capaces de aislar debidamente este componente de los diferentes servicios. No existe una forma nativa que permita ni realizar pruebas ni inyectar *mocks* al componente.

Para remediarlo, una posible manera solución sería utilizar una **variable en la cabecera** de las peticiones que nos indicase que se trata de un *test* para que, en vez de realizar las llamadas al resto de componentes, a nivel interno, desde la parte que sí que podemos configurar) se devuelvan algunos datos por defecto.

Para **automatizar estas pruebas**, se pueden realizar desde el exterior peticiones con el *flag* de testeo marcado. Hay que destacar que, si se atiende a la definición estricta de *test* unitarios, esta operativa **no se corresponde exactamente un test unitario** debido a que se realizan peticiones desde un servicio externo hacia el OSB, perdiendo así el principio de aislamiento.

Por desgracia, dadas las características del producto y sus limitaciones, esta forma de operar representa la **manera más cercana disponible** de realizar un *test* unitario para el orquestador OSB.

***Nota**: El Oracle Service Bus no está expresamente diseñado para la realización de este tipo de pruebas sobre él, y no vamos a disponer de los datos de cobertura sobre este componente.*

Implementación de tests en un servicio web SOAP .NET

El **servicio web SOAP .NET** será el componente sobre el que se van a implementar la mayoría de las pruebas unitarias debido a su **funcionalidad** consistente en obtener la información a través de consultas a la BD y procesarla. Esta operativa contiene una gran carga funcional que debe ser debidamente validada.

A la hora de realizar los tests, se debe de tener cuidado para que **en ningún momento se realice una conexión** con ningún otro componente (PE, la BD debe estar *mockeada*). También se debe tener cuidado de que, al acceder al código para lanzar las pruebas, el acceso no haga través de una petición SOAP, sino que se efectúe de manera directa desde el código fuente.

Implementación de tests en una BD

Teniendo en cuenta que la base de datos es un componente cuyo **objetivo es almacenar información** y en ningún caso va a incluir ningún tipo de funcionalidad, este **tipo de test no se deben aplicar** sobre este componente.

3.3.2. Herramientas

Para la implementación de las pruebas unitarias va a ser necesario utilizar un conjunto de herramientas diferentes que se ajusten a las tecnologías sobre las que están construidos cada uno de los diferentes módulos de nuestro sistema.

Herramientas de testeo unitario en el frontal (aplicación Angular)

Para probar aplicaciones desarrolladas sobre el ***framework* Angular** (el caso del frontal de nuestro sistema), comúnmente, como principal alternativa se utilizan las herramientas *Jasmine* [43] y *Karma* [44].

Jasmine* es la *suite* de *testing sobre la que se desarrollarán las pruebas (la librería base) y ***Karma* es el ejecutor** desarrollado por los propios ingenieros de Angular cuya finalidad es ejecutar los *test* en dicho entorno. *Karma* dispone de una **versión CLI** de tal forma que permite que los test de Jasmine sean ejecutados a través de una consola de comandos.

Estas tecnologías disponen de diversos *plugins*, como por ejemplo *Karma Test Explorer* [45], que facilitan **integración con *Visual Studio Code*** (el IDE que se utilizaría para el desarrollo de este componente). Mediante estas integraciones es posible visualizar tanto los resultados de ejecución, como un árbol con los test disponibles permitiendo además su ejecución.

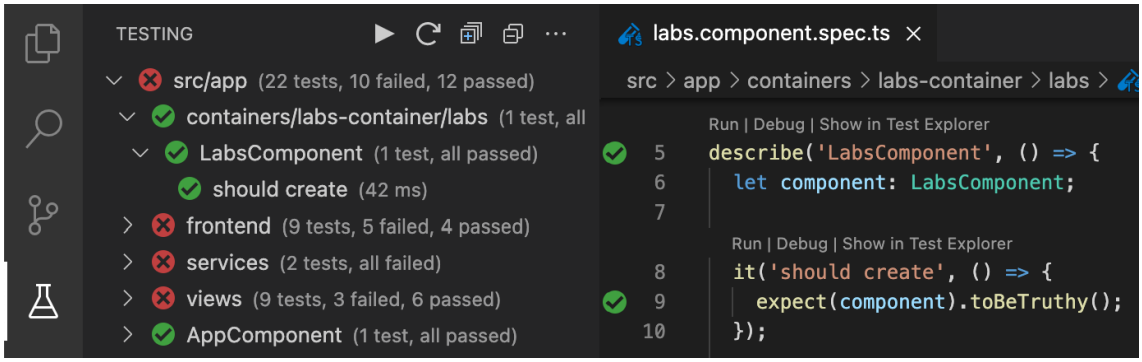


Ilustración 10: Vista resumen de la ejecución de un conjunto de test realizados para Angular a través del plugin Karma Test Explorer. La ilustración proviene de [45].

Herramientas de testeo unitario en backend desarrolladas en C# .NET (servicios web)

Para los **componentes *backend* desarrollados en C# .NET**, la librería de desarrollo de *testing* más recomendada es ***NUnit*** [37]. *NUnit* es un *framework* de código abierto diseñado para la codificación de pruebas unitarias.

NUnit se puede **integrar** de manera sencilla a través de un *plugin* [46] **con *Visual Studio***. A través de esta integración va a ser posible visualizar los diferentes *test* disponibles, ejecutarlos y observar los diferentes resultados obtenidos.

Además, *NUnit* dispone de una **aplicación comandos de consola** lo que permite su integración con autómatas (*Jenkins* para la solución que se está evaluando en este trabajo).

Para asegurar la estanquidad de las pruebas en un único componente y realizar los diferentes ***mocks***, es posible utilizar la herramienta ***Moq*** [42], otro *framework* de código abierto diseñado para dicho propósito.

Herramientas de testeo unitario para orquestadores (OSB)

El orquestador ***Oracle Service Bus* no está diseñada para permitir ser probada** a través de un framework de testeo de manera nativa. La herramienta **no dispone de un sistema que permita añadir *mocks*** de manera sencilla permitiendo así aislar su componente del resto.

Con estas condiciones, no es posible afrontar un *test* unitario de manera sencilla. No obstante, como este componente puede llegar a implementar lógica de negocio resulta importante plantear un acercamiento de dichos test para este componente.

Para solucionar la no posibilidad de la realización de pruebas, se pueden realizar, por cada *test*, **las llamadas de manera externa** (desde otro componente), a través de la librería *NUnit*, la misma que se emplea empleada para el resto de componentes *backend*. Adicionalmente, para permitir la realización de *mocks,* se van a añadir, dentro de la lógica interna del módulo, diferentes flujos que en función de un *flag* deriven en una llamada o una respuesta por defecto.

Este tipo de **operativa permite aplicar pruebas** que, aunque no sean unitarias al completo, permiten asegurar que la lógica interna que se encuentra en el OSB funciona de manera correcta. Algunos problemas y desventajas de emplear esta operativa de pruebas son:

- Teóricamente hablando no se están realizando test unitarios.
    - No se aplican en un entorno 100% estanco.
    - No son especialmente ágiles.

- Se prueba la funcionalidad a grandes rasgos, no en pequeñas funcionalidades.

- Definir los resultados esperados puede es complejo de implementar.
- Resulta muy costoso en términos de complejidad aumentar los casos de prueba.

Obtención de la cobertura de código

Para permitir la evaluación del ***code coverage*** de cada uno de los módulos, se van diferenciar los productos en dos grandes grupos:

- **Proyectos cuyos *test* están basados en NUnit y programados a través de *Visual Studio* (*backend* salvo OSB)**. Para estos proyectos *Visual Studio*, en su versión *Enterprise*, ofrece de manera nativa la posibilidad de evaluar esta propiedad de manera nativa [47].

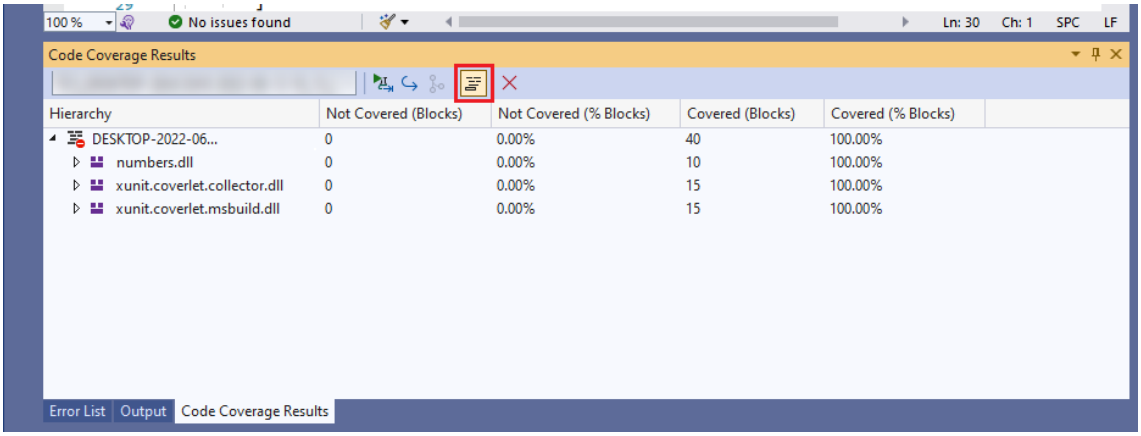


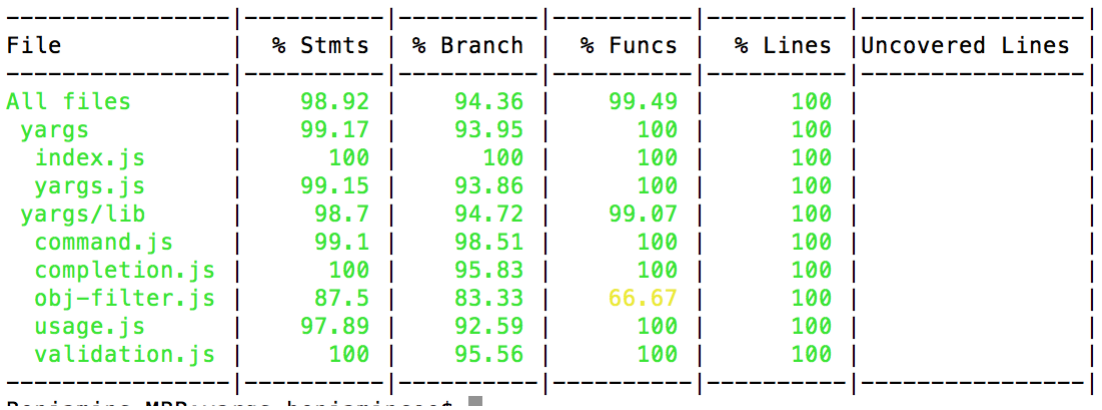
Ilustración 11: Pestaña resultados de code coverage de la implementación nativa en Visual Studio Enterprise edition. La imagen está tomada de la documentación del propio componente [47].

De manera alternativa, en el caso de que no sea posible el empleo de la versión *Enterprise* de *Visual Studio*, existe la posibilidad de utilizar la herramienta *NCover* [48]. Esta herramienta permite obtener el *code coverage* y generar reportes con dichos resultados. Dispone de una interfaz gráfica para facilitar su uso al usuario.

- **Proyectos cuyos test están basados en Jasmine y Karma (frontal)**. En este caso, las propias herramientas de *testing* utilizadas están diseñadas para utilizar e incluyen de manera nativa la herramienta *Istanbul Code Coverage* [39] para medir la cobertura.

  Se trata de una herramienta de código abierto que nos va a permitir, tras la ejecución de los *test*, explorar los resultados de cobertura de código obtenidos. Los resultados se pueden apreciar desde la terminal de comandos, o de manera gráfica a través de un fichero de reporte que por defecto se guardará en la ruta:

{path_proyecto}/coverage/index.html.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	98.92	94.36	99.49	100	
yargs	99.17	93.95	100	100	
index.js	100	100	100	100	
yargs.js	99.15	93.86	100	100	
yargs/lib	98.7	94.72	99.07	100	
command.js	99.1	98.51	100	100	
completion.js	100	95.83	100	100	
obj-filter.js	87.5	83.33	66.67	100	
usage.js	97.89	92.59	100	100	
validation.js	100	95.56	100	100	

Ilustración 12: Visualización de los resultados de code coverage obtenidos a través de la herramienta Istanbul. La imagen está tomada de la página web de la herramienta [49].

La evaluación del ***test coverage*** no requiere su inserción en el **proceso de integración continua**. Se trata de una ayuda para los desarrolladores que permite evaluar si los casos de prueba existentes para su producto son suficientes o no. Es por eso por lo que, al contrario que con el resto de las herramientas, gana relevancia la posibilidad de visualizar los resultados de manera gráfica frente a su ejecución desde terminal.

Nota: *Debido a sus limitaciones y a la operativa que se está aplicando para realizar las pruebas en el orquestador, no es posible obtener los datos de code coverage de dicho componente.*

3.4. Pruebas de integración

Introducción a las pruebas de integración

Las **pruebas de integración** constituyen el siguiente paso lógico, dentro del proceso incremental de prueba de un producto software, a las pruebas unitarias. Su objetivo es, una vez comprobado el correcto funcionamiento de los componentes de manera individual (a través de *test* unitarios), **verificar el correcto funcionamiento de dichos módulos en grupo**.

Para ello, es necesario que cada uno de los módulos se encuentre debidamente aislado para evitar interferencias entre ellos. A la hora de lanzar las pruebas, éstas no se aplican a partir del código fuente, sino que se ejecutan directamente a nivel de componente.

Aunque su principal foco es la detección de defectos producidos por la integración de dos o más módulos software, **también son útiles para evaluar el ajuste y**

funcionamiento general del sistema (evaluar, en términos generales, si el comportamiento funcional del sistema es correcto). Al igual que pasaba con los *test* unitarios, resulta positivo que estas pruebas queden automatizadas para facilitar su reproducibilidad en el futuro.

Características y ventajas de las pruebas de integración

Los *test* de integración son complejos y requieren de una **gran cantidad de recursos** para ser ejecutados. Durante estas pruebas, se deben mantener varios componentes activos y se debe tener especial cuidado en que una prueba no afecte al rendimiento de la otra.

Los recursos consumidos se ven reflejados tanto en términos de memoria, como temporales. Para la ejecución de un test de este tipo, previamente, **deben levantarse en paralelo todos los productos a probar**, después ejecutarse las pruebas (que al requerir la comunicación entre componentes van a ser más lentas que las unitarias) y por último cerrarse los componentes.

Las principales **ventajas** del uso de este tipo de pruebas son:

- Aseguran el correcto ensamblado de módulos programados incluso aunque estos implementen diferentes lógicas y o tecnologías.
- Inhabilitan la pérdida de información relevante a través de las comunicaciones.
- Amplían la cobertura de pruebas funcionales de los *test* unitarios, mejorando la robustez de las aplicaciones.

Enfoques para la planificación de pruebas de integración

A la hora de la planificación de las pruebas de integración, existen diferentes enfoques teóricos para afrontar dicho problema:

- **Pruebas de integración incrementales**. Es un enfoque que, como su nombre indica, consiste en partir de un módulo base ir añadiendo componente a componente, los diferentes módulos del sistema hasta llegar al sistema completo.
  La estrategia se puede enfocar de manera ascendente (de menor número de componentes a mayor), de manera descendente (de mayor número de componentes a menor) o en *sándwich* (una mezcla de las anteriores).
- **Pruebas de integración *Big Bang***. En este enfoque no se pretende comprobar cada una de las interacciones entre los componentes una a una, sino que su objetivo es probar el sistema de una única tacada, considerando toda la solución como una unidad global única (independientemente del número de módulos que la compongan).

3.4.1. Cómo implementarlos en el sistema

La clave para una correcto diseño e implementación de los test de integración es tener claro cuáles son los **aspectos que se quieren probar** en cada caso de prueba y establecer cuál es la estrategia de definición de pruebas.

Estrategia de definición de pruebas

Dada la arquitectura de la solución práctica que se está evaluando, la estrategia de definición de pruebas que más se amolda es una **incremental ascendente**. Esto se debe a que, mediante esta técnica, es posible hacer los mismos saltos que hace la información en el sistema (naciendo en la BD, y sufriendo transformaciones hasta llegar al frontal del sistema).

Siguiendo esta estrategia, se va a comenzar en los puntos más básicos de la solución: la BD y el servicio web SOAP que la consulta. Tras la realización de las pruebas pertinentes, se añadirá el orquestador OSB. Después se añadirá el servicio web API REST y, por último, la aplicación frontal.

Pruebas de integración BD + servicio web SOAP

Para la realización de esta prueba, en primer lugar, es **necesario definir y establecer los datos de prueba** dentro de la unidad mínima del sistema (en este caso la BD). Una vez identificados y establecidos estos datos, se va a proceder al lanzado de las pruebas.

Para ello en se van a lanzar una **batería de pruebas** contra en servicio web SOAP encargado de extraer los datos de la BBDD (el servicio debe estar enlazado con la BBDD real). Durante esta prueba se debe comprobar que se obtienen todos los datos necesarios y que no se producen pérdidas de información.

Nota: *Nótese que, a este nivel, es posible que los datos ya no concuerden exactamente con los originales ya que podrían haber sufrido modificaciones debidas a la lógica del negocio.*

Pruebas de integración BD + servicio web SOAP + orquestador

Tras verificar el correcto funcionamiento de los dos primeros componentes, se debe dar el salto al orquestador. **La lógica a aplicar es exactamente la misma** que para los componentes ya probados, pero partiendo desde otro componente: hacer peticiones al orquestador, el cual se comunicará con el servicio web SOAP, quien que en última instancia se conectará con la BD.

De nuevo, **se deberá comprobar que se obtienen todos los datos requeridos** y que no

se producen pérdidas de información. Al igual que con la primera batería de pruebas, llegados a este punto los datos pueden no concordar con los originales almacenados en la BD al haber sufrido transformaciones lógicas fruto de su paso por los diferentes módulos.

Nota: *Nótese, que en este caso no hay que utilizar el flag propuesto para los test unitarios de este componente, porque precisamente, lo que queremos probar, es la integración con el resto de componentes.*

Pruebas de integración BD + servicio web SOAP + orquestador + servicio web API REST

Tras verificar el correcto funcionamiento de los tres primeros componentes, se debe dar el salto al servicio web API REST. De nuevo, el **modus operandi es el mismo**: hacer un conjunto de peticiones REST a dicho componente y comprobar que los resultados obtenidos concuerdan con los esperados tras la lógica de negocio derivada del viaje de los datos por los diferentes componentes.

Nótese que, al ser este el punto de entrada desde el exterior al sistema, llegados a este punto **se podrían llegar a requerir ciertos permisos para aplicar algunas operaciones**. Para ello, es necesario generar unas credenciales de acceso al sistema para el autómata que se encargará de realizar las pruebas.

Nota: *Las credenciales deben estar debidamente protegidas y nunca deben estar añadidas directamente sobre código (puesto que supondrían una brecha de seguridad severa).*

Pruebas de integración del sistema completo (todos los componentes)

Por último, se debe comprobar la **integración completa del sistema** añadiendo el frontal de la aplicación. En este nuevo set de pruebas se debe prestar atención al hecho de que al ser el objetivo del test comprobar la integración de este componente sobre el resto (y no la funcionalidad del frontal), las pruebas deben atacar directamente el nexo de unión entre la aplicación frontal y el servicio API REST (el punto en el que nacen las llamadas) y no la lógica que se le pueda aplicar después a estos datos (ya sea mostrándolos u añadiendo alguna capa de lógica de negocio).

3.4.2. Herramientas

Como es habitual, en función del componente que se esté probando y la tecnología sobre la que esté desarrollado, va a ser necesario aplicar una serie de tecnologías u

otras para aplicar las pruebas de integración.

Herramientas para pruebas de integración BD + servicio web SOAP

Las pruebas del **primer bloque de componentes** (BBDD + Servicio Web SOAP), se pueden realizar a través de llamadas al servicio ejecutadas desde la misma librería de testeo que se ha propuesto para los *test* unitarios (*NUnit*). Para su correcto funcionamiento, será necesario configurar debidamente el enrono de ejecución de las pruebas para que los componentes sean estancos y puedan conectarse entre sí.

Nota: *Nótese que, en este caso, no va a resultar necesario el uso del framework Moq, debido a que es precisamente en estos test en donde se pretende probar funcionalidad de conectividad y comunicación entre componentes.*

Para la realización de las pruebas se **debe considerar** que en esta ocasión se está **trabajando con operaciones asíncronas**. Además, estas pruebas tienen un sobrecoste derivado de levantar y mantener todos los componentes involucrados. Para remediarlo, resulta interesante aprovechar las posibilidades de **paralelismo** de pruebas que ofrece *NUnit*. Mediante este paralelismo va a ser posible reducir los tiempos de espera de las tareas (durante la espera se realizan otras tareas), aligerando así la carga temporal de las pruebas de integración.

Herramientas para pruebas de integración BD + servicio web SOAP + orquestador

Para el siguiente bloque de pruebas (BBDD + Servicio Web SOAP + orquestador OSB), se pueden utilizar las mismas herramientas y que para el bloque anterior.

Herramientas para pruebas de integración BD + servicio web SOAP + orquestador + servicio web API REST

Para la realización del siguiente bloque de pruebas (BBDD + Servicio Web SOAP + orquestador OSB + servicio web REST), existen diversas opciones de **herramientas** cada una con una operatividad diferente:

1. **Aplicar la misma lógica que con los bloques anteriores** (utilizar la librería NUnit para terminar llamando a la API y comprar los resultados obtenidos con los esperados). La principal ventaja de esta opción es que se no se añaden más dependencias software a terceros.
2. **Utilizar de una herramienta especializada en llamadas REST**, como, por ejemplo, *Postman* [50].
   Postman es una plataforma API para desarrolladores que permite realizar todo tipo de llamadas, asignaciones a variables, comprobaciones y operaciones relacionadas con llamadas a APIs.
   Dispone de la posibilidad de ejecutarse a través de CLI y de un plugin [48] que

le permite integrarse con el IDE *Visual Studio Code*.
La principal ventaja es que es una herramienta más especializada para este tipo de acciones y la principal desventaja que se aumenta la deuda técnica y que según los requerimientos podría requerir de licencia.

3. **Usar las herramientas integradas dentro de *Visual Studio Code*** (en sus últimas versiones incluye herramientas propias similares a Postman de manera nativa). La lógica a aplicar sería la misma que en el punto anterior.
   La ventaja que se gana sobre el punto anterior es que se reduce la deuda técnica (puesto que el IDE los vamos a utilizar igualmente) y la principal desventaja que resulta más compleja la codificación enfocada para un entorno de automatización a través de CLI.

A la hora de seleccionar una de las opciones, se deben considerar las necesidades de los test. En todo momento se debe priorizar el hecho de no aumentar la **deuda técnica** de la solución. Es por esto, por lo que, en primera instancia, si las pruebas se pueden realizar a través de la primera opción (NUnit), ésta debe ser la elegida. En caso contrario, la siguiente opción será las herramientas integradas en *Visual Studio Code*, y, por último, si con ninguna de las anteriores opciones es posible la realización de las pruebas, se optará por la opción de Postman.

Herramientas para pruebas de integración del sistema completo (todos los componentes)

Para el bloque de pruebas más completo de todos (incluyen todos los módulos y son realizadas desde el frontal), se pueden utilizar las **mismas herramientas que se emplean para las pruebas unitarias**: *Jasmine* y *Karma*.

En esta ocasión, habrá que prestar especial atención en enfocar las pruebas directamente al componente encargado de establecer la conexión con la API REST, y **no incluir en las pruebas componentes con funcionalidades propias del frontal** que estén por encima de dicha conexión.

3.5. Pruebas de Interfaz de usuario (UI)

Introducción a las pruebas de interfaz de usuario (UI)

Las **pruebas de interfaz de usuario** (también conocidas por sus siglas en inglés UI o incluso GUI según el contexto), son el tipo de *test* a más alto nivel de aquellos cuya funcionalidad es probar el correcto funcionamiento de los componentes de la aplicación que vamos a ver en este trabajo.

Son unas pruebas que se realizan desde la parte frontal de la aplicación y se centran en **evaluar tanto el rendimiento, como la funcionalidad general** desde los elementos visuales de una aplicación. Como es lógico, al no utilizar mocks, de manera indirecta, a través de la interfaz se está probando todos los módulos de la aplicación (tanto a nivel de componente, como de integración).

Características y ventajas de las pruebas de interfaz de usuario (UI)

Son las pruebas funcionales más **costosas** tanto en lo temporal (pues se aplica el flujo completo de la aplicación) como a nivel de **requisitos técnicos y recursos** (requieren de todos los componentes).

Como es habitual, resulta muy útil tener las pruebas automatizadas para permitir su reproducibilidad (PE: justo antes de un despliegue a producción). Algunas de las **ventajas** de este tipo de pruebas son:

- Aumentan los casos de prueba vistos anteriormente.
- Reducen el testeo de la UI realizado de manera manual.
- Garantizan el funcionamiento del sistema a nivel global.
- Son una fuente de seguridad para el usuario final.

Enfoques para la planificación de pruebas de interfaz de usuario (UI)

Al igual que nos sucedía con las pruebas de integración, existen diferentes **metodologías de aproximación** a las pruebas. Las más importantes son las siguientes:

- **Pruebas funcionales**. Son una serie de pruebas que se centran en comprobar las funcionalidades de la aplicación a nivel completo (interfaz de usuario incluida). Pueden ser programadas tanto a mano, como grabadas a través de una aplicación externa (que luego traduce la grabación a un test replicable)
- **Pruebas de aceptación**. Son una batería de pruebas que pretenden replicar el comportamiento completo de un usuario final (PE: un registro en la aplicación). Generalmente se graban a través de un autómata y son de gran utilidad de cara a los despliegues.
- **Pruebas de interfaz gráfica de usuario**. Son una serie de pruebas que se centran especialmente en comprobar que la interfaz gráfica se desemplea satisfactoriamente, restándole importancia al resto de los aspectos de la prueba. Son especialmente relevantes sobre todo para realizar comprobaciones de aplicaciones móviles, en donde las dimensiones de la pantalla pueden variar enormemente en función del dispositivo.

3.5.1. Cómo implementarlos en el sistema

Para la implementación de las pruebas hay que centrarse únicamente en **la aplicación del frontal** (en el marco objetivo del trabajo desarrollada en el *framework* Angular). En la solución evaluada, como se trata de una aplicación relativamente sencilla (un canal de entrada), los casos de prueba van a ir principalmente enfocadas a las **pruebas de aceptación** (validación del cumplimiento de los requisitos). No obstante, si dadas las dimensiones y/o complejidad de la aplicación se considerase necesario, también se podría plantear el diseño de pruebas funcionales.

La decisión de si merece la pena añadir, de manera adicional, una batería de **pruebas de interfaz gráfica** o no debe depender principalmente en el uso objetivo de la aplicación y si se va a utilizar con diferentes terminales hardware o no (en el caso del sistema de ejemplo de este trabajo, no se dan dichas circunstancias).

La forma de realizar dichas **pruebas de interfaz** consiste en diseñar un conjunto de casos de prueba que representen acciones de diferentes usuarios sobre la aplicación, programarlos (ya sea a través de la grabación de un autómata o programarlos de manera manual) y posteriormente comprobar que dicho flujo se comporta de manera esperada en la aplicación.

En estas pruebas se debe comprobar con todo lujo de detalles el correcto funcionamiento de la aplicación. Desde el correcto posicionamiento de los botones y popups hasta el contenido de las tablas, títulos y demás elementos visuales.

Durante la ejecución de las pruebas se pueden duplicar algunas de ellas modificando las proporciones de la ventana de la UI para verificar el correcto escalado de los elementos.

3.5.2. Herramientas

En esta ocasión **herramientas** a emplear para la realización de los tests son librerías diseñadas para la **realización de pruebas e2e** (end to end). Si bien es cierto que existen herramientas genéricas para el testeo e2e de todo tipo de aplicaciones web (como podría ser *Selenium* [51]), en el caso de herramientas desarrolladas en Angular se recomienda el uso de principalmente 2 herramientas: *Cypress* [52] y *Protractor* [53].

Las tres opciones (*Selenium, Cypress* y *Protractor*) disponen de **sendas opciones autómatas para grabar** el uso de un usuario de la aplicación de manera natural y

posteriormente replicarlo como un test.

Adicionalmente, las tres disponen también de la capacidad de lanzarse en un **entorno CLI** (y por lo tanto de automatizarse). Las 3 permiten obtener **reportes y capturas** de pantalla del estado de la ventana en caso de error.

Aunque ***Protractor*** es una herramienta expresamente diseñada para Angular, ésta no debe ser la aplicación principal que escogida. Se trata de una herramienta que está poco a poco **cayendo en desuso** (en beneficio de las otras) y en agosto 2023 dejará de tener soporte oficial.

Tanto *Cypress* como *Selenium* son herramientas de código abierto. Dejando de lado que ambas opciones tienen un gran **respaldo por parte de la comunidad**, dentro de la misma comunidad destaca más el uso de *Cypress*. Además, esta librería es más completa y permite realizar un mayor tipo de pruebas de manera nativa. Considerando todo esto, Cypress resulta en una mejor opción, por lo que es la herramienta escogida para la codificación de estas pruebas.

3.6. Pruebas de carga y estrés

Introducción a las pruebas de carga y estrés

Las **pruebas de carga y estrés** son el último tipo de pruebas que se van a abarcar en a lo largo de trabajo. Este tipo de pruebas tienen un objetivo completamente diferente al que hemos visto hasta ahora: su objetivo no es tratar de comprobar el correcto funcionamiento de una aplicación a nivel funcional, sino que se trata de **descubrir cuáles son los límites de carga** (cantidad de usuarios simultáneos, tipos de usuarios, etc.) que es capad de soportar el sistema. Adicionalmente, también suelen ser utilizadas para **identificar cuellos de botella** en sistemas multicomponente.

Características y ventajas de las pruebas de carga y estrés

Las pruebas de carga y estrés **requieren de una gran capacidad de cómputo** y de un entorno completamente apartado (puesto que al llevarlo al límite se convierte en un entorno inestable). Dicho entorno, **se debe ser una réplica exacta** de la arquitectura en producción la cual vaya a estar exclusivamente dedicada a estas pruebas.

Debido a su alto coste, **estas pruebas no se realizan de manera habitual**, sino que se realizan en momentos concretos del desarrollo, bajo demanda o en momentos claves del desarrollo tales como:

- Despliegue de la versión inicial del producto.

- Implementación de muchas funcionalidades desde la última prueba.
- Cambios estructurales en la aplicación.
- Estudio de migración de tecnologías.

Algunas de las **ventajas** que nos ofrecen este tipo de test son:

- Descubrimiento de las limitaciones del sistema.
  - Descubrimiento de cuellos de botella presentes en el sistema.
  - Evaluación de la estabilidad y robustez del sistema (si se cae o no)
- Garantizar la fiabilidad y la solidez del sistema.
- Validar la escalabilidad de la solución.

Enfoques para la planificación de pruebas de carga y estrés

Debido a su rareza y aunque son pruebas cuyo lanzamiento se realiza a través de **herramientas automatizadas**, se tratan tipo de pruebas que por lo general suelen supervisarse de forma manual a través de herramientas de visionado del estado de las máquinas (como podría ser un explorador de tareas).

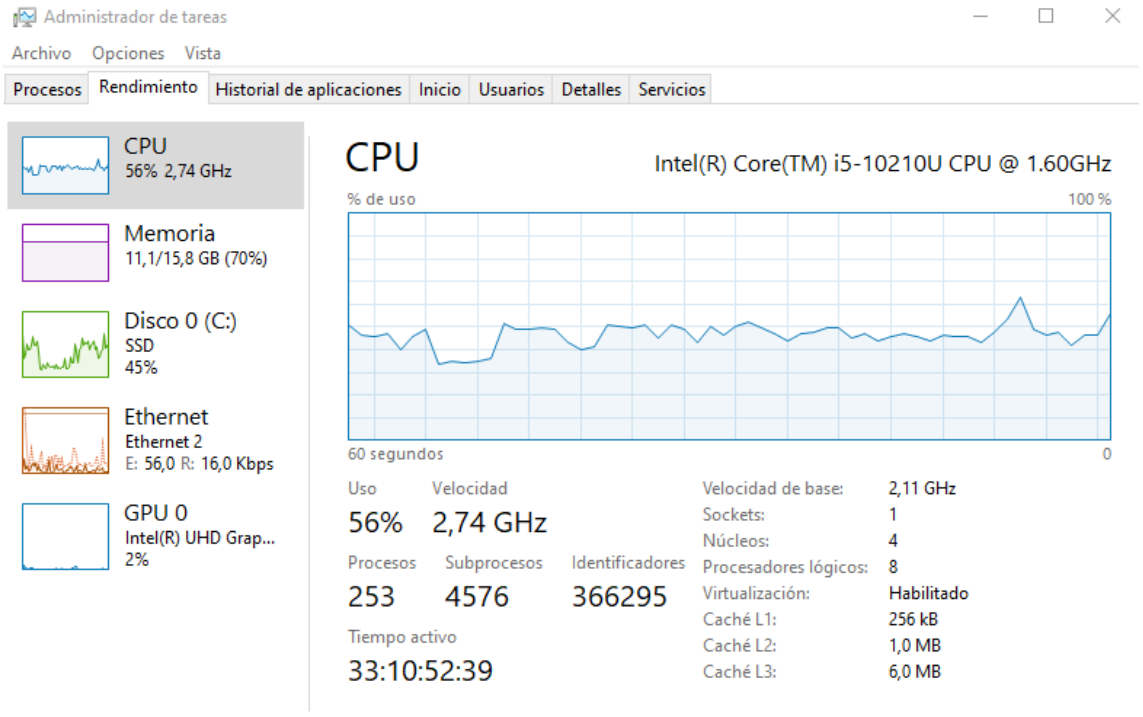


Ilustración 13: Cuadro de administrador de tareas en el que se puede evaluar el estado de los diferentes componentes hardware de un equipo durante la ejecución de unos tests.

A grandes rasgos, existen dos tipos de focos de pruebas:

- Periodos cortos con una gran cantidad de carga.
- Periodos largos con una cantidad de carga más moderada.

3.6.1. Cómo implementarlos en el sistema

Dada la criticidad del ejemplo práctico que se está evaluando en el trabajo, estas pruebas tienen una especial relevancia. Mediante las mismas, se va a identificar la **cantidad máxima de usuarios concurrentes** que es capaz de soportar el sistema, permitiendo así ser capaces de prever, en base a los datos históricos, posibles necesidades de optimización de componentes o escalado de recursos.

Para que los datos obtenidos sean de valor, resulta esencial que las pruebas se realicen en un **entorno estanco y con una arquitectura hardware idéntica** a la que tendrá la aplicación en el entorno de producción.

Tipos de operaciones y método de aplicación

Por la arquitectura de la solución aplicada, a la hora de plantear estos test, se deben considerar dos tipos de operaciones:

1. **Operaciones "simples"** que no requieren de encolado por parte el orquestador.
2. **Operaciones "críticas"** que requieren de encolado por parte el orquestador.

Dado el interés funcional de conocer los límites de las pruebas críticas, van a cobrar especial relevancia aquellas pruebas que se centran en este tipo de operaciones.

El modus operandi consiste tomar una operación o un conjunto de operaciones (según el set de pruebas) e ir aumentando su peso o cantidad de operaciones simultáneas hasta que el sistema se sature.

Durante este proceso resulta más que interesante **monitorizar la utilización de recursos** de los diferentes sistemas para evaluar el escalado a medida que se realizan las pruebas. Los enfoques de las pruebas serán los siguientes:

- Descubrir cuál es la cantidad máxima de información entrante que es capaz de procesar el sistema en **una única entrada** a través de operaciones críticas.
  En caso de disponer de los datos de utilización de las herramientas en años anteriores, resultará muy útil contrastar los resultados obtenidos junto con dicha información para evaluar el estado global de la aplicación.
- También resulta interesante descubrir cuál es la **cantidad máxima de operaciones "críticas" simultáneas** que es capaz de aceptar nuestro sistema sin saturarse en un intervalo corto de tiempo.
  Como recordatorio, el motivo por el que se incluye el orquestador es, precisamente, evitar que el sistema se caiga en un estado de sobrecarga de este tipo de operaciones, garantizando la entrada de todas ellas.

De nuevo, en caso de disponer de datos históricos resulta de especial interés realizar una comparación junto con los datos disponibles.

- Realización de pruebas que nos permitan descubrir **cuál es la cantidad máxima de operaciones "simples"** simultáneas que es capaza de aceptar nuestro sistema sin saturarse.
  En esta ocasión se debe tener en cuenta que, al no estar utilizando el orquestador, es más probable que los servidores terminen colapsando tras la realización de la prueba. Resulta interesante comparar los resultados obtenidos en este conjunto de pruebas junto con el anterior, para ver cuál es el peso efectivo de cada una de las pruebas sobre el sistema.
- Realización de un conjunto de pruebas con **operaciones mixtas**. Estas operaciones pretenden simular toda la interacción de un usuario con la aplicación de principio a fin.
  Para ello, se suele emplear un software de captura el cual a través de la interacción de un usuario con la aplicación genere una prueba que replique dicha interacción. Tras las pruebas, dispondremos de una estimación de la cantidad de usuarios reales que es capaz de soportar nuestro sistema.

3.6.2. Herramientas

Al igual que ha sucedido con el resto de las pruebas, se requieren de diferentes herramientas para cada uno de los conjuntos de test que pretendemos aplicar.

En el caso del primer grupo de pruebas, aunque como se verá más adelante existen **herramientas específicas diseñadas para la captura y relanzado de operaciones**, resulta más práctica la construcción de una herramienta propia debido a la lógica de negocio incrustada en las pruebas. Estos casos las pruebas consisten en ir modificando la llamada para añadirle complejidad y no en aumentar el número de repeticiones de la misma (que es la acción menos habitual de las herramientas generales).

La codificación de ese escalado de información para las operaciones podría no ser trivial y requerir el desarrollo de cierta lógica del negocio. En cualquier caso, la herramienta casera debe estar enfocada (en la medida de lo posible) a su puesta en marcha en procesos de integración continua.

Para conseguirlo, resulta interesante desarrollarla como una aplicación de consola. Considerando los requisitos no funcionales de nuestra aplicación objetivo, el *framework* de desarrollo escogido será una aplicación de consola .NET desarrollada a través de *Visual Studio*.

Para la **captura de las diferentes peticiones realizadas** desde el frontal y su posterior relanzado aumentando la cantidad de llamadas, se puede utilizar la herramienta *JMeter* [54].

Este software permite **capturar las diferentes llamadas** que se realizan desde el frontal mediante una grabación, y posteriormente, **relanzar estas mismas llamadas** configurando tanto la cantidad de veces, como el espaciamiento entre ellas. Es la herramienta ideal para todas las pruebas consistentes en aumentar la cantidad de llamadas por unidad de tiempo.

	FRONTAL (ANGULAR)	**SERVICIOS WEB (C# .NET)**	**OSB**	**BBDD**
PRUEBAS UNITARIAS	**Jasmine + Karma + Istanbul Code Coverage**	**NUnit + Moq + NCover**	***NUnit + Moq***	***NA***
PRUEBAS DE INTEGRACIÓN	**Jasmine + Karma**	**NUnit / Postman (sólo API)**	***NUnit***	***NA***
PRUEBAS DE INTERFAZ DE USUARIO	**Cypress / Selenium**	***NA***	***NA***	***NA***
CARGA Y ESTRÉS	***Herramienta desarrollada por el equipo + JMeter***			

Ilustración 14: Tabla en la que se resumen las diferentes herramientas de testing propuestas para la verificación de los diferentes componentes de la solución. En el caso de las pruebas de integración, la columna hace referencia a el componente del que nacen (independientemente de que posteriormente afecte al resto).

Capítulo 4 – *Branching*

4.1. Introducción

El objetivo de este capítulo es explorar un conjunto de **estrategias de Branching** de código fuente y analizar cuáles son sus principales ventajas y desventajas. Todo ello tendrá como objetivo final **brindar una estrategia adecuada** que permita desarrollar el hipotético sistema con la mayor facilidad posible y cumpliendo todos los objetivos propuestos.

A lo largo del capítulo, se deberán considerar los siguientes **requisitos funcionales**:

- Simplificación de los procesos de desarrollo y de mantenimiento
- Desarrollo en diferentes despliegues
- Automatización de los despliegues y entregas de producto
- Definición y seguimiento de estándares de calidad de código
- Permisibilidad de aplicación de metodologías ágiles
  - Control de versiones
  - Integración y despliegue continuos

De la misma forma, se van a considerar los siguientes **requisitos no funcionales**:

- Control de versiones mediante GIT

4.2. Conceptos previos

Introducción conceptual al control de versiones

Se denomina como ***branching*** a una **estrategia de gestión de desarrollo**, enfocada sobre el código fuente, que permite a los diferentes trabajadores de un producto, elaborar diferentes características o prestaciones (*features* en el argot) en un mismo proyecto de manera simultánea, controlada, organizada y segura.

La idea nace a partir a partir del concepto **rama** (*branch* en el argot) de código. Una rama es una bifurcación que se produce en el código fuente, partiendo de un estado concreto común (a su rama madre), momento a partir del cual los diferentes desarrolladores pueden realizar cambios en el código fuente de cada una de las ramas sin afectar dichos cambios al resto de las ramas.

Al igual que se pueden crear bifurcaciones, también existe posibilidad de juntar dos ramas. A esta operación se le denomina ***merge*** y tiene como objetivo agregar las funcionalidades desarrolladas en las dos ramas que se están fusionando en una única rama (generalmente una de las originales).

Al combinarse, es habitual que en ambos flujos de desarrollo se hayan modificado las mismas partes del código. A esta situación se le conoce como ***conflicto,*** suele ser bloqueante y deben de ser resueltos antes de poder llevarse a cabo la fusión. Tras la unión, no necesariamente deben destruirse las ramas originales.

Existen unos **tipos concretos de *merges***, los cuales no pueden ser realizados por un único desarrollador, sino que requieren que los cambios que se están asignando sean revisados por al menos un miembro adicional del equipo. A este tipo de *merge* se le denomina diferente en función de la herramienta de control de versiones que se esté utilizando. Las nomenclaturas más comunes son ***pull request*** de GitHub [55] y ***merge request*** de GitLab [56].

También existe un concepto denominado ***cherry-pick***, el cual consiste en la aplicación selectiva de ciertos cambios de una rama de código A a otra rama de código B, sin la necesidad de añadir en B todos los cambios que se hayan sucedido en A. Para que este concepto tenga sentido, necesariamente la rama de código B debe ser posterior a la rama de código A.

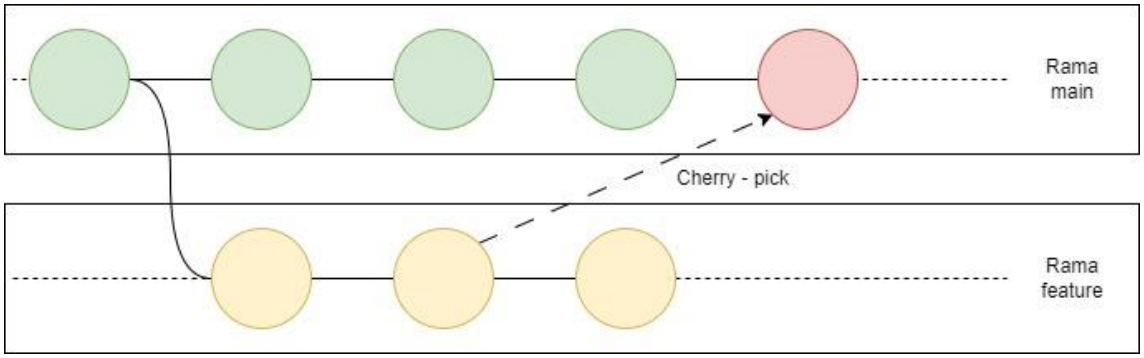


Ilustración 15: Esquema de cambios en el que se encuentra un cherry - pick.

Nota: *los diferentes conceptos mencionados en este apartado se encuentran integrados dentro de la herramienta GIT* [13] *de tal forma que el desarrollador sólo debe centrarse en la realización de los cambios y la resolución de los conflictos.*

Ventajas y desventajas del uso de ramas durante el proceso de desarrollo

Las estrategias de branching han ganado especial relevancia a partir del uso de herramientas de control de versiones que permiten realizar las operaciones de creación, unión y destrucción de ramificaciones de manera totalmente automática.

De entre las herramientas de control de versiones que permiten realizar estas operaciones destaca GIT, que es la herramienta que se va a considerar a lo largo de este trabajo para el control de versiones de la solución.

Algunas de las **ventajas** del uso de ramas son:

- **Paralelismo y separación de funcionalidades.** Las ramas permiten a los desarrolladores trabajar en diferentes funcionalidades y/o funciones de forma aislada. Esta particularidad permite estructurar organizar los nuevos cambios del producto y agilizar el desarrollo.
- **Estanqueidad y experimentación.** Las ramas, al definir un entorno en el que los cambios de una no interfieren con las otras, son ideales para realizar diferentes tipos pruebas de concepto. Si la prueba sale bien se continuará su desarrollo, y si no, se eliminará la rama en cuestión.
- **Control de versiones.** Las ramas permiten mantener un registro temporal de los diferentes estados del desarrollo, pudiendo mantener así diferentes versiones de un mismo producto al mismo tiempo.

Por el contrario, algunas de sus **desventajas** son:

- **Aumento de la complejidad del proceso de desarrollo.** Cuantas más ramas tiene un producto, más complejo puede resultar el desarrollo del mismo. Los miembros del equipo deben tener en cuenta cuál es el estado previo de cada

una de las ramas antes de comenzar a desarrollar sobre ellas.

- **Conflictos de fusión.** Solventar los diferentes conflictos ocurridos durante una fusión de código puede llegar resultar muy complejo y laborioso para los desarrolladores.
- **Mantenimiento.** Cuando un mismo producto tiene un gran número de ramas, el mantenimiento se puede volver más desafiante. Cada una de las diferentes ramas puede requerir una serie de actualizaciones y ajustes específicas. Este tipo de "problema" se intensifica cuando existen diferentes ramas cada una asociada a una versión del producto en el entorno final.

4.3. Tipos de estrategias de *branching*

Existen gran cantidad de estrategias de branching y, por ende, diferentes tipos de enfoque que las agrupan. Este tipo de agrupaciones tiene en cuenta factores como:

- Flujos de trabajo.
- Enfoque para integración y despliegue continuos.
- Esquema de ramificaciones.
- Ciclos de vida de las ramas.
- Necesidades del equipo y producto.

Dos de los enfoques más comunes de branching son el ***Branch based*** y **el *Trunk based***. En este caso, la principal diferencia reside en el **flujo de trabajo**, y en donde se van a centrar los desarrolladores en trabajar.

Estrategias branch based

La idea que existe detrás de las estrategias ***branch based*** es que los diferentes miembros del equipo de desarrollo trabajen las funcionalidades en una rama secundaria expresamente creada para dicha funcionalidad, manteniendo así la rama principal siempre con la última versión estable.

Los cambios no tienen por qué aplicarse directamente sobre la rama secundaria, sino que pueden realizarse a través de **ramas de corta duración** que posteriormente se fusionarán con la secundaria. Una vez se han desarrollado todas las funcionalidades en la rama secundaria, se realiza un volcado de todas estas funcionalidades a la rama principal, quedando así esta última actualizada y estable.

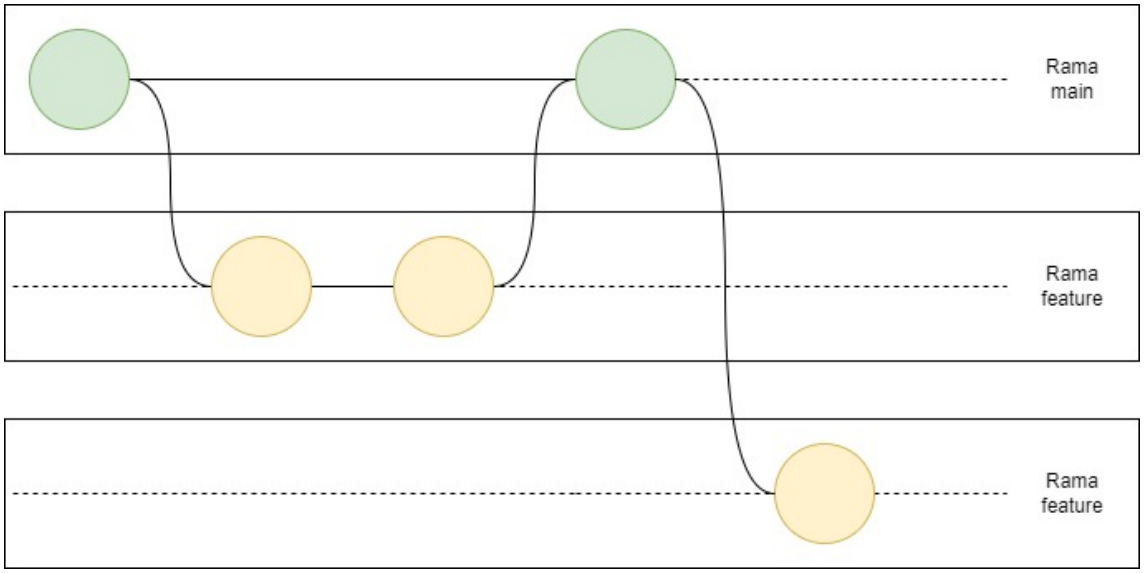


Ilustración 16: Diagrama que ilustra un ejemplo de flujo de desarrollo empleando una estrategia branch based.

Por lo general, lo que pretende este enfoque es tener en todo momento una versión completamente estable en la rama principal. Como **ventajas**, destacan:

- **Aislamiento**. Las ramas separadas permiten separar el trabajo en progreso del trabajo terminado, lo que favorece enormemente la estabilidad de los diferentes entornos de desarrollo.
- **Paralelismo**. Las ramas separadas permiten realizar varios desarrollos por separado, lo que resulta especialmente útil en equipos de desarrollo grandes.
- **Fácil reversión de los cambios**. Al aplicarse todos los cambios a través de *merges* resulta sencillo revertir cambios en el caso de que se hayan detectado errores (sólo hay que retroceder un *merge*, en vez de varios *commits*).

En cambio, sus **contras** pasan por:

- **Coste de integración**. El merge de la rama secundaria con la principal puede ser especialmente costoso. Esto puede desembocar en una disminución del ritmo de trabajo.
- **Pérdida de visibilidad.** Esta pérdida viene derivada de la cantidad de ramas empleadas. Se puede mitigar si se van eliminando las ramas conforme se van integrando los cambios.
- **Flexibilidad**. Mediante esta estrategia, no va a ser posible mantener varias versiones del producto en paralelo. Una vez generada una versión, esta debe persistir intacta.

Estrategias trunk based

En las estrategias ***trunk based***, la lógica es justo la opuesta. El desarrollo se realiza en **una única rama principal** (generalmente llamada *main*, *master* o *trunk*). La idea es que

los desarrolladores trabajen lo más cercano a esta rama, ya sea a través de una rama de muy corta duración que posteriormente será anexionada a la principal, o directamente introduciendo los cambios sobre ella.

La decisión sobre si trabajar directamente sobre la rama principal o hacerlo a través de ramas de corta duración suele depender de la cantidad de desarrolladores del equipo.

Una vez el equipo ha desarrollado por completo una versión o funcionalidad, se genera una nueva rama desde master con el nombre de la versión en cuestión. En el caso de detectarse errores en alguna rama de versión, estos serán corregidos en la rama principal y posteriormente llevados a la rama en cuestión.

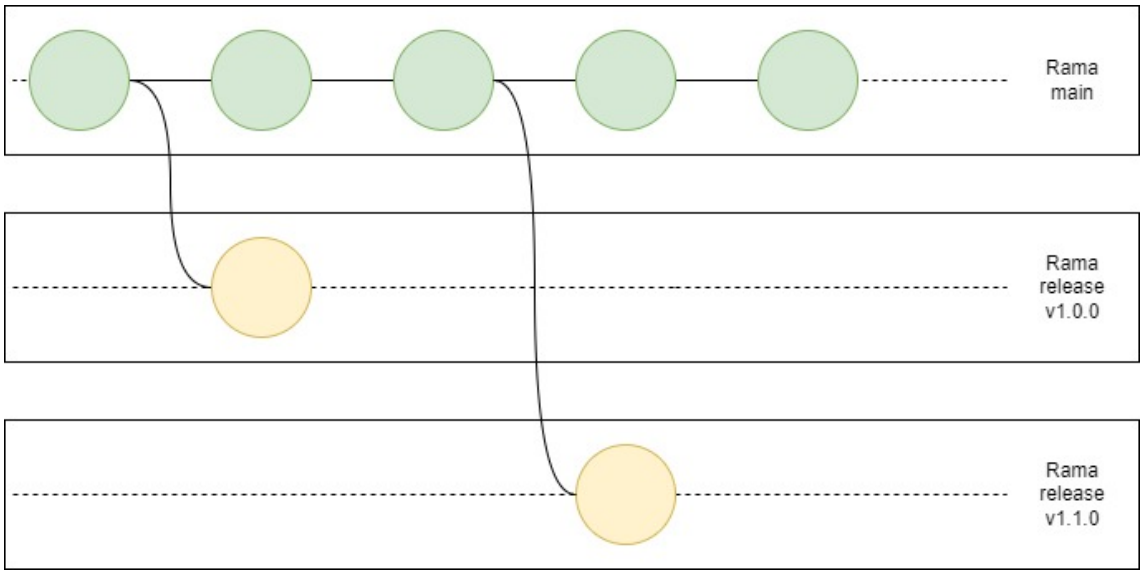


Ilustración 17: Diagrama que ilustra un ejemplo de flujo de desarrollo empleando una estrategia trunk based.

Lo que pretende este enfoque es minimizar la cantidad de ramas de desarrollo, disminuyendo así la complejidad del proyecto. En general, como resultado, el flujo de trabajo suele ser más simple y directo. Como **ventajas**, destacan:

- **Simplicidad**. Tanto las integraciones realizadas como los conflictos son menos frecuentes y de menor tamaño, lo que facilita su integración en el proceso de desarrollo.
- **Visibilidad**. Al encontrarse todos los cambios en la rama principal resulta sencillo obtener una vista completa tanto del estado, como del histórico del proyecto.
- **Flexibilidad**. Es un método de trabajo que aporta una gran flexibilidad a la hora de desemplear las labores de desarrollo. Entre estas flexibilidades, destaca la posibilidad de poder disponer de varias versiones del producto en paralelo.

En cambio, sus **desventajas** son:

- **Aislamiento**. La disminución de la cantidad de ramas puede derivar a la integración de manera no deseada de cambios en desarrollo.
- **Paralelismo**. La disminución de las ramas de desarrollo dificulta las labores de desarrollo de equipos grandes, en donde hay que abordar varios desarrollos al mismo tiempo.

4.4. Desarrollo *branch based*

4.4.1. *GitFlow*

Conceptos básicos de GitFlow

GitFlow es una de las estrategias más conocidas y su objetivo es prescribir una base de orquestación precisa y pautada que permita seguir de la manera relativamente sencilla el flujo de desarrollo del proyecto. Esta estrategia se basa principalmente en dos ramas:

- **Rama *main***: es la rama en la que se encuentra el registro de las diferentes entregas oficiales que se han ido realizando a lo largo del producto.
- **Rama *develop***: es la rama en la que se encuentran las últimas versiones estables de los cambios realizados, pero que no necesariamente forman parte todavía de una release oficial.

En ningún momento es posible realizar cambios directamente sobre ninguna de estas dos ramas. Como veremos, todos los cambios deben realizarse a partir de *merges* desde ramas auxiliares.

Desarrollo a través de GitFlow

El modus operandi a seguir para realizar el desarrollo de una nueva funcionalidad es el siguiente:

1. Se genera una nueva rama naciente a partir de la última versión de la rama ***develop***. A este tipo de rama se le suele denominar como ***feature***.
2. En la rama ***feature*** se incluye todo el desarrollo de la nueva funcionalidad.
3. Una vez finalizado el desarrollo, la rama generada para dicho propósito se fusiona con la rama ***develop***, quedando así reflejados los cambios en dicha rama.

A medida que las funcionalidades se van terminando de desarrollar, llegará un punto en el que la rama ***develop*** tendrá los cambios suficientes como para querer generar y

realizar una nueva entrega del producto. Cuando esto suceda, se realizarán las siguientes acciones:

1. Se genera una nueva rama naciente a partir de la última versión de la rama ***develop***, la cual incluirá las nuevas funcionalidades. A este nuevo tipo de rama se le denomina ***release***.
2. En esta nueva rama, se revisan todas las funcionalidades desarrolladas. En el caso de que se detecten fallos, se realizarán las correcciones pertinentes sobre esta misma rama.
3. Una vez comprobadas las funcionalidades, se fusiona la rama ***release*** con la rama ***master***. De este merge nace la versión que se entregará al cliente. Adicionalmente, la rama ***release*** también se fusiona a la rama ***develop***, para que esta última incluya las diferentes correcciones implementadas durante el ciclo de vida de la rama ***release***.

Por último, existe un último escenario, en el cual se detecta un fallo posterior a la entrega del producto que debe de ser corregido. Cuando esto sucede, la forma de actuar es la siguiente:

1. Se genera una nueva rama a partir de la última versión de la rama ***master***. A este nuevo tipo rama se le denomina ***hotfix***.
2. En esta nueva rama, se corrigen los fallos detectados con la mayor brevedad posible.
3. Por último, tras corregirlos, la rama ***hotfix*** se fusiona tanto con la rama ***main*** como con la rama ***develop***, para que en ambas ramas queden reflejados los cambios.

En el siguiente esquema, se puede apreciar un ejemplo de un posible flujo de desarrollo empleando la estrategia *GitFlow*:

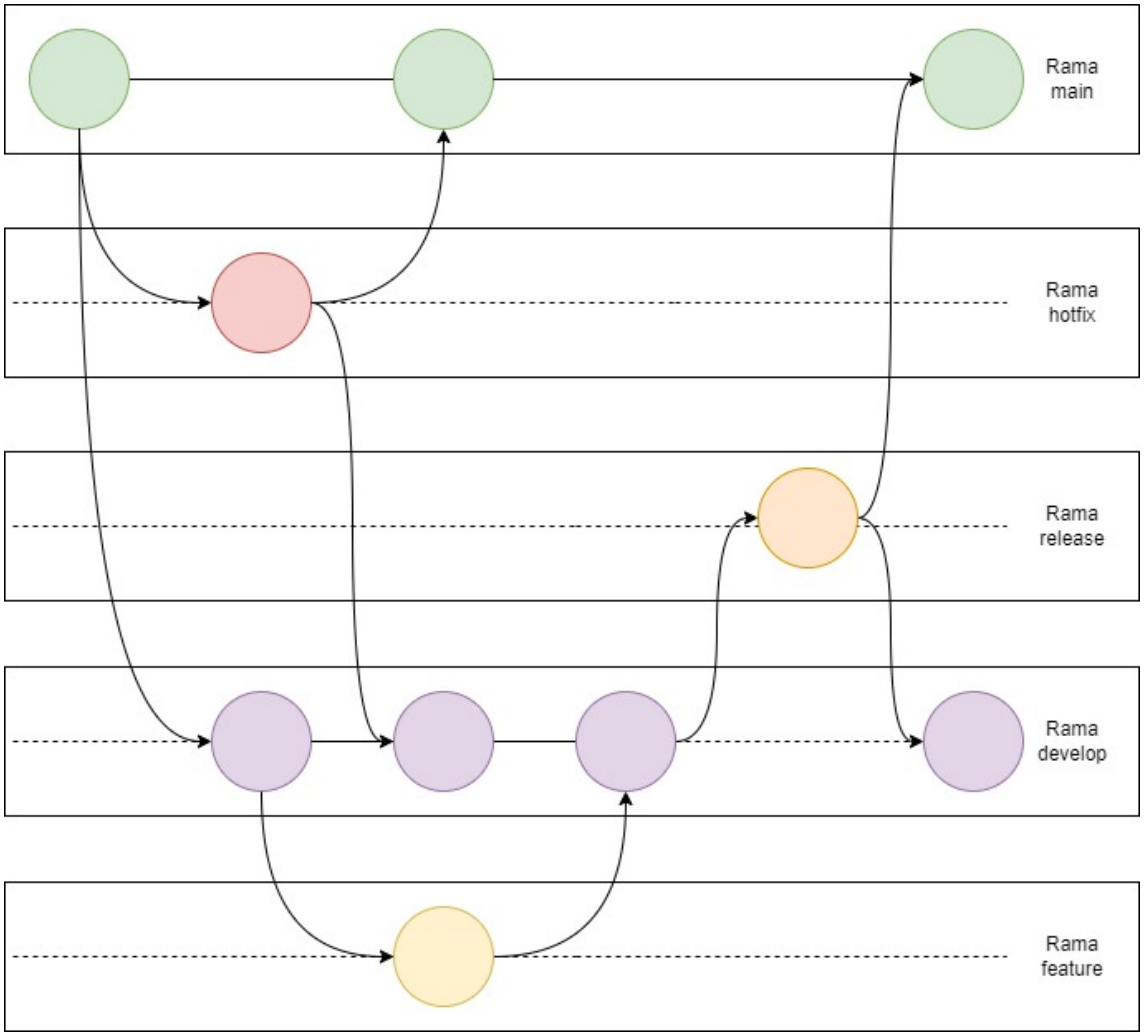


Ilustración 18 : Diagrama que ilustra un ejemplo de flujo de desarrollo empleando la estrategia GitFlow.

Ventajas y desventajas de GitFlow

Algunas de las **ventajas** de esta estrategia son las siguientes:

- Es muy **completa** y tiene un plan a seguir para la mayoría de los escenarios posibles durante el desarrollo.
- Al tener una **rama dedicada al producto entregado**, es sencillo evaluar la evolución del producto a lo largo del tiempo.
- **Útil en equipos y desarrollos grandes**, en donde cada desarrollador puede enfocarse en un tipo de desarrollo concreto.

En cambio, algunas de sus **desventajas** son:

- **La casuística es elevada**, y puede ser complejo seguirla sin cometer errores.
- En productos pequeños puede **aumentar mucho la complejidad del desarrollo**.
- Al **requerir de muchos *merges*** el histórico del repositorio puede llegar a ensuciarse.

- Al requerir de tantas ramas y merges, **no es una estrategia optimizada para la entrega continua** (CD).

4.4.2. *GitHubFlow*

Conceptos básicos de GitHubFlow

GitHubFlow es una estrategia propuesta por el equipo de desarrollo de la herramienta de gestión de código GitHub [55] que nace a partir de la estrategia *GitFlow*, y que pretende, a través de una serie de simplificaciones, ganar agilidad sobre la estrategia *GitFlow*.

Esta nueva estrategia se va a diferenciar principalmente de la rama *GitFlow* en que se va a pasar de las dos ramas anteriormente mencionadas (***main*** y ***develop***) a una única rama principal y estable denominada (***master***).

Desarrollo a través de GitHubFlow

La estrategia no diferencia los diferentes tipos desarrollos (correcciones de fallos de código, desarrollo de funcionalidades…). Para todos ellos, el modus operandi es el mismo:

1. Se genera una rama auxiliar a partir de la rama ***master***.
2. Sobre esta rama se realiza el cambio para el cual se ha generado la propia rama (una corrección concreta o el desarrollo de una *feature*).
3. Una vez completado el desarrollo, se solicita un ***pull request*** en el cual la mayor cantidad posible de miembros del equipo van a tomar parte para tratar de mejorar los cambios.
4. (Opcional). En el caso de que no se haya realizado ningún *merge* sobre ***master*** desde el nacimiento de la rama, se recomienda desplegar el contenido de la propia rama para verificar el correcto funcionamiento de la misma.
5. Por último, una vez comprobados todos los cambios introducidos, se fusiona la rama con ***master*** y se despliega el producto con los nuevos cambios (en el caso de que ya se hubiese realizado el despliegue a través del paso 4, se recomienda volver a realizarlo).

Aunque no es obligatorio, con la finalidad de simplificar el historial de desarrollo, desde GitHub recomiendan eliminar cada una de las ramas una vez se ha realizado el *merge* de la misma con ***master***. En el siguiente flujo se puede apreciar un desarrollo realizado mediante esta estrategia:

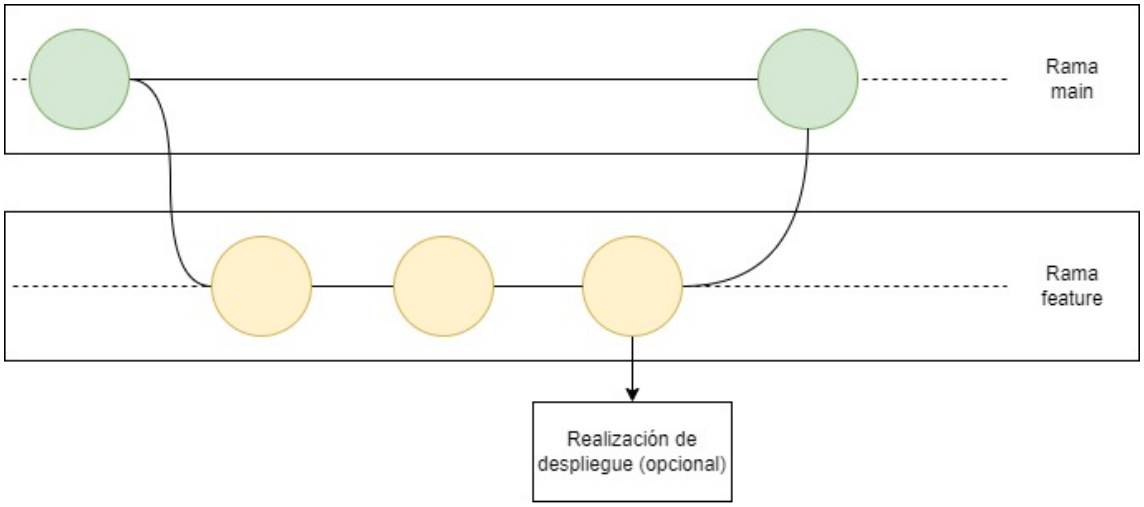


Ilustración 19: Ejemplo de flujo de cambios siguiendo una estrategia GitHubFlow.

Ventajas y desventajas de GitHubFlow

Las principales **ventajas** de *GitHubFlow* son las siguientes:

- Es una estrategia **simple y fácil de seguir**.
- **Funciona especialmente bien con equipos y desarrollos sencillos**.
- Está muy **enfocada a la entrega continua** de producto (CD).

Por contra, sus **desventajas** son las siguientes:

- **Requiere de una capacidad de despliegue y retorno muy buena** (sobre todo si se realizan los despliegues desde la rama como se indica en el paso 4).
- Si se realizan muchos desarrollos a la vez se pierde la capacidad de desplegar el código de la propia rama (una de sus ventajas). Esto suele suceder en equipos de desarrollo y productos grandes.
- En caso de no eliminarse las ramas, puede resultar **complejo** entender cuál es **el motivo de ser de cada una de las ramas existentes**.

4.4.3. *GitLabFlow*

Conceptos básicos de GitLabFlow

GitLabFlow es una estrategia propuesta por el equipo de desarrollo de la herramienta de gestión de código GitLab [56], que al igual que *GitHubFlow*, nace a partir de la estrategia *GitFlow*. En esta ocasión, lo que se pretende conseguir a través de esta estrategia es tratar de reducir al mínimo posible la cantidad de errores posibles en producción.

Esta nueva estrategia se va a diferenciar principalmente de la rama *GitFlow* en que se va a pasar de las dos ramas anteriormente mencionadas (***main*** y ***develop***) a una rama por cada uno de los entornos de desarrollo (master/desarrollo, pre-producción, producción, etc.).

Desarrollo a través de GitLabFlow

Durante el desarrollo y corrección de errores, el flujo propuesto desde GitLab es el siguiente:

1. Se genera una rama auxiliar a partir de la rama ***master***.
2. Sobre esta rama se realiza el cambio para el cual se ha generado la propia rama (una corrección concreta o el desarrollo de una *feature*).
3. Una vez completado el desarrollo, se solicita un ***merge request*** en el cual la mayor cantidad posible de miembros del equipo van a tomar parte en tratar de mejorar los cambios.
4. Por último, esta rama se anexiona sobre la rama ***master***.
5. (Sólo para correcciones de errores). En el caso de que el desarrollo realizado sobre la rama sea una corrección urgente que deba desplegarse de manera inmediata, se permite la realización de ***cherry-picks*** que incluyan las correcciones de esta rama sin necesidad de pasar por los *merge request* de las ramas de los entornos anteriores (explicados en el siguiente párrafo).

En el momento en el que se han desarrollado una cantidad de nuevas funcionalidades adecuada en la rama de *master*, éstas funcionalidades viajan de dicho entorno al siguiente a través de un ***merge request***. Después viajarán de dicho entorno al siguiente a través de otro ***merge request*** y así sucesivamente hasta llegar a producción. Este tipo de operaciones pueden llegar a ser complejas y pueden acaparar una gran parte del tiempo de desarrollo.

Nota: *nótese que un mismo cambio se va a revisar en cada uno de los merge request que se realicen hasta llegar a producción, lo que debería reducir el riesgo de que lleguen fallos a producción.*

En el siguiente esquema se puede apreciar un posible flujo de desarrollo realizado a través de esta estrategia de desarrollo:

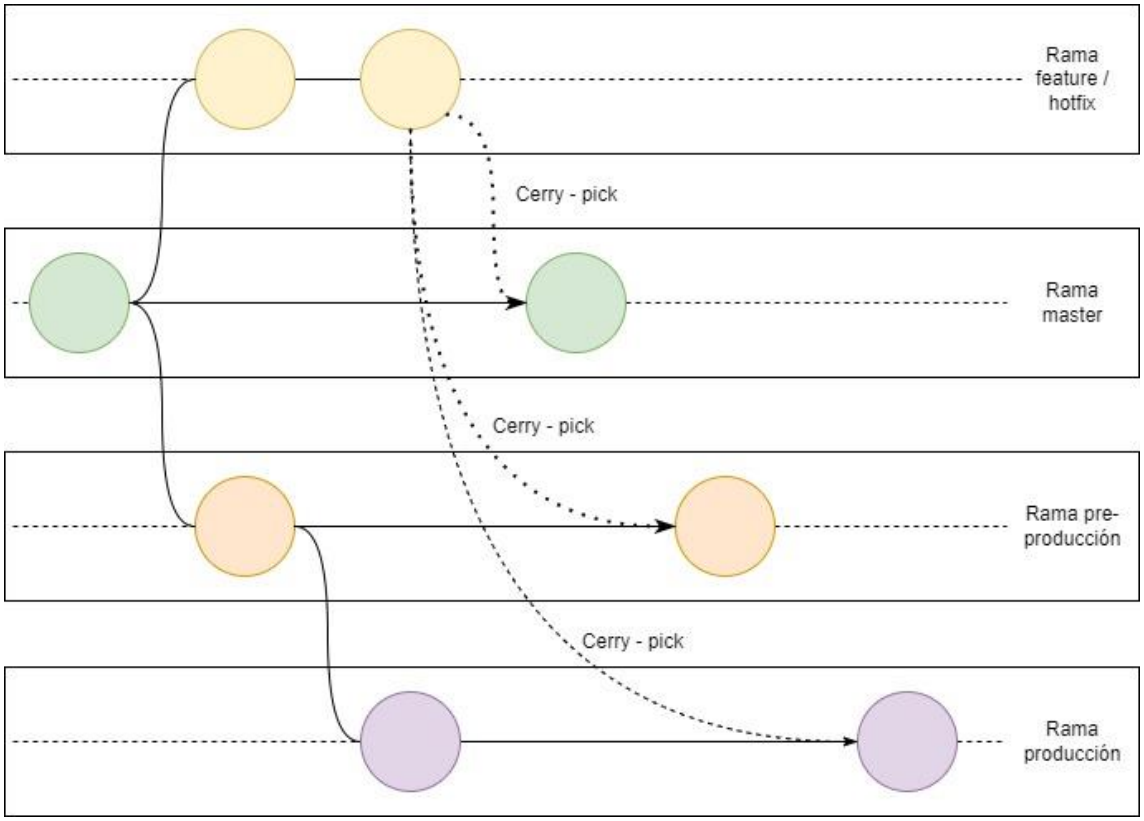


Ilustración 20:Ejemplo de flujo de cambios siguiendo una estrategia GitLabFlow.

Ventajas y desventajas de GitLabFlow

Algunas de las **ventajas** de este sistema son:

- Gran **robustez del código** desplegado en producción, revisado por los desarrolladores en diversas ocasiones.
- **Trazabilidad del código desplegado** en cada uno de los entornos en todo momento.
- **Funciona bien independientemente del tamaño del equipo y del proyecto**.

En cambio, sus principales **desventajas** pueden ser:

- Se pierde una **gran cantidad de tiempo revisando** el mismo código de manera manual.
- **Reduce la agilidad de la entrega del producto**. La necesidad de realizar tantas revisiones de código de manera permanente ralentiza el proceso de entrega del continuo.
- Los ***merge request* pueden ser muy complejos** cuando aglutinan gran cantidad de cambios.

4.5. Desarrollo *trunk based*

Conceptos básicos de desarrollo trunk based

Las estrategias *trunk based* se centran en desarrollar los avances del código dentro de la **rama principal** para, posteriormente, generar una nueva rama por cada una de las **raleases** (versiones) del producto que se generen.

El objetivo que persigue esta estrategia no es otra que ser capaces de mantener por separado cada una de las versiones del producto liberadas.

Desarrollo a través de desarrollo trunk based

El modus operandi de la estrategia es el siguiente: Existe una rama principal ***master***, que es sobre la que se van a introducir todos los cambios. Es posible realizar los cambios tanto directamente sobre la propia rama *master* como a través de ramas de tipo ***feature*** como se ha visto en otras estrategias como *GitHubFlow*.

La mayor diferencia la encontramos a la hora de **desplegar el producto**. Mientras que en las estrategias vistas hasta ahora los diferentes cambios que se deseaban desplegar se anexionaban a la rama de entorno correspondiente, en este caso lo que se va a hacer es, una vez agregados los cambios a *master*, crear una nueva rama la cual va a representar a una versión concreta del producto.

En el caso de que se descubran **errores** en una versión, las correcciones pertinentes se realizarán directamente sobre **master**, y se llevarán a la rama de la versión correspondiente a través de un ***cherry-pick*** que permitirá generar una nueva versión del producto desplegado.

En la ilustración 17 es apreciar un posible flujo de desarrollo empleado utilizando esta estrategia de ramificaciones.

Ventajas y desventajas de desarrollo trunk based

Algunas de las **ventajas** de esta estrategia son:

- **Permite mantener varias versiones del producto a la vez**. Cada una de estas versiones se encontrará alojada en una rama.
- **Tiene un flujo sencillo y fácil de seguir**. Al sólo disponer de una rama principal de cambios, es más sencillo seguir los cambios introducidos.
- Está bastante **enfocada al despliegue continuo** (CD). Al no requerir de muchos *merges,* los despliegues no se ralentizan tanto como en las estrategias que

requieren de muchas ramas.

Algunas de las **desventajas** de esta estrategia son:

- **Aumento de la dificultad de mantenimiento del producto**. Al tener que mantener varias versiones del producto en diferentes ramas se ralentiza el mantenimiento del mismo.
  Los fallos posibles que se detecten pueden afectar a diferentes versiones desplegadas en diferentes ramas y sus correcciones deben anexionarse en todas ellas.
- **Pérdida de aislamiento y paralelismo.** Al trabajarse el desarrollo, en su mayoría, en una única rama, se pierde la capacidad de aislamiento paralelismo que ofrecen las ramas.
- **Dificultad para la reversión de los cambios.** Debido a la diversificación de las versiones del código, si se decide revertir un cambio se podría requerir realizar dichos cambios en todas las ramas existentes.

4.6. Desarrollo *ReleaseFlow*

Conceptos básicos de ReleaseFlow

ReleaseFlow es una estrategia propuesta por el equipo de Azure [57] de Microsoft la cual está diseñada expresamente para funcionar de la manera lo más óptima posible en **equipos de desarrollo y proyectos muy grandes**. Como se va a ver a continuación, ReleaseFlow aporta un enfoque que se asemeja a las estrategias *trunk-based*, pero con sus propios matices.

Desarrollo a través de ReleaseFlow

Toda la estrategia se basa sobre tener una **rama principal** en la que se realizan incrementos lo más pequeños posibles, y, cada unidad de tiempo, generar una nueva rama ***release*** que aglutine todos los desarrollos relacionados hasta la fecha. De esta forma, el desarrollo de desemplea a través de los siguientes pasos:

1. Se crea una rama de la menor duración posible nacida de la rama principal (***master***).
2. Sobre esta rama se realiza el mínimo incremental posible lógico, denominado ***topic*** (el incremental suele ser mucho más pequeño que el desarrollo de una funcionalidad completa).
3. Se fusionan los cambios con la **rama principal** y se destruye la rama ***topic***. Este merge se realiza siempre de la manera lo más ágil posible, por lo que

se suelen evitar los merges del tipo **pull request** y similares.

Una vez transcurrida la unidad de tiempo deseada (generalmente en torno a 2 semanas o un sprint en el caso de que se trabaje mediante metodologías ágiles), se genera una nueva rama ***release*** que va a representar la siguiente versión del producto.

***Nota**: En el caso de que se encuentre un fallo grave la versión desplegada, se permite realizar el merge del/los topic(s) que lo arreglan tanto a **master**, como a la rama **release** de la versión (que será nuevamente desplegada).*

El siguiente esquema muestra un ejemplo de desarrollo a través de esta estratega de ramas:

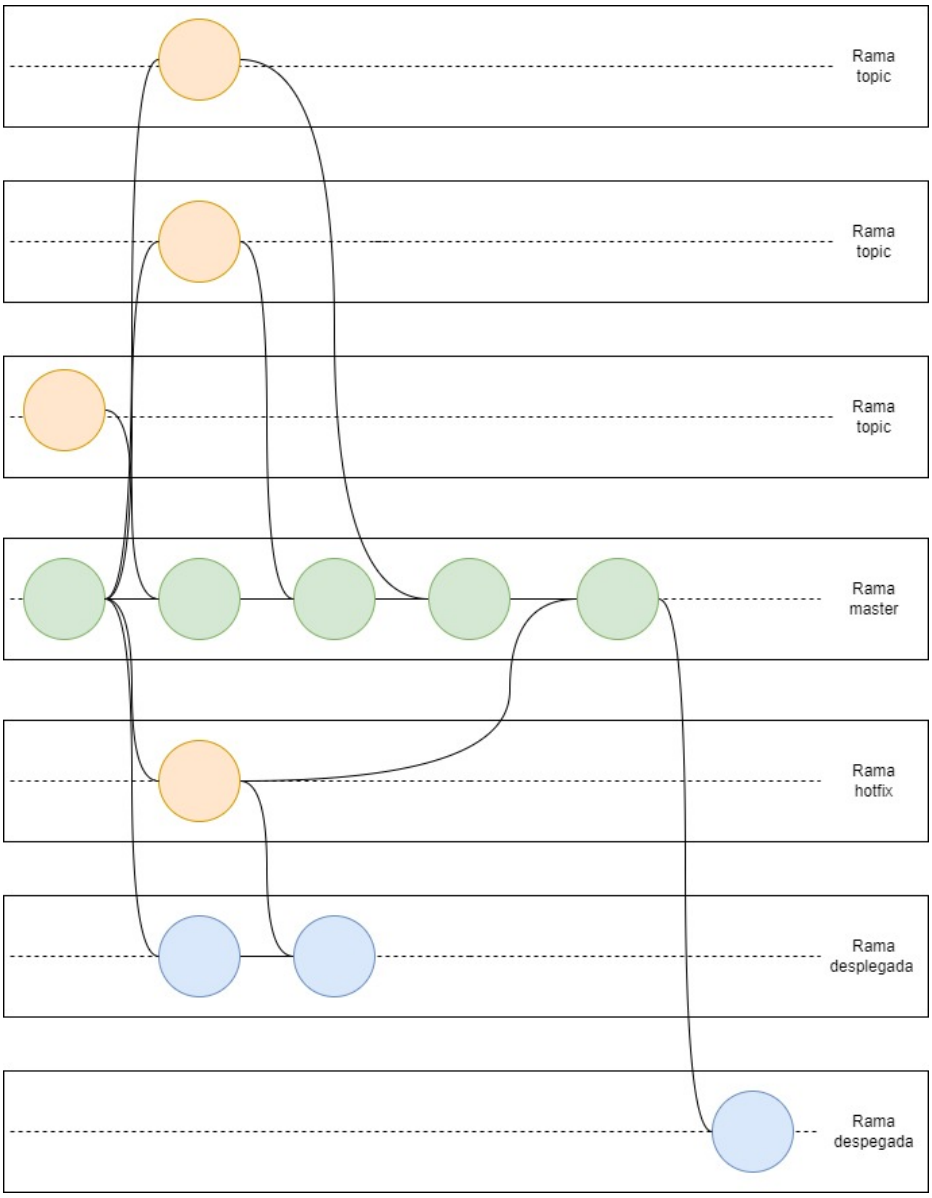


Ilustración 21: Ejemplo de flujo de cambios siguiendo una estrategia ReleaseFlow.

Ventajas y desventajas de ReleaseFlow

Las principales **ventajas** de la estrategia son:

- **Funciona bien con equipos o proyectos grandes y ofrece una gran capacidad de paralelismo**. Su diseño, enfocado en perder el mínimo tiempo en aplicar cambios y ramas de corta duración están enfocados a este objetivo.
- Muy **poca pérdida de tiempo en tareas que no son puro desarrollo**. Debido a la necesidad de aplicación inmediata de los cambios, se suprimen tareas de revisión de código manuales y similares.

En cambio, sus principales **desventajas** son:

- Casi **nula trazabilidad del desarrollo**. Al realizarse muchos cambios y con cambios muy pequeños, resulta prácticamente imposible trazar el desarrollo de una funcionalidad concreta.
- **Alejamiento de la entrega continua y despliegue de código poco probado**. Los despliegues se realizan por incrementos temporales y no funcionales. El código desarrollado podría desde tener que esperar todo el incremental a ser desarrollado hasta ser desplegado al instante sin ser debidamente probado.
- **Requiere de un sistema de despliegue muy rápido** para corregir fallos de código. Este problema se intensifica debido a la realización de despliegues que incluyen funcionalidades poco probadas.
- **Requiere de una infraestructura de testeo** muy avanzada. Para mitigar la no posibilidad de realización de pruebas durante el desarrollo del código, resulta esencial la realización de test automatizados que prueben las aplicaciones.

4.7. Desarrollo *Master-only flow*

Conceptos básicos de master-only flow

La estrategia master-only flow, es una estrategia que se basa en **la no utilización de ramas** a la hora de desarrollar software. Puede parecer un concepto contra intuitivo, pero el objetivo que se persigue con esta estrategia es tratar de reducir al mínimo la cantidad de tiempo invertida en lo relativo a la gestión de ramas (creación, desarrollo, fusión…).

Se basa sobre la premisa de que no va a ser necesario invertir tiempo en la unión de los diferentes desarrollos del equipo si esta combinación se realiza de manera natural durante el desarrollo del producto.

Técnicas para emplear master-only flow

Como todos los miembros del equipo están trabajando y añadiendo cambios sobre la misma rama, la clave de esta estrategia reside en ser capaces de realizar estos trabajos en paralelo **generando el menor número de conflictos posible** y solventándolos en el **menor tiempo posible**. Para ello, se suelen emplear las siguientes técnicas:

- **Automatizaciones avanzadas y baterías de test pesadas**. Disponer de un conjunto de test suficientemente robusto y la correcta automatización de los mismos, permite saber en tiempo real si los cambios que se están introduciendo a través de un *commit* afectan negativamente al producto que ya se encontraba desarrollado.
- **El equipo debe de ser maduro**, debe tener automatismos y estándares y a poder ser debe contar con experiencia trabajando juntos. Estas cualidades permiten al equipo sincronizarse mejor reduciendo el número de conflictos que se puedan realizar durante el desarrollo.
- **El equipo debe de tener experiencia con la gestión de conflictos**, pues va a ser un problema habitual durante el desarrollo.
- Al no disponer de revisiones de código durante las fusiones de ramas, todo el código que se añada a la rama debe de ser de calidad. En este aspecto, es bastante común emplear técnicas de **programación en pareja** (pair programming en el argot) que permitan cerciorar que el código que se despliega en master sea directamente de calidad.

Nota: *los conflictos que se dan en esta estrategia tienen la misma funcionalidad que los que se dan al realizar cambios a través de ramas. Estos conflictos enfrentan la versión local contra la versión que se encuentre en el repositorio común.*

Ventajas y desventajas de master-only flow

Algunas de las **ventajas** de esta forma de desarrollar son las siguientes:

- Ideal para **equipos y/o productos muy pequeños.** Este tipo de equipos y productos no requieren de la paralización perdida al no emplear ramas que sí que pueden requerir equipos y productos de mayor tamaño.
- **Reducción del tiempo invertido en tareas diferentes a la programación**. La pérdida de tiempo asociada a revisiones de código y gestiones de ramas es eliminada del proceso de desarrollo.
- **Invita a la realización de test robustos**. Al reducirse los ámbitos de prueba durante procesos como los *merge request*, resulta interesante la posibilidad de automatizar al máximo la capacidad del testeo del propio sistema.

En cambio, sus **desventajas** son:

- **Requiere de un equipo con unas cualidades muy concretas.**
- **Pérdida de visibilidad.** La no utilización de ramas dificulta entender el motivo de ser de cada uno de los cambios y puede dificultar el seguimiento del desarrollo en el caso de que se produzcan varios desarrollos en paralelo.
- **Pérdida de la posibilidad de mantener varias versiones al mismo tiempo**. Esta acción no es posible realizarla al no disponer de ramas para cada una de las versiones. Sí que se pueden generar versiones de estados concretos del producto, pero estas versiones no pueden ser corregidas.
- **Muy difícil de llevar a cabo en equipos y/o proyectos grandes**. Esto se debe a la pérdida de paralelismo asociada a la no utilización de ramas requerida para este tipo de desarrollos.
- **Los conflictos son frecuentes y duros.** Al aplicar los cambios directamente sobre master, en el momento en el que hay más de una persona trabajando sobre el mismo proyecto la aparición de conflictos se vuelve muy frecuente.
- **Pueden existir momentos de inestabilidad en el producto.** Estos momentos se pueden deber al despliegue de cambios poco probados directamente sobre master.

4.8. Aplicación sobre la solución

Consideraciones para la selección de una estrategia

A grandes rasgos, resulta muy complejo elegir una estrategia como la más óptima para realizar un desarrollo. Cada una de las estrategias funciona mejor en algunos aspectos y peor en otros.

Para la correcta elección de una metodología de *branching*, **se debe analizar el caso sobre el que se quiere aplicar** y seleccionar aquella que más se encaje con sus requisitos y necesidades. Algunos de los aspectos a considerar y sus respectivos datos para la solución asociada al trabajo son:

- **Dimensiones del equipo de trabajo**. En el caso de la solución que se está evaluando en este trabajo, en torno a 9 personas. Factores como la experiencia también pueden resultar relevantes, aunque no se van a incluir en este análisis.
- **Necesidad (o no) del mantenimiento de varias versiones del producto**. En el caso de la solución a aplicar no existe esta necesidad.
- **Disponibilidad de un entorno de pruebas automatizadas**. La realización de

pruebas, así como la automatización de procesos son uno de los requisitos principales de la solución analizada en este trabajo.

- **Duración y tamaño del desarrollo a realizar**. Considerando las dimensiones del equipo de desarrollo, la necesidad de brindar mantenimiento y carga de trabajo de la aplicación, se podría considerar que el caso práctico tiene una duración media.
- **Metodologías a aplicar**. En el caso de la solución que se está evaluando en el trabajo, como requisito se deben emplear las metodologías SCRUM, integración continua y despliegue continuo.

Evaluación de las diferentes estrategias para su empleo durante el desarrollo de la solución

GitFlow es la estrategia de branching más equilibrada y completa de evaluadas. Describe de manera cerrada la forma en la que se desarrolla en el equipo, ayudando así a la trazabilidad de los cambios.

Su mayor debilidad reside en la cantidad de ramas necesarias para su uso, y la pérdida de tiempo que ello conlleva. Dadas las características del caso práctico, *GitFlow* es una estrategia que encaja en el marco de desarrollo de este proyecto.

GitHubFlow es, dentro de las estrategias que emplean ramas, la más sencilla de todas las evaluadas. Se trata de una estrategia muy versátil y que combina de manera ideal con la entrega continua.

Una de sus desventajas es que la sencillez de la propia estrategia hace que se pueda perder cierta trazabilidad del desarrollo del producto. De nuevo, dadas las características del producto a desarrollar, *GitHubFlow* también es una estrategia que encaja bien en el marco de desarrollo.

GitLabFlow presenta su mayor ventaja en la seguridad extra que produce el realizar varias revisiones del código durante el desarrollo y de manera previa al despliegue. El coste que ello tiene es aumentar la cantidad de tiempo invertido en revisiones de código.

Considerando el contexto de la aplicación sobre la que se está enmarcando el trabajo, las desventajas de *GitLabFlow* pesan más que sus ventajas.

Aunque el sistema tiene ciertos requisitos de robustez, el hecho de que esta estrategia retrase los procesos de entrega (lo cual choca con la metodología CD) y de que la robustez requerida puede ser probada a través de un buen proceso de automatización de pruebas (pruebas que están consideradas dentro de los requisitos de esta solución), se concluye que *GitLabFlow* no es la estrategia más adecuada para este caso de uso.

El desarrollo ***trunk based*** es una de las pocas estrategias evaluadas que permite realizar el mantenimiento de varias versiones del producto desplegadas. Por el contrario, aportar este versionado tiene la desventaja de que puede llegar a reducir el tiempo de desarrollo y mantenimiento de las futuras funcionalidades.

Trunk based, sería una de las estrategias candidatas a considerar si un requisito fuese tener la necesidad de mantener varias versiones del producto desplegadas a la vez. Sin embargo, como la solución a considerar en este trabajo no dispone de dicho requisito se desestima esta opción.

ReleaseFlow, es una estrategia diseñada para su empleo en equipos y proyectos de un muy gran tamaño. Su principal desventaja es que se pierde gran parte de la trazabilidad del desarrollo.

Considerando los aspectos anteriormente mencionados de la solución objetivo, para un desarrollo de las dimensiones que se están tratando (tanto a nivel de producto, como de personal), no existe la necesidad de aplicar una estrategia que se centre especialmente en su desempeño en trabajos y/o equipos de gran tamaño.

Esta desestimación se intensifica si se considera la pérdida de trazabilidad que implica su uso. *ReleaseFlow* no es una de las estrategias prioritarias a seguir para este tipo de soluciones.

Por último, ***Master-only flow***, es una estrategia que funciona bien en unos casos de uso muy concretos.

Debido los grandes requisitos que tiene la estrategia en ámbitos como: nivel de personal, falta de diseño para el despliegue continuo (requisito de la solución objetivo) o posibles problemas durante su empleo con aplicaciones de mayor tamaño, *Master-only flow* tampoco es una de las candidatas a seguir.

Selección de la estrategia a seguir

Tras el análisis general de las diferentes estrategias a evaluar, sólo existen dos estrategias que destacan como candidatas para su aplicación en el caso práctico sobre el que se ambienta esta solución *GitHubFlow* y *GitFlow*.

Si bien cualquiera de las dos opciones encajaría debidamente en el marco de desarrollo, ***GitHubFlow*** tiene unos matices que hacen que se trate de una mejor opción para la solución que estamos tratando. Estos matices son:

1. **La *GitHubFlow* está diseñada para ser más rápida y sencilla que *GitFlow***. Esto finalmente se traduce en que la estrategia va a encajar mejor con las metodologías SCRUM, integración continua y despliegue continuo que forman

parte de los requisitos de la solución a tratar.

2. Como requisitos, **la aplicación va a disponer de un sistema de automatización de pruebas, y de despliegues**. La necesidad de la disposición de estas condiciones es la principal desventaja de la estrategia *GitHubFlow*. Como la solución a considerar tiene cubiertas estas necesidades como requisitos, resulta sencillo el empleo de dicha estrategia y el uso de sus ventajas.

Capítulo 5 – Automatización del proceso de desarrollo (CI / CD)

5.1. Introducción

El objetivo de este capítulo es brindar a nuestro caso práctico, de una serie de **procesos automatizados** que permitan, de la manera lo más eficiente posible, el seguimiento las metodologías de desarrollo de integración y entrega continua. Estos procesos van a estar directamente relacionados con los capítulos vistos hasta ahora.

Con dichos objetivos en mente, a lo largo de este capítulo vamos a considerar los siguientes **requisitos funcionales**:

- Simplificación de los procesos de desarrollo y mantenimiento
- Disminución de errores (tanto de producto como de desarrollo)
- Desarrollo en diferentes despliegues
- Automatización de los despliegues y entregas de producto
- Definición y seguimiento de estándares de calidad de código
- Desarrollo y aplicación de pruebas
    - Pruebas unitarias
    - Pruebas de integración
    - Pruebas de UI
    - Pruebas funcionales
    - Pruebas de carga y rendimiento

- Resolución de problemas típicos
- Permisibilidad de aplicación de metodologías ágiles
    - Control de versiones
    - Gestión de tareas
    - Integración y despliegue continuos

De la misma forma, también vamos a considerar los siguientes **requisitos no funcionales**:

- Control de versiones mediante GIT
- Aplicabilidad en diferentes formatos de tecnologías WEB
    - Simple Object Access Protocol (SOAP)
    - API REST (representational state transfer)
- Empleo de tecnologías desarrolladas por Microsoft
    - *Visual Studio*
    - *Visual Studio Code*
    - Microsoft SQL Management Studio
    - IIS Express
    - Windows 10
- Empleo del *Oracle Service Bus* (OSB)
- Empleo de sistemas de BBDD relacionales
- Empleo del repositorio de artefactos Nexus

5.2. Conceptos previos

5.2.1. *Continuous integration* (CI)

Se conoce como **integración continua**, *continuos integración*, o por su acrónimo CI a una metodología de desarrollo software consistente en la realización de actualizaciones progresivas en el menor tiempo posible. De manera previa a su integración con el resto del producto, estas actualizaciones deben ser correctamente probadas y revisadas por los desarrolladores. La automatización de las pruebas y la introducción de los cambios juegan un papel esencial en el correcto desempleo de esta metodología. Siguiéndola, se consiguen los siguientes objetivos:

- **Aceleración del proceso de integración de las nuevas funcionalidades**. Al realizar integraciones constantes del producto, las funcionalidades llegan antes a los entornos comunes. Este concepto se da, aunque estas integraciones se hagan en incrementales más pequeños que con un proceso de desarrollo

tradicional.

- **Detección fallos de concepto o definición del producto con mayor rapidez**. Al llegar las nuevas funcionalidades con mayor frecuencia a los entornos comunes, se realizan pruebas de concepto sobre las mismas con mayor rapidez que si no se aplicase CI. Esto finalmente se traduce en la detección prematura de fallos de concepto o definición.
- **Validación del correcto funcionamiento del incremental**. El proceso de integración continua obliga a, de manera prematura a la integración de una funcionalidad, validar su correcto desempleo funcional. Estas validaciones deben hacerse de manera automatizada y deben probar el producto en su totalidad y no sólo el incremental a introducir.
- **Permite la realización de entrega continua (CD)**. La aceleración del proceso de integración abre la posibilidad de entregar las nuevas funcionalidades en cuanto son desarrolladas, naciendo así el concepto de la entrega continua.

Para llevar a cabo esta idea de manera eficaz, resulta esencial disponer, de manera previa al desarrollo del producto de:

- **Una infraestructura que permita demostrar que el código desarrollado es correcto**. Esta infraestructura debe ser capaz de probar el código en su totalidad y debe ser capaz de replicar las pruebas realizadas con frecuencia y velocidad (se va a requerir de su uso durante cada integración).
- **Automatización de los procesos relativos a las integraciones de código**. Esta automatización se realiza a través de *pipelines* y debe incluir dentro del proceso tanto la construcción, como el *testeo* del código. Forma un papel esencial para que el aumento de las integraciones no suponga un sobrecosto temporal inabordable.
- **Empleo de un flujo de trabajo y gestión de ramas adecuado para esta metodología**. Para el empleo de CI, es necesaria la elección de una metodología de trabajo que permita este tipo de desarrollos (no todas las metodologías de trabajo cumplen los requisitos necesarios).
  En este aspecto funcionan bastante bien las metodologías ágiles y las gestiones de ramas que permitan una rápida integración de los cambios sin sobrecostes en el proceso de integración.

5.2.2. *Continuous delivery* (CD)

***Nota**: el acrónimo CD es ambiguo. Puede referirse tanto a continuous delivery, como a continuous deployment. En lo sucesivo, a no ser que se especifique lo contrario, siempre*

que se emplee va a hacer referencia a continuous delivery.

Se conoce como **entrega continua**, continuous delivery, o por su acrónimo CD a otra metodología de desarrollo, la cual consiste en la realización de entregas al cliente en el menor tiempo posible (PE: en el caso de una metodología ágil basada en sprints, realizar al menos, una entrega por dicha unidad de tiempo).

El CD se basa tanto en la velocidad de integración de cambios, como en la reducción del tamaño de los mismos, conceptos conseguidos a través del empleo de la integración continua. Reducir el tamaño del incremental entregado permite:

- **Facilitar la depuración y testeo del producto.** Al ser el incremental de menor tamaño, las pruebas aplicadas sobre el mismo son más cortas y sencillas.
- **Detección prematura de funcionalidades finalmente no deseadas.** Al entregarse las funcionalidades con mayor frecuencia, el usuario final puede probar estas funcionalidades con mayor antelación de cómo se haría a través de una metodología tradicional.
  De estas pruebas pueden nacer posibles errores de definición de funcionalidades que no se tuvieron en cuenta en los procesos anteriores de desarrollo.
- **Mejora la sensación de velocidad de desarrollo.** Al realizarse las entregas al usuario final con mayor frecuencia, la percepción de la continuidad del desarrollo del producto de éste mejora. Esto se suele traducir en una valoración más positiva por su parte al proceso de desarrollo del producto.

La principal ventaja de esta metodología reside en el **ahorro de tiempo**. La detección de funcionalidades no deseadas de manera prematura permite un gran ahorro de tiempo puesto que sólo se habrá perdido el tiempo de desarrollo del pequeño incremental de la funcionalidad. En cambio, mediante el uso de una metodología de entregas a largo plazo, se podría haber llegado a perder todo el tiempo de desarrollo de la funcionalidad por completo debido a la no detección de la funcionalidad no deseada derivada de no entregar el producto antes.

Como requisito, al igual que sucedía con la CI, para la entrega continua también resulta esencial la **automatización** de los procesos de producción y entrega del producto. La entrega continua, va a ampliar los procesos de automatización ya desarrollados para la integración continua (los *pipelines*), de tal forma que también permita realizar la entrega del producto de manera automática, rápida y fiable.

Para más información sobre la integración o entrega continuas, se puede consultar el libro Continuous Delivery, de Jez Humble y David Farley [58].

5.2.3. *Pipelining*

Conceptos básicos de los pipelines

Se conoce como ***pipeline*** al mecanismo de automatización encargado de llevar a cabo el flujo que permite la aplicación de las metodologías CI y CD. Arquitectónicamente hablando, los *pipelines* están compuestos por **etapas**. Dichas etapas, a su vez, agrupan conjuntos de **tareas** dentro del *pipeline*. Un ejemplo podría ser la etapa Test, en donde se puede, por ejemplo, establecer una tarea por cada uno de los tipos test.

Se puede establecer una analogía de estos conceptos junto con una saga literaria: la saga completa sería el *pipeline*, que forma una estructura lógica completa y variada. De la misma forma, los libros, que forman parte del mismo tema global, pero están más centrados en ciertos aspectos serían las etapas. Y, por último, los capítulos (la unidad más específica) serían las diferentes tareas.

Lógica de ejecución de los pipelines

Durante la **ejecución del pipeline**, cada una de las etapas que componen al pipeline se ejecutan **de manera secuencial**, que, a su vez, harán lo propio con las diferentes tareas.

Tras la ejecución de una tarea, ésta se marca con uno de los siguientes estados resultado: **success** (éxito, si todo ha ido bien), **warning** (advertencia, si se ha dado algún error leve) o **error** (en caso de error grave). Estos estados se escalan desde las tareas hacia la etapa en la que se encuentran de la siguiente manera:

- Si una tarea finaliza como *error*, la etapa será finalizada como ***error*** (independientemente del estado del resto de tareas). Cuando esto sucede, se suele abortar el resto de ejecución de las tareas puesto que, al haberse detectado un error, los cambios serán descartados.
- Si han finalizado todas las tareas, ninguna ha finalizado en *error* y al menos una ha finalizado como *advertencia*, la etapa en su totalidad será marcada como ***advertencia*** (en esta ocasión no se aborta la ejecución del resto de tareas puesto que los cambios no se van a descartar).
- Si han finalizado todas las tareas y todas ellas han finalizado en *éxito*, la etapa será marcada como ***éxito*** también.

La lógica de escalado desde las etapas hasta el *pipeline* en su totalidad es la misma, quedando así definido el resultado final del *pipeline*.

Arquitectura de un pipeline

Las diferentes etapas que componen los *pipelines*, pueden variar enormemente en función del producto y de la metodología a aplicar. No obstante, en rasgos generales, todo *pipeline* correcto y completo enfocado a su uso para CI/CD, debe incluir, al menos las siguientes etapas (siguiendo el mismo orden):

1. **Test**. Como su nombre indica, esta es la etapa en la que se va a probar el código desarrollado. En este aspecto, resulta esencial que los test se encuentren actualizados respecto del código y sean lo más completos posible.
   Dependiendo del motivo de la ejecución y la criticidad de su tipo (si es sólo un desarrollo local, o se trata de un despliegue o similar), podrían variar los tipos de test que ejecuta (PE: sólo unitarios). Esta etapa no debe poder finalizarse como *warning*.
2. **Quality**. Esta es la etapa encargada de comprobar que todos los componentes del proyecto involucrados siguen los estándares de calidad acordados por el equipo.
   En el caso de que alguno de los estándares definidos no sea correspondido, las herramientas de *Code Quality*, muchas veces disponen de mecanismos para corregir los problemas automáticamente. Estos comportamientos se suelen desactivar en los *pipelines* (pese a su correcto funcionamiento) debido a que:
   1. Dichas correcciones no han sido supervisadas por ningún desarrollador y este código iría directamente sobre el cliente.
   2. Dichas correcciones no han sido probadas durante los test (puesto que la etapa es anterior).

   En su lugar, es habitual terminar la etapa como *advertencia* y no como *error* para que dichos cambios se corrijan por un desarrollador.
   OJO: hay que tener mucho cuidado con este tipo de prácticas, puesto que, en el caso de extenderse en el tiempo o suceder de manera habitual, podría darse el caso de que los desarrolladores se acostumbrasen a las *advertencias* y terminasen tratándolas como *éxitos*.
3. **Release**. Esta etapa está encargada de construir el binario o empaquetado del producto y desplegarlo donde corresponda. Esta etapa varía enormemente en función del proyecto y su tipo (puede requerir de un empaquetado y subida a un repositorio de artefactos, del lanzamiento de scripts en BBDD…).
   En el caso de existir documentación sobre el producto o el versionado del mismo, durante esta etapa también se debe desplegar, junto con el producto, esta información.
   Como es lógico, release debe de ser la última etapa debido a que sólo se debe entregar el producto en el caso de que se hayan pasado todas las métricas de seguridad definidas. Esta etapa no debe poder finalizarse como *warning*.

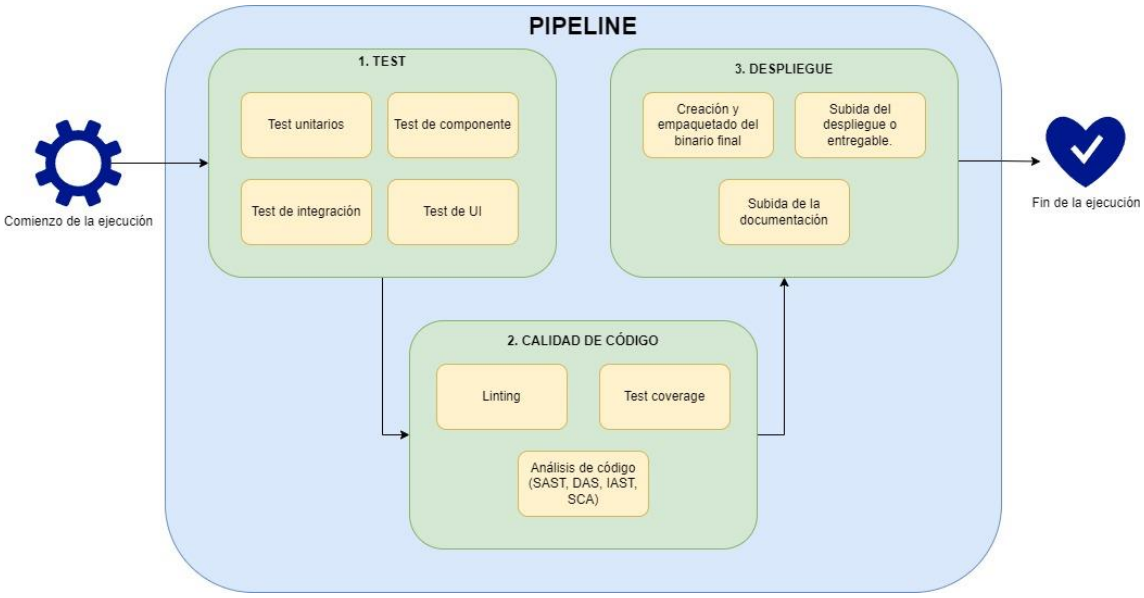


Ilustración 22: Esquema de la arquitectura básica de un pipeline enfocado a CI/CD.

Adicionalmente, aunque con un carácter más opcional, también es habitual contar las siguientes etapas:

1. **Prepare**. Esta es una etapa que debe ejecutarse la primera de todas. Su objetivo es preparar el entorno en el que se va a realizar la compilación del producto. Durante su ejecución se preparan y cachean todas las dependencias del producto para que puedan ser reutilizadas durante todo el ciclo de vida el pipeline.
   Normalmente, durante esta etapa también es habitual descargar el código fuente del producto. No realiza comprobaciones y no debe poder finalizarse como *warning*.
2. **Docs**. Esta es la fase encargada de generar la documentación de manera automática. Es habitual que los proyectos orientados al empleo de metodologías CI/CD se encuentren configurados para ser capaces de autogenerar la documentación del proyecto. De esta forma, la documentación del producto queda siempre actualizada y los desarrolladores no tienen que realizar esfuerzo activo durante la creación de la misma.
   Según la criticidad de la documentación generada, la etapa podría llegar a finalizar como *warning*, aunque es más habitual finalizar como *error* (debido a su relación con el proceso de despliegue).

Entornos de ejecución de los pipelines

Para garantizar la estanquidad de los procesos de ejecución de *pipelines* y su no dependencia de entornos de desarrollo concretos, las diferentes tareas que componen

los *pipelines* se ejecutan en máquinas diferentes a aquellas en las que se desarrolla. Estas máquinas pueden estar tanto alojadas dentro de la **infraestructura propia** como en un **servicio cloud** externo como podría ser AWS.

Independientemente del lugar físico en el que se encuentren, estas máquinas ejecutoras deberán tener sus propias credenciales de acceso a la información de tal forma que se les permita realizar todas las acciones que sean necesarias para el desempeño de sus tareas. Estas credenciales deben identificar al *pipeline* de manera inequívoca, para que, si en algún momento se da un error, éste sea fácil de localizar.

Nota: *Se debe tener especial cuidado a la hora de definir esta configuración de credenciales para no generar vulnerabilidades del sistema.*

Adicionalmente, se debe garantizar que cada una de las tareas que se ejecuten en dichas máquinas y sus resultados no interfieran con el resto de tareas que también se ejecutan en las mismas máquinas (ya sean posteriores, anteriores o paralelas). Para conseguir esto es habitual que las diferentes tareas se ejecuten sobre contenedores Docker [59].

Para aumentar la velocidad de ejecución de las tareas y reducir el peso del sistema de *pipelining* se suelen definir diferentes **contenedores Docker** que permiten ejecutar las diferentes tareas pero que contienen únicamente aquellas herramientas que sean estrictamente necesarias para su ejecución. Para que esta lógica funcione, además de la definición de los contenedores, es necesario que los ejecutores de tareas se enlacen con el producto (PE: un ejecutor de .NET asociado a un producto .NET).

A través de dichos contenedores se evita la necesidad disponer de una máquina que tenga instaladas todas las dependencias de todas las herramientas de todas las tecnologías que se usen en la empresa.

Visualización de los resultados de los pipelines

Como la ejecución de los pipelines se realiza en máquinas diferentes a las empleadas para el desarrollo, es necesario hacer **accesibles desde el exterior** todos los reportes y resultados derivados de la ejecución del pipeline (informes de *testing*, *code quality*, documentación generada…). Es habitual que se almacene el histórico de todos los reportes generados hasta la fecha.

Para simplificar los procesos de desarrollo, las direcciones a dichos recursos deben de ser fáciles de acceder y encontrar. En este aspecto, muchas herramientas de gestión de versionado de código fuente permiten, dentro de los repositorios de código, añadir unas **pegatinas** (botones de acceso directo) relacionadas con el estado y los reportes de la última ejecución del *pipeline*.

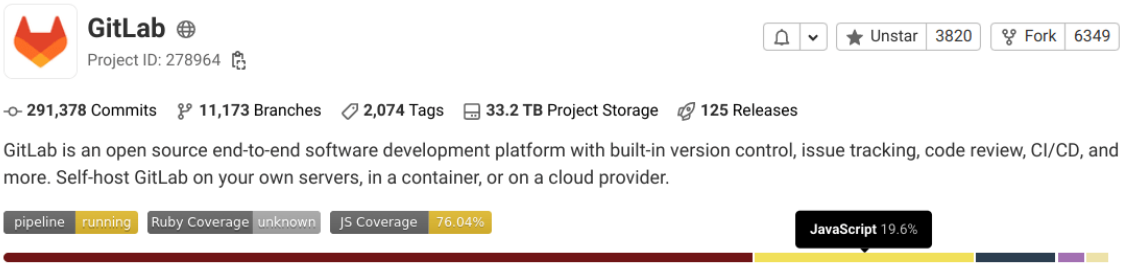


Ilustración 23: Ejemplo de la vista repositorio en GitLab en el que se aprecian pegatinas que incluyen algunos estados de la ejecución del pipeline. La imagen está tomada de la documentación oficial de GitLab [60].

5.2.4. *Jenkins*

Introducción a Jenkins

Jenkins es una herramienta de automatización de código abierto que permite poner en práctica los *pipelines* necesarios para el correcto desarrollo del producto dentro de las metodologías CI/CD.

En la práctica, Jenkins actúa como un **servidor** el cual es capaz de conectarse con los repositorios de código y los componentes que componen el resto de la infraestructura y entornos de desarrollo. A través de dicha conectividad y, mediante una serie de hilos ejecutores, es capaz de llevar a cabo en la máquina que lo aloja, todas las tareas de validación, testeado, compilación o despliegue de tenga definida el *pipeline*.

Para mantener la estanquidad de las ejecuciones que realice, es recomendable que el servidor que aloja Jenkins no se encuentre en ninguno de los equipos de desarrollo, y que se trate de una máquina externa (ya sea integrada en la infraestructura propia, o a través de un servicio cloud).

Interfaz gráfica de Jenkins

Para facilitar el acceso al servidor, Jenkins dispone de una **interfaz gráfica de usuario** (GUI) montada sobre una aplicación web. Desde dicha aplicación web Jenkins ofrece, además de una interfaz amigable, la posibilidad de **configurar la herramienta** de una manera cómoda y sin necesidad de acceder directamente a la máquina en la que se encuentra alojada la herramienta. Las principales acciones y configuraciones que se pueden realizar desde dicha aplicación web son:

- **Instalación de *plugins*.** A través de los *plugins*, se facilita la realización de ciertas tareas, como la sincronización con un repositorio GIT, que de otra manera se deberían hacer de manera manual.
- **Configuración global de la aplicación.** Desde la interfaz es posible establecer la

configuración básica de la aplicación. Algunos de los aspectos más relevantes de esta configuración están relacionados con la gestión de usuarios la protección de acceso a la herramienta.

- **Configuración de claves de acceso.** A través de la interfaz es posible configurar las diferentes claves de acceso a servidores y servicios de manera segura. De esta forma se evita su guardado en el fichero de definición de *pipeline* u opciones similares que podrían suponer una vulnerabilidad para la solución final.
- **Generación de vistas personalizadas.** Desde la interfaz se pueden configurar las diferentes pantallas de visualización de tal forma que sólo se muestre la información deseada por el usuario.
- **Organización de repositorios.** Desde la aplicación web es posible establecer la relación entre *Jenkins* y los diferentes repositorios que van a alojar el código de los diferentes componentes de la solución.
- **Realización y visualización de ejecuciones.** Desde la GUI es posible hacer todas las acciones de lanzado y visualización de las diferentes ejecuciones que se hayan realizado a lo largo del tiempo.

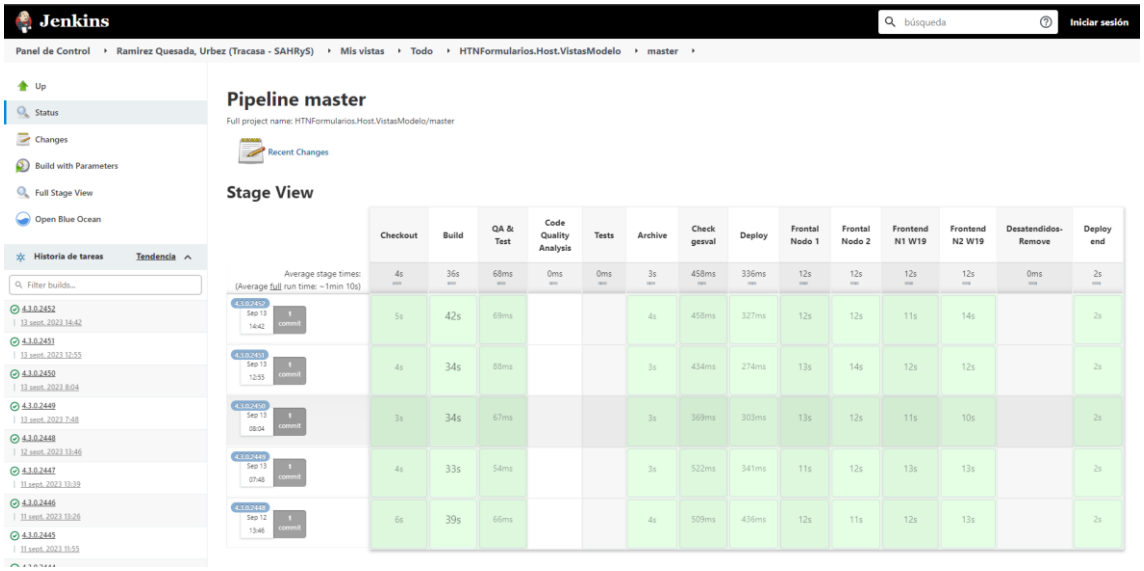


Ilustración 24: visualización del estado de las últimas ejecución de la rama master de un proyecto a través de la GUI de Jenkins.

***Nota:** no se deben confundir los conceptos de "configuración de la herramienta" y "definición de pipelines". Configurar la herramienta hace referencia a la configuración propia de la herramienta: la cantidad de repositorios, configuraciones de la vista, disparadores… En cambio, la definición del pipeline hace referencia a que tareas, etapas y acciones debe hacer un pipeline concreto.*

También es importante considerar que la "definición de pipelines" se puede establecer

tanto desde la "configuración de la herramienta", como a través de unos dicheros de configuración denominados Jenkinsfile.

Métodos de ejecución de los pipelines

Para la ejecución de los *pipelines*, Jenkins dispone de dos opciones principales. La primera y más básica consiste en acceder **desde la GUI** al *pipeline* del repositorio en cuestión y, de manera manual, seleccionar las etapas a ejecutar y lanzar el proceso.

La segunda de las opciones consiste en configurar y establecer una **relación entre *Jenkins* y los repositorios de código**. A través de esta configuración, es posible, mediante el uso de un conjunto de *webhooks* realizar ejecuciones en *Jenkins* de manera **automatizada** a partir de los cambios que se realicen en los repositorios.

A través de esos mismos *webhooks* es posible definir los diferentes **tipos de ejecución** para un mismo proyecto dependiendo de si el disparador es un commit en una rama, un merge, un tag…

La principal ventaja de esta segunda opción es que el desarrollador no debe de tener que realizar de manera manual las ejecuciones cada vez que desee aplicar unos cambios en el código fuente. Esto agiliza el proceso de desarrollo y lo simplifica puesto que los diferentes desarrolladores no necesitan memorizar que tareas debe ejecutar el pipeline según qué tipo de cambio que se esté aplicando.

Otra de las ventajas de la segunda opción es que es posible hacer que se le envíe al desarrollador una notificación (por ejemplo, a través de un correo electrónico) en el que se le indique el estado final del pipeline.

La posibilidad de la ejecución automática es esencial para la correcta implementación de las metodologías CI/CD. Dicha automatización permite al equipo de desarrollo "despreocuparse" del estado del pipeline (salvo si se producen errores) y centrarse en desarrollar las aplicaciones.

Nota: *Los resultados de cualquiera de los dos métodos de ejecución son plenamente visibles desde la GUI de Jenkins.*

Definición del pipeline

Para definir la configuración arquitectónica y operativa de los diferentes *pipelines* que pueden llegar a componer el ecosistema del producto, Jenkins pone a disposición de los desarrolladores dos posibilidades.

La primera posibilidad consiste en definir, **desde la propia GUI** tanto la arquitectura como las tareas a realizar por el *pipeline*. Resulta algo compleja y tediosa debido a su poca flexibilidad. Esta es la opción original que ofrecía la herramienta, pero con el

tiempo ha ido cayendo en desuso en favor de la segunda opción.

La segunda posibilidad consiste en depositar en la raíz de los repositorios de código un fichero ***Jenkinsfile*** [61], en el cual se van a definir las diferentes acciones que componen el pipeline, así como sus dependencias, entornos de ejecución, tareas, etc.

Jenkins, de manera automática, buscará entre los repositorios que tenga debidamente configurados, todas aquellas ramas que tengan un Jenkinsfile. Desde la GUI mostrará dichos repositorios y ramas. Desde la misma vista se puede tanto visualizar los resultados como ejecutar los *pipelines*.

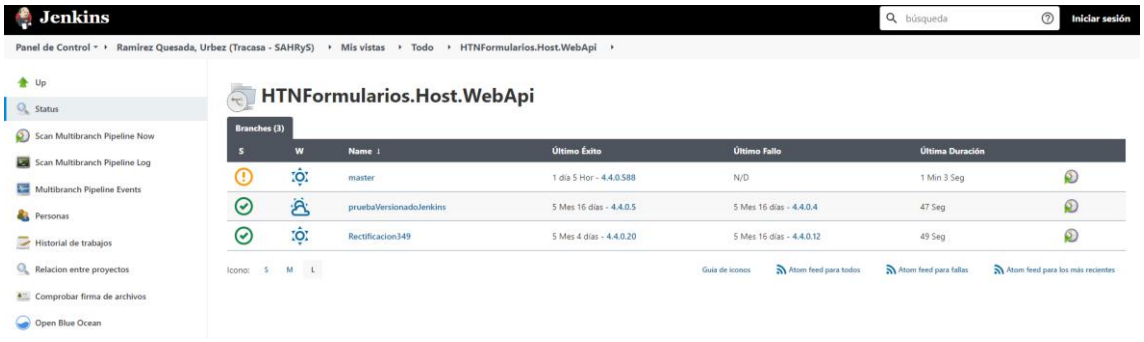


Ilustración 25: Vista de las diferentes ramas de un repositorio que disponen de un Jenkinsfile y el estado de la última ejecución de dichas ramas.

En el caso de que se quieran realizar arquitecturas cerradas, pero con cierta configuración específica para cada repositorio, es posible realizar una combinación entre ambas opciones. A través el método clásico se fija una arquitectura que posteriormente se nutrirá del fichero de configuración Jenkinsfile.

Nota: *nótese que, mediante la opción del Jenkinsfile, resulta muy sencillo (alterando únicamente el Jenkinsfile) establecer cambios de comportamiento en el pipeline en función da rama de código que se esté empleando.*

5.3. Arquitectura del *pipeline* propuesto

Una vez presentados los diferentes conceptos básicos sobre las metodologías CI/CD, los *pipelines* y la herramienta de automatización que se va a emplear en la solución a tratar, es preciso definir **la arquitectura e hitos del *pipeline* propuesta**.

Este *pipeline* debe amoldarse al caso práctico y pretende englobar y poner en práctica

todos los conceptos presentados hasta ahora en los diferentes capítulos de la manera lo más automatizada posible.

5.3.1. Etapa 1 – Preparación

La primera de las etapas que va a componer el *pipeline*, es la etapa de **preparación**. Durante esta etapa, el pipeline va a descargar las diferentes **dependencias** necesarias para el compilado, testeado y despliegue de la solución propuesta. Dichas dependencias quedarán cacheadas para facilitar su uso en etapas y tareas posteriores. Para su implementación, es necesario estas dependencias queden debidamente reflejadas en el fichero de configuración Jenkinsfile.

Adicionalmente, durante esta etapa, también se va a proceder a la descarga de la última versión **del código fuente**. Esta versión, como es lógico, contendrá los nuevos cambios introducidos que deberán ser probados durante la propia ejecución del *pipeline*.

En esta primera etapa también se va a generar una versión binaria del código fuente. El binario fruto de dicha compilación también debe ser cacheado y será reutilizado en etapas posteriores (como, por ejemplo, en la de test).

5.3.2. Etapa 2 – *Test*

Durante la segunda etapa, el *pipeline* va a **verificar** el correcto funcionamiento del código desarrollado. Ésta es la etapa más importante del pipeline, puesto que es en donde se va a decidir si el código propuesto es válido o no para su aceptación y posterior despliegue. Este proceso es esencial para poner en práctica debidamente una metodología CI.

Durante esta etapa, se van a emplear todos los **test automatizados** desarrollados por el equipo para garantizar que no se haya roto el código previo, y que las nuevas funcionalidades cumplen con los requisitos establecidos. Estas pruebas deben estar programadas y ejecutadas siguiendo la lógica y herramientas descritas en "Capítulo 3 – Testing". En el caso de la solución propuesta en este trabajo, encontramos 3 tipos de test automatizados:

- **Pruebas unitarias.** Rápidas y centradas en el correcto desempleo funcional.
- **Pruebas de integración.** De velocidad media y centradas en la correcta comunicación entre componentes.

- **Pruebas de interfaz de usuario.** Lentas y centradas en probar la solución en su totalidad.

Debido a la **alta cantidad de ejecuciones y cambios** introducidos fruto del empleo de la metodología CI y al gran peso temporal y tecnológico de la ejecución de todas las pruebas de código, no resulta práctico lanzar, para cada cambio, la totalidad de las pruebas disponibles. De no ser así, se estaría reduciendo la velocidad de desarrollo derivada de la espera a la ejecución de las pruebas. Esta reducción chocaría de manera directa con el empleo de la metodología CI.

Para afrontar las ejecuciones sin disminuir el ritmo de desarrollo, se van a definir dos **tipos de ejecuciones** las cuales se emplearán en función del tipo de disparador que haga efectiva la ejecución del *pipeline*:

1. **Batería de pruebas "simple"**. Esta batería de pruebas está compuesta únicamente de los **test unitarios**. La batería pretende ser lo más rápida posible y a su vez comprobar la validez funcional de los diferentes componentes del producto.
   Esta variante de ejecución se va a llevar a cabo siempre que los cambios que se estén introduciendo no tengan un impacto crítico sobre la solución, es decir, que tengan como destino la **misma rama** de la que nacen (o formen parte del proceso de creación de una nueva) o que no se esté realizando ni un despliegue ni un *merge*.
   Mediante el empleo de únicamente las pruebas unitarias se pretende obtener un ahorro de tiempo a costa de la no validación completa de la solución fruto de que en este punto no se va a desplegar el producto y por lo tanto no es crítico revisarlo a nivel global.
2. **Batería de prueba "completa"**. Esta batería de pruebas está compuesta de **la totalidad de las pruebas automatizadas** disponibles para la solución. Se trata de conjunto de pruebas más amplio y pesado que el anterior.
   Esta variante de ejecución se va a emplear siempre que los cambios que producen la ejecución del *pipeline* formen pare un proceso más crítico que un *commit* normal. Este tipo de procesos se dan en el caso de un ***merge*** de una rama a otra, o de que el producto que se está evaluando vaya a ser **desplegado** en etapas posteriores.
   Su objetivo es conseguir el **máximo nivel de verificación** del sistema independientemente del coste temporal y tecnológico que esto suponga. Mediante estas pruebas se pueden llegar a encontrar errores que la batería de prueba "sencilla" no ha sido capaz de encontrar.

***Nota**: todas las herramientas de testing propuestas en el capítulo correspondiente disponen de una versión CLI. Esta versión las habilita para ser ejecutados desde una*

consola de comandos y, por ende, a través de una ejecución de un pipeline de Jenkins.

Tras la ejecución de las diferentes pruebas, todos los **resultados obtenidos** deben quedar disponibles para su posterior revisión de la manera más cómoda posible para el desarrollador. En este aspecto, es habitual tanto mostrar los resultados directamente por pantalla durante la ejecución del pipeline, como generar un reporte que será accesible de manera externa con todos los resultados de las pruebas.

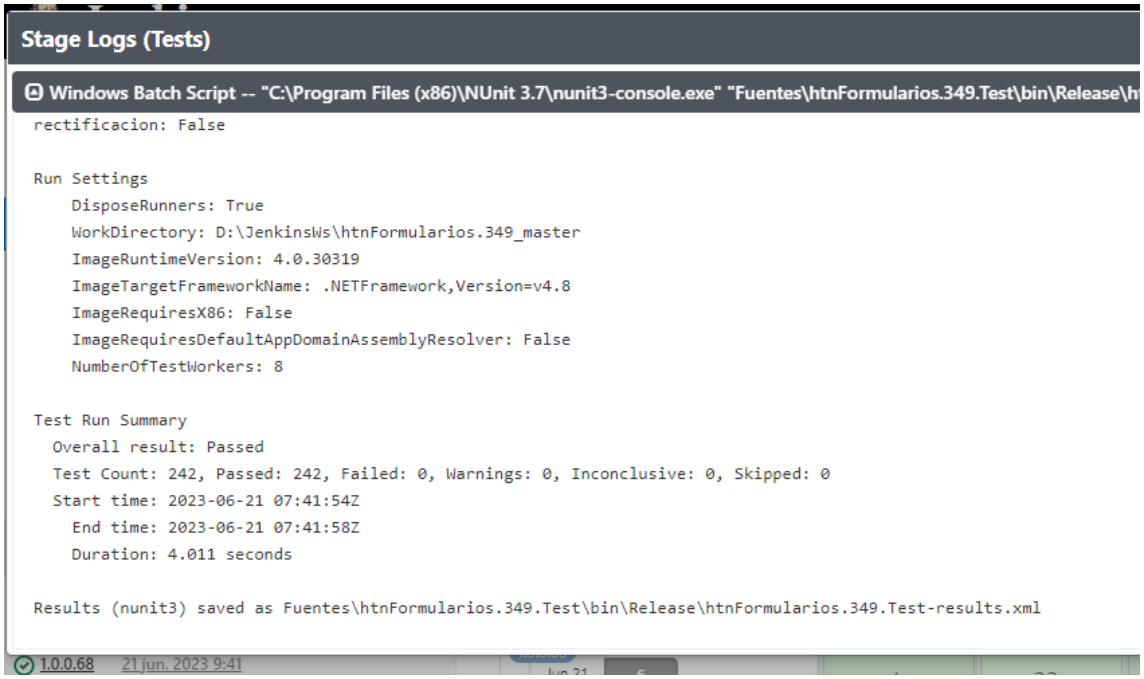


Ilustración 26: Vista de los resultados obtenidos durante la ejecución de unos test unitarios a través de la GUI de Jenkins. En la imagen también se aprecia la ruta en la que se ha almacenado el reporte que incluye dichos resultados.

***Nota:** al tratarse de la etapa la que garantiza tanto el correcto desarrollo del producto como el correcto empleo de la metodología CI, es esencial que, en el caso de que se detecte cualquier fallo durante su ejecución, se pare por completo el pipeline y se descarten los nuevos cambios introducidos en el producto.*

5.3.3. Etapa 3 – Calidad de código

Durante la siguiente etapa, el *pipeline* va a evaluar las diferentes configuraciones de **calidad de código** establecidas por el equipo. Estas comprobaciones se deben a realizar sobre todo el código que forme parte del producto que esté siendo tratado en el *pipeline.* Para la realización de las validaciones, se deben emplear las diferentes herramientas de calidad de código propuestas en los apartados correspondientes (ver "Capítulo 2 – Code Quality") a través de sus versiones CLI.

La forma de visualización de los **resultados** debe ser la misma que en la etapa de test. Los resultados de las diferentes evaluaciones deben estar disponibles tanto desde la GUI durante la ejecución, como a través de los diferentes reportes generados por las herramientas (los cuales deben ser accesibles por todos los desarrolladores).

Durante la realización de las comprobaciones, los diferentes errores de calidad detectados se deben clasificar en dos grupos:

- **Errores críticos.** Estos son los errores relacionados con la seguridad, vulnerabilidades, *explotis*, dependencias deprecadas…
  Estos errores deben ser tratados como un **error** de aplicación y deben ser corregidos de manera previa a la inserción de los cambios. En el caso de detectarse un error de este tipo se debe parar la ejecución del *pipeline* y finalizar como error.
- **Errores leves.** Estos son aquellos errores relacionados con el estilo de código, simplificaciones, duplicidades de código…
  Aunque estos errores no necesariamente deben ser tratados como error, sí que es necesario corregirlos de cara a futuras integraciones. En el caso de detectarse un error de este tipo, se debe finalizar la etapa como **advertencia** permitiendo continuar con el resto del flujo del *pipeline*. Una excepción de este comportamiento se da en el caso de los despliegues, en donde estos errores también deben considerarse como errores.

Con la finalidad de evitar que el equipo de desarrollo se acostumbre a ver estados de advertencia llegando a asimilarlos como normales e ignorándolos, se van a añadir, además de la lógica ya mencionada, las siguientes restricciones:

- **No permitir finalizar dos *pipelines* consecutivos como advertencia.** En el caso de que un *pipeline* vaya a finalizar como advertencia, de manera previa a su finalización se debe comprobar el estado de su anterior ejecución.
  En el caso de que dicho estado sea una *advertencia o un error*, la ejecución pertinente deberá finalizar como error.
- **Establecimiento de un umbral máximo.** Mediante un pesaje basado en el tipo de error y su criticidad es posible calcular la gravedad global de los errores detectados. A través de dicho valor es posible marcar la etapa como error en el caso de que la cantidad de errores supere el umbral establecido.

Nota: *algunas de las herramientas de validación (especialmente aquellas relacionadas con el ámbito de la seguridad) pueden requerir de una gran cantidad de tiempo para su ejecución. Aunque en la solución propuesta no está originalmente considerado, para la calidad de código también sería posible definir dos flujos de desarrollo dependiendo del*

tipo de ejecutor del pipeline (mismo concepto que el aplicado para la etapa de test).

5.3.4. Etapa 4 – Despliegue

La última etapa del *pipeline* de la solución propuesta va a ser la encargada de la realización del **despliegue del producto** de un entorno a otro. Esta es una **etapa opcional** que será únicamente ejecutada en el caso de que el motivo de la ejecución del *pipeline* sea un despliegue. La etapa tan sólo debe ser ejecutada en el caso de que todas las etapas anteriores hayan sido ejecutadas de manera satisfactoria.

Durante su ejecución se van a construir los diferentes **binarios** a partir del código fuente en su forma más óptima posible permitida desde el compilador. Posteriormente, dichos binarios, junto con toda la documentación asociada existente (la cual deberá existir de manera previa en el repositorio) y sus dependencias, serán debidamente empaquetados, subidas al repositorio de artefactos Nexus y llevados a los diferentes servidores que compongan el nuevo entorno sobre el que se esté desplegando el producto.

Nota: *la versión del artefacto generada y el entorno objetivo proviene de un tag relacionado con el commit que provoque la ejecución del pipeline. Dicho tag seguirá la nomenclatura* <entorno>_V<versión>. Jenkins es capaz de abstraer dichos valores y ponerlos en uso durante la ejecución del pipeline.

Mediante las subidas al repositorio de artefactos, el equipo de desarrollo podrá mantener un **histórico** de todas las versiones que han existido desplegadas en los diferentes entornos a lo largo del tiempo.

Nota: *en esta etapa de deben realizar todos los despliegues que sea técnicamente posible. En el caso de la solución objetivo, esta condición incluye todos los componentes salvo el orquestador OSB. Lamentablemente, los despliegues del componente OSB requieren de su realización de manera manual.*

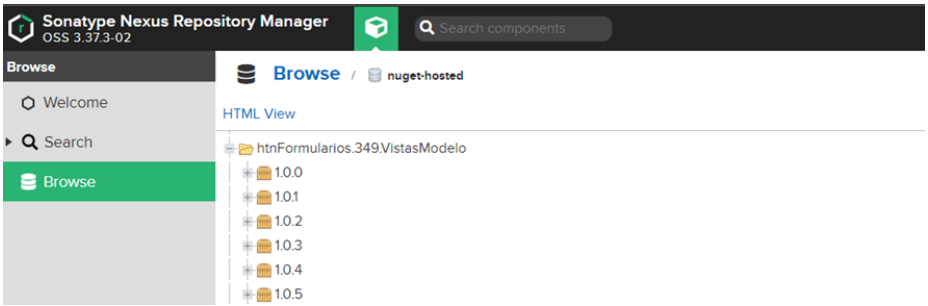


Ilustración 27: Ventana del repositorio de artefactos Nexus donde se puede comprobar el histórico de las diferentes

versiones desplegadas a lo largo del tiempo.

5.4. Implantación durante el proceso de desarrollo de la solución

Métodos de ejecución propuestos para el pipeline

El método principal de ejecución del pipeline que se va a emplear durante el desarrollo de la solución va ser **automático**. Para ello, es necesario sincronizar, a través de un conjunto de webhooks los diferentes repositorios de código con Jenkins. Principalmente se deben diferenciar 4 tipos de ejecuciones diferentes:

- **Ejecución manual.** Los diferentes implicados acceden a la GUI de Jenkins y optan por ejecutar desde dicha interfaz el *pipeline*. La etapa de despliegue no debe poder ser ejecutada mediante esta forma de actuar. La batería de ejecución de los test a lanzar será seleccionada de manera manual.
- **Ejecución por *commit*.** Esta ejecución automatizada a través de *webhook* se da cuando un desarrollador realiza un *commit* sobre una rama de código. Esta ejecución lanzará el conjunto sencillo de pruebas de los test y no permitirá la ejecución del despliegue.
- **Ejecución por *merge*.** Esta ejecución automatizada a través de *webhook* se da cuando un desarrollador realiza un *merge* de dos ramas de código. Esta ejecución lanzará el conjunto completo de pruebas de los test y no permitirá la ejecución del despliegue.
- **Ejecución por *tag* (despliegue).** Esta ejecución automatizada a través de *webhook* se da cuando un desarrollador realiza cualquier cambio de código que vaya ligado a un *tag* con la nomenclatura <entorno>_V<versión>. Esta ejecución lanzará el conjunto completo de pruebas de los test y ejecutará la etapa de despliegue, haciéndolo efectivo para el entorno y versión indicados.

Nota: *en el caso de la ejecución automatizada, se pueden producir situaciones en las que, a través de un mismo evento de desarrollo, se estén realizando combinaciones commit-tag o merge-tag en una única ejecución del pipeline. En el caso de darse esta situación, la operativa de ejecución que prevalece es la del tag.*

Flujo de desarrollo con el pipeline

La estrategia de *branching* propuesta para el desarrollo de la solución (***GitHubFlow***) consiste, a grandes rasgos, en la realización de una rama por cada cambio que se necesite introducir en el producto.

Nota: *En el apartado "GitHubFlow" se encuentra una descripción más detallada del funcionamiento de dicha estrategia de branching.*

Mediante el seguimiento de esta metodología, conforme se realicen los diferentes desarrollos de producto, el *pipeline* se irá ejecutando de manera **automatizada** a través de los *webhooks* configurados.

Nota: *aunque la ejecución manual del pipeline es posible, ésta debe tener un papel secundario y sólo debe ejecutarse en situaciones excepcionales producidas durante el proceso de desarrollo.*

Durante este proceso de desarrollo, Git y Jenkins se comunicarán sabiendo diferenciar entre un tipo de acción u otra de manera automática. En el caso de que se deseen realizar *commits* o *merges* que no impliquen un despliegue, no se requieren acciones adicionales por parte del desarrollador.

En cambio, si se quiere realizar un **despliegue**, el desarrollador deberá añadir en el repositorio de código un ***tag*** con la nomenclatura <entorno>_V<versión>.

Nota*: los diferentes entornos englobados dentro de <entorno> son todos los descritos en "Capítulo 7 – Problema: Entornos de desarrollo", salvo el entorno local, el cuál dadas sus características no requiere de despliegue.*

Para hacer referencia a los entornos sin alargar en exceso los diferentes tags, se van a emplear, de manera descriptiva, las tres primeras letras en mayúscula del entorno objetivo (DES, PRE, VAL o PRO).

Para poder desplegar una versión a un entorno, es obligatorio que dicha versión haya sido previamente desplegada a los entornos anteriores.

Nota: *las versiones englobadas dentro de <versión> estarán compuestas por 3 números separados por puntos (PE: 1.0.0). No debe ser posible la creación de un tag con una versión anterior a una ya creada.*

En el caso de que se esté realizando un despliegue desde una rama, la <versión> debe incluir tanto la última versión de la rama de la que nace, como un subíndice que permita tanto identificarla como despliegue de rama como versionarla dentro de los diferentes despliegues de dicha rama (PE: la primera versión desplegada desde una rama que nace de la versión 1.0.5, tendrá la versión 1.0.5.1-<ID rama>).

Cualquier forma de **despliegue** que no sea a través del *pipeline* debe quedar inhabilitada para aquellos componentes cuyo despliegue sea automatizable. En el caso de la solución estudiada, el único componente que no cumple con estos requisitos es el **orquestador OSB**. Siempre que se quiera realizar un despliegue que incluya este

componente, tras la ejecución del pipeline, el desarrollador debe desplegar de manera manual el orquestador.

Tras la ejecución de un pipeline, al desarrollador que haya sido responsable de realizar el *trigger* que ha terminado en la ejecución de un pipeline, le llegará una **notificación resumen** con el resultado. En el caso de que el resultado de ejecución del pipeline sea fallido, el desarrollador deberá solucionar en el menor tiempo posible el problema detectado.

Capítulo 6 – Problema: Realización de validaciones *frontend* y *backend*

6.1. Introducción

El objetivo de este capítulo es definir un plan de acción para la resolución de un problema común que se da en una gran cantidad de aplicaciones similares a la evaluada en este trabajo: como **plantear y gestionar la necesidad de validar** la información procesada por la aplicación (tanto en frontal, como en *backend*).

6.2. Descripción del problema

Contexto del problema

Con el auge del modelo cliente-servidor y las aplicaciones web, la separación entre los componentes frontales y *backend* de una solución se ha vuelto mucho más notable que en las aplicaciones de escritorio, en donde la diferencia es prácticamente conceptual (el frontal está relacionado con la parte de la aplicación que interactúa con el usuario y el *backend* con la que no).

En el caso de las aplicaciones web, esta diferenciación se realiza, además de a un nivel lógico, a un nivel arquitectónico. Las diferentes **aplicaciones frontales y *backend*** que componen una solución están desarrolladas en componentes software

completamente diferentes y forman estructuras funcionales independientes. Esta situación obliga a las aplicaciones web a enfrentarse a las siguientes adversidades:

- **Diferencias tecnológicas.** Las tecnologías empleadas para el desarrollo de la parte frontal y *backend* pueden ser completamente diferentes. En el caso de la solución estudiada en este trabajo, se diferencian Angular y TypeScript para el frontal frente a aplicaciones C# .NET en el caso del *backend*.
- **Diferentes equipos de ejecución.** Es muy común que los diferentes componentes frontales y *backend* que componen una solución sean ejecutados en equipos completamente diferentes. Habitualmente, las aplicaciones frontales se ejecutan en el equipo del usuario final, mientras que las *backend* se ejecutan en servidores alojados en otro lugar (por ejemplo, en un servicio cloud).
  Esta separación permite que el equipo del usuario final sólo conozca la parte del código y los binarios correspondientes a la parte frontal.
- **Acceso a *backend* sin necesidad de pasar por el frontal.** En el caso de las aplicaciones montadas sobre API REST generalmente es posible, mediante el establecimiento de un conjunto de conexiones con la API REST, emplear toda la funcionalidad de la aplicación sin necesidad de pasar por el componente frontal ofrecido.

Estas diferenciaciones permiten afirmar que, en el fondo, en las aplicaciones web montadas sobre APIs REST, **la aplicación** principal **reside en el *backend*** y que la parte frontal es una ayuda facilitadora que se le ofrece al usuario de las aplicaciones para simplificar y mejorar la usabilidad del sistema a través de una interfaz de usuario amigable.

Desarrollo del problema

Prácticamente todas las aplicaciones, independientemente de su ámbito, requieren de una serie de **validaciones, comprobaciones y acciones** que les permitan desempeñar correctamente las funciones para las que están diseñadas. Estas tareas desempeñan un papel crucial tanto a nivel de seguridad, como a nivel de negocio.

En el caso de las aplicaciones clásicas de escritorio, estas operaciones se pueden desempeñar en cualquier parte del flujo, puesto que necesariamente, todos los usuarios deben ajustarse al uso de la aplicación al completo. En cambio, en las aplicaciones web montadas sobre APIs REST, esta flexibilidad desaparece debido a que el usuario no necesariamente debe hacer uso de la aplicación frontal para emplear la funcionalidad de la aplicación (alojada en el *backend*).

Para ser capaces de establecer las validaciones en el punto indicado, es necesario definir una **frontera que separe el flujo obligatorio del opcional** de la aplicación. A

partir de esta frontera (en dirección al flujo obligatorio) va a ser en donde, necesariamente, todas las validaciones de obligado cumplimiento (especialmente aquellas relacionadas con la seguridad de la aplicación) deben ser realizadas (puesto que es la única forma de asegurar al 100% su cumplimiento).

Nota: *según el contexto del sistema objetivo, podría darse el caso de que no necesariamente todas las validaciones sean de obligado cumplimiento. Aquellas no lo sean, podrían llegar a ejecutarse al otro lado de la frontera.*

En el caso de las aplicaciones web montadas sobre APIs REST, esta **frontera** se encuentra precisamente en **el propio API REST**. Una vez el usuario realiza una llamada REST y esta accede al sistema *backend*, el usuario ya no dispone de más posibilidades de interacción con el flujo la aplicación salvo la espera a la respuesta de la petición dada por el sistema.

Duplicidad de las validaciones y problemas asociados

El **requerimiento técnico de realizar las validaciones** a partir de la frontera de seguridad establecida **en *backend*** puede provocar una ralentización del proceso de validación y por ende un empeoramiento en términos de usabilidad del sistema. Para detectar un fallo en los datos introducidos, el usuario deberá esperar a que el sistema *backend* procese la totalidad de los datos y le devuelva aquellas inconsistencias encontradas.

Para empeorar las cosas, con la finalidad de no aumentar la cantidad de datos a procesar por los servidores *backend*, las llamadas con la información a validar generalmente se realizan una vez se cumplen ciertos hitos (como, por ejemplo, haber rellenado toda la información posible).

Con la finalidad de que esta pérdida de usabilidad no se haga efectiva, es posible **introducir en paralelo las mismas validaciones** que se realizan en el *backend* en el frontal. De esta forma, el usuario obtiene feedback inmediato (no debe esperar al envío y procesado de datos por parte de los servidores) y siente el proceso de uso de la aplicación más ágil que en el caso original.

Nota: *no necesariamente es obligatorio incluir la totalidad de las validaciones en la parte frontal para mejorar los resultados. Habitualmente, son validaciones sencillas como comprobaciones de formato o tipos de datos las que se realizan tanto en el frontal, como en el backend.*

No obstante, no es oro todo lo que reluce y, la aplicación de esta duplicidad supone un reto en diferentes aspectos. En primer lugar, las variaciones tecnológicas existentes entre el frontal y el *backend* hacen que sea necesario buscar una forma óptima de desarrollar estas validaciones sin que se produzcan desfases entre ellas.

Adicionalmente, la realización de ciertas validaciones desde la capa frontal puede debilitar la seguridad del sistema por los siguientes motivos:

1. **Liberación del código fuente.** Las validaciones que se realizan en la parte frontal de la aplicación se realizan en el equipo del cliente, y, por lo tanto, éste tiene acceso al proceso de validación. Esto puede provocar ataques de caja blanca o ingeniería inversa.
2. **Impedimentos para el control y restricción de intentos de acceso.** Al realizarse ciertas comprobaciones desde frontal, se reduce la capacidad de control y resulta mucho más complejo restringir la cantidad de reintentos de acceso, conceptos clave en términos de seguridad informática.

Encaje del problema con la solución estudiada

Este problema afecta de manera directa al canal de entrada estudiado durante este trabajo. Para su correcto funcionamiento, es necesario que para todas las casillas se validen tanto los **tipos de datos**, como el **formato introducido**. Dada la criticidad y sencillez de estas validaciones, se va a valorar positivamente la posibilidad de que sean insertadas en el frontal.

Además, la solución también requiere de validación de **relaciones entre casillas** para cumplir con las necesidades del negocio. Por último, el sistema objetivo también debe realizar todas las comprobaciones pertinentes en lo referente al **control de acceso de datos**.

6.3. Posibles soluciones

6.3.1. Suprimir las validaciones en el frontal

Planteamiento de la solución

Una de las maneras más sencillas de lidiar con este problema es **eliminando las validaciones de la parte frontal**. Al trasladar la totalidad de las comprobaciones al *backend* del proyecto, se garantiza que la cantidad de esfuerzos requeridos tanto para realizar como para mantener y controlar el correcto funcionamiento de todo el sistema de reglas sea mínima. No obstante, mediante la aplicación de esta solución, en lo que a **experiencia del usuario** se refiere, se produce un decremento significativo derivado de la espera en el procesado de datos.

Enfoques para llevarla a cabo

A grandes rasgos, existen dos **enfoques de operatividad** que permiten lidiar con el envío de los datos desde frontal para su validación. Según el tipo de validaciones y el tiempo requerido para su procesado se deberá elegir uno u otro. Estos enfoques son los siguientes:

- **Intercambio de mensajes cliente – servidor constante**. Este enfoque se basa en el intercambio constante de mensajes de validación independientemente de las acciones del usuario. Este intercambio se puede realizar, por ejemplo, en intervalos constantes de tiempo.
  El objetivo de esta operativa es tratar de reducir el tiempo de espera que tiene el usuario final desde que introduce los datos en el sistema hasta que estos son validados.
- **Intercambio de mensajes cliente – servidor bajo demanda.** Este enfoque se basa en la realización del intercambio de mensajes cuando se produzcan ciertas interacciones con la aplicación web. Estas acciones suelen ir ligadas a la lógica de negocio y pueden ir desde completar ciertas secciones, hasta a realizar envíos de manera manual cuando el usuario lo desee.
  Esta operativa obtiene peores resultados en cuanto a usabilidad del sistema, pero a cambio, reduce en gran medida la cantidad de tráfico generado por la aplicación.

Existen dos enfoques principales que permiten gestionar la información enviada durante las llamadas de verificación:

- **Envío de todos los datos que han sido introducidos por el usuario.** Esta operativa es más sencilla de implementar, pero por el contrario produce un sobrecosto derivado del envío y validación de datos que ya han sido enviados previamente.
- **Envío de tan solo aquellos datos que hayan variado desde el último envío.** Esta operativa es más compleja de implementar, pero a cambio disminuye el coste derivado del envío y validación de datos (aquellos datos que no han sido modificados desde el último envío).

Ventajas y desventajas de la solución

Las principales **ventajas** de suprimir las validaciones del frontal son:

- **Aumento de la dificultad de posibles ataques**. Al disponer de todos los controles de validación directamente integrados en el *backend* del proyecto, el código y la lógica de dichos controles no está expuesta al público. Esto finalmente se traduce en que resulta más complejo aplicar técnicas de caja

blanca que atacan de manera directa a las comprobaciones realizadas a los datos.

- **Mejora de la mantenibilidad**. Disponer de una única capa de validación hace que sea más sencillo tanto crear como mantener las diferentes validaciones del sistema. Además, al estar integradas las validaciones en un único componente, se reducen la cantidad de dependencias técnicas y la cantidad de tests a mantener (al ser un sólo componente el que debe de ser probado).

Asimismo, las principales **desventajas** de aplicar esta solución son:

- **Empeoramiento de la usabilidad de la aplicación**. Al no disponer de validaciones en la parte frontal, las validaciones de datos tardan mucho más en llegar al usuario final.
- **Aumento del tráfico de datos**. El depender por completo del *backend* para la realización de validaciones aumenta significativamente la cantidad de tráfico generado por cada cliente. Esto se debe a que, para su funcionamiento tienen que viajar tanto los datos, como los errores detectados y las correcciones de los mismos.
- **Reducción de las capas de validación**. Aunque realizar las validaciones en el frontal no es obligatorio, sí que aporta ventajas a nivel funcional en términos de seguridad (modelo del queso suizo de causalidad de accidentes).
  Además, las validaciones van a ser realizadas una única vez y, por lo tanto, aumenta tanto su criticidad como propensión a fallos más severos. Esta problemática gana especial relevancia en aquellas comprobaciones que tengan que ver con temas de seguridad, como pueden ser inyecciones SQL.

***Nota**: existen herramientas de desarrollo específicas que permiten que las aplicaciones resultantes trasladen casi la totalidad lógica al backend de la aplicación (incluyendo comportamientos exclusivos de la parte frontal), convirtiendo así a la parte frontal en un mero cascarón visual comunicado con el backend. Una de estas herramientas es Blazor* [62].

En este trabajo no se han considerado expresamente este tipo de soluciones debido a que, por definición, el frontal la aplicación que estamos tratando necesariamente estará desarrollado en Angular.

Adicionalmente y debido al alto tráfico de la aplicación objetivo en momentos concretos, el gran aumento de carga que supone para los servidores estas herramientas puede ser fatal.

6.3.2. Duplicación de las validaciones

Planteamiento de la solución

Otra forma de afrontar este problema es **duplicar las validaciones** del *backend* y para disponer de validaciones espejo en el entorno frontal. Mediante esta solución, se busca **mejorar la usabilidad** de la aplicación (disponiendo el cliente de los datos validados de manera inmediata) y reducir la carga de trabajo de los servidores *backend*.

Enfoques para llevarla a cabo

Afrontar esta solución supone un **reto técnico** debido a las **diferencias tecnológicas** existentes entre las aplicaciones. Para llevarla a cabo, existen diferentes tipos de **enfoques** en función de cómo afrontar este desarrollo duplicado. Durante este trabajo se van a considerar y evaluar los siguientes enfoques (cada uno de ellos dispone de un su apartado posterior):

- Duplicación basada en desarrollo.
- Duplicación basada en transpilación de código.
- Duplicación basada en configuración y generación automatizada de código.

Ventajas y desventajas de la solución

Las principales **ventajas** de realizar las validaciones en ambos entornos son:

- **Mejora de la usabilidad**. Al realizar las validaciones desde el frontal es posible comprobar los datos de mantra instantánea desde que son introducidos en el sistema, pudiendo así informar de manera inmediata al usuario de la validez de los mismos.
- **Disminución del tráfico de datos**. Al no requerir del envío de los datos para su validación, sólo es necesario que los datos viajen del frontal al *backend* en aquellas ocasiones en las que los datos se quieran procesar (guardar un borrador, realizar un envío de datos, etc.).
- **Aumento de las capas de validación**. Al validarse los datos en diferentes capas de validación, se puede llegar a producir en una mejora de la seguridad derivada de fallos en una tecnología o librería concreta (modelo del queso suizo de causalidad de accidentes).

Por el contrario, las principales **desventajas** de este tipo de operativa son:

- **Disminución del desarrollo y mantenibilidad del producto**. Al tener las

validaciones tanto en el frontal, hay que duplicar todas las tareas de trabajo durante el ciclo de desarrollo del producto (introducción de mejoras, diseño, generación y lanzamiento de test…). Además, es posible que se produzcan desincronizaciones en las validaciones fruto de los diferentes desarrollos realizados en cada una de las tecnologías.

- **Acceso a ataques específicamente diseñados**. Al disponer desde el cliente del código fuente, así como las diferentes validaciones realizadas, es más sencillo para un hipotético atacante diseñar una batería de pruebas maliciosas que estén enfocadas en atacar los puntos débiles de dichas validaciones.

6.3.2.1. Duplicación basada en desarrollo

Planteamiento del enfoque

En este enfoque serán los propios desarrolladores quienes, a través de un **desarrollo** del producto **realizado "a mano"** programen tanto las reglas frontales como las *backend*. Esto implica invertir **mayor cantidad de tiempo** por parte de los desarrolladores para llevar a cabo un conjunto de tareas que podrían realizarse de manera más ágil si sólo se desarrollasen en el *backend*.

Este tipo de enfoque puede introducir algunas **mejoras en términos de seguridad** (además de las generales de la solución). Si las reglas frontales son desarrolladas por un conjunto de personas diferente al de las reglas de *backend*, podrían llegar a producirse variaciones de la validación fruto de un enfoque diferente de afrontar la comprobación que aumentan la robustez del producto.

Ventajas y desventajas del enfoque

Las principales **ventajas** de realizar la duplicidad a mano son:

- **Control sobre el producto**. En todo momento, el equipo de desarrollo obtiene un código que hace exactamente lo que esperan de él. Adicionalmente, este código, presumiblemente, siempre va a cumplir todas las aquellas reglas de calidad de código impuestas por el equipo.
- **Independencia**. Esta solución permite que las reglas que se realizan en el frontal y en el *backend* tengan sus propios matices y diferencias.
  Según el proyecto, existen cierto tipo de validaciones y acciones facilitadoras que sólo resultan útiles durante la formación de los datos y que no requieren de su ejecución en *backend*. El canal de entrada estudiado en este trabajo dispone de este tipo de validaciones.
  Esta flexibilidad también puede resultar útil a la hora de realizar pruebas de concepto.

- **Sencillez, reducción de dependencias y menor abstracción**. El código generado de manera manual suele derivar en una sencillez, legibilidad e independencia que el código desarrollado mediante otras técnicas.

Asimismo, sus principales **desventajas** son:

- **Empeoramiento del ciclo de desarrollo**. El ciclo de desarrollo se complica y se alarga. Para realizar tareas que podrían ser resueltas mediante una única codificación, va a ser necesario realizar dos.
  Esta duplicidad se esparce en todos ámbitos del flujo de desarrollo: mantenimiento, testing, nuevos desarrollos…
- **Mayor riesgo de errores**. Al tener que realizar una misma tarea varias veces, es más posible que se produzcan errores durante el desarrollo de alguna de ellas. Debido a la dualidad, estos errores pueden ser muy longevos puesto que puede llevar mucho tiempo detectarlos.
- **Aumento de la complejidad para mantener la consistencia**. Durante el desarrollo del producto pueden darse situaciones confusas durante la detección de errores. Cada vez que se detecte un error, de manera previa a la corrección es necesario evaluar si dicho error se encuentra en uno de los componentes o en ambos.

6.3.2.2. Duplicación basada en transpilación de código

Planteamiento del enfoque

Este enfoque se basa en un conjunto de herramientas llamadas **transpiladores**. Estas herramientas traducen de mantra automática el código fuente de un lenguaje de programación a otro.

La idea es que, utilizando este tipo de herramientas los programadores **únicamente desarrollen las diferentes validaciones en una de las tecnologías y componentes.** Posteriormente, mediante el empleo de los transpiladores, éstas validaciones se traducen al lenguaje de programación del otro componente.

Nota: *El proceso de transpilación se puede integrar de forma sencilla en el compilador e incluso en el pipeline de automatización.*

Transpiladores

Existen diversas herramientas de transpiración que se basan en los lenguajes sobre los que se basa nuestro sistema. La transpilación se puede hacer en cualquiera de las dos direcciones (desde lenguajes *backend* hacia frontal o viceversa). En el caso de la

solución existen las siguientes opciones:

- **Transpilación de JavaScript y TypeScript a C# .NET.** Enfocar esta traducción supone un reto debido a que en muchas ocasiones hay que pasar de un lenguaje no tipado (JavaScript) a uno fuertemente tipado y orientado a objetos (C#). Esta situación hace que actualmente no existan herramientas que permitan esta funcionalidad.
  Si pese a ello se optase por esta dirección, una posible solución puede ser el empleo de un lenguaje puente para. Como es lógico, la sobrecarga de traducciones podría derivar en un código de menor calidad (sobre todo en términos de legibilidad).
- **Transpilación de C# .NET a JavaScript y TypeScript.** El enfoque de esta traducción resulta más sencillo debido a que hay que pasar de un lenguaje tipado y orientado a objetos (C#), a otros con menor tipado y mayor flexibilidad programática (JavaScript y TypeScript).
  Existen diferentes herramientas que permiten realizar la labor de transpilación en esta dirección en función del tipo de proyecto .NET que se esté utilizando como base. Entre ellas destacan:
  - Bridge.NET [63] para proyectos NET Framework.
  - H5 [64] para proyectos .NET Core.

Resulta importante considerar que, aunque existen herramientas operativas y con mantenimiento activo, también existen muchos proyectos similares de relevancia (muchas veces recomendados por delante de los actualmente existentes) que han quebrado, han quedado desactualizadas o han quedado a medias en la realización de sus objetivos. Además, en el caso de las herramientas existentes y funcionales, la información y documentación disponible es escasa y tienen unos requisitos técnicos de importancia considerable.

La selección de una buena herramienta de transpilación es clave para el éxito de este enfoque puesto que se está estableciendo una relación de obligado cumplimiento entre la solución y la herramienta de transpilación. Esta relación puede provocar **deuda técnica y la ralentización** de la solución para adaptarse de futuras versiones de los lenguajes.

Una posible solución a estos problemas es desarrollar un **transpilador propio** que se ajuste en gran medida a las necesidades concretas del sistema en un momento dado. No obstante, un desarrollo de dicha índole no es trivial y requiere de una gran cantidad de recursos.

Ventajas y desventajas del enfoque

Las principales **ventajas** de la traspiración de código son:

- **Simplificación del proceso de desarrollo**. Se reducen la cantidad de tareas a realizar de manera activa por los programadores durante el desarrollo de la solución. Tan solo es necesaria la codificación de las validaciones para uno de los componentes. No obstante, sí que es necesario generar test para las dos tecnologías, aunque el código de una de ellas sea autogenerado.
- **Las validaciones de frontal y *backend* son las mismas**. Al ser la validación de un componente, una traducción de la del otro, el funcionamiento interno de tanto las validaciones realizadas en el frontal, como las validaciones realizadas en el *backend* será el mismo (incluso en el caso de el que exista un fallo en la validación). Esta no-dualidad impide que la solución se beneficie del empleo del modelo del queso suizo de causalidad de accidentes pese a realizar las validaciones en dos capas.

En cambio, sus principales **desventajas** son:

- **Pérdida de legibilidad**. El código generado por un autómata, aunque esté basado en el código realizado por un desarrollador, suele sufrir grandes pérdidas de legibilidad derivadas de la lógica funcionamiento interno del autómata.
  Esta pérdida se hace especialmente patente si se realiza una comparativa entre el código transpirado y el generado de manera manual.
- **Pérdida de control**. Los desarrolladores pierden por completo el control sobre el código que se genera. Este código podría no cumplir con los estándares de calidad propuestos por el equipo.
  Aunque sea posible que el equipo modifique los ficheros generados, este tipo de operativa carece de sentido puesto que, las mismas modificaciones deberían realizarse cada vez que se establezca cualquier cambio sobre el código original (puesto que se vuelven a traducir los ficheros).
- **Deuda técnica**. Al basar el proceso de desarrollo en una herramienta de este tipo, se está adquiriendo una deuda técnica del proceso de desarrollo que durará a lo largo de todo el ciclo de vida del proyecto.
  Con el desarrollo de nuevas funcionalidades en los lenguajes, muchas herramientas de este tipo deben adaptarse para poder utilizarlas. Esa espera trunca de manera directa la posibilidad de desarrollar el producto en las últimas versiones que nos ofrece la tecnología escogida de manera inmediata. Para reducir estos tiempos de espera, resulta crucial escoger una herramienta que limite lo mínimo posible el desarrollo.
- **Posibilidad de errores en los compiladores**. Los compiladores, al igual que

cualquier otro tipo de aplicación, pueden contener fallos funcionales.
Estos fallos podrían provocar que se generase código defectuoso el cual podría llegar a propagarse por toda la solución en caso de no detectarse.
Además, en caso de detectarse, podría darse el caso de que el fallo detectado no disponga de una fácil solución y que se requiera de un cambio la forma de programar la solución final.

6.3.2.3. Duplicación basada en configuración y generación automatizada de código

Planteamiento de la solución

Este enfoque representa un punto medio entre los dos vistos hasta ahora. Su objetivo es tomar cada uno de los puntos fuertes de dichos enfoques y tratar de eliminar sus debilidades.

El enfoque se basa en que los desarrolladores **definan una configuración de validación** (en vez de programar la propia validación) y que, a través de una **herramienta** de diseño interno (en lo posterior nos referiremos a ella como motor), **se genere el código** fuente para cada una de las tecnologías en base a la configuración indicada. Para realizar la definición de las reglas, se pueden aplicar diferentes estrategias y tecnologías.

Definición de reglas mediante un fichero genérico

Una opción es la definición de las reglas mediante un **fichero común** en un formato de texto, como podría ser **JSON o XML**. En dicho fichero quedarían reflejadas las diferentes reglas que el motor debe generar.

La principal ventaja de esta opción es la **flexibilidad** que ofrecen estos formatos para definir las reglas. Asimismo, su principal desventaja reside en la dificultad de implementar un motor tan genérico y la facilidad a través de la cual se podrían cometer errores humanos.

Definición de reglas mediante DMN

Esta opción es una extensión de la selección de un fichero genérico de configuración. La idea es añadirle a dicho fichero una capa una superior de modelado a través del lenguaje **Decision Model and Notation (DMN)** [65]. DMN es un estándar diseñado por Object Management Group (OMG) [66] que está específicamente diseñado para alojar un modelo de decisiones y reglas.

Este estándar de modelado adicionalmente está planteado de tal forma que permite aplicar sobre él el estándar **Friendly Enough Expresion Leanguaje (FEEL)**. El empleo de FEEL facilita la comprensión de las validaciones configuradas y realizadas el sistema para aquellos miembros del proyecto que no tengan un carácter técnico.

Dish

decision

Hide details

U	Input +		Output +	
	Season	How many guests	Dish	
	season	guestCount	desiredDish	
	string	integer	string	Annotation
1	"Fall"	<= 8	"Spareribs"	-
2	"Winter"	<= 8	"Roastbeef"	-
3	"Spring"	<= 4	"Dry Aged Gourmet Steak"	-
4	"Spring"	[5..8]	"Steak"	Save money
5	"Fall", "Winter", "Spring"	> 8	"Stew"	Less effort
6	"Summer"	-	"Light Salad and a nice Steak"	Hey, why not!?
+	-	-	-	-

Ilustración 28: Tabla de decisión definida con el estándar DMN. En esta tabla se puede apreciar el empleo del estándar FEEL a través de DMN. Ejemplo tomado de [67].

Los modelos realizados con DMN pueden **ser representados de manera gráfica** a través de **Business Process Model and Notation (BPMN)** [68], otro estándar diseñado por OMG que permite la visualización manera gráfica de procesos.

Existen muchas **herramientas**, entre ellas *Visual Studio* Code mediante un *plugin* [69], que **permiten modelar** a través de ellas un **diagrama DMN**. Estas herramientas simplifican el proceso de definición de reglas y disminuyen la probabilidad de los errores humanos.

A nivel técnico, los diagramas de DMN se almacenan **en ficheros XML** lo que le permite un fácil procesado a través del motor y la posibilidad de estudiar sus cambios mediante el control de versiones.

La principal ventaja de esta definición de reglas es el uso de interfaces gráficas que reducen el error humano y aumentan la legibilidad. Su principal desventaja reside en la pérdida de flexibilidad que se produce durante la definición de las reglas.

Definición de reglas en BBDD

Otra opción es establecer la **definición de las reglas directamente sobre la base de datos** del sistema. Para llevar a cabo esta definición, existen patrones de diseño como *specification pattern*, que permiten establecer una base de configuración sencilla para la definición de reglas.

Las reglas fruto del empleo de ***specification pattern*** están diseñadas para ser procesadas por un motor, devolviendo verdadero o falso en función del resultado

obtenido. El empleo de este patrón simplifica el proceso de desarrollo del motor puesto que establece unos conceptos que las definiciones deben seguir.

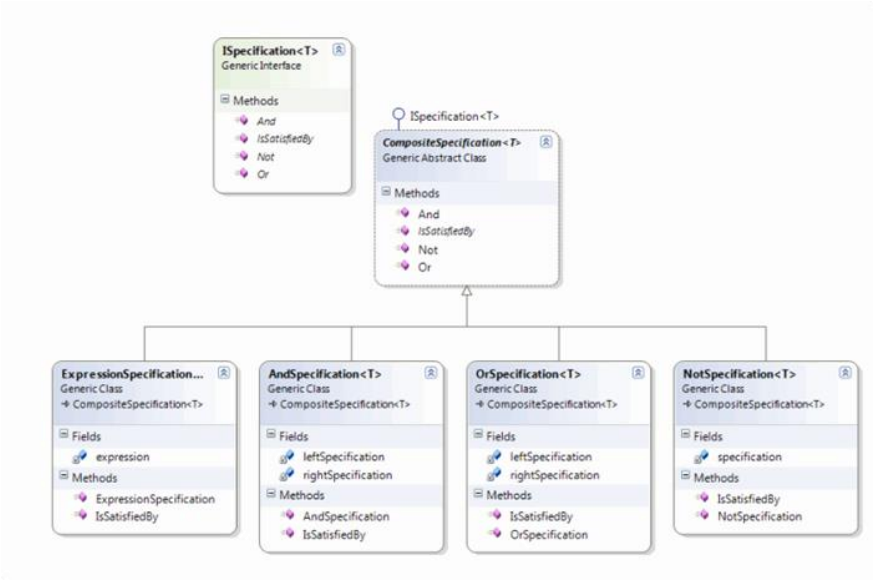


Ilustración 29: Estructura de tablas empleada por el patrón de diseño Specification Pattern. La imagen ha sido tomada de [70].

La principal ventaja de esta definición es, que al estar montada sobre una BBDD relacional, la posibilidad de cometer errores humanos durante a la definición de reglas disminuye. Como desventaja destaca el hecho de que se pierde gran flexibilidad de definición, pudiendo llegar a limitar el tipo de reglas aplicables según la situación.

Ventajas y desventajas del enfoque

Las principales **ventajas** del empleo de la duplicación basada por configuración son:

- **Simplificación del proceso de desarrollo**. Durante todo el proceso de desarrollo de la aplicación, se elimina la necesidad de programar ningún tipo de validación en beneficio del desarrollo de una configuración cuya realización es más sencilla.
  Cabe destacar que, aunque no se programen las validaciones de manera expresa, sí que resulta interesante la realización de test que permitan probar el correcto funcionamiento de las validaciones (tanto frontales como *backend*).
- **Flexibilidad del reglaje**. El disponer de uno o varios motores que se nutren de unas reglas definidas para generar el código, permite la posibilidad utilizar intencionadamente diferentes lógicas de aplicación para una misma regla de validación sin necesidad de desarrollarlas activamente. Esto permite aplicar el modelo del queso suizo de causalidad de accidentes.
- **Disminución de la posibilidad de errores**. Una vez los motores están desarrollados, la posibilidad de cometer un error durante la codificación de

validaciones disminuye mucho debido al gran grado de automatización del proceso de desarrollo y a la simplicidad de la definición de las reglas.
Esta ventaja se intensifica si se utilizan, para la definición de las reglas, las opciones de DMN o *specification pattern*.

Asimismo, las posibles **desventajas** del empleo de esta solución son:

- **Aumento de la cantidad de componentes**. El equipo de desarrollo debe planificar, diseñar, mantener y emplear uno (o varios) componentes nuevos encargados de generar el código de validación de la aplicación (motores).
  Estos motores deben estar incluidos en el proceso de desarrollo (empleo de todas las técnicas y herramientas vistas en los capítulos anteriores) e integrados debidamente en el pipeline.
- **Complejidad del motor**. El desarrollo de los motores de reglas que se nutren de una configuración base no es sencillo. Requiere de una planificación y una cantidad de recursos relativamente grandes.
- **Deuda técnica del motor de reglas**. El motor de reglas puede suponer una carga en términos de deuda técnica según las tecnologías que emplee.
  Además, este componente debe estar siempre actualizado (con toda la carga adicional de desarrollo extra que ello conlleva), lo cual puede verse lastrado por posibles dependencias externas sobre las que no se tiene control.

6.4. Solución escogida para la solución

Considerando las diferentes necesidades de validación de la solución estudiada y su disparidad en lo que a criticidad para el sistema se refiere, se van a proponer diferentes soluciones para cada uno de los tipos de validación.

Para las **validaciones relacionadas con el control de sesiones, la protección de datos y la seguridad general** la mejor opción es realizar las validaciones exclusivamente en la parte *backend* del sistema. Estas validaciones tienen un valor muy importante y resulta esencial mantener su integridad. Mediante la realización de validaciones exclusivamente en el *backend*, éstas operaciones tan cruciales se protegen de posibles ataques de caja blanca realizados con la finalidad de robar información o atentar contra la protección de datos.

De manera excepcional, **las validaciones de seguridad relacionadas con la con inyecciones de código o la introducción maliciosa de datos** van a ser evaluadas tanto en el frontal, como en el *backend*. La idea es tratar de detectar de manera prematura

los datos maliciosos, reduciendo así la cantidad de carga de este tipo de datos en *backend* y reduciendo así el riesgo de caer infectado. Para dificultar la realización de posibles ataques, la duplicidad de estas validaciones debe estar desarrollada a mano y **empleando diferentes enfoques de validación**. El enfoque de validación aplicado en el *frontend* también deberá ser utilizado en el *backend* (de manera no exclusiva).

Por último, para las **validaciones relacionadas con el tipado, formato y validez de los *datos***, la mejor opción es realizar una validación duplicada tanto en *frontend* como en *backend*. Este tipo de validaciones son de las que el usuario más se beneficia durante el uso de la aplicación y, por lo tanto, las que más permiten mejorar la usabilidad de la solución mediante su integración en el frontal.

Para su puesta en escena se propone emplear **la duplicidad basada en generación de código a partir de un fichero de configuración**. Más concretamente, su implementación mediante el estándar DMN.

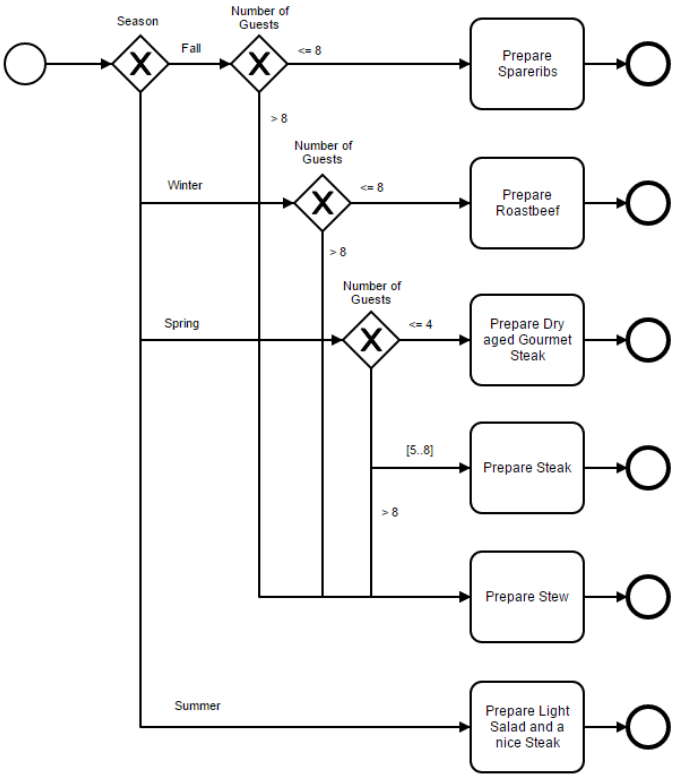


Ilustración 30: Ejemplo de diagrama DMN representado a través del estándar BPMN. El ejemplo está tomado de [67].

Las reglas que se pueden generar con ese sistema de definición encajan a la perfección con los requisitos del sistema estudiado. Además, su forma de diseño FEEL permite el envío y/o acceso a las reglas configuradas a los *product owners* (perfiles de menor carácter técnico) de tal forma que pueden comprobar su correcta definición sin la necesidad de tener conocimientos técnicos o de preguntar al equipo de desarrollo.

Capítulo 7 – Problema: Entornos de desarrollo

7.1. Introducción

En este capítulo se van a considerar las diferentes necesidades derivadas del proceso de producción de software con la finalidad de **definir una infraestructura de entornos** que permita a un equipo de desarrollo desemplear sus actividades tanto de desarrollo como despliegue de una manera ágil, segura, lógica y ordenada.

7.2. Requisitos de los equipos de desarrollo

Para entender los requisitos existentes durante el desarrollo de un producto software, primero es necesario comprender, a grandes rasgos, cuál es el **proceso de creación de código** que se lleva a cabo en un desarrollo en equipo. Este proceso por lo general sigue los siguientes pasos:

1. Un desarrollador realiza y prueba unos cambios del producto en su equipo.
2. Esos cambios puestos en común con los realizados por el resto de desarrolladores.
3. Tras la aprobación los cambios, estos son entregados al cliente.

Dicho proceso hace patente la necesidad de la existencia de **más de un entorno**. Cada entorno va a contener una versión estanca diferente del código en función del objetivo que persiga (entorno de desarrollo base, entorno del cliente…). La existencia de varios entornos fuerza a tener un proceso en el que el código se mueve de un entorno a otro (despliegue).

Los **despliegues** son operaciones delicadas y deben tratarse con sumo cuidado. Durante éstos, es posible provocar errores funcionales en la aplicación (pese a encontrarse el código debidamente desarrollado) y por lo tanto también deben ser probados.

Nota: *con la finalidad de garantizar que el producto se despliega correctamente, es muy habitual que el despliegue del producto se haga a través de herramientas de automatización. Para más información al respecto, ver "Capítulo 5 – Automatización del proceso de desarrollo (CI / CD)".*

Con estos conceptos en mente, es posible identificar las siguientes **necesidades** que deben ser satisfechas por la arquitectura de entornos:

- El producto debe estar siempre disponible para el usuario final.
- El producto al que tiene acceso el usuario debe estar debidamente probado.
- El proceso de despliegue del código también debe ser probado.
- Debe existir un punto de encuentro que permita juntar el trabajo desempleado por los trabajadores.
- Debe existir un proceso que permita a los programadores desarrollar en sus equipos sin afectar al resto.

A raíz de este problema, desde la industria software se han diseñado y puesto en práctica diferentes tipos de entornos. Cada uno de estos entornos se encuentra estandarizado y a través de ellos es posible solventar las necesidades anteriormente descritas.

7.3. Tipos de entornos

7.3.1. Producción (PRO)

Producción es el último entorno dentro del proceso de software. El objetivo de este entorno es que el código alojado en su interior sea **utilizado por los usuarios finales** como el producto en cuestión.

Éste debe de ser el único entorno al que dichos usuarios tengan acceso. En este sentido, es habitual restringir y monitorizar los permisos de acceso incluso de los desarrolladores para evitar fugas de información o actualizaciones indebidas de los datos.

Para que el software haya llegado hasta este entorno, previamente se han debido **realizar todas las pruebas** exigidas por el proceso de desarrollo al código que se va a implantar. Bajo ningún concepto debería realizarse un despliegue en este entorno de una pieza de código que no haya sido debidamente probada.

Como es lógico, debe ser el **entorno más estable y probado** de todos. Es por ello, por lo que es el entorno que va a tener siempre la versión más antigua y estable de todos. En el caso de que se produzca cualquier tipo de incidencia, fallo o caída en él, solventarla se convierte en el principal objetivo del equipo de desarrollo a corto plazo.

7.3.2. Validación (VAL)

El entorno de **Validación** es el entorno **inmediatamente anterior a Producción**. Se trata de un entorno puente cuyo objetivo es brindar un **entorno de pruebas** que sea lo más parecido posible a producción (tanto a nivel arquitectónico, como hardware y software).

En este entorno se deben realizar todas las pruebas funcionales necesarias al producto para darlo por válido y enviarlo a Producción (se trata de la última barrera antes del despliegue a los usuarios finales). Generalmente, estas pruebas no se limitan a las pruebas programadas por el equipo de desarrollo, sino que estas suelen ser complementadas con pruebas realizadas a mano por negocio.

Este es el entorno idóneo para la **realización de pruebas de carga y estrés** del producto. Estas pruebas deben realizarse sólo en momentos puntuales puesto que tienen una gran cantidad de requisitos y convierten el entorno en inestable.

En términos de código desplegado, Validación siempre debe tener **la misma versión de código que se encuentre desplegada en Producción** o en su defecto, una inmediatamente superior. En el caso de ser una inmediatamente superior, dicha versión debe de ser desplegada en Producción en cuanto se finalicen sus pruebas y se apruebe su despliegue, quedando así estabilizadas las versiones.

***Nota**: En arquitecturas que no requieren de ningún tipo despliegue manual, este entorno se fusiona junto con PRE-producción.*

7.3.3. PRE-producción (PRE)

El entorno de **PRE-producción** es el entorno que se encuentra **a medio camino entre el de Validación y el de Desarrollo**. Se trata de un entorno controlado cuyo objetivo principal es **comprobar si el despliegue** que se está gestando desde desarrollo hacia producción es correcto a nivel técnico y de dependencias (a nivel funcional se evalúa en el entorno de validación).

Una vez los nuevos cambios se encuentran en este entorno a través de un despliegue, el producto resultante es testeado. Si todas las pruebas son superadas con éxito, dicho despliegue quedará validado podrá realizarse en Validación. En caso contrario, el despliegue será descartado y PRE-producción deberá volver al estado previo al del despliegue fallido.

Al ser un entorno en el que se prueba el correcto funcionamiento de diferentes despliegues, se trata de un entorno con **cierta inestabilidad**. En términos de versionado de código, aunque no se encuentra tan ligado a Producción como si se encuentra el entorno de Validación, el entorno de PRE-producción debe de tener una **versión posterior, pero cercana** tanto a la que se encuentra en **Producción** como en Validación.

***Nota**: En arquitecturas que no requieren de ningún tipo despliegue manual, este entorno se fusiona junto con Validación.*

7.3.4. Desarrollo (DES)

El entorno de **Desarrollo**, es el primer entorno del proceso de creación de aplicaciones software en el que se ponen en común los últimos cambios generados por los desarrolladores. Se encuentra a **medio camino entre Local y PRE-producción**. Su objetivo **es fusionar todos los cambios más recientes** introducidos por el equipo de desarrollo en un entorno común y validar que no se producen interferencias entre ellos.

Durante el desarrollo, una vez se finalizan los cambios en Local, éstos son inmediatamente llevados a Desarrollo. De esta forma, Desarrollo dispone, presumiblemente, de las **últimas funcionalidades desarrolladas estables**.

Debido a la rápida inserción de los nuevos cambios en este entorno, Desarrollo es un entorno con **cierta inestabilidad**. En términos generales, se podría decir que este

entorno contiene la base del estado actual del proyecto.

7.3.5. Local (LOC)

El entorno **Local**, es el primer entorno del proceso de desarrollo. Tras él, **el siguiente entorno es Desarrollo**. El objetivo principal de este entorno es permitir el **desarrollo de las funcionalidades de manera estanca**. De esta forma, mientras se produce el código, no se bloquean o rompen los cambios que ya se encuentren desplegados en los entornos comunes. Es el único de todos los entornos que se ejecuta por completo en la máquina en la que se desarrolla el resto del producto.

De entre todos los entornos, Local es el entorno que **incluye las funcionalidades más nuevas** (puesto que incluye las que están en desarrollo activo). La inclusión de estas funcionalidades lo convierte en un entorno **altamente inestable**.

Este entorno tiene una **peculiaridad a nivel de operatividad**. Para todos los entornos mencionados hasta el momento, los despliegues se realizan en una única dirección (Desarrollo -> PRE-producción -> Validación -> Producción).

Local incluye tan sólo el incremental de los cambios generados por un desarrollador en su máquina. Esto significa que, por sí solo, Local nunca va a tener los cambios realizados por el resto de miembros del equipo. Sin embargo, para llevar a cabo las operaciones de desarrollo de código, en ocasiones es necesario disponer de dichos cambios.

Para ello, es habitual traerlos de Desarrollo (en donde ya se encuentran probados y estables). De esta forma **el intercambio de versiones entre local y desarrollo es bidireccional** (Local <-> Desarrollo).

7.4. Arquitectura de entornos propuesta para la solución estudiada

La propuesta arquitectónica a nivel de entornos que va a emplear la solución estudiada va a incluir **todos los tipos de entornos de desarrollo** definidos en el apartado anterior. Gracias a ellos, va a ser posible solventar todas las necesidades de desarrollo definidas.

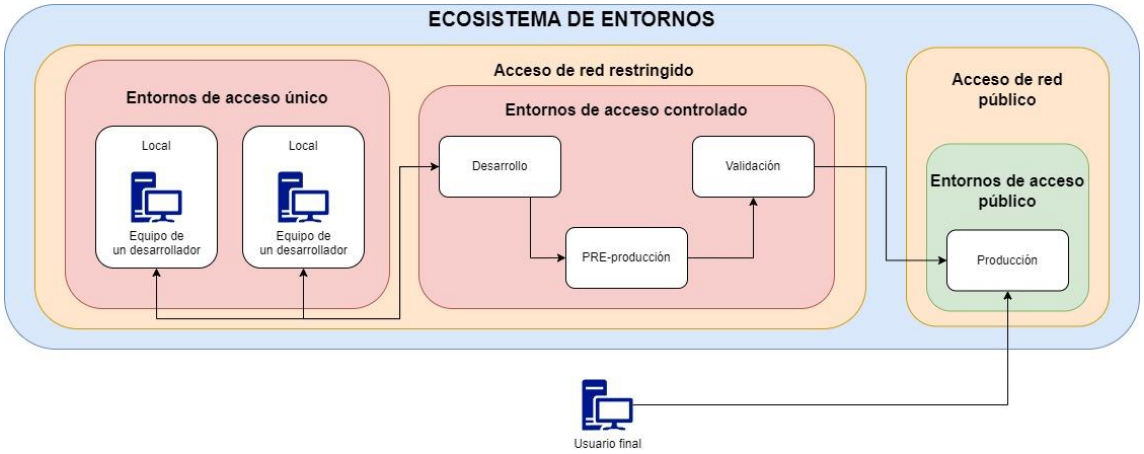


Ilustración 31: Arquitectura a nivel de entorno de la solución propuesta.

Pese a que para la mayoría de los componentes (salvo el OSB), el paso de un entorno al inmediatamente superior se realiza de manera automatizada y siempre a través del pipeline definido, **no es posible eliminar el entorno de Validación**. La **dependencia técnica del *Oracle Service Bus*** (la cual no está pensada para desplegarse de manera automática) obliga a realizar parte del despliegue del producto de manera manual, dotando así de sentido a la existencia de dicho entorno.

Para traer los cambios que se encuentren alojados en **Desarrollo a Local**, no se va a emplear el *pipeline* definido. Aunque a nivel técnico es posible su implementación en dicho mecanismo, ésta carecería de sentido puesto que en el *pipeline* se realizan muchas acciones (como *testing* o *code quality*) que para este tipo de despliegues no aportarían valor.

La manera más sencilla de traer estos cambios consiste en definir y construir un ***script***, el cual sea capaz de conectarse con las zonas en las que se encuentran los binarios del entorno de desarrollo y traerlos al entorno local. De esta forma, el entorno local queda actualizado con el código alojado en desarrollo.

Considerando los requisitos de la aplicación y para, no añadir dificultad técnica al proyecto, este *script* puede estar montado sobre una aplicación de consola de .NET programada desde el propio *Visual Studio*.

Capítulo 8 – Resultados, conclusiones y líneas futuras

8.1. Resultados y conclusiones

Tras la realización del trabajo, puedo concluir que, a día de hoy existen muchos **procesos, metodologías y herramientas** diseñadas con la finalidad de **mejorar los procesos de desarrollo** (tanto a nivel de creación de código, como de empaquetado, despliegue y entrega del producto).

Como se ha podido apreciar a lo largo del trabajo, el mundo de las **aplicaciones web no es una excepción**. Durante el desarrollo del mismo, ha sido posible facilitar opciones resolutivas para los diferentes problemas que han ido apareciendo. Como resultado**, la solución propuesta cumple con los todos objetivos del trabajo** (aplicados para un caso real, pero enfocados desde un punto de vista genérico). Además, ha sido posible la **resolución de algunos problemas clave** del desarrollo web como son la relación de validaciones y la selección de entornos de desarrollo.

A través de la **solución propuesta**, se establece un flujo de trabajo que mediante el cual, gracias a una serie de herramientas y a la automatización de procesos es posible mejorar y simplificar los procesos de desarrollo. Además, mediante el empleo de dicha solución, es posible evaluar la calidad del código desarrollado, mantener un proceso de pruebas constante, mantener uno estándares de calidad y definir un flujo de trabajo todo ello enmarcado en el empleo de metodologías ágiles y CI/CD.

El **grado de automatización** general conseguido a través de la solución propuesta para el sistema estudiado **es muy alto**. De hecho, de no ser por el orquestador, el grado de automatización de tareas de comprobación y de despliegue sería total. No obstante, pese a las dificultades que ha añadido dicho componente, se han podido solucionar todos los problemas encontrados de una forma relativamente sencilla.

Tras concluir el proyecto, puedo afirmar que **los resultados obtenidos** a nivel personal, académico y profesional son **muy positivos**.

A nivel académico, la realización de este proyecto ha reforzado mis conocimientos sobre la usabilidad de las herramientas de control de versiones basadas en GIT, sistemas de testeo de aplicaciones web y herramientas de evaluación de calidad de código. Además, mis conocimientos sobre automatización de procesos, *pipelining*, flujos de trabajo, metodologías ágiles y desarrollo de aplicaciones web también se han visto incrementados.

A nivel profesional, el hecho de haber podido realizar mi TFM en colaboración con una empresa ha sido muy enriquecedor. Esta experiencia me ha permitido enfocar el trabajo hacia un caso real el cual se puede beneficiar de los conocimientos adquiridos durante este estudio. Además, esta situación me ha permitido apoyar y contrastar el estudio frente a los conocimientos de mis compañeros.

Por último, **a nivel personal**, la realización de este trabajo me ha permitido reafirmar la importancia de la automatización de procesos, el *testing* y la calidad de código en el proceso de desarrollo software. Sin estas prácticas, el desarrollo software resultaría más complejo, sobre todo a la hora de desplegar y comprobar su funcionamiento lo que finalmente deriva en problemas a largo plazo.

8.2. Líneas futuras

No caben dudas de que, **el proceso de desarrollo y la automatización de procesos**, son dos áreas que están en **constante evolución**. Durante el desarrollo de los proyectos, pueden producirse modificaciones en sus requisitos que afectan de manera directa al proceso de desarrollo y que, por lo tanto, requieran de la modificación de dichos procesos. Es por esto por lo que, a lo largo de la vida útil del producto, todos estos procesos deben de ser revisados frecuentemente, llegando a realizar cambios de mantenimiento en las *pipelines* o herramientas propuestas en caso de que fuese necesario.

Un punto de avance sobre la solución propuesta, puede ser el **estudio de validadores**

de código más avanzados. En este sentido, este trabajo se ha centrado sobre todo en el análisis estilístico, pruebas de seguridad de aplicaciones estáticas (SAST) y análisis de composición de software (SCA). No obstante, existe muchas otras técnicas de validación que pueden ser estudiadas e incorporadas al proceso de desarrollo propuesto por este trabajo. Ejemplos de estas técnicas son las pruebas de seguridad de aplicaciones dinámicas (DAS) o las pruebas de seguridad de aplicaciones interactivas (IAST).

Lo mismo sucede con el *testing.* La solución propuesta incluye pruebas unitarias, pruebas de integración pruebas de interfaz de usuario y pruebas de carga y estrés. No obstante, existen michos **otros tipos de *testing*** (pruebas de aceptación, pruebas de usabilidad, pruebas de rendimiento…) los cuales podrían combinarse con los ya estudiados incrementando así la capacidad de validación del sistema. Estos nuevos test podrían incluirse dentro del *pipeline* si se considerase oportuno.

Otro punto de interés lo podemos encontrar en la **expansión de la automatización** realizada a través del *pipeline*. Además de los ya incluidos, existen otros campos de interés como la generación de documentación automatizada o la comprobación de los despliegues realizados mediante llamadas de salud. Estas dos etapas cumplimentan muy bien con las ya propuestas ya que facilitan todavía más el desarrollo y despliegue del producto.

También puede resultar interesante el estudio de pequeñas **modificaciones de la aplicación evaluada** y apreciar como esas modificaciones afectarían a la solución propuesta. Esto aumentaría el grado de generalización de los proyectos sobre los que se puede aplicar la solución.

Por último, la **búsqueda de más problemas comunes** sobre el sistema estudiado, así como las soluciones propuestas para los mismos resultaría muy interesante para aumentar los horizontes de la solución.

Capítulo 9 –Bibliografía

[1] K. Gene, J. Humble, P. Debois, and J. Willis, *The DevOPS Handbook*. 2016.
[2] N. Forsgren, J. Humble, and K. Gene, *Accelerate: The Science of Lean Software and DevOps*. 2018.
[3] «Angular». https://angular.io/ (accedido 11 de septiembre de 2023).
[4] «JavaScript With Syntax For Types.» https://www.typescriptlang.org/ (accedido 11 de septiembre de 2023).
[5] BillWagner, «Documentos de C#: inicio, tutoriales y referencias.» https://learn.microsoft.com/es-es/dotnet/csharp/ (accedido 11 de septiembre de 2023).
[6]. «NET | Crear. Probar. Implementar.», *Microsoft*. https://dotnet.microsoft.com/es-es/ (accedido 11 de septiembre de 2023).
[7] «Visual Studio: IDE y Editor de código para desarrolladores de software y Teams», *Visual Studio*. https://visualstudio.microsoft.com/es/ (accedido 11 de septiembre de 2023).
[8] «Visual Studio Code - Code Editing. Redefined». https://code.visualstudio.com/ (accedido 11 de septiembre de 2023).
[9] MashaMSFT, «SQL Server technical documentation - SQL Server». https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16 (accedido 11 de septiembre de 2023).
[10] John-Hart, «IIS Express Overview», 3 de diciembre de 2020. https://learn.microsoft.com/en-us/iis/extensions/introduction-to-iis-express/iis-express-overview (accedido 11 de septiembre de 2023).
[11] Atlassian, «Scrum: qué es, cómo funciona y cómo empezar», *Atlassian*. https://www.atlassian.com/es/agile/scrum (accedido 11 de septiembre de 2023).
[12] mijacobs, «Team Foundation Server - TFS», 22 de marzo de 2022. https://learn.microsoft.com/en-us/previous-

versions/azure/devops/all/overview?view=tfs-2017 (accedido 11 de septiembre de 2023).
[13] «Git». https://git-scm.com/ (accedido 11 de septiembre de 2023).
[14] «Jenkins», *Jenkins*. https://www.jenkins.io/ (accedido 11 de septiembre de 2023).
[15] «Sonatype Nexus Repository - Binary & Artifact Management | Sonatype». https://www.sonatype.com/products/sonatype-nexus-repository (accedido 11 de septiembre de 2023).
[16] «Oracle Service Bus | Oracle España». https://www.oracle.com/es/middleware/technologies/service-bus.html (accedido 11 de septiembre de 2023).
[17] R. Cecil, *Clean Code: A Handbook of Agile Software Craftsmanship*. 2008.
[18] «Linter IDE Tool & Real-Time Software for Code | Sonar». https://www.sonarsource.com/products/sonarlint/ (accedido 11 de septiembre de 2023).
[19] «GitHub - StyleCop/StyleCop: Analyzes C# source code to enforce a set of style and consistency rules.» https://github.com/StyleCop/StyleCop (accedido 11 de septiembre de 2023).
[20] «ReSharper: The Visual Studio Extension for .NET Developers by JetBrains», *JetBrains*. https://www.jetbrains.com/resharper/ (accedido 11 de septiembre de 2023).
[21] «EditorConfig». https://editorconfig.org/ (accedido 11 de septiembre de 2023).
[22] gewarren, .«NET code style rule options - .NET», 11 de octubre de 2022. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/code-style-rule-options (accedido 11 de septiembre de 2023).
[23] «Prettier · Opinionated Code Formatter». https://prettier.io/index.html (accedido 11 de septiembre de 2023).
[24] «Node.js», *Node.js*. https://nodejs.org/es (accedido 11 de septiembre de 2023).
[25] «Options · Prettier». https://prettier.io/index.html (accedido 11 de septiembre de 2023).
[26] «Prettier - Code formatter - Visual Studio Marketplace». https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode (accedido 11 de septiembre de 2023).
[27] «SQLFluff». https://sqlfluff.com/ (accedido 11 de septiembre de 2023).
[28] «SQL Enlight», *SQL Enlight*. https://sqlenlight.com/sql-enlight/ (accedido 11 de septiembre de 2023).
[29] «SonarQube 10.2». https://docs.sonarsource.com/sonarqube/latest/ (accedido 11 de septiembre de 2023).
[30] «SonarLint for Visual Studio 2022 - Visual Studio Marketplace». https://marketplace.visualstudio.com/items?itemName=SonarSource.SonarLintforVisualStudio2022 (accedido 11 de septiembre de 2023).
[31] «typescript-eslint». https://typescript-eslint.io/ (accedido 11 de septiembre de 2023).
[32] «ESLint - Visual Studio Marketplace». https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint (accedido 11 de septiembre de 2023).

[33] «Configuration Files - ESLint - Pluggable JavaScript Linter». https://eslint.org/docs/latest/use/configure/configuration-files (accedido 11 de septiembre de 2023).
[34] «Configuration Files (New) - ESLint - Pluggable JavaScript Linter». https://eslint.org/docs/latest/use/configure/configuration-files-new (accedido 11 de septiembre de 2023).
[35] «SQL Enlight - Visual Studio Marketplace». https://marketplace.visualstudio.com/items?itemName=Ubitsoft.SQLEnlight (accedido 11 de septiembre de 2023).
[36] «JUnit 5». https://junit.org/junit5/ (accedido 11 de septiembre de 2023).
[37] «NUnit.org». https://nunit.org/ (accedido 11 de septiembre de 2023).
[38] «pytest: helps you write better programs — pytest documentation». https://docs.pytest.org/en/7.4.x/ (accedido 11 de septiembre de 2023).
[39] «Istanbul Code Coverage», *GitHub*. https://github.com/istanbuljs (accedido 11 de septiembre de 2023).
[40] Atlassian, «What is Code Coverage?», *Atlassian*. https://www.atlassian.com/continuous-delivery/software-testing/code-coverage (accedido 11 de septiembre de 2023).
[41] «Code Coverage, el resultado de tus pruebas unitarias», *OpenWebinars.net*, 14 de enero de 2020. https://openwebinars.net/blog/code-coverage-el-resultado-de-tus-pruebas-unitarias/ (accedido 11 de septiembre de 2023).
[42] «moq». Moq, 9 de septiembre de 2023. Accedido: 11 de septiembre de 2023. [En línea]. Disponible en: https://github.com/moq/moq
[43] «Jasmine documentation home». https://jasmine.github.io/pages/docs_home.html (accedido 11 de septiembre de 2023).
[44] «Karma». Karma, 9 de septiembre de 2023. Accedido: 11 de septiembre de 2023. [En línea]. Disponible en: https://github.com/karma-runner/karma
[45] «Karma Test Explorer (for Angular, Jasmine, and Mocha) - Visual Studio Marketplace». https://marketplace.visualstudio.com/items?itemName=lucono.karma-test-explorer (accedido 11 de septiembre de 2023).
[46] «NUnit 3 Test Adapter - Visual Studio Marketplace». https://marketplace.visualstudio.com/items?itemName=NUnitDevelopers.NUnit3TestAdapter (accedido 11 de septiembre de 2023).
[47] Mikejo5000, «Code coverage testing - Visual Studio (Windows)», 19 de junio de 2023. https://learn.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2022 (accedido 11 de septiembre de 2023).
[48] «NCover | .NET Code Coverage for .NET Developers». https://www.ncover.com/ (accedido 11 de septiembre de 2023).
[49] «Istanbul, a JavaScript test coverage tool.», *istanbuljs-website*. http://istanbul.js.org/ (accedido 11 de septiembre de 2023).
[50] «Postman API Platform | Sign Up for Free», *Postman*. https://www.postman.com (accedido 11 de septiembre de 2023).
[51] «Selenium», *Selenium*. https://www.selenium.dev/ (accedido 11 de septiembre

de 2023).
[52] «JavaScript Component Testing and E2E Testing Framework | Cypress». https://www.cypress.io/ (accedido 11 de septiembre de 2023).
[53] «Protractor - end-to-end testing for AngularJS». https://www.protractortest.org/#/ (accedido 11 de septiembre de 2023).
[54] «Apache JMeter - Apache JMeter™». https://jmeter.apache.org/ (accedido 11 de septiembre de 2023).
[55] «GitHub: Let's build from here», *GitHub*. https://github.com/ (accedido 11 de septiembre de 2023).
[56] «The DevSecOps Platform». https://about.gitlab.com/ (accedido 11 de septiembre de 2023).
[57] «Servicios de informática en la nube | Microsoft Azure». https://azure.microsoft.com/es-es (accedido 11 de septiembre de 2023).
[58] J. Humble, D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 2010.
[59] «Docker: Accelerated Container Application Development», 10 de mayo de 2022. https://www.docker.com/ (accedido 11 de septiembre de 2023).
[60] «Repository | GitLab». https://docs.gitlab.com/ee/user/project/repository/ (accedido 11 de septiembre de 2023).
[61] «Using a Jenkinsfile», *Using a Jenkinsfile*. https://www.jenkins.io/doc/book/pipeline/jenkinsfile/ (accedido 11 de septiembre de 2023).
[62] «Blazor | Compilación de aplicaciones web cliente con C# | .NET», *Microsoft*. https://dotnet.microsoft.com/es-es/apps/aspnet/web-apps/blazor (accedido 11 de septiembre de 2023).
[63] «Bridge 17.10.1». https://nuget.org/packages/Bridge/ (accedido 11 de septiembre de 2023).
[64] «GitHub - theolivenbaum/h5: 🚀 The next generation C# to JavaScript compiler». https://github.com/theolivenbaum/h5 (accedido 11 de septiembre de 2023).
[65] «Decision Model and Notation™ (DMN™) | Object Management Group». https://www.omg.org/dmn/ (accedido 11 de septiembre de 2023).
[66] «OMG Standards Introduction | Object Management Group». https://www.omg.org/about/omg-standards-introduction.htm (accedido 11 de septiembre de 2023).
[67] «DMN Tutorial», *Camunda*. https://camunda.com/dmn/ (accedido 11 de septiembre de 2023).
[68] «BPMN Specification - Business Process Model and Notation». https://www.bpmn.org/ (accedido 11 de septiembre de 2023).
[69] «DMN Editor - Visual Studio Marketplace». https://marketplace.visualstudio.com/items?itemName=redhat.vscode-extension-dmn-editor (accedido 11 de septiembre de 2023).
[70] G. Rangaraj, «Specification pattern in C#», *CodeProject*, 17 de octubre de 2013. https://www.codeproject.com/Articles/670115/Specification-pattern-in-Csharp (accedido 11 de septiembre de 2023).