Universidad Zaragoza

Trabajo Fin de Grado

Grado en Ingeniería Informática

Metodología para mejorar la calidad de la entrega continua de proyectos software desplegados sobre plataformas de gestión de contenidos

Autor/es

José Ignacio Hernández

Directora

Alexandra García Domenech

Ponente

Francisco Javier Nogueras Iso

Escuela de Ingeniería y Arquitectura

2021

Resumen

El proceso de desarrollo de software se encuentra en constante evolución. Por ello, muchas empresas apuestan por la implementación de metodologías de integración y entrega continua en sus proyectos. Esta decisión agiliza los trabajos de detección y depuración de errores, así como asegura unos mínimos de calidad en el producto mediante pruebas en todas las etapas del desarrollo.

Hoy en día la gran mayoría de equipos trabajan utilizando un sistema de control de versiones que a su vez se desea que sirva como plataforma para desplegar ese producto. Se hace interesante la idea de poder aunar todo el proceso de desarrollo, revisión y entrega en una misma plataforma, permitiendo un acceso fácil a todas las partes implicadas.

Este Trabajo de Fin de Grado estudia el estado actual de la integración y entrega continua en proyectos sobre plataformas de gestión de contenido (CMS) como Drupal en un entorno de trabajo real, y propone la mejora e implantación de una metodología de entrega continua utilizando control de versiones y tecnologías de virtualización.

La metodología de entrega continua se implementa mediante el uso de 'tuberías' homogéneas sobre el sistema de integración continua de GitLab CI. Este procedimiento incluye la automatización de la ejecución de pruebas de análisis estático, pruebas unitarias, funcionales, seguridad y rendimiento utilizando diferentes frameworks.

Este trabajo se ha desarrollado en el ámbito de la empresa Hiberus Digital Business, S.L. Esto ha permitido validar la metodología propuesta sobre un caso real.

Palabras claves: Entrega continua, Integración continua, Control de versiones, CMS, GitLab

Índice

1. Introducción y objetivos

1.1. Contexto y motivación

El marco de este Trabajo de Fin de Grado (TFG) se desarrolla en un entorno de trabajo real. La empresa en cuestión es Hiberus Digital S.L, quienes se definen como "una de las principales compañías de tecnología españolas. Especializados en servicios de consultoría de negocio, desarrollo tecnológico, transformación digital y outsourcing. Prestamos servicios a organizaciones públicas y privadas a las que ayudamos a satisfacer sus necesidades de negocio." [1]

Una de las muchas áreas de especialización es el desarrollo de portales web de contenidos, para lo cual usan (entre otros) *Drupal* [2], uno de los sistemas de gestión de contenidos basado en web más utilizados en la industria.

El modelo de Drupal se relaciona con el patrón *Presentation Abstraction* and *Control* (PAC), dónde diferentes componentes tienen su propia presentación y controlador. De esta forma, si se considera que una página de *Drupal* contiene diferentes bloques, cada uno tiene su propio modelo abstracto, vista y controlador. Además, tal como se puede ver en la Figura I Drupal se compone de tres piezas claves:

- Core: Conjunto de funcionalidades y servicios que traer Drupal por defecto (incluye librerías, modules, themes, etc.)
- Modules: Es el apartado donde se definen la mayoría de componentes funcionales y lógicas de los mismos. Se pueden instalar módulos de terceros que modifiquen el comportamiento de Drupal y añadan nuevas funcionalidades (contributed, desarrollados por la comunidad y aprobados por Drupal), o desarrollar módulos propios (custom)
- Themes: Proveen las plantillas HTML, estilos CSS e incluso scripts en el lado del cliente para la capa de presentación.

Cuando Hiberus decide lanzar un nuevo proyecto sobre Drupal, el trabajo que se debe realizar se centra fundamentalmente en los componentes *modules* y *themes*. Cada cliente requiere módulos personalizados y una presentación acorde con la imagen de cada empresa.

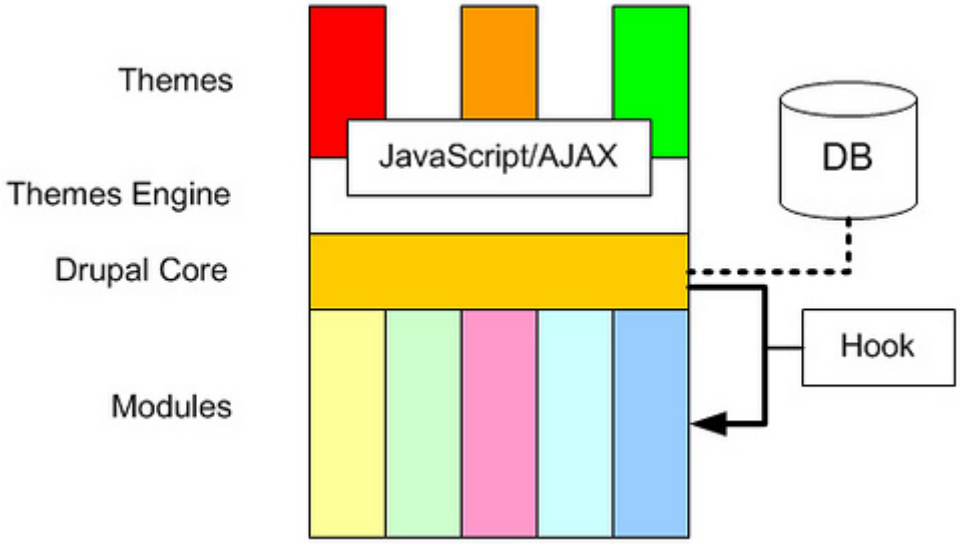


Figura I - Arquitectura de Drupal [3]

En Hiberus se encuentran más de 60 desarrolladores en el equipo de Drupal. Por ello, existe la motivación de crear una línea de despliegue lo suficientemente completa de manera que aporte información a los desarrolladores para detectar errores previos a producción, y es en este contexto donde se encuadra este TFG.

Además, conviene aclarar que este despliegue está relacionado con los conceptos de "integración y entrega continua". La **entrega continua (***Continuous Delivery, CD***)** es la práctica que te permite, si quieres hacerlo, desplegar automáticamente en el entorno de desarrollo o producción cada integración exitosa del software. [4]. Existe otro término que muchas veces se confunde por entrega continua, se habla de **despliegue continuo (***Continuous Deployment, CD***)** a aquella práctica que va un paso más allá y cada cambio pasa todos los escenarios de la línea de despliegue de producción, sin intervención humana, y solo un fallo en las pruebas podrá denegar ese cambio a producción final. La **integración continua (***Continuous Integration, CI***)** es la práctica de comprobar que cada cambio que actualiza el repositorio de trabajo se integra correctamente con lo anterior, supone un requisito para la entrega continua. [5] Generalmente cada persona del equipo integra sus cambios al menos una vez por día, lo que lleva a numerosas integraciones diarias. Para conseguirlo se pueden poner en marcha automáticamente diferentes tipos de pruebas y análisis cada vez que se actualice.
Uno de los elementos claves de la entrega/integración continua es la inclusión de test orientados a automatizar las pruebas a distintos niveles: pruebas de desarrollo, pruebas de sistema, pruebas de aceptación e incluso la evaluación del software sin ejecutarlo. La Figura II muestra, a grandes rasgos, los diagramas de flujo de cada una de estas prácticas y su alcance.

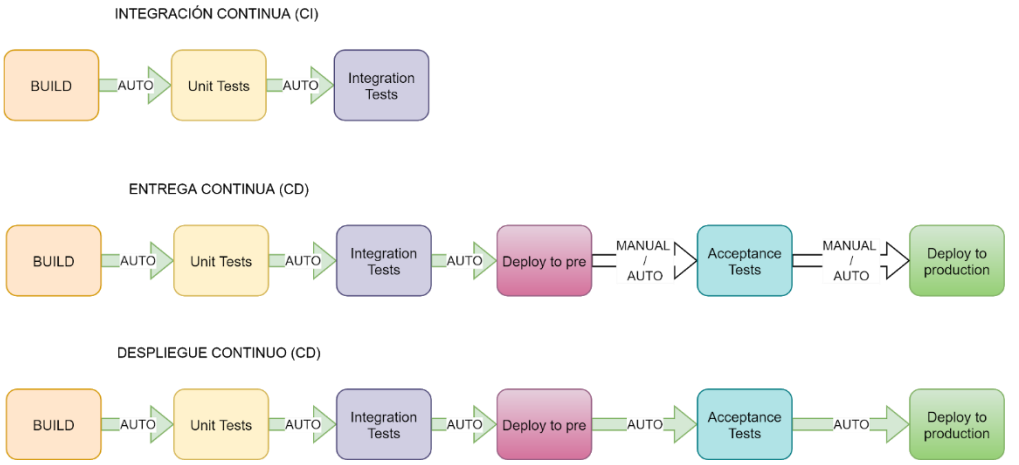


Figura II - Diagramas de flujo de CI , CD (Continuous Delivery) y CD (Continuous Deployment)

En el marco de este proyecto nos hemos decantado por la integración y entrega continua, ya que el matiz de que existen clientes a los que reportar y que disponen de la palabra final para efectuar cambios o no, limita los despliegues automáticos a producción. No obstante, se aboga por realizar despliegues intermedios en preproducción, sobre el que se aplicarán técnicas de pruebas varias, como se detalla en puntos posteriores.

1.2. Objetivos

El objetivo principal de este TFG es establecer una metodología de integración y entrega continua que sirva para cualquier proyecto informático desplegado sobre plataformas de gestión de contenidos Drupal (CMS).

En concreto, esta propuesta de metodología tiene como subobjetivos concretos crear una infraestructura adaptable donde se puedan configurar los pasos necesarios en una entrega continua e integrar scripts de pruebas automatizados de los siguientes tipos de pruebas:

- Pruebas de análisis estático de la calidad del código: Comprueban la corrección del código antes de que este sea ejecutado, puede detectar errores, vulnerabilidades de seguridad y sugerir cambios. Todo ello se debe realizar en función de un estándar establecido.
- Pruebas funcionales: Comprueban que el software ejecutado se corresponda con la descripción de funcionalidades descritas. Esto será diferente para cada tipo de proyecto y dependerá del documento de especificación de requisitos.
- Pruebas de seguridad: Analizan el proyecto en busca de vulnerabilidades directas (en el propio código) o heredadas por dependencias de terceros.
- Pruebas de rendimiento: Son pruebas no funcionales que permiten verificar que se cumplen unos límites dentro de los cuales se considera válido el rendimiento de un sitio web. Recogen mediciones de cómo funciona el proyecto en diferentes entornos.
- Pruebas de accesibilidad: Comprueban el grado en el que personas con dificultades físicas puedan utilizar el entorno sin mayor problema. Se pueden establecer límites a partir de los cuales se considera válido.

Además, en esta propuesta metodológica se ha impuesto como restricción utilizar la plataforma GitLab CI. Para el resto de tipos de pruebas sí que se ha dado completa

libertad en este TFG para estudiar distintas alternativas e integrar aquellas que mejor se adapten a las necesidades de la empresa y los proyectos que se desarrollan.

Finalmente, para probar la aplicabilidad de la metodología se pretende probar el desarrollo sobre un proyecto piloto y una vez aprobada se llevará a proyectos en producción.

1.3. Tecnologías utilizadas

La metodología de integración y entrega continua diseñada se ha implementado mediante *pipelines* homogéneas sobre el sistema de integración continua de GitLab CI. La elección de la plataforma sobre la que se implementa el sistema CI/CD viene decantada por el entorno de trabajo que se utiliza en Hiberus y desarrollos previos. El entorno de trabajo se organiza en base a repositorios privados alojados en GitLab y que están asociados al grupo de desarrollo de Drupal de Hiberus. Por ello, encaja a la perfección la oportunidad de encapsular toda la metodología en una misma herramienta como es GitLab. Ya existía una *pipeline* básica que simplemente se encargaba de construir y desplegar cada proyecto, pero con este TFG se ha formalizado el uso de *pipelines*, se han mejorado la funcionalidades que proporcionan y se han documentado.

El procedimiento de entrega continua propuesto incluye también la automatización de análisis estático del código, pruebas funcionales o de seguridad con las siguientes herramientas:

- El apartado de pruebas de análisis estático, y también dinámico, de la calidad del código vendrán de la mano de las herramientas PHP Code Sniffer (estático) y PHPStan.
- Las pruebas funcionales se llevan a cabo con el framework de testing Cypress.
- El análisis de seguridad viene tanto por la herramienta Local PHP Security Checker como por la batería de herramientas de análisis estático de seguridad que incorpora GitLab (SAST y Secret Detection).
- Finalmente, se utilizan las herramientas K6 y Lighthouse CI que proporcionan reportes de rendimiento (configurables a ciertos niveles que se quieran alcanzar). Lighthouse también devuelve métricas de accesibilidad. Para este último punto también se incluye adicionalmente una herramienta que incorpora el propio GitLab (accessibility).

Además, para testear la metodología sobre sistemas CMS, se desarrollará un proyecto piloto con tecnología Drupal y PHP virtualizado sobre contenedores Docker.

La elección de estas herramientas en concreto se justificará en los puntos posteriores, donde se explica la investigación realizada que ha llevado a la conclusión de sugerir estas soluciones.

1.4. Estructura de la memoria

Este documento se encuentra compuesto por 5 puntos y 3 anexos. El contenido de cada uno es el siguiente:

- 1. **Introducción y objetivos**: apunta de manera general cual es el contexto en el que se desarrolla el trabajo, así como ciertos conceptos claves y los objetivos que se marcan. También hace una pequeña descripción de las herramientas que se han utilizado.
- 2. **Metodología**: recoge la propuesta de metodología que se desarrolla en este trabajo, detalla cada apartado que se integran en la propuesta y el estudio de

cada herramienta utilizada, así como la motivación para ser usada en este marco de proyecto.

- 3. **Implementación de la metodología sobre GitLab CI/CD y Docker**: declara cómo se orquesta todo, las tecnologías involucradas y detalla cada paso que implementa la *pipeline* que aplica la metodología.
- 4. **Resultado**: se expone el resultado final de la metodología aplicada sobre un proyecto piloto y finalmente sobre un proyecto de empresa real, Marcotran.
- 5. **Gestión, conclusiones y trabajo futuro**: indica valoraciones finales y competencias obtenidas, así como propuestas de mejora y ciertos aspectos de la gestión del proyecto.
- Los tres anexos se dividen en:
    - ANEXO I: se indican aspectos interesantes de la configuración de las herramientas utilizadas, así como una revisión del resultado que estas generan. También se incluye un apartado de errores comunes encontrados y su solución.
    - ANEXO II: documentación que se ha realizado en el contexto del proyecto real de empresa Marcotran, se ha escrito en lenguaje inglés.
    - ANEXO III: incluye el cronograma que recoge las actividades desarrolladas durante este proyecto y las fechas en las que se han ejecutado.

2. Metodología

Esta sección describe el flujo de trabajo que se propone para la integración y entrega continua del proyecto que se plantea con GitLab CI/CD, así como las pruebas que se proponen incluir dentro de la metodología.

En particular, se profundiza sobre las herramientas que se han investigado y se han propuesto para cada uno de los tipos de pruebas que se especificaron como subobjetivos en la sección 1.2.

2.1. Propuesta de flujo de trabajo con GitLab CI/CD

GitLab [6] es una herramienta colaborativa para el control de versiones en desarrollo software basado en Git. Comenzó como un proyecto open source y derivó en dos versiones distintas, GitLab CE (Community Edition) y GitLab EE (Enterprise Edition), lo cual se traduce en un modelo de negocio open-core [7]. Más adelante (ver sección 2.4.2) se indicará en qué limitaciones se traduce la versión de GitLab utilizada en el proyecto, debido al modelo de suscripciones que ofrece GitLab añadiendo funcionalidades y soporte. Entre los muchos servicios que ofrece GitLab, se incluye un sistema de integración y entrega continua denominado *GitLab CI/CD***.**

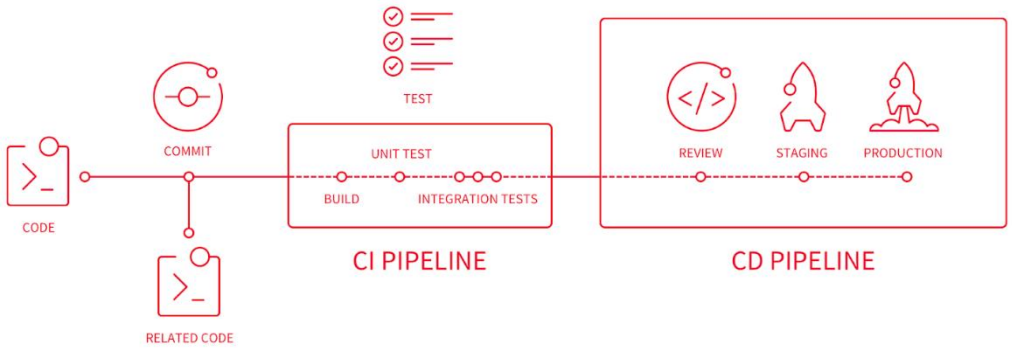


Figura III - Diagrama de flujo GitLab CI/CD [8]

GitLab CI/CD permite a los desarrolladores asociar sus repositorios con tuberías (*pipelines*) que construyan, realicen pruebas y desplieguen sus proyectos. Este poderoso motor, similar a otras opciones disponibles en el mercado, como *Jenkins*, *Circle CI*, *Travis CI*, *Team City* o *GitHub Actions* entre otros, permite personalizar cómo serán estas tuberías en función de la estructura del repositorio o las necesidades del equipo.

La arquitectura se basa en unos denominados GitLab ***runners***, máquinas virtuales aisladas que se encargan de ejecutar el trabajo definido por GitLab CI/CD y devolver los resultados producidos. Se puede configurar un runner por proyecto y este ser reutilizado para otros indicándose en la configuración del repositorio. El tipo de trabajo es maestro - esclavo, por lo que puede darse el caso de que una *pipeline* quede pendiente de ejecución ya que el runner asociado está siendo utilizado por otro repositorio.
Cada runner tendrá su configuración específica. Para aquellos proyectos con requisitos similares se pueden configurar ***Shared runners*** de manera que, si esta opción está activada, el trabajo a ejecutar en cualquier proyecto puede ser cogido por cualquiera de los runners indicados como “compartidos”. Utilizan una política de uso justo, evitando a proyectos que creen cientos de trabajos y usen todos los runners compartidos

disponibles. [9] Existe una versión más acotada que se denomina ***Specific runners***, los cuales son exclusivos para los proyectos que así lo indiquen, y no se encuentran disponibles para ningún otro repositorio. Una capa de configuración más son los ***Group runners***, permiten asociar un conjunto de runners a todos los proyectos pertenecientes a un grupo de GitLab. Siguen políticas de uso FIFO.

En el caso de los proyectos Drupal de Hiberus se propone utilizar un runner específico (**Specific runner**), cuya configuración se detalla en el apartado de arquitectura.

Dado que esta metodología se piensa aplicar con proyectos basados en Drupal, a continuación, se detalla la estructura de los repositorios de Drupal en Hiberus. Por lo general, se encuentran organizados en diferentes ramas. Las principales son ***develop***, ***qa*** y ***master***. Adicionalmente, dependiendo del estado del proyecto, se pueden crear subramas puntuales *como* ***feature/X*** o ***release/sprintX*****.**
Cada una representa un estado concreto del desarrollo:

- Develop: Rama que integra los cambios realizados en las ramas de desarrollo aisladas (features/x), se une (*merge*) a QA después de pasar las pruebas en el entorno de integración.
- QA: Rama de entorno de preproducción, previa a que el portal desarrollado salga a producción. Se crea al finalizar un sprint, tras finalizar las pruebas en release/sprintX y será el entorno desplegado sobre el que el cliente realice sus pruebas de aceptación. Asegurando que lo que se va a desplegar en el entorno de producción no va a fallar y cumple con lo requerido.
- Master: Es la rama que representa el entorno de producción, estado estable y desplegado del proyecto.
- Feature/X: Rama local de desarrollo de la característica X (generalmente para proyectos Drupal, módulo), que se integrará en develop
- Release/sprintX: Es la rama que representa el entorno de preproducción en desarrollo. En este punto suelen realizarse pruebas end-to-end (e2e). Se supone que todos los problemas de calidad de código ya han sido solucionados en las ramas de feature y develop. Una vez finalizado el sprint se pasa el contenido de esta rama a qa.

Con este flujo en mente, se plantea integrar la solución para la lógica entre ramas que se presenta en la Figura IV.

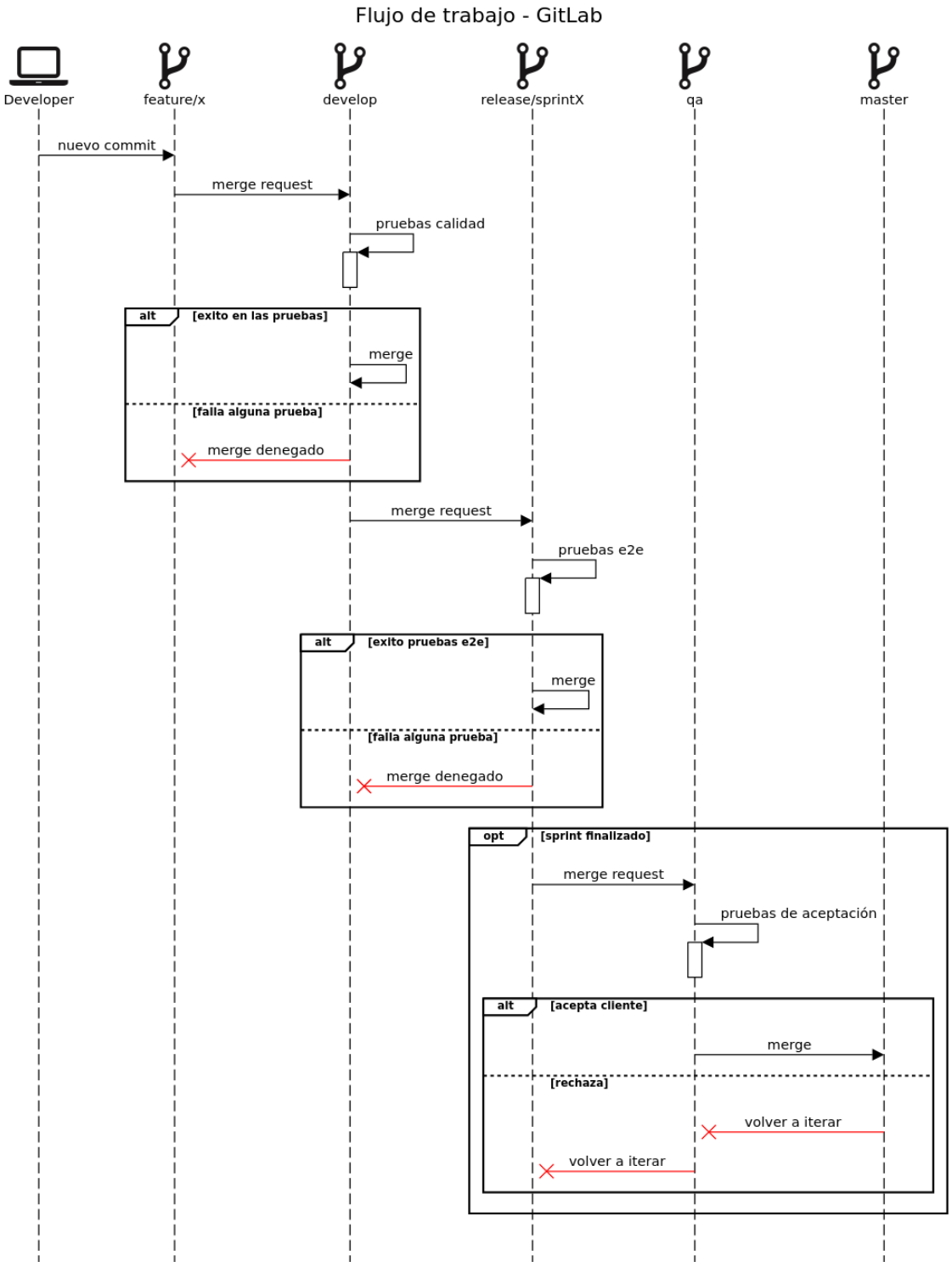


Figura IV - Diagrama de secuencia para GitLab Drupal en Hiberus

El diagrama presentado en la Figura IV es el flujo de trabajo deseado entre ramas, de manera que si falla alguna de las pruebas pertinentes no se efectúa el *merge* y debe ser revisado por el desarrollador para corregir los errores. En el momento en el que se inicia el proyecto no hay ninguna prueba de calidad de código o integración. También cabe destacar que las pruebas de aceptación en la rama *qa* especifican que el cliente podrá acceder a un entorno de preproducción desplegado, y si este se encuentra conforme, se pasarán los cambios al entorno de producción.

GitLab permite especificar sobre qué ramas o en qué ocasiones (*merge requests*) se ejecutarán los trabajos (*jobs*) especificados, así como añadir excepciones de ramas sobre las que no se deben ejecutar ciertos *jobs*. De forma similar a otros motores que soportan CI/CD, la acción de la *pipeline* asociada al repositorio se especifica en un fichero *gitlab-ci.yml* en la raíz del mismo. Este consiste principalmente de escenarios (*stage*) en los que se divide la *pipeline*. Cada escenario se compondrá de trabajos (*job*) que se encargan de tareas en forma de scripts y los cuales pueden generar artefactos (*artifacts*) como resultado de la ejecución.
Cada *job* se puede dividir a su vez en varios elementos, uno de ellos es la imagen (*image*) docker que configura el entorno sobre el que trabajara. Si recordamos, previamente hemos hablado también de un runner específico, el cual deberá soportar la definición de docker in docker.
Finalmente, con el fin de reducir tiempos de ejecución de cada *job*, se puede definir las llamadas estrategias de caché. De manera que se puede cachear directorios entre cada *job* para que no sea necesario recompilar dependencias en cada uno.

Tal y como se ha comentado en la introducción, este entorno de control de versiones viene impuesto por parte de la empresa. Pero una vez investigado su poderoso sistema de integración y entrega continua, supone la herramienta perfecta para aunar todo el proceso de construcción, pruebas y despliegue en un mismo entorno. Esto facilitará las tareas de revisión, evitando tener que saltar entre diferentes portales y simplificando el sistema de notificaciones. Solo recibiremos correos electrónicos de GitLab para informar del estado de nuestro despliegue.

2.2. Integración de pruebas relacionadas con el análisis de la calidad del código

Los analizadores de código estático surgen de la necesidad de detectar errores sin necesidad de ejecutar el código. Un analizador estático lee el texto del código y trata de encontrar patrones típicos de errores. Uno de los ejemplos más claros de este tipo de análisis es el analizador que se incluye en la mayoría de los entornos de desarrollo integrados (IDE) como PHPStorm, Visual Studio Code o IntelliJ IDEA. Estos analizadores indican visualmente problemas comunes de código como son variables no definidas, errores en tipos en llamadas a funciones o asignaciones. Algunos son capaces de analizar otros menos comunes como son estilos de código, longitud de líneas de código, etc.

Con el fin de liberarse de la dependencia del IDE que utilice el desarrollador, se propone integrar este tipo de análisis y detecciones en la propia *pipeline*. De esta manera se ofrece un reporte para que el desarrollador pueda consultarlo en cada *commit* y así mantener un registro del progreso de mejora en la calidad del código.

Siendo PHP uno de los lenguajes principales utilizados en el desarrollo de proyectos Drupal, se estudiaron las opciones existentes para asegurar la calidad del código.
Una de las características de un lenguaje interpretado como PHP en comparación a lenguajes compilados como Java es que no se debe esperar al compilador. Pero surge un problema, ¿cuándo se da cuenta de que hay errores? Al no compilar, pueden existir muchos más errores en tiempo de ejecución. A diferencia de los lenguajes compilados, no nos daremos cuenta de que hay un fallo hasta que se ejecute la línea de código en cuestión que lo genera. Por ello surge la necesidad de reducir el tiempo de análisis manual / visual que llevan a cabo los desarrolladores, en favor de analizar el código de forma estática automatizada.

Cuando se buscan soluciones del tipo de analizadores de código estático aplicable a PHP , existe un gran número de posibilidades [10] y esto complica el trabajo de elegir la mejor solución. No obstante, si se restringe la búsqueda de analizadores a aquellos que soporten estándares de código Drupal, sobresalen como mejores prácticas los nombres de PHP Code Sniffer y PHPStan.

Este tipo de herramientas aplicadas a nuestro caso de uso como es un proyecto de Drupal, permiten personalizar ciertas características de calidad de código, detección de errores y sugerencias en el análisis del proyecto. En concreto las pruebas se deben centrar sobre aquellos dos pilares fundamentales que comentamos en la introducción de los proyectos de drupal: *modules* y *themes*. Suponen la mayor parte de los esfuerzos de desarrollo y se propone que estos análisis de calidad se centren exclusivamente en aquellos módulos y temas que se hayan desarrollado de manera independiente. Excluyendo aquellas que vengan por defecto en el *core* de Drupal o incluidas por terceros.

Centrándonos en los dos nombres mencionados anteriormente, también surgen dudas sobre si utilizar ambas herramientas o solamente una. En las siguientes subsecciones se explica detalladamente cómo funcionan.

2.2.1. PHP Code Sniffer

PHP Code Sniffer [11] se compone de dos scripts de PHP. El principal, denominado **phpcs** , es el encargado de tokenizar archivos con extensión de PHP, JavaScript y CSS para detectar violaciones en el estándar de código. El segundo tipo de script se corresponde a **phpcbf**, el cual se encarga de corregir automáticamente estas violaciones detectadas por el principal. En nuestro caso, no haremos uso de esta funcionalidad ya que proponemos generar reportes para que sean los desarrolladores quienes corrijan una vez sea denegada la integración y mantengan un estándar de código en todo el proyecto.

PHP Code Sniffer viene con una serie de estándares de código PHP predeterminados (PEAR, PSR2, MySource, Zend, Squiz, PSR12 y PSR1) que se pueden aplicar en sus análisis. No obstante, estos no son completamente aplicables a lo que recomienda la propia organización de Drupal para desarrollar nuevos módulos. Es por ello que la documentación oficial de Drupal establece una guía para registrar los estándares denominados **Drupal** y **Drupal Practice**, para que puedan ser utilizados por PHP Code Sniffer. Adicionalmente, la documentación de Drupal incluye una guía explicando cada criterio que sigue su estándar de código [12] .

2.2.2. PHPStan

A priori, PHPStan [13] se define como una herramienta de análisis estático que sustituiría el rol de un compilador en otro lenguaje de programación. Es open source, de uso gratuito y se encuentra basado en el PHP Parser creado por Nikita Popov.

PHPStan usa técnicas de caché para mostrar el mismo resultado de análisis si detecta que la fuente a analizar no ha cambiado desde la última vez que se ejecutó. Para definir la configuración de PHPStan se puede indicar en el propio comando que lanza el script **phpstan analyse** o utilizar un fichero de extensión **.neon** en el que se puede indicar que directorios serán analizados, así como niveles estrictos que debe superar el análisis para no considerarse peligros o errores. También permite diferentes tipos de salidas

para sus análisis (tablas, json, compatible con teamcity o GitHub, etc.). La que nos interesa es el formato en tabla, ya que agrupa los errores por fichero de manera colorida y es la preferible para lectura humana. Y así será, porque se mostrará en la salida del *job* de GitLab que ejecute el análisis para una posterior revisión por un desarrollador si esta falla.

PHPStan cuenta con una gran documentación, la cual es útil para interpretar los errores que este te lanza.
Uno de los puntos no tan buenos de PHPStan es que los errores que presentan son de tipo de texto, por lo cual complica la tarea si se desea clasificar o filtrar los errores más comunes ya que se deberá hacer mediante expresiones regulares. También complica a la hora de generar estadísticas.
En comparativa PHPStan es notablemente más rápido que otras opciones como Phan (analizador estático de código desarrollado inicialmente por Rasmus Lerdorf. Esta herramienta se centra más en evitar falsos positivos) y entre un 20-50% más rápido que Psalm (solución también open source de análisis estático para PHP desarrollada por la compañía Vimeo). [14]

No obstante, todavía no se ha hablado de cómo se aplica al desarrollo en Drupal. Esto es porque de manera similar a PHP Code Sniffer, no funciona por defecto con el estándar de Drupal. PHPStan no sabe cómo cargar ninguno de los ficheros, ya que PHPStan depende de poder cargar las clases a través de un autoloader generado por dependencias de composer. Cuando habilitas un módulo o tema, no se añade información de cómo cargar sus ficheros al autoload.

Para atajar estos problemas, existe una extensión de PHPStan desarrollada por Matt Glaman en la cual incluye a este paradigma el poder analizar proyectos Drupal y sus dependencias.

2.2.3. Conclusiones

Entonces, ¿en qué se diferencian PHP Code Sniffer y PHPStan?
PHP Code Sniffer se define como una herramienta de análisis estático que puede ayudarnos a encontrar errores en nuestro código. Cuando se ejecuta PHP Code Sniffer, cada fichero se tokeniza y parsea individualmente. No obstante, no es capaz de detectar si código, métodos o clases referenciadas realmente existen en otros ficheros. Tampoco es capaz de detectar cambios en las interfaces de un método.

Por ejemplo, si un método espera una interfazA pero en realidad se le pasa un interfazB. , un analizador de código estático no es capaz de notificar esta discordancia.

Aquí entra una de las diferencias claves de PHPStan y su análisis dinámico. Una de las claves de este analizador es que, si encuentra clases desconocidas, tratará de cargar estas automáticamente y entender su interfaz. Si por algún motivo, no es capaz de resolver esa clase, devolverá un error.

2.3. Integración de pruebas funcionales

En el contexto del desarrollo de software, las pruebas funcionales recogen diferentes tipos de validaciones con el fin de demostrar que un producto funciona correctamente a ojos del usuario. Cada prueba se suele asociar a un caso de uso concreto y puede implementarse en diferentes niveles.

- Pruebas unitarias: Valida específicamente el funcionamiento de un fragmento concreto del software. Puede ser una operación lógica del código, y sirve para asegurar la calidad de cada elemento interno del software que conforma el producto final.
- Pruebas de integración: Suelen ocurrir una vez finalizadas las pruebas unitarias, comprueban el conjunto de elementos del software y la comunicación entre estos.
- Pruebas funcionales de sistema o de aceptación: Se realizan al final del ciclo de pruebas, de manera que se cercioran de que cumple con las especificaciones del caso de uso que representa. La diferencia entre el nivel de sistema y aceptación es que las primeras son realizadas por la parte proveedora (empresa de desarrollo) y las segundas están pensadas desde el punto de vista de la parte aceptante (cliente).

En el caso que nos interesa, estas últimas pruebas de aceptación suelen venir establecidas por unos requisitos definidos por los clientes, y en la mayor parte de los casos son estos últimos los que se encargan mediante una demo del producto de comprobar las funcionalidades del sistema. Por lo cual, se decide centrar los esfuerzos de este tipo de pruebas en la metodología en aquellas pruebas de integración / funcionalidad que aseguren tener un estado funcional conforme a requerimientos para que a continuación pueda desplegarse al entorno de preproducción para ser aceptado por el cliente.

Se estudiaron qué tipo de herramientas podían encajar con la naturaleza web del proyecto y se plantearon alternativas como *Selenium* [15] (pruebas / automatización de navegador), *Playwright* (pruebas sobre multi plataforma, posibilidad de ejecutar sobre todos los navegadores modernos), *Cucumber* (no se trata de una herramienta específicamente diseñada para pruebas sobre navegador, pero soporta automatización sobre navegador mediante *Selenium WebDriver*), *Robo* (framework para correr tareas PHP utilizado en otros proyectos del departamento de CMS de Hiberus), *Puppeteer* (pruebas de navegador *headless* para Google Chrome) o *Cypress***.**

Uno de los estándares en la industria es el uso de *Selenium* para realizar pruebas funcionales contra la interfaz gráfica del navegador. Ideado como una grabación de pasos de un usuario en Firefox, desembocó en *Selenium WebDriver* que ofrece una interfaz de lenguaje para registrar acciones como las realizaría un usuario.

Para que funcione de la manera más genérica posible, en este punto se propone el uso de una herramienta de testing funcional llamada Cypress. Cypress es un framework de pruebas funcionales basado en JavaScript.
Cypress [16] se diferencia así mismo de *Selenium* en la aproximación que toma en su arquitectura. *Selenium* y la mayoría de herramientas operan fuera del navegador y ejecutan comandos a través de la red, sin embargo, *Cypress* está basado en un proceso de servidor *Node*. Por lo tanto, la comunicación y sincronización entre tareas es constante, optimizando su servicio y produciendo resultados más consistentes. No obstante, *Cypress* también opera en la capa de red para leer y modificar el tráfico web durante su ejecución.

Frente al resto de opciones, obviando comparativas de velocidad y diferencias de arquitectura entre ellas, se valora el discurso de *Cypress* en el que apunta que aunque se puede utilizar perfectamente sobre aplicaciones ya desplegadas, realmente se encuentra optimizado para ser utilizado en el desarrollo de día a día ya que hasta es posible utilizarlo sin depender de un backend operativo (simplemente permite interceptar

peticiones y completarlas con una respuesta predefinida, similar al concepto de los *mocks*).

Cypress aúna principalmente tres tipos de pruebas:

- e2e: Permite realizar pruebas sobre cualquier contenido que se pueda ejecutar sobre un navegador. Incluye operaciones típicas como visitar la página, realizar acciones y aserciones que comprueben los efectos de esta. A efectos prácticos es la automatización de las mismas acciones que realizaría un usuario mediante interfaz gráfica.
- Componente: Se trata de una funcionalidad experimental en el momento de este trabajo. Supone el uso de la librería de pruebas de *Cypress* para realizar los mismos objetivos que logran frameworks de pruebas como **jest** o **mocha**, siendo el resultado la prueba de componentes concretos del proyecto de manera aislada, sin necesidad de que se carguen sobre una página. Los componentes se renderizan mediante un navegador virtual denominado jsdom.
- API: Puede realizar llamadas HTTP, permitiendo también realizar pruebas sobre la API expuesta de tu proyecto.

Una de las ventajas añadidas de Cypress es que permite el uso de plugins, tanto verificados como de la comunidad, que añaden funcionalidades tan destacadas como la comparación del apartado visual de una página a partir de una captura de pantalla de referencia. Funciona tal que considerará inválida la comparación si esta supera un límite establecido en la diferencia de píxeles.

Todo este tipo de pruebas están a disposición de lo que consideren necesario los desarrolladores y propietarios del producto. Implementadas mediante scripts de *JavaScript* supone un paso más en la automatización del ciclo de vida del producto y la persistencia de los cambios.

Otra de las características que incluye es la posibilidad de grabar los resultados de los tests, tanto como artefactos para descargar en local, como mediante el uso de un panel de control alojado en una web y que sirve a modo de registro visual del historial completo de ejecuciones de la herramienta *Cypress*. En ese registro puedes acceder a los artefactos (capturas de pantalla o vídeos) que te genera, reproducirlos y ver paso a paso cómo se ha desarrollado el test ejecutado.

Toda esta versatilidad supone que para la ejecución de pruebas funcionales se proponga la solución ofrecida por *Cypress*, coincidiendo con la experiencia de los desarrolladores en JavaScript y la naturaleza de pruebas en entorno local aplicado a la *pipeline*. No obstante, una de las sugerencias que se hace teniendo en cuenta el estándar de la industria, es el uso de *Selenium* para reproducir testing sobre el escenario de navegador ya desplegado en preproducción para asegurar así también la compatibilidad entre navegadores web que ofrece *Selenium webdriver*. Pues uno de los claros puntos negativos de *Cypress* es el tiempo de ejecución inicial que pesa demasiado para hacer pruebas cortas en entornos desplegados, pero no supone tal gasto en pruebas locales.

2.4. Integración de pruebas de seguridad

Se encuentra en la naturaleza de estos proyectos el utilizar *composer* y dependencias externas. Composer es una herramienta para la gestión de dependencias en PHP: permite declarar librerías de las que dependerá el proyecto y se encarga de instalarlas y actualizarlas cuando se construya el proyecto

Estas dependencias externas pueden ser susceptibles de riesgos de seguridad y ciertamente el tamaño en proyectos tan grandes hacen muy complicada la gestión de monitorizar si las nuevas vulnerabilidades pueden afectar de manera directa o indirecta al trabajo desarrollado.

Inicialmente se propuso una de las herramientas más referenciadas de análisis de seguridad en proyectos PHP, se trata de Security Checker de Sensiolabs [17], pero se descarta ya que indican en el repositorio oficial que los servicios web han dejado de funcionar al final de enero de 2021. En su lugar, recomiendan una herramienta similar que realiza la misma tarea, pero de manera "local". También se encuentra otra característica, en este caso relacionada con la seguridad que integra la plataforma de GitLab. Se trata de una plantilla de análisis de seguridad denominada GitLab SAST.

2.4.1. Local PHP Security Checker

Se trata de Local PHP Security Checker [18], una herramienta de línea de comando que comprueba si la aplicación PHP tiene dependencias vulnerables. Para ello lo comprueba contra la base de datos PHP Security Advisories Database, la cual se mantiene constantemente actualizada mediante contribuciones de usuarios. Más adelante se detalla cómo sirvió para reportar una vulnerabilidad en el proyecto real.

2.4.2. GitLab SAST

Entre las muchas herramientas que ofrece la plataforma de GitLab para integración y entrega continua, se encuentran aspectos de Seguridad. Mediante los cuales se pueden añadir fácilmente a la *pipeline* herramientas para analizar el código y detectar vulnerabilidades conocidas, denominado Static Application Security Testing (SAST) [19]. Puede incluso analizar la historia de Git para detectar si existen posibles filtraciones de secretos

- SAST: Similar a la situación anterior con Local PHP Security Checker, GitLab incluye una batería de pruebas para analizar el código frente a vulnerabilidades conocidas. Soporta una gran cantidad de lenguajes, en nuestro caso PHP, incluye la phpcs-security-audit que supone una serie de reglas de PHP Code Sniffer para puro PHP y compatibilidad con reglas específicas de Drupal 7.
- Secret Detection [20]: En nuestro caso, al ser repositorios privados y solo accesibles mediante *vpn* no debería ser un problema crucial. Pero sorprendentemente se han detectado vulnerabilidades por mostrar credenciales sin cifrar en repositorios. Una herramienta como Secret Detection, integrada en el entorno de GitLab CI/CD supone una vulnerabilidad cubierta, aseguramiento de buenas prácticas y ahorro de tiempo sustancial en la detección de casos vulnerables.
    - Entre las claves por defecto que detecta se encuentran: **Cloud services**(Amazon Web Services (AWS), Google Cloud Platform (GCP), Heroku API), **Claves de cifrado**(PKCS8, RSA, SSH, PGP, DSA, EC), **Redes sociales**(Facebook API, Twitter API), **Cloud SaaS**(GitHub API, Shopify API, Slack Token, Slack Webhook, Stripe API, Twilio API, Claves de API que comiencen por la cadena api-), **Contraseñas en URL**, **Número de Seguridad Social de Estados Unidos**

Como se mencionó en la introducción, uno de los "inconvenientes" de GitLab es la existencia de distintos modelos de suscripción, lo cual limita por ejemplo ciertas funcionalidades en el uso de esta herramienta. No obstante, GitLab SAST se encuentra

disponible también para utilizarse en el nivel básico gratuito, tal como se puede observar en la Figura V.

Capability	In Free	In Ultimate
Configure SAST Scanners	✓	✓
Customize SAST Settings	✓	✓
View JSON Report	✓	✓
Presentation of JSON Report in Merge Request	○	✓
Address vulnerabilities	○	✓
Access to Security Dashboard	○	✓
Configure SAST in the UI	○	✓
Customize SAST Rulesets	○	✓

Figura V - Funcionalidades SAST disponibles en cada nivel de GitLab

Como ocurría con los otros análisis, también es posible establecer un límite del nivel de riesgo a partir del cual se considera reportar una vulnerabilidad mediante variables.

2.5. Integración de pruebas de rendimiento

En el ámbito del software, se define a las pruebas de rendimiento como los procesos por los cuales se somete al sistema que queremos probar a una carga de trabajo específica con la finalidad de medir los resultados que proporciona (velocidad, tiempo de respuesta, fiabilidad, estabilidad) [21] . Las pruebas de rendimiento se ejecutan con los siguientes objetivos en mente:

- **Localizar cuellos de botella**: Teniendo en cuenta la calidad del servicio frente a un número elevado de usuarios, permiten identificar qué aspecto en concreto del sistema está lastrando al resto frente al trabajo que se le requiere.
- **Cumplimiento de requisitos de servicio:** Aseguran que se corresponden los **SLA (Service Level Agreement)** negociados y cumplen las expectativas de nuestros clientes.
- **Medir la capacidad del sistema y planear escalabilidad:** Una vez tienes unas medidas que representan cómo se comporta tu sistema ante cargas altas de trabajo, es útil hacer uso de estas para planear la escalabilidad de tu proyecto conforme crece la base de usuarios que lo utilizan.

Existen diferentes tipos de pruebas de rendimiento:

- **Carga**: Determinan el comportamiento del sistema frente a ciertas cargas específicas de trabajo. Se indican unos objetivos esperados de la ejecución de estas pruebas. Sirven para identificar problemas de rendimiento generales y a continuación tratar de corregirlos.
- **Estrés:** Determinan el punto a partir del cual un sistema deja de funcionar debido a carga extrema. Ayuda a identificar la utilización de los recursos del sistema sobre cargas muy altas de trabajo.

- **Estabilidad**: A diferencia de las anteriores, este tipo de pruebas se realizan para una carga moderada de trabajo, pero durante un periodo de tiempo alargado. Ayudan a identificar problemas que puedan presentarse tras un largo periodo de uso pero que de otra manera pasarían desapercibidos.
- **Alto tráfico "esporádico"**: Similar al caso de las pruebas de estrés, comprueban el comportamiento del sistema frente a un repentino aumento en el número de usuarios / peticiones. Sirve para identificar problemas bajo ciertas circunstancias especiales, como podría ser una venta de artículos durante un tiempo determinado u otros eventos.
- **Volumen:** En estos escenarios se expone al sistema frente a un alto volumen de datos, bien sea mediante poblar la base de datos o el almacenamiento con diferentes tipos de datos, de manera que refleje el comportamiento frente a altas cargas de este tipo de trabajo.

En el caso de los gestores de contenidos, el producto final puede llegar a ser usado por miles de usuarios de manera simultánea, haciendo crítica esta etapa del proceso de pruebas.

Se propone para el marco de esta metodología la aplicación de pruebas de rendimiento una vez se despliega en el entorno de desarrollo de preproducción.
Similar a lo que ocurría en las pruebas funcionales, también se encuentra una herramienta por excelencia establecida en lo que es el estándar de la industria, denominada *JMeter*. Una de los puntos claves de esta herramienta es que dispone de interfaz gráfica mediante la cual diseñar planes de prueba, siendo así un punto de entrada sencillo. Se pueden partir de plantillas y bloques de tests gráficos para implementar los primeros tests. También existen algunos inconvenientes en el uso de *JMeter*, su implementación de 1 hilo por 1 usuario virtual (*VUs*) supone un alto coste en memoria y el formato XML de sus scripts hace que sea menos legible para compartir y colaborar.

Realizando este estudio uno se encuentra con otra clara candidata para la implementación de pruebas de rendimiento, la más joven k6. También de naturaleza *open source*, toma una orientación algo distinta a *JMeter* y se basa principalmente en código JavaScript para diseñar sus pruebas, sin interfaz gráfica exceptuando un servicio en la nube que dispone de panel de control para el usuario. Otro punto es el bajo consumo de memoria que supone k6 frente a otras soluciones (implementando rutinas del lenguaje *Go* para representar los usuarios virtuales, además la carencia de interfaz gráfica también reduce consumo de recursos en el sistema), lo que permite a una sola instancia de la herramienta el poder emular más usuarios simultáneos virtuales (hasta 30.000 - 40.000, generando hasta 300.000 peticiones por segundo) que otras herramientas de prueba como *JMeter*. Lo cual será más que suficiente para nuestro caso de prueba [22]
Para ello se hace uso de la herramienta k6 que permite configurar de manera sencilla mediante imagen de *docker* en la *pipeline*. Permitiendo realizar tests de carga e indicando por ejemplo el número de usuarios que atacaran la aplicación y definir criterios para el cumplimiento de ese test. La posibilidad de establecer *thresholds* a nivel de test se valora como clave y siendo varias las posibilidades de personalización. Tales como:

- x% de las veces el tiempo de respuesta debe ser inferior a los indicados "y" milisegundos.
- x% de peticiones fallidas o y% de peticiones exitosas
- etc.

En cuanto al reporte de resultados, k6 no dispone de visualizaciones incluidas de su análisis (más allá de un resumen de los parámetros en las pruebas end to end, que se

verá en el caso real), por ello también permite tanto exportar los resultados en formatos para scripting como CSV o JSON, y la integración con otras herramientas de visualización como Amazon CloudWatch, Grafana, k6 Cloud (solución propia de la herramienta), Datadog o Netdata entre otras.

2.6. Integración de pruebas de accesibilidad

El uso de herramientas software o de la web está pensado para el consumo humano, sin importar barreras físicas. Uno de los muchos propósitos de la web es unir personas, mejorar la calidad de vida de los usuarios, pero por desgracia inclusive a día de hoy no está al alcance de todos. Al existir un amplio registro de usuarios las necesidades varían y se hace necesario la aplicación de pruebas que aseguren (en la mayoría de los casos y en la medida de lo posible) el mayor grado de accesibilidad al contenido que se desea consumir.

El Consorcio World Wide Web (W3C) es una comunidad que trabaja conjuntamente para desarrollar estándares Web. Uno de sus objetivos es la accesibilidad web (para la cual crearon una división específica de su comunidad, Web Accessibility Initiative, WAI), sobre la que indica lo siguiente:
“La Web debe ser accesible para brindar igualdad de acceso y oportunidades a personas con capacidades diversas. De hecho, la Convención de las Naciones Unidas sobre los derechos de las personas con discapacidad reconoce el acceso a las tecnologías de la información y las comunicaciones, incluida la Web, como un derecho humano básico.” [23]

Queda claro que los beneficios de un buen diseño y navegación guiado por la accesibilidad beneficia a todas las personas (usuarios o propietarios). En el caso de los usuarios aumenta la usabilidad y con ello disminuye el nivel de entrada de acceso a la utilización del sitio web. Además, los motores de búsqueda benefician a aquellos sitios web bien estructurados y contenidos bien detallados, dotándolos de un mejor posicionamiento y apariencia en los resultados de las búsquedas que se realicen.

Para medir el nivel de accesibilidad de un sitio web el W3C definió unas directrices que han sido actualizadas con el paso del tiempo hasta la última iteración en 2018 de las llamadas Guías de Accesibilidad para el Contenido Web (WCAG 2.1) con una nueva propuesta programada para 2021.
Esta guía define una serie de Niveles de conformidad que indican el grado de accesibilidad [24]:

- **A**: Estipula unos requisitos mínimos para que la web no se considere inaccesible. Algunas de las características que menciona son la inclusión de:
  - Operable: Navegación mediante teclado, posibilidad de entrada de acciones mediante ratón / punteros o algo distinto a teclado.
  - Alternativas de texto para contenido no textual.
  - Contenido y fondo distinguibles entre sí.
  - Legible
- **AA**: Debe satisfacer el nivel A, se considera un nivel aceptable de accesibilidad y añade ciertas características como:
  - Contraste de color suficiente (nivel 4:5:1)
  - Elementos de navegación consistente.
  - Formularios con campos lo suficientemente descriptivos y bien etiquetados, así como sugerencias ante errores.
  - Encabezados y subencabezados usados en orden lógico.

    - Legibilidad no restringida por la orientación del dispositivo (vertical u horizontal) , excepto si es imprescindible para presentar un contenido específico.
    - Texto escalable.
- **AAA**: Debe satisfacer los niveles de conformidad anteriores, se considera el nivel óptimo. Es difícil de lograr mantener en la totalidad del sitio web, pero añade ciertas características interesantes:
    - Contraste de color mínimo 7:1
    - Lenguaje de signos para audio o vídeo.
    - Navegable en su totalidad por teclado, sin dependencias del tiempo (combinaciones de teclas o esperas)
    - Deshabilitar animaciones, excepto si es esencial para la funcionalidad o información asociada.

Teniendo en cuenta lo anterior se propone dedicar un escenario concreto de la *pipeline* a realizar un análisis de accesibilidad. Este tipo de reportes también entraron en juego en un apartado previo sin tenerlo consideración ya que la herramienta *Lighthouse* se incorpora en el mismo escenario que *Cypress* para pruebas de integración local. *Lighthouse* se encarga de producir un análisis de calidad de la página web indicada a partir de criterios de rendimiento (se ve lastrado al estar el sistema construido por Docker en un runner de GitLab, más detalle en el apartado de arquitectura y configuración de la *pipeline*), puntos conforme a la accesibilidad web junto a prácticas para mejorarla, y también es posible obtener un puntaje respecto a criterios de *SEO* (optimización para motores de búsqueda).

La opción directa que aparece cuando se investigan herramientas de accesibilidad que se integren bien con GitLab es una solución de similares características a las comentadas en SAST. Se trata de pa11y, una herramienta *open source* de medición de accesibilidad para sitios web que GitLab integra mediante el uso de una plantilla de *job* [25]. Se opta por utilizar esta utilidad, pero con el fin de darle un punto mayor de configuración se encuentra una aproximación en el siguiente proyecto [26] para realizar análisis mediante *pa11y-ci* y con el archivo de configuración de *pa11y-ci* [27] que podremos editar para adaptarnos a nuestros requisitos.
El funcionamiento es el siguiente:

- GitLab *template for* pa11y: Facilita la integración de ejecutar pa11y por defecto en las urls indicadas mediante una variable de entorno de la *pipeline*.
- pa11y-ci: Mediante la configuración de un *job* de la *pipeline* se especifica que utiliza una imagen dockerizada del propio *GitLab-ci-utils* que leerá un archivo denominado .pa11y.json donde se especificará la configuración y las urls que se probarán. Pudiendo invocar también a la herramienta pa11y-ci-reporter-html, que como su propio nombre indica sirve para presentar mediante visualización HTML los resultados recogidos por el pa11cy-ci y facilita mediante código de colores la gravedad en cuanto a accesibilidad de cada elemento inspeccionado

3. Implementación de la metodología sobre GitLab CI/CD y Docker

En este capítulo se trata en detalle cómo se establece la metodología propuesta en este TFG, basado en las prácticas y herramientas comentadas anteriormente. Para ello utilizaremos el sistema de integración continua que ofrece GitLab y describiremos su configuración. Con el fin de separar lo que es la aplicación de la infraestructura, se propone utilizar Docker (plataforma abierta que permite mediante contenedores optimizados y modulares el integrarse sencillamente en un entorno ágil) y sus contenedores desacoplados para ejecutar cada escenario de manera aislada y lo más paralela posible. De esta forma, se puede desarrollar el flujo de trabajo propuesto en la sección 2.1 y más concretamente en el diagrama de secuencia de la Figura IV.

3.1. Arquitectura de la solución

La arquitectura propuesta para implementar el flujo de trabajo tiene tres componentes: el servidor de GitLab, los GitLab Runners y el fichero de configuración YAML. **GitLab** en sí mismo mediante su servidor se encarga de coordinar los procesos, monitorizar los repositorios, poner en marcha las *pipelines*, etc. Luego se encuentran los mencionados **GitLab Runners**, que se encargan de ejecutar los pasos del proceso de la *pipeline*. Estos pueden correr en cualquier plataforma que soporte el lenguaje Go. Normalmente estos procesos de ejecución se localizan en contenedores de Docker. Esto garantiza que cada vez que se ejecute la *pipeline* el entorno será el mismo. Cuando el motor de GitLab Ci detecta un cambio en el código (git *push*) este acciona la *pipeline* y valida el archivo de configuración YAML. A continuación, procesa el orden de los *stages* y *jobs*, los cuales pasan a un estado “pendiente”, a la espera de ser elegidos por un runner. El primer paso que realiza el runner al comunicarse con GitLab es clonar el repositorio. Posteriormente, ejecuta la acción definida por el *job* y se completa con estado de éxito o fracaso, y genera si así lo requiere *artifacts* que enviará al servicio de GitLab CI. Cada transición de *job* en la *pipeline* acciona el servicio de GitLab Ci que actualiza el estado de la *pipeline* y pasa los siguientes *jobs* que deben ser ejecutados. Finalmente, el último componente clave es el archivo de configuración **YAML** (un estándar de serialización de datos, legible para humanos, y que normalmente es usado para escribir archivos de configuración), el cual indica que queremos ejecutar en la *pipeline*.

En nuestro caso, se quiere que GitLab cree *job*s que trabajen tanto con el código del repositorio, como con la aplicación Drupal construida dentro de la *pipeline*. El conjunto de *job*s y sus tecnologías se muestran en el diagrama de despliegue de la Figura VI.

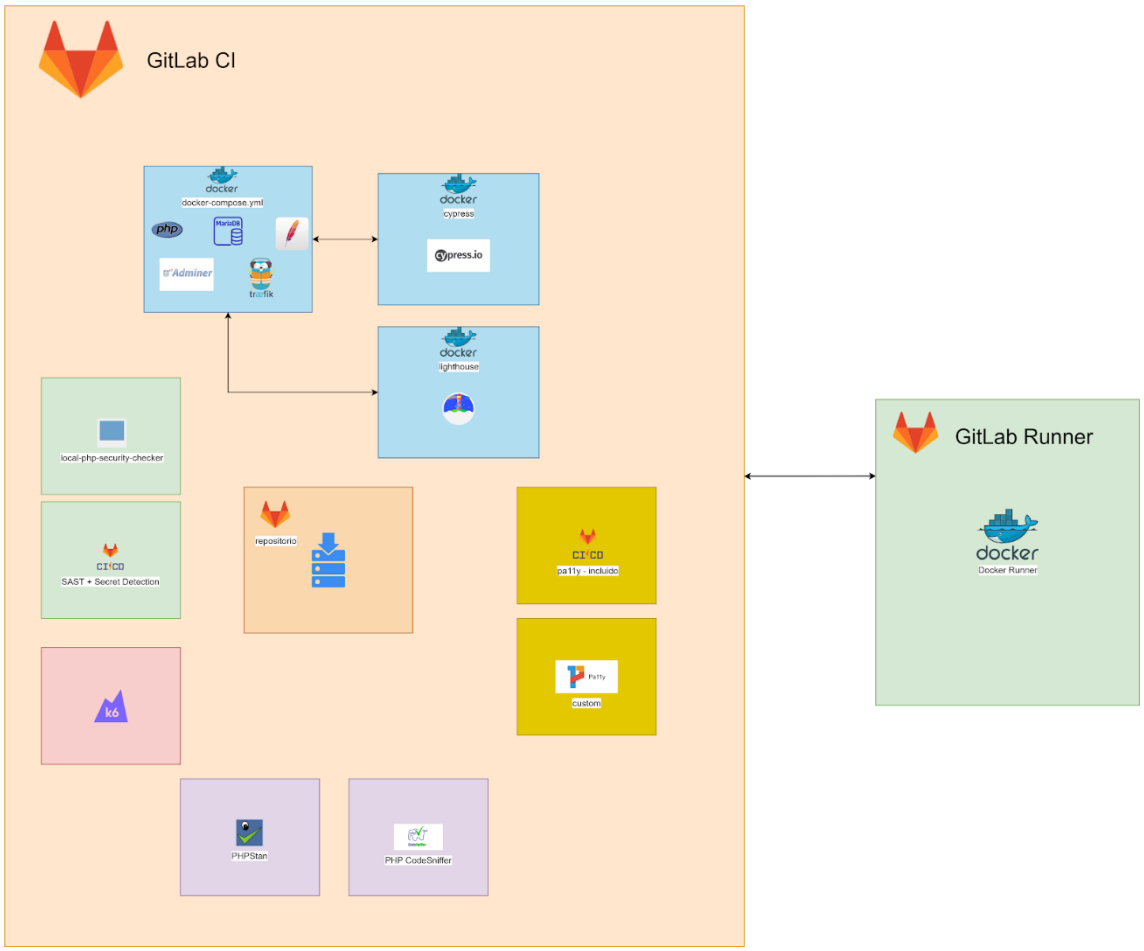


Figura VI - Arquitectura del sistema en GitLab CI

Para lograr la construcción de la aplicación de Drupal mediante Docker se utilizará el servicio denominado **Docker in Docker**. Esta imagen de Docker automáticamente inicializa un docker Daemon, el cual necesitamos para construir y comunicar contenedores en el propio entorno de GitLab CI. [28] El servicio docker:dind se relaciona a los *jobs* que indiquemos en la *pipeline* (o si definimos el servicio de forma global, se relaciona con todos los *jobs*)

Para arrancar la aplicación se requiere de una variedad de contenedores, y con el fin de orquestar esta puesta en marcha se utiliza la herramienta compose. Requiere que definamos los servicios que componen nuestra aplicación en un archivo denominado docker-compose.yml. Adicionalmente se definen unos archivos del mismo tipo, pero representando los servicios de Cypress y Lighthouse que se ejecutarán sobre la aplicación dockerizada.

Necesitamos del servicio *docker*:*dind* para montar la estructura mediante *docker compose*, permitiendo arrancar y para las imágenes desde el entorno de CI.

La aplicación Drupal se basa principalmente en los servicios de docker: *php*, *mariadb*, *adminer*, *apache* y *traefik*. Luego existirán dos servicios en archivos diferentes que representarán a *Cypress* y *Lighthouse* CI ejecutadas sobre esa aplicación levantada en "local" con el fin de realizar esas pruebas **funcionales** y reporte de calidad incluso previas al despliegue en preproducción. Si estas pruebas no resultan exitosas se detiene la *pipeline*, por el contrario, si se completan se pasa a la segunda fase que son los **analizadores estáticos** PHP Code Sniffer y PHPStan. De la misma manera, se pasaría

a una tercera fase que incluye el análisis de GitLab **SAST**, detección de secretos y local-php-security-checker. Tras este punto se realizarán las pruebas de **accesibilidad** mediante a11y de GitLab y la versión personalizada de pa11y. A partir de aquí se considera incluir el proceso de **despliegue** al entorno de preproducción. Una vez este finalice se realizarán pruebas **e2e** mediante el uso de nuevo de lighthouse ci, pero esta vez sobre este entorno desplegado donde sí que tendrá mayor valor los resultados que aporte en cuanto a rendimiento y buenas prácticas. Y finalmente se llevará a cabo pruebas de **rendimiento** mediante trabajos de carga en k6.

3.2. Configuración de la pipeline

En esta sección se explica de forma detallada cómo se configura la *pipeline* sobre un proyecto de Drupal vacío creado de propio para probar la puesta en marcha de la solución propuesta. En la sección 4 se indica cómo se adapta esta *pipeline* para proyectos concretos.

Tal y como se ha comentado en la sección anterior 2.1 de GitLab, se puede definir una estrategia de caché para la *pipeline*. En este caso se opta por incluir los directorios propios de la compilación de dependencias de los proyectos de Drupal, estos serán:

- vendor: Contiene las librerías externas de las que depende Drupal core (por ejemplo, symfony, drush... En nuestro caso ahí se añadirán phpstan, squizlabs/php_codesniffer entre otras). Estas dependencias las carga el autoloader. Pero no todas las dependencias se instalan aquí. Por ejemplo, si requiere de módulos de la comunidad, se instalarán en el /modules/contrib.
- web/core: Incluye librerías externas y propias a Drupal, módulos, temas, perfiles. Todo requerido por el core de Drupal.
- web/libraries: Librería de terceros no incluidas por core, pero lo suficientemente comunes para que se incluyan aquí.
- web/modules/contrib: Diferenciando entre **contrib** y **custom** hace que sea más sencillo seguir la traza y desarrollo de los módulos. En *contrib* se incluyen aquellos módulos creados por la comunidad, mientras que en *custom* se encuentran los creados propiamente por los desarrolladores del proyecto en cuestión
- web/profiles/contrib: Los perfiles definen pasos adicionales que ejecutar tras la instalación base de Drupal, pueden ofrecer funcionalidades adicionales o cambios de comportamiento en el sitio. También se pueden diferenciar entre contribuciones de la comunidad (*contrib*) y perfiles desarrollados específicamente (*custom*).
- web/themes/contrib: Similar a como ocurre con los módulos y perfiles, existen temas desarrollados por la comunidad. Estos temas permiten cambiar la apariencia del sitio web Drupal. Un tema puede ser base de otros subtemas, los cuales también se encontrarán en este directorio.
- drush/: Aquí se encuentra la configuración específica de *Drush* para este proyecto. *Drush* es una utilidad de línea de comandos para comunicarte con Drupal. En nuestro caso se utiliza para instrucciones que devuelvan la información/estado del proyecto, actualicen o modifiquen la base de datos.

En resumen, se incluyen todos aquellos directorios generados en la construcción del proyecto para que no sea necesario compilar en cada paso. De manera adicional, se incluye el directorio de los reportes generados por lighthouse ci, por si se desea consumir en otro escenario ajeno al que ejecuta lighthouse.

- ./.lighthouseci/ : el tipo de archivos que genera lighthouse es el siguiente
    - flags-***.json (flags que utiliza el navegador google chrome)

    - Por cada run que hace de análisis genera resultados del tipo
        - lhr-1617088238124.html - equivalente a lo que almacena en temporary public storage o server
        - lhr-1617088238124.json - toda la información recogida en forma de JSON

La estrategia de caché podrá ser extendida por cada *job* que genere estos directorios (*push*) o que los consuma (*pull*).

Adicionalmente, otro elemento que se puede añadir son plantillas (*templates*), las cuales representan configuraciones de las que pueden extender cada *job* con el fin de no repetir código. Algunas de estas configuraciones son:

- image: Imagen de Docker que utilizarán los *jobs* que extiendan de esta plantilla.
- stage: Escenario al que pertenecerán los *jobs* que extiendan de esta esta plantilla.
- services: Define una imagen de Docker que asociará a la imagen definida previamente para los *jobs* que extienda de esta plantilla. De manera que estos servicios serán accesibles mediante red para la imagen principal del job que lo utilice. (es el caso del servicio *docker*:*dind* mencionado anteriormente) Estos servicios son creados mediante contenedores que utilizarán esa imagen y se comunicarán con el contenedor del job. [29]
- dependencies: Expresa dependencia entre *jobs* tal que los *jobs* que extiendan de esta plantilla no serán ejecutados hasta que termine completamente la ejecución de los *jobs* indicados en esta lista

Con el fin de dividir de forma clara y concisa el trabajo, se definen los siguientes escenarios:

- **Build**: Este escenario actualmente instala las dependencias de composer y sigue una estrategia de caché de tipo push. Por lo cual se pasará el directorio *vendor* que genere con todo lo necesario para que se ejecuten los siguientes *jobs*.
- **E2e-local-tests**: Este escenario implementa pruebas funcionales de Cypress y análisis de Lighthouse sobre la plataforma Drupal en “local”, sin desplegar. Hace uso de docker-compose.yml para la puesta en marcha de la aplicación, docker-compose-cypress.tests.yml para el servicio de Cypress y docker-compose-lhci.tests.yml para el servicio de Lighthouse. El tráfico es dirigido por traefik entre Apache y Adminer. Para el correcto funcionamiento de estas pruebas, es necesario obtener la base de datos que se encuentra remotamente y así actualizar la base de datos que utilizará en la versión local. Esto último se consigue mediante SSH.
- **Tests**: Este escenario realizará los tests asociados al análisis estáticos de código con PHP Code Sniffer y PHPStan.
- **Sast-tests**: Este escenario aúna los tests de seguridad que GitLab Sast considera que se pueden ejecutar sobre el proyecto, la detección de secretos en el repositorio y la versión implementada de local-php-security-checker.

En este punto se considera que si todo ha ido correctamente se realizará un escenario de **deploy** a un entorno de preproducción.

- **Accessibility**: Este escenario combina el análisis de accesibilidad de GitLab a11y y la versión implementada de pa11y que toma un archivo de configuración personalizado para un punto de mayor de detalle en las pruebas.
- **E2e-tests:** Similar al anterior escenario de e2e-local-tests, pero esta vez se realizan sobre el despliegue previo. Utilizará una imagen de docker de Lighthouse sobre GitLab para el análisis de calidad, a diferencia del escenario

local en el que el *job* debía montar la imagen de manera indicada, aquí se indica mediante el parámetro image que se describió anteriormente para los *jobs*.

- **Load-tests:** Este escenario realizará las pruebas de rendimiento sobre la aplicación desplegada mediante el uso de la herramienta k6.

La Figura VII muestra la sucesión de los escenarios que se integran en la *pipeline*, tal como se ha descrito en los dos párrafos anteriores.

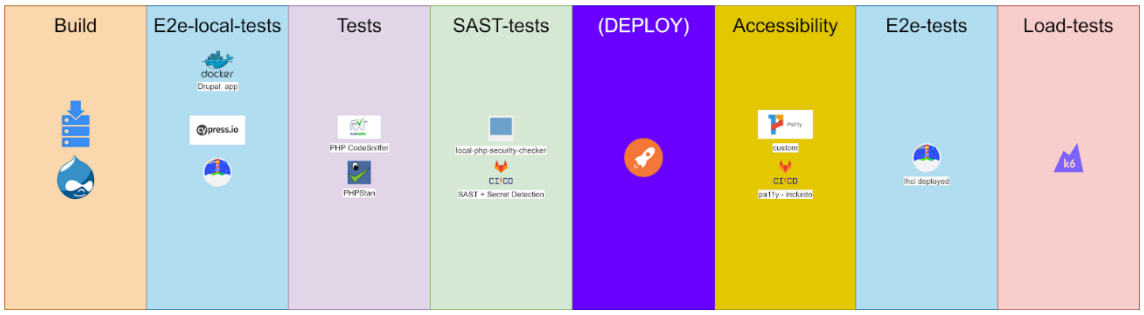


Figura VII - Pipeline propuesta

Los *jobs* que se encuentran en el stage Tests son los que hacen uso de esas plantillas mencionadas anteriormente. Estos *jobs* están en el mismo stage porque comparten gran parte de las características necesarias para ejecutar sus trabajos. Así mismo, depende de que ya hayan terminado los *jobs* de Build y E2e-local-tests.

4. Puesta en marcha de la metodología

Para comprobar el correcto funcionamiento de la propuesta de metodología a seguir, se propone aplicarla sobre un proyecto piloto y más adelante sobre un proyecto en producción concreto. De esta forma, se pretende demostrar que la metodología es fácilmente adaptable a diferentes proyectos basados en Drupal.

4.1. Proyecto piloto

Con el fin de aplicar la propuesta de metodología, se crea un proyecto piloto vacío basado en Drupal. Se basa en una estructura de proyecto de Drupal que dispone de lo básico para desarrollar, la cual Hiberus tenía creada en GitHub para proyectos previos [30].

Se realiza un *dump* (volcado) de una base de datos ya definida, con el fin de que la web no lleve directamente a la página de instalación de Drupal. Y el aspecto del sitio web se puede ver en la Figura VIII.

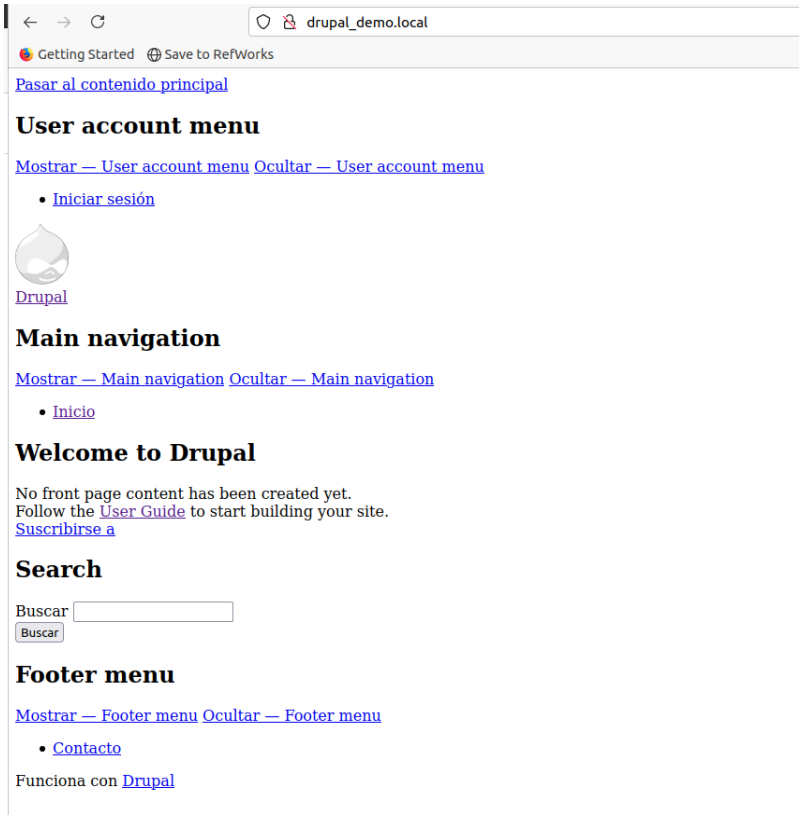


Figura VIII - Proyecto demo (vista inicial por defecto de Drupal)

En la Figura IX se pueden observar cómo resulta sobre GitLab los *stages* propuestos en la sección 3.2 (Figura VII). Respecto al *stage* Sast-tests, cabe destacar que aparte del *job* especificado de local-php-security-checker y el *job* encargado de secret_detection, se incluyen un conjunto de reportes SAST porque GitLab detecta automáticamente su compatibilidad con el proyecto [19]. Estos reportes son:

- eslint-sast: Reglas de ESLint para detectar vulnerabilidades de seguridad de Node. Aunque la propia desarrolladora del plugin que utiliza ese job alerta de

que detecta muchos falsos positivos que deben ser comprobados por un humano.

- nodejs-scan-sast: Otra herramienta de análisis para aplicaciones de Node.
- phpcs-security-sast: similar al local-php-security-checker , pero se decide incluir la versión específica de local para poder presentar en la terminal el resultado del deporte, y no solamente mediante *artifact* como limitan los SAST de GitLab.
- semgrep-sast: Otra herramienta de análisis estático de seguridad, soporta una gran cantidad de lenguajes (Go, Java, JavaScript, JSON, Python, Ruby, TypeScript, etc. Así como otros lenguaje como PHP que se encuentran en fase de alpha) y arroja también recomendaciones del tipo de “no usar RegExp() en una variable, ya que un atacante puede hacer DOS con expresiones regulares de larga duración”.

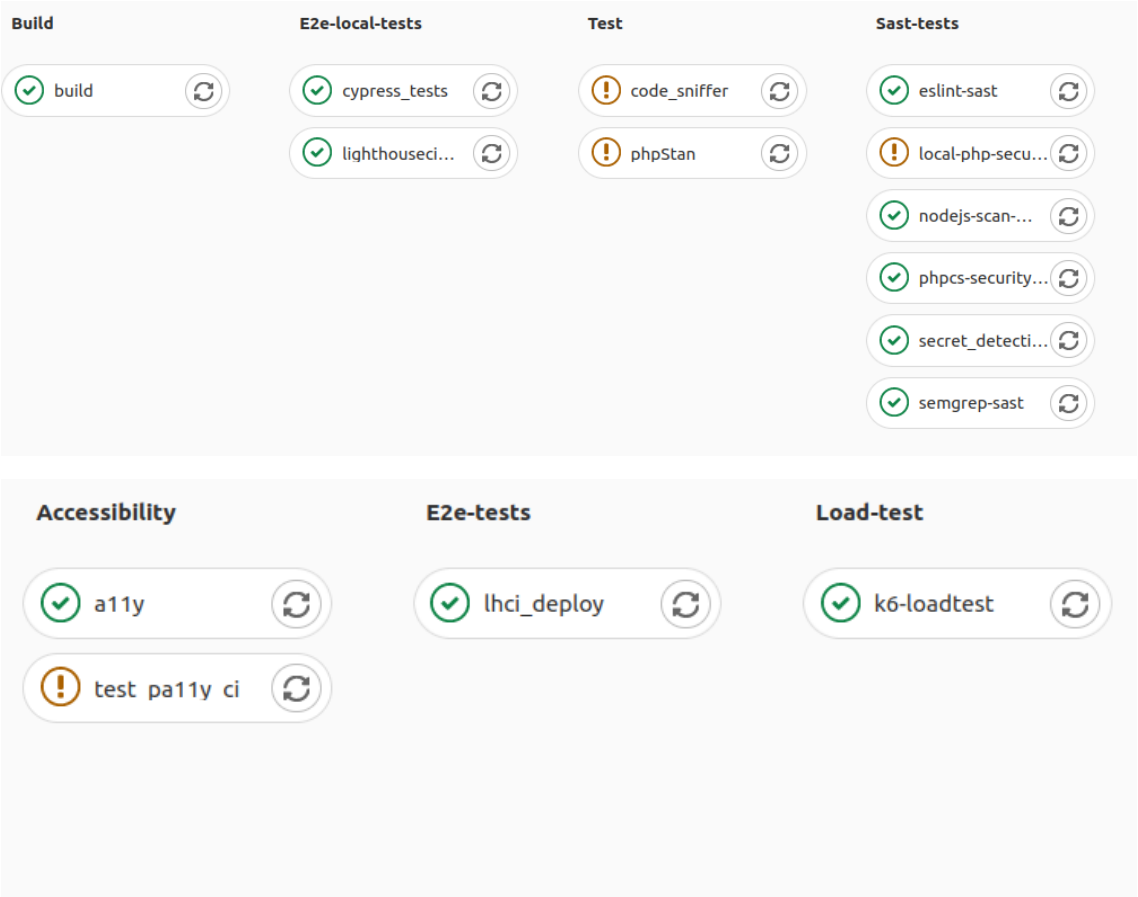


Figura IX - Pipeline de proyecto piloto en GitLab CI

El análisis de calidad de código en condiciones normales se realiza sobre lo que desarrolla uno propiamente, excluyendo así las partes de módulos, perfiles o temas que vengan por defecto en los proyectos de drupal o desarrollados por terceros ya que raramente se van a modificar. En este proyecto piloto al no existir módulos *custom* se toma prestado como referencia un módulo cualquiera de un proyecto real desarrollado por Hiberus. En este caso se ha utilizado el módulo *core* desarrollado para DKV Seguros, el cual define entidades, controladores y elementos base. No es necesario probar su correcta integración ya que simplemente se utilizará para realizar el análisis del código estático mediante las herramientas PHP Code Sniffer y PHPStan. sin llegar a ser ejecutado.

Tal y como se describe en la Figura VII, inicialmente se realiza la compilación de dependencias en el *stage build*. A continuación, se pasa al escenario *E2e-local-tests* que producirá los resultados indicados en los Anexos correspondientes a *Cypress* y Lighthouse CI. Respecto a los resultados que producen todos los *jobs*, todos son exitosos a excepción de los que se indican con el símbolo de alerta  . Esto significa que considera que el *job* ha fallado, pero por conveniencia para el resto de trabajos a ejecutar se permite ese fallo. En condiciones reales, la propuesta de metodología considera que si un *job* falla se detiene la *pipeline* y no se prosigue hasta que sea solucionado y se ejecute correctamente.

En este caso fallan las pruebas referentes a:

- code_sniffer y phpstan: Detecta los errores correspondientes del módulo core de DKV. Esto se elige para ejemplificar que si detecta y reporta los fallos y recomendaciones pertinentes.
- local-php-security checker: Detecta la misma vulnerabilidad que su equivalente SAST de GitLab, pero la diferencia se basa en que GitLab considera que su *job* se ha ejecutado correctamente y sube el *artifact* con el reporte de vulnerabilidades de seguridad. No obstante, local-php-security-checker reporta inmediatamente la vulnerabilidad referente a la versión de *composer* utilizada (véase ANEXO I.3) y realiza su salida con código de error. Por lo cual GitLab considera fallido el *job*, pero se permite este "fallo" para no paralizar la *pipeline*.
- test pa11y ci: Es una situación equivalente a la descrita anteriormente. GitLab considera que su *job* de accesibilidad se ha ejecutado correctamente y sube el *artifact* con el reporte de consideraciones de accesibilidad. Estas consideraciones se deben a que en la web del proyecto piloto no se han seguido ningún criterio de accesibilidad, es la interfaz por defecto de los proyectos de Drupal. No obstante, la versión personalizada de la herramienta *test pa11y ci* reporta las consideraciones de accesibilidad y realiza su salida con código de error al no considerarse un análisis completamente exitoso. Por lo cual GitLab considera fallido el *job*, pero se permite este "fallo" para no paralizar la *pipeline*.

En este proyecto no se realizan ningún tipo de despliegue por lo cual los *jobs* que se proponen ejecutar sobre proyectos desplegados apuntan a páginas webs reales ajenas al proyecto piloto. Es el caso de los escenarios referentes a Accesibilidad, test e2e sobre desplegado y tests de carga, que en la propuesta de metodología se realizan sobre el entorno de preproducción.
El tiempo total del proceso de la *pipeline* es de una media de **20 minutos**. Un tiempo considerable, pero que debido a la cantidad del número de escenarios que se realizan y puesto en comparación al ahorro que supone a las alternativas manuales, se considera asequible en el marco del proyecto en el que se trabaja.

4.2. Proyecto Marcotran

Una vez comprobado el correcto funcionamiento de la propuesta de metodología a seguir descrita anteriormente sobre un proyecto piloto, se propone llevar a la práctica esta misma y demostrar con pocas modificaciones el que es fácilmente adaptable a diferentes proyectos basados en Drupal.

En el contexto de este TFG, se ha elegido como proyecto real el desarrollado por Hiberus para la empresa Marcotran. Marcotran es una empresa que proporciona

servicios de transporte de mercancías fundada en 1979 y con sede social en Zaragoza. Marcotran tiene cobertura internacional en más de 50 países (España, EU, extracomunitarios en África América y Rusia) y cuenta con más de 1600 empleados. La empresa Hiberus fue contratada para desarrollar la web de la empresa donde se gestiona la presentación de los servicios que proporciona, noticias, ofertas de trabajo y permite acceder también a un área privada para usuarios con credenciales. La vista principal de la web en producción se puede observar en la Figura X.

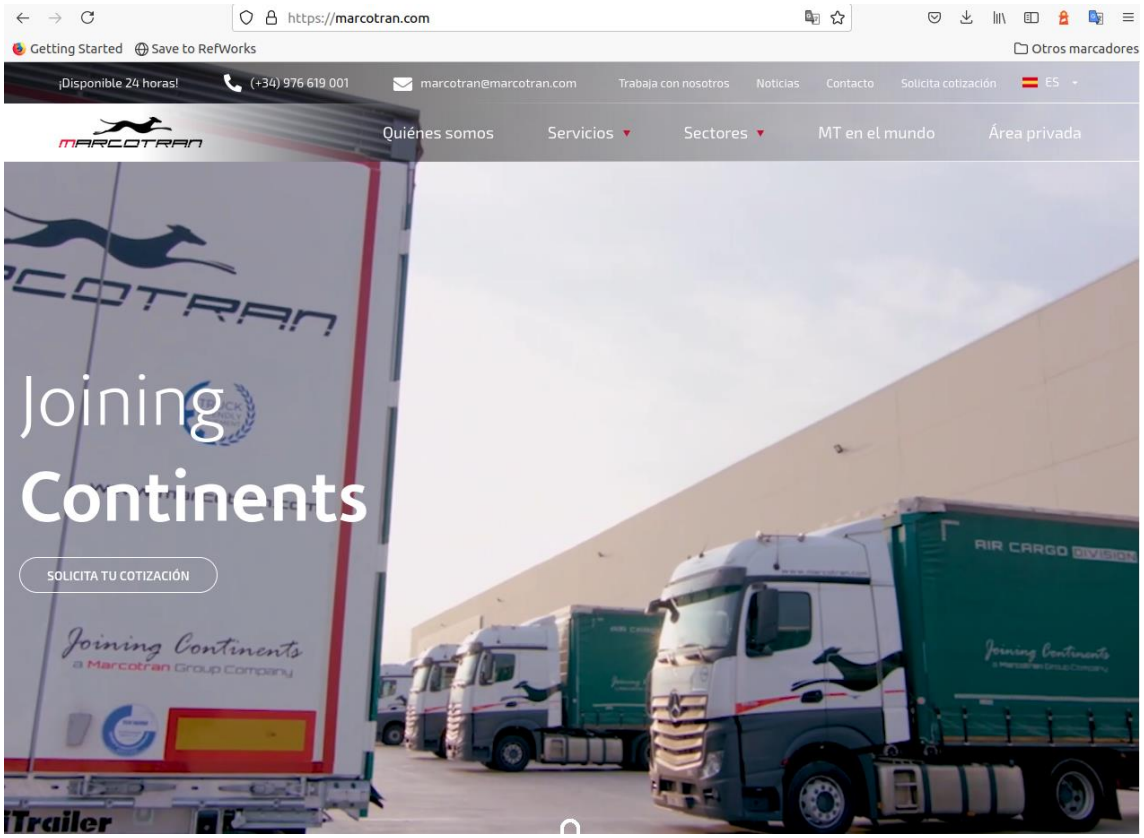


Figura VIII - Proyecto Marcotran (vista inicial)

Antes de incorporar la metodología propuesta en este TFG, la *pipeline* de despliegue inicial de este proyecto realizaba únicamente la fase de deploy. Esta se componía de un total de tres *jobs* (uno por cada rama principal "develop", "qa" y "master") los cuales se encargan de asociar un entorno en el que van a ser desplegado, indicando un nombre y url estáticas. En el caso de la rama *develop* el nombre es *Development* y la url http://marcotran.des.demohiberus.com .

Al abordar el proyecto de Marcotran, se observó que una de las principales diferencias entre el proyecto piloto que se desarrolló y este real es la estructura que decidieron utilizar. Esta diferencia implica que sustituyen el directorio de web (básico en la estructura de Drupal) por uno denominado docroot. En este directorio se incluirán los mismos subdirectorios core, libraries, modules, profiles, sites y themes. Este cambio de denominación supuso un fallo en el *job* de PHPStan ya que no podía determinar dónde se encuentra la raíz del proyecto y requerirá modificaciones internas que se detallarán en el anexo de Errores comunes encontrados.

En la Figura XI se muestra la *pipeline* que ha adaptado para el proyecto Marcotran. La *pipeline* es similar a la del proyecto piloto, pero con algunas diferencias. Los primeros escenarios se traducen de manera idéntica al proyecto piloto, exceptuando los cambios

pertinentes en la configuración para que se corresponda con la estructura que sigue el proyecto de Marcotran. Para las pruebas E2e-local-tests se realiza un *dump* de la base de datos con el contenido de la web, pero para agilizar las pruebas se realiza excluyendo los recursos de imágenes y vídeos que se encuentran en remoto en otro directorio. La visualización y resultados se encuentran en los correspondientes anexos de Cypress (anexo I.4) y Lighthouse CI (anexo I.5). A continuación, se observa que el servicio SAST de GitLab incluye los analizadores de seguridad que se describen anteriormente y añade a su vez otra herramienta como es spotbugs-sast pero que no aporta una diferencia de análisis con respecto al resto.

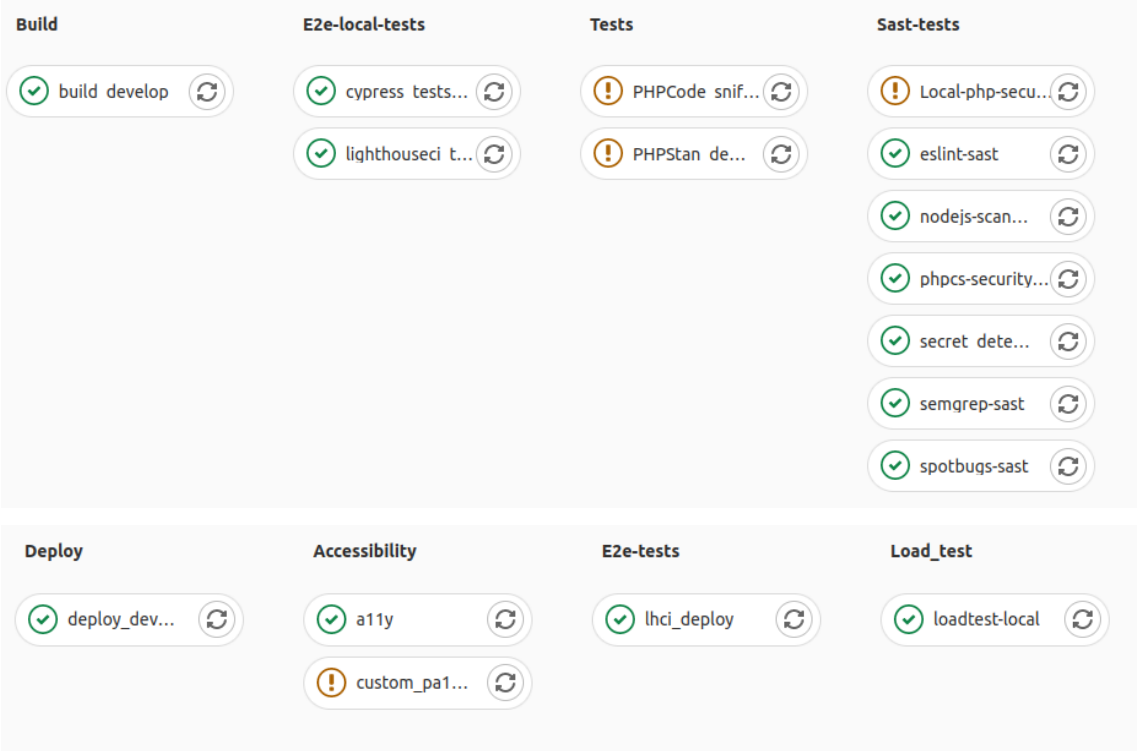


Figura IX - Pipeline de proyecto Marcotran en GitLab CI

Posteriormente se produce el *deploy* (este escenario ya venía incluido en la *pipeline* previamente), y a continuación se realizan los escenarios pertinentes de Accesibilidad, pruebas E2e y pruebas de carga sobre el entorno en el que se han desplegado.

El tiempo total del proceso de la *pipeline* es de una media de **30-32 minutos**. Un tiempo considerable, pero que puesto en comparación al ahorro que supone frente a pruebas manuales y cobertura de casos de uso, se considera asequible en el marco del proyecto en el que se trabaja. Los detalles adicionales referentes a la configuración y resultados que produce esta *pipeline* se localizan en el Anexo I.

Adicionalmente, se elaboró una documentación lo más completa posible para detallar cómo ha sido la configuración de cada escenario. Esta documentación se incluye en formato de README en *markdown* a la rama *develop* del proyecto, el cual es accesible en este documento en el ANEXO III.

5. Gestión, conclusiones y trabajo futuro

5.1. Gestión del proyecto

Este TFG se ha realizado dentro de la empresa Hiberus entre el 8 de febrero y el 23 de junio de 2021 siguiendo un horario de lunes a viernes de 08:00 a 15:00. Esto resulta en un total 665 horas dedicadas para la realización de este proyecto. Este horario incluye horas de reuniones con el tutor académico, así como las horas adicionales de trabajo, investigación y redacción que requería este TFG.

Toda la planificación se configura inicialmente mediante el cronograma orientativo disponible en la Figura XII, donde se estimaron las diferentes tareas tanto a alto nivel como ciertos detalles del proceso que se iba a seguir.

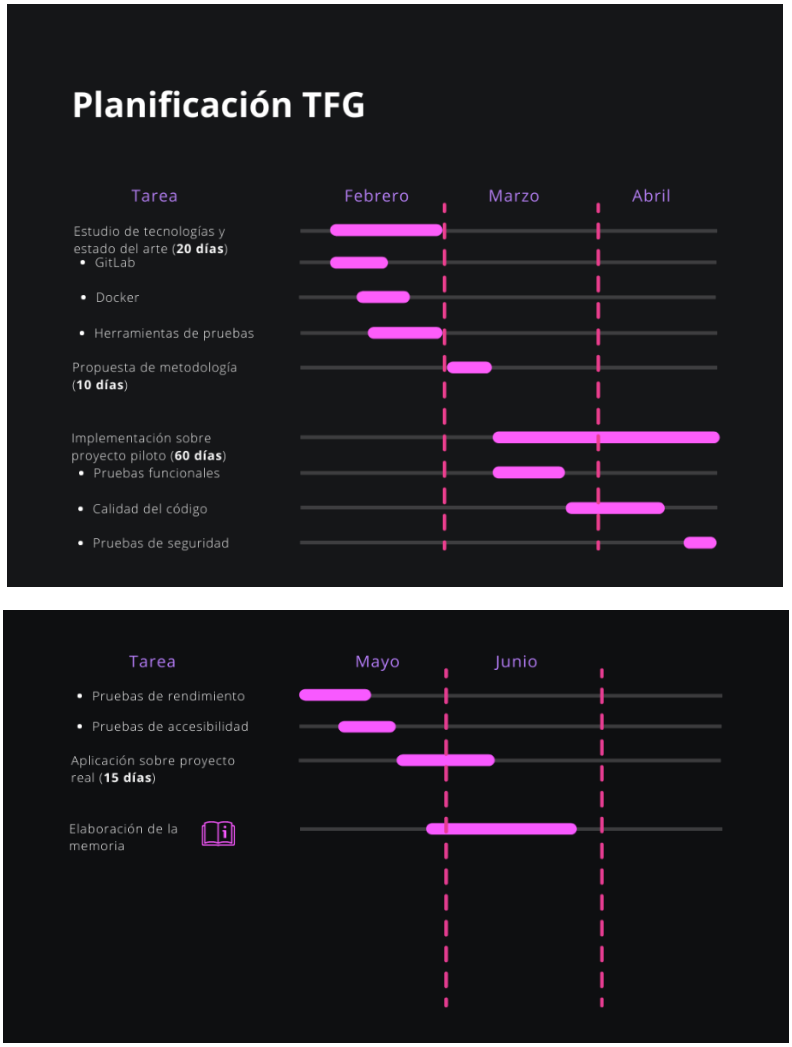


Figura X - Planificación a alto nivel del TFG

El resultado final de la organización del trabajo y tareas implementadas se puede consultar en el Anexo III. Ahí se puede observar cómo ciertas tareas de configuración al inicio del trabajo se demoraron más de lo previsto y han supuesto retrasos que se acumulan para el resto de tareas propuestas.

5.2. Conclusiones

El proyecto se puede presentar como una base genérica de metodología que puede utilizarse para cualquier proyecto de software en desarrollo que esté basado en integración continua sobre plataformas de versionado de código. En este caso se trabajó con un tipo de proyecto de Drupal en la plataforma de GitLab al ser un requisito impuesto por la empresa en la que se desarrolló este TFG.

Se considera que se han cumplido los objetivos iniciales propuestos, habiendo realizado un amplio estudio de las herramientas requeridas para completar e implementar el trabajo.

Con la aplicación de esta propuesta de metodología y mejora sobre el sistema de Integración Continua que tenía en desarrollo Hiberus, se demuestra que asegura la calidad del código, detección de vulnerabilidades y ofrece la estructura para asegurar la calidad del producto software antes y después de estar desplegado. Adicionalmente se observa que esta metodología mejora la productividad en el sentido de que supone un ahorro de tiempo a los desarrolladores para corregir errores y prevenirlos en un futuro, así como generar reportes que aseguran que se cumplen los estándares de códigos impuestos. Esta retroalimentación que produce la *pipeline* se puede traducir en una buena relación entre coste y beneficio para quienes desean implementar esta solución, puesto que el mantenimiento del servicio y de las herramientas propuestas es mínimo.

Desde un punto de vista personal este TFG ha supuesto profundizar tanto en conceptos teóricos como en tecnologías que no se habían tenido a penas experiencia de trabajo durante la carrera. Es el caso de la integración y entrega continua, que si bien es cierto se puso en práctica en muchas asignaturas de la rama, por contratiempos siempre se ha dedicado menos tiempo en el desarrollo a la automatización de estos procesos. En cuanto a las tecnologías ocurre algo similar con Docker, analizadores de código, automatización de pruebas sobre entornos desplegados.

Finalmente se valora muy positivamente la oportunidad de realizar el trabajo en un entorno empresarial real. Este entorno aporta una visión de cómo se realizan prácticas vistas en asignaturas de ingeniería del software que igual pasan desapercibidas o no se imagina que beneficios tienen en la industria.

5.3. Trabajo futuro

Aunque el trabajo se considera completado, existe margen de mejora sobre lo desarrollado. Durante el desarrollo aparecen herramientas que finalmente no se implementaron, pero que podrían mejorar la cobertura de calidad del producto y la monitorización de los servicios que intervienen en el sistema de integración continua. Algunas de estas herramientas serían SonarQube [31], para mejorar la calidad de código. Existen otras herramientas centradas en la monitorización de recursos como es Prometheus [32], que recogería métricas de los servicios de GitLab y que podrían ser visualizadas mediante Grafana. De esta manera se podría analizar el uso y detectar posibles mejoras en la infraestructura de integración continua. Adicionalmente, se podría considerar implementar esta metodología sobre otras plataformas como Jenkins o Travis si el entorno de trabajo fuera distinto a GitLab o se prefiriera externalizar a estas herramientas. Aunque supongan importantes diferencias en la implementación, los conceptos y herramientas que se han propuesto son fácilmente integrables en otras plataformas.

6. Bibliografía

1. Hiberus. Hiberus Tecnología, Expertos En Consultoría De Negocio y Tecnología. [Internet]. 2021. Available from: www.hiberus.com/

2. Drupal.org. Drupal [Internet]. 2021. Available from: www.drupal.org/about

3. webgeometrics. Drupal Basics and Workflow Architecture [Internet]. Available from: http://www.webgeometrics.com/drupal-basic-architecture/

4. Fowler M. Continuous Integration [Internet]. 2006. (martinfowler.com). Available from: https://martinfowler.com/articles/continuousIntegration.html

5. Béjar R, Zarazaga Soria FJ. PS-1-Diseño-Proyectos_4-Construcción-Automática.pdf. 2020.

6. Wikipedia. GitLab [Internet]. 2021. Available from: https://en.wikipedia.org/wiki/GitLab

7. Wikipedia. Open-core model [Internet]. 2021. Available from: https://en.wikipedia.org/w/index.php?title=Open-core_model&oldid=1025858222

8. ETSI (Instituto Europeo de Normas de Telecomunicaciones) Readme Ci Help [Internet]. Available from: https://forge.etsi.org/rep/help/ci/README.md

9. GitLab. Configuring runners in GitLab [Internet]. Available from: https://docs.gitlab.com/ee/ci/runners/

10. Static analysis tools for PHP [Internet]. Available from: https://github.com/exakat/php-static-analysis-tools

11. Squizlabs. PHP Code Sniffer GitHub [Internet]. Available from: https://github.com/squizlabs/PHP_CodeSniffer

12. Drupal.org. Coding standards [Internet]. 2020. Available from: https://www.drupal.org/docs/develop/standards/coding-standards

13. Mirtesová P. PHPStan: Find Bugs In Your Code Without Writing Tests! [Internet]. 2016. Available from: https://phpstan.org/blog/find-bugs-in-your-code-without-writing-tests

14. Matiukhin M. PHP static code analysis based on the example of PHPStan, Phan and Psalm [Internet]. 2018. Available from: https://medium.com/bumble-tech/php-code-static-analysis-based-on-the-example-of-phpstan-phan-and-psalm-a20654c4011d

15. Selenium HQ Browser Automation [Internet]. Available from: https://www.selenium.dev/

16. Cypress Architecture [Internet]. Available from: https://docs.cypress.io/guides/overview/key-differences#Architecture

17. Potencier F. SensioLabs Security Checker [Internet]. Available from: https://github.com/sensiolabs/security-checker

18. Potencier F. Local PHP Security Checker [Internet]. Available from: https://github.com/fabpot/local-php-security-checker

19. GitLab. Static Application Security Testing (SAST) [Internet]. Available from: https://docs.gitlab.com/ee/user/application_security/sast/index.html

20. GitLab. Secret Detection [Internet]. Available from: https://docs.gitlab.com/ee/user/application_security/secret_detection/index.html

21. Lucena Herrera C. Qué es una prueba de rendimiento de Software [Internet]. 2019. Available from: https://openwebinars.net/blog/que-es-prueba-de-rendimiento-software/?utm_source=youtube&utm_medium=descripcion&utm_campaign=que-es-prueba-rendimiento

22. van der Hoeven N. Comparing k6 and JMeter for load testing [Internet]. 2021. Available from: https://k6.io/blog/k6-vs-jmeter/

23. W3C. Accessibility [Internet]. Available from: https://www.w3.org/standards/webdesign/accessibility

24. W3C. How to Meet WCAG (Quickref Reference) [Internet]. Available from: https://www.w3.org/WAI/WCAG21/quickref/

25. GitLab. Accessibility testing [Internet]. Available from: https://docs.gitlab.com/ee/user/project/merge_requests/accessibility_testing.html

26. Goldenthal A. GitLab pa11y-ci [Internet]. Available from: https://gitlab.com/gitlab-ci-utils/gitlab-pa11y-ci

27. Pa11y Configuration file [Internet]. Available from: https://github.com/pa11y/pa11y#configuration

28. Majid H. How to use Gitlab CI, Pytest and docker-compose together [Internet]. 2020. Available from: https://dev.to/hmajid2301/how-to-use-gitlab-ci-pytest-and-docker-compose-together-1p4j

29. GitLab. Services [Internet]. Available from: https://docs.gitlab.com/ee/ci/services/

30. Hiberus. Drupal Development project [Internet]. Available from: https://github.com/hinternet/drupal-project

31. SonarQube. GitLab Integration [Internet]. Available from: https://docs.sonarqube.org/latest/analysis/gitlab-integration/

32. Prometheus. Available from: https://prometheus.io/

Anexos

ANEXO I - Configuraciones y resultados en GitLab CI

En este anexo se detallan indicaciones, si procede, sobre las herramientas y cómo se presentan los resultados

1. Configuración de PHP Code Sniffer

En la Figura XIII se puede observar un ejemplo de salida de PHP Code Sniffer aplicado sobre un módulo custom de Drupal para la aplicación de Marcotran.

```
----------------------------------------------------------------------
FILE: .../docroot/modules/custom/marcotran_core/marcotran_core.tokens.inc
----------------------------------------------------------------------
FOUND 10 ERRORS AND 1 WARNING AFFECTING 10 LINES
----------------------------------------------------------------------
 19 | WARNING | [ ] Unused variable $info.
 53 | ERROR   | [x] Namespaced classes/interfaces/traits should be
    |         |     referenced with use statements
 66 | ERROR   | [x] Case breaking statements must be followed by a
    |         |     single blank line
 69 | ERROR   | [x] Namespaced classes/interfaces/traits should be
    |         |     referenced with use statements
 70 | ERROR   | [x] Case breaking statements must be followed by a
    |         |     single blank line
 73 | ERROR   | [x] Namespaced classes/interfaces/traits should be
    |         |     referenced with use statements
 74 | ERROR   | [x] Case breaking statements must be followed by a
    |         |     single blank line
 77 | ERROR   | [x] Namespaced classes/interfaces/traits should be
    |         |     referenced with use statements
 78 | ERROR   | [x] Case breaking statements must be followed by a
    |         |     single blank line
 90 | ERROR   | [x] Concat operator must be surrounded by a single
    |         |     space
 90 | ERROR   | [x] Concat operator must be surrounded by a single
    |         |     space
----------------------------------------------------------------------
```

Figura XI - Ejemplo de salida PHP Code Sniffer

Estos errores en el análisis estático vienen provocados por la siguiente pieza de código. Claramente se puede observar lo referenciado en la Figura XIV entre las líneas 53 y 66.

```php
function marcotran_core_tokens($type, $tokens, array $data, array $options,
  $replacements = [];
  if ($type === 'marcotran') {
    $current_uid = \Drupal::service( id: 'current_user')->id();
    $user = \Drupal\user\Entity\User::load($current_uid);
    foreach ($tokens as $name => $original) {
      switch ($name) {
        case 'organization_chart':
          $config = \Drupal::service( id: 'config.factory')->get('marcotran_
          $organization_chart = $config->get("organization_chart.fid");
          $organization_chart_url = "";
          if (!empty($organization_chart) && $file = File::load($organizati
            $uri = $file->getFileUri();
            $organization_chart_url = Url::fromUri(file_create_url($uri))->
          }

          $replacements[$original] = $organization_chart_url;
          break;
```

Figura XII - Extracto de código del módulo custom marcotran_core_tokens

2. Configuración de PHPStan

En la Figura XIII se puede observar un ejemplo de salida de PHPStan con nivel de control de calidad **5** aplicado sobre todos los módulos custom de Drupal para la aplicación de Marcotran. Este nivel se puede editar mediante una variable en GitLab CI. Se observan recomendaciones y un número total de errores encontrados al final de la salida.

```
------ ------------------------------------------------------------------------
 Line   marcotran_ws/src/Ws/MarcotranWs.php
------ ------------------------------------------------------------------------
 98     Unsafe usage of new static().
        💡 See:
        https://phpstan.org/blog/solving-phpstan-error-unsafe-usage-of-new-st
        atic
 193    Variable $response might not be defined.
 310    Call to an undefined method Exception::getResponse().
------ ------------------------------------------------------------------------
------ ------------------------------------------------------------------------
 Line   marcotran_ws/src/Ws/WsMockHandler.php
------ ------------------------------------------------------------------------
 48     Parameter #1 $status of class GuzzleHttp\Psr7\Response constructor
        expects int, string given.
 405    Parameter #1 $min of function rand expects int, string given.
 405    Parameter #2 $max of function rand expects int, string given.
------ ------------------------------------------------------------------------
[ERROR] Found 197 errors
```

Figura XIII - Ejemplo de salida PHPStan

Estos errores en el análisis estático vienen provocados por la siguiente pieza de código en la Figura XVI. Dónde claramente se puede observar lo referenciado en las líneas 193. En este punto podría darse el caso de que, si no entra en ninguna de las condiciones anteriores del try y catch, la variable response no esté inicializada.

```
167
168       if ($type == "employee") {
169
170         $arguments = [
171           'email' => $email,
172           'document_number' => $document_number,
173         ];
174
175         $service = new ApiManager( service: "download_payroll", $arguments);
176         $response = $service->execute();
177
178         return $response;
179       }
180
181     }
182     catch (ClientException $e) {
183       $status = $e->getCode();
184       $response_exception = $e->getResponse();
185       $responseBodyAsString = $response_exception->getBody();
186       $response['status_response'] = $status;
187       $response += json_decode($responseBodyAsString,  assoc: TRUE);
188
189       $this->setLoggerMessage( ws: "Download documents", $response);
190
191     }
192
193     return $response;
194   }
```

Figura XIV - Extracto de código del módulo custom marcotran_ws

A continuación, se detalla un error destacable que se han encontrado durante la prueba de herramienta PHPStan en la *pipeline*. En la terminal del *job* de **PHPStan** se mostraba el siguiente error:

> RuntimeException thrown in /builds/drupal/projects/marcotran/drupal/vendor/mglaman/phpstan-drupal/src/Drupal/DrupalAutoloader.php on line 73 while loading bootstrap file /builds/drupal/projects/marcotran/drupal/vendor/mglaman/phpstan-drupal/drupal-autoloader.php: Unable to detect Drupal at

Este error sucede cuando se trabaja con una estructura de proyecto diferente a la estándar y Drupal Finder no es capaz de autodeterminar ni drupalRoot ni drupalVendorRoot en el fichero DrupalAutoloader.php
Para resolver este problema se decidió incluir de manera directa el valor de estas dos variables, resultando en el código de la Figura XVII:

```
//    $this->drupalRoot = $drupalRoot;
      $this->drupalRoot =
"/builds/drupal/projects/marcotran/drupal/docroot";

//    $this->autoloader = include $drupalVendorRoot . '/autoload.php';
      $this->autoloader = include
"/builds/drupal/projects/marcotran/drupal/vendor" . '/autoload.php';

//    $this->serviceYamls['core'] = $drupalRoot .
'/core/core.services.yml';
      $this->serviceYamls['core'] =
"/builds/drupal/projects/marcotran/drupal/docroot" .
'/core/core.services.yml';
      $this->serviceClassProviders['core'] =
'\Drupal\Core\CoreServiceProvider';
```

Figura XVII - Extracto de código de DrupalAutoloader.php en Marcotran

3. Configuración de Local PHP Security Checker

En la Figura XVIII se puede observar un ejemplo de salida del análisis de vulnerabilidades de seguridad aplicado sobre producción de Marcotran.

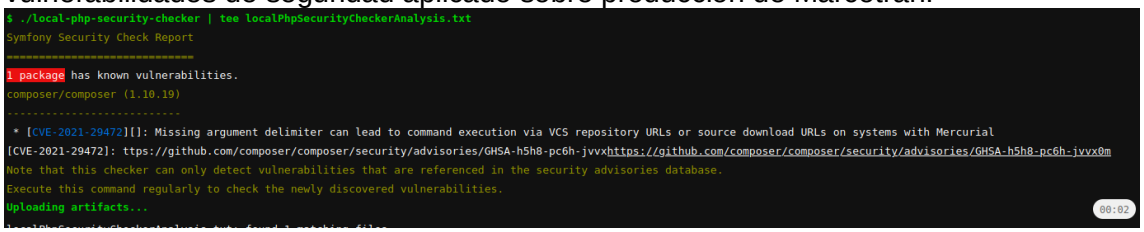


Figura XVI - Ejemplo de salida Local PHP Security Checker

En este ejemplo se detecta una posible vulnerabilidad detectada el 27 de abril por la cual se podría realizar inyección de comandos si se pasara a composer el input de un usuario, como detalla aquí
https://github.com/composer/composer/security/advisories/GHSA-h5h8-pc6h-jvvx

También se recomienda actualizar la dependencia a la versión de composer ya parcheada

4. Configuración de Cypress

Se realiza un script básico que ejecutará Cypress sobre el proyecto piloto. Tal como se puede observar en la Figura XIX, realiza las siguientes acciones:

- carga la ruta base
- comprueba que contiene la opción de “Contacto”, y en tal caso hace click en ella
- comprueba que la URL incluye la ruta /contact
- Finalmente, rellena si exista el campo con id edit-mail y comprueba su valor

```
describe('The Home Page', () => {
  it('successfully loads', () => {
    cy.visit('/')
    cy.screenshot("homepage")
    cy.contains('Contacto').click()

    cy.url().should('include', '/contact')
    cy.get('[id=edit-mail]')
        .type('fake@email.com')
        .should('have.value', 'fake@email.com')

    cy.screenshot("contact")
  })
})
```

Figura XVII - Ejemplo de prueba en JavaScript para Cypress (Proyecto piloto)

En las Figura XX y Figura XXI se pueden observar unos ejemplos de los artefactos (capturas de pantalla) que realiza Cypress sobre el proyecto piloto.

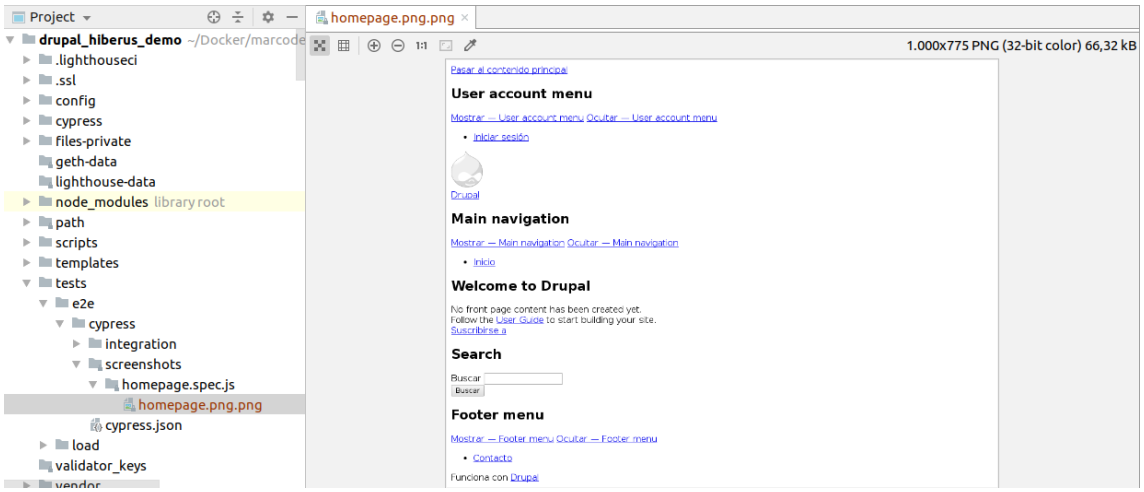


Figura XVIII - Captura de pantalla Proyecto piloto (vista inicial)

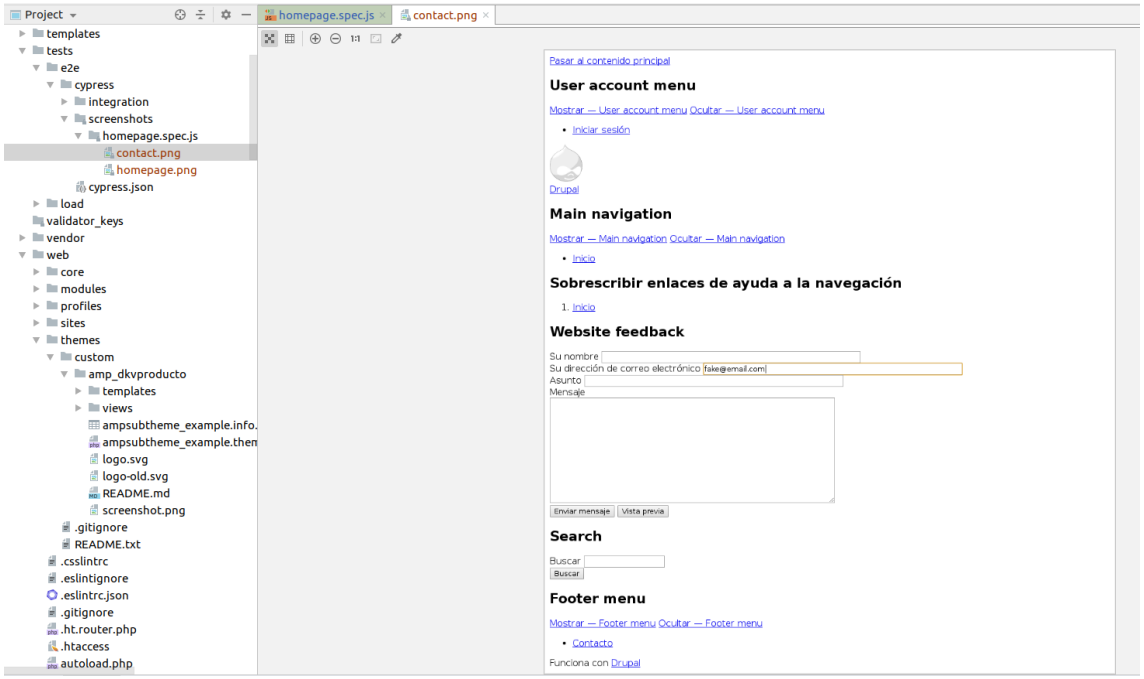


Figura XIXI - Captura de pantalla Proyecto piloto (vista contacto)

Para el caso de **Marcotran**, aunque la finalidad tampoco es el diseño de la prueba, se realiza el siguiente script (véase Figura XXII) para comprobar el correcto funcionamiento de la infraestructura de Cypress:

- carga la ruta base
- comprueba que contiene la opción de “Quiénes somos”, y en tal caso hace click en ella (la opción force es para que obvie los elementos de HTML en los que se encuentra anidado el *nav link* efectivo y así mejorar la legibilidad del test simplemente por su nombre de referencia)
- comprueba que la URL incluye la ruta /quienes-somos/sobre-marcotran
- comprueba que contiene la opción de “Solicita cotización”, y en tal caso hace click en ella
- comprueba que la URL incluye la ruta /solicita-cotización
- Finalmente, rellena si exista el campo de nombre de la compañía y comprueba su valor

```
describe( description: 'The Home Page', specDefinitions: () => {
  it( desc: 'successfully loads', func: () => {
    cy.visit('/')
    cy.screenshot("homepage")
    cy.contains('Quiénes somos').click({force: true})
    cy.url().should('include', '/quienes-somos/sobre-marcotran')
    cy.screenshot("quienesSomos")
    cy.contains('Solicita cotización').click({force: true})
    cy.url().should('include', '/solicita-cotizacion')

    cy.get('[id=edit-field-cot-company-0-value]')
        .type('fake company S.A')
        .should('have.value', 'fake company S.A')

    cy.screenshot("solicitaCotizacion")
  })
})
```

Figura XX - Ejemplo de prueba en JavaScript para Cypress (Proyecto Marcotran)

En la Figura XXIII se puede observar un ejemplo del artefacto (captura de pantalla) que realiza Cypress sobre la pantalla inicial de Marcotran. Cabe mencionar que realiza diversas capturas de pantalla mediante *scroll* de la página y las une de manera resulta en la pantalla completa indicada. También se debe destacar que las imágenes y vídeos que sí que se encuentran en producción aquí no se extraen para mejorar tiempos de respuesta y recursos, como se menciona en el apartado de puesta en marcha

Figura XXI - Captura de pantalla Proyecto Marcotran (vista inicial)

En la Figura XXIV se puede observar un ejemplo del artefacto (captura de pantalla) que realiza Cypress sobre la pantalla de solicitar cotización de Marcotran. (se puede observar cómo el campo de nombre de compañía ha sido rellenado con lo indicado)

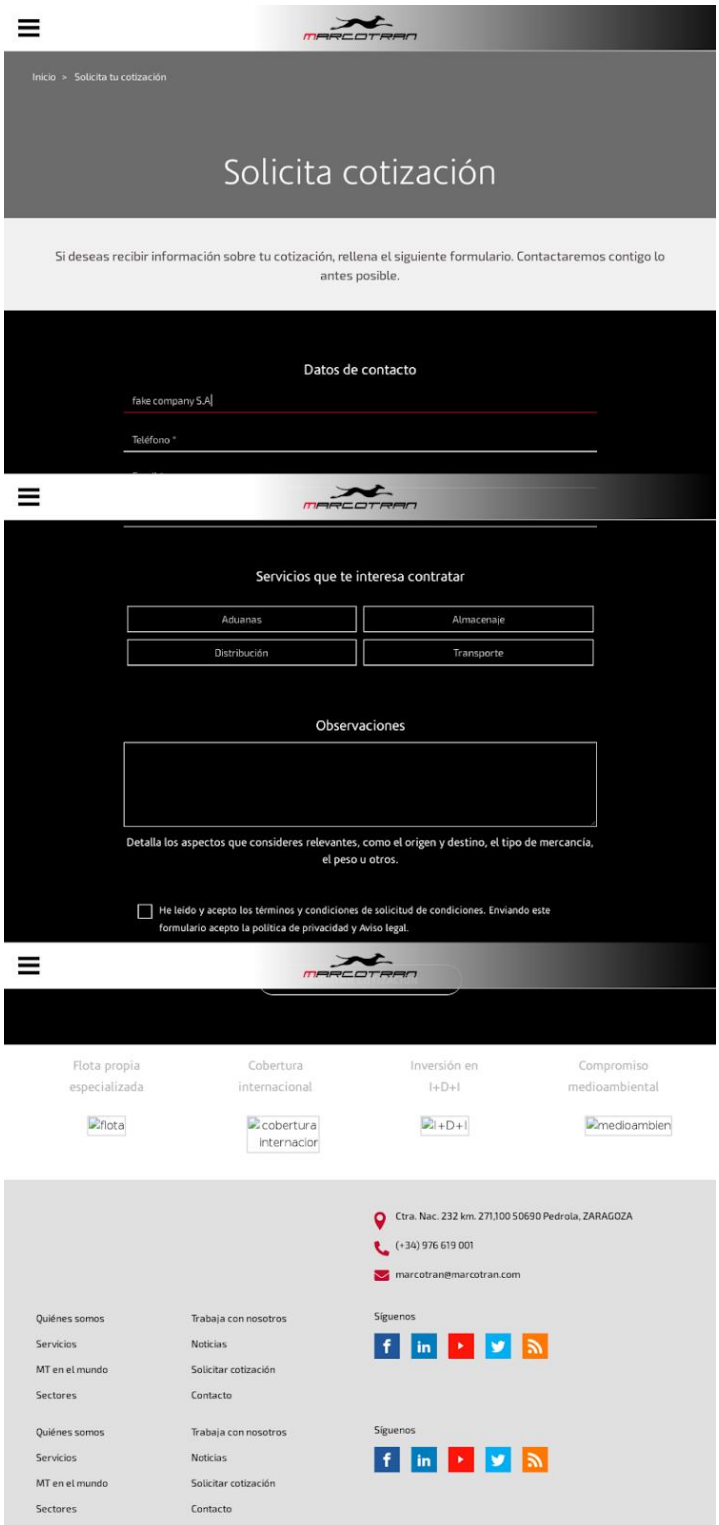


Figura XXII - Captura de pantalla Proyecto Marcotran (vista solicitar cotización)

En la Figura XXV se puede observar un ejemplo de salida en terminal de Cypress aplicado sobre entorno de Docker para la aplicación de Marcotran en GitLab CI.

```
Running: homepage.spec.js...                                              (1 of 1)
Estimated: 11 seconds


The Home Page

  ✓ successfully loads (32930ms)


1 passing (33s)


(Results)

┌──────────────────────────────┐
| Tests:        1              |
| Passing:      1              |
| Failing:      0              |
| Pending:      0              |
| Skipped:      0              |
| Screenshots:  3              |
| Video:        false          |
| Duration:     32 seconds     |
| Estimated:    11 seconds     |
| Spec Ran:     homepage.spec.js |
└──────────────────────────────┘


(Screenshots)

- /e2e/cypress/screenshots/homepage.spec.js/homepage.png (1000x3578)
- /e2e/cypress/screenshots/homepage.spec.js/quienesSomos.png (1000x4398)
- /e2e/cypress/screenshots/homepage.spec.js/solicitaCotizacion.png (1000x2140)


(Uploading Results)

- Done Uploading (1/3) /e2e/cypress/screenshots/homepage.spec.js/solicitaCotizacion.png
- Done Uploading (2/3) /e2e/cypress/screenshots/homepage.spec.js/homepage.png
- Done Uploading (3/3) /e2e/cypress/screenshots/homepage.spec.js/quienesSomos.png
```

Figura XXIII - Ejemplo de salida Cypress

5. Configuración de Lighthouse CI

Se configura el siguiente entorno de Lighthouse, donde indicamos la URL sobre la que se hará el reporte de rendimiento, así como que el destino dónde se almacena el resultado será temporal. Aquí podría configurarse una máquina a modo de servidor de almacenamiento de los reportes generados por Lighthouse.

Para el proyecto piloto, en fase de pruebas e2e locales sobre Docker, el fichero de configuración de Lighthouse se indica como en la Figura XXVI. Se indica explícitamente que esa URL base solamente pase por las categorías de *performance y accesibility* puesto que no representan webs reales y categorías como SEO son omisibles para aligerar la ejecución.

```
{
  "ci": {
    "collect": {
      "url": "http://apache",
      "settings": {
        "chromeFlags": "--no-sandbox",
        "onlyCategories": [
          "performance",
          "accessibility"
        ]
      }
    },
    "upload": {
      "target": "temporary-public-storage"
    }
  }
}
```

Figura XXIVI - Archivo de configuración de Lighthouse (lighthouserc.json)

En la Figura XXVII se puede observar un ejemplo de salida de Lighthouse Ci aplicado sobre la versión de preproducción del proyecto piloto. Claramente se aprecia que no se debe tener en cuenta las métricas de rendimiento en este punto de preproducción, ya que el entorno dispone de varios elementos que relegan la ejecución como son Docker, GitLab Ci y el propio lighthouse. Las métricas de accesibilidad no se ven lastradas por esto, ya que analizan el código y apartado visual (sin tener en cuenta tiempos de carga)

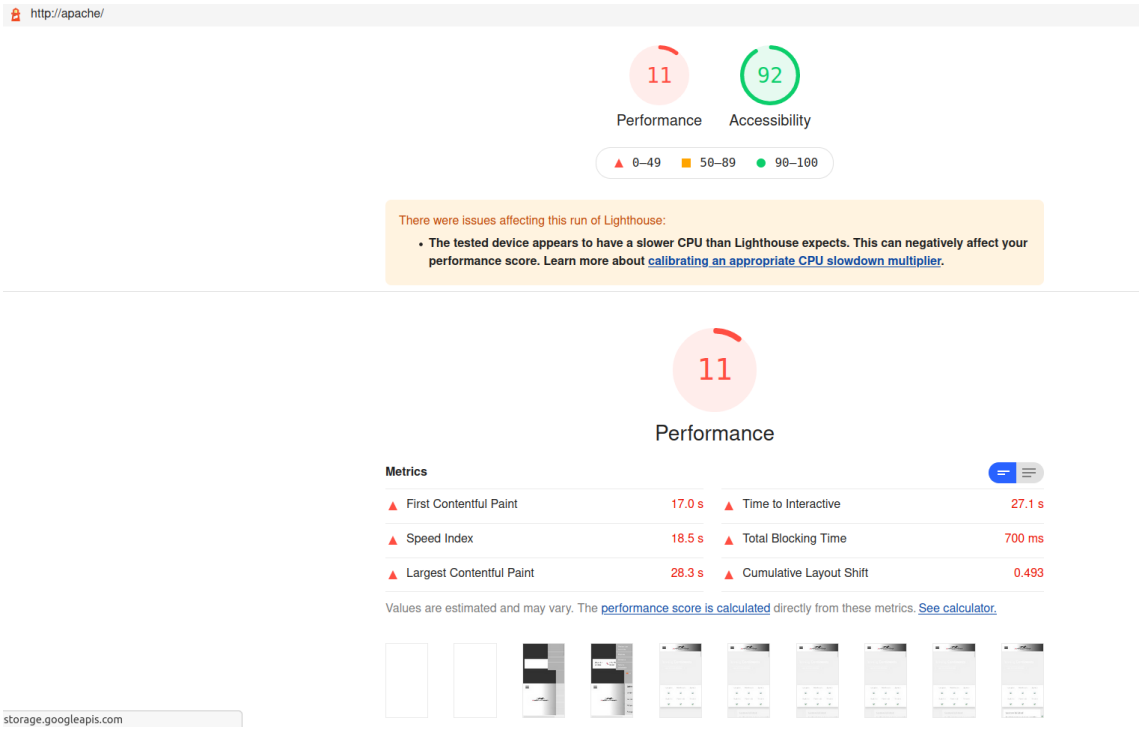


Figura XXV - Ejemplo de salida en formato web de Lighthouse

Será sobre ejemplos desplegados dónde sí que se deba tener en cuenta este resultado de performance. Por ejemplo, en el Proyecto real de Marcotran.

El proyecto de Marcotran cuenta con el siguiente fichero de configuración (lighthouserc_deploy.json) que se muestra en la Figura XXVIII.

```
{
  "ci": {
    "collect": {
      "url": "https://marcotran.com/",
      "settings": {
        "chromeFlags": "--no-sandbox",
        "skipAudits": ["full-page-screenshot"]
      }
    },
    "upload": {
      "target": "temporary-public-storage"
    }
  }
}
```

Figura XXVI - Archivo de configuración de Lighthouse sobre Marcotran

En la Figura XXIX se puede observar un ejemplo de salida de Lighthouse Ci aplicado sobre la versión de producción de Marcotran.

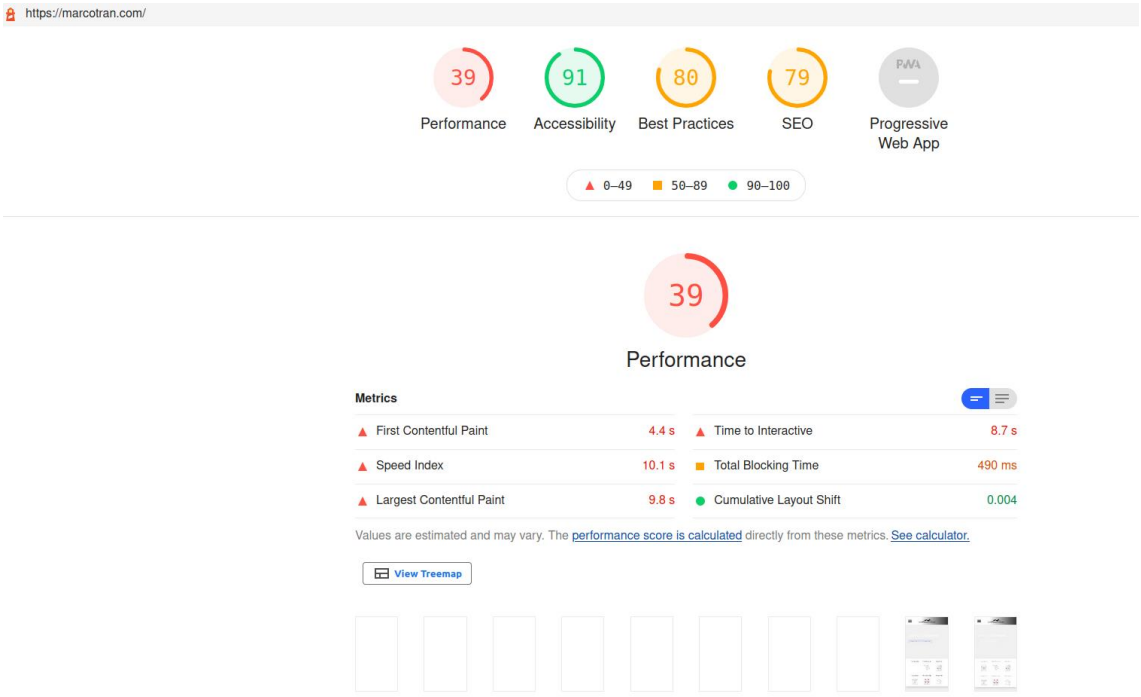


Figura XXVIIX - Ejemplo de salida en formato web de Lighthouse

A continuación, se detalla un error destacable que se han encontrado durante la prueba de herramienta Lighthouse en la *pipeline*. En el contenido de la terminal de **Lighthouse ci** se mostraba el siguiente error:

> Error: Unable to determine current hash with git rev-parse HEAD

Este error se ha resuelto incluyendo las siguientes variables referentes Lighthouse en el fichero **.env**:

> LIGHTHOUSE_VERSION=latest
> LHCI_BUILD_CONTEXT__CURRENT_BRANCH=$CI_COMMIT_REF_NAME
> LHCI_BUILD_CONTEXT__CURRENT_HASH=$CI_COMMIT_SHA
> GIT_BRANCH_NAME=master

6. Configuración de GitLab SAST, Secret detection y Accesibilidad

En este punto se muestran algunos de los posibles reportes que brindan las herramientas analizadoras de seguridad de GitLab SAST, Secret detection o Pa11y (que se utilizará para generar reportes de accesibilidad)

En la figura XXX se puede observar el siguiente fragmento del artefacto en JSON generado por una de las herramientas analizadoras, semgrep-sast.


```
 "id":
"ded348268d2db34a43df55061b695ce7542e005ebaf7b31f99fea017cadcc36e",
      "category": "sast",
      "name": "Found non-literal argument to RegExp Constructor",
      "message": "Found non-literal argument to RegExp Constructor",
      "description": "RegExp() called with a variable, this might allow
an attacker to DOS your application with a long-running regular
expression.",
      "cve":
"docroot/themes/custom/h_admin/js/jquery.min.js:e3b0c44298fc1c149afbf4c8
996fb92427ae41e4649b934ca495991b7852b855:security/detect-non-literal-reg
exp",
      "severity": "Medium",
      "confidence": "Unknown",
      "scanner": {
      "id": "eslint",
      "name": "ESLint"
      },
      "location": {
      "file": "docroot/themes/custom/h_admin/js/jquery.min.js",
      "start_line": 2,
      "end_line": 2
      },
```


Figura XXVIII - Ejemplo de salida de semgrep-sast

Para el análisis de accesibilidad, como se detalla en la sección 2.6, a parte del incluido por GitLab, se configura un análisis por medio de **Pa11y.** Se configura mediante el siguiente JSON (véase figura XXXI), en el fichero .pa11y.json y se rebaja la exigencia del análisis a WCAG2A, sobre la URL de producción de Marcotran

```
{
 "defaults": {
   "chromeLaunchConfig": {
     "args": [ "--no-sandbox" ]
   },
   "includeWarnings": true,
   "standard": "WCAG2A"
 },
 "urls": [
   {
     "url": "https://marcotran.com/"
   }
 ]
}
```

Figura XXIXI- Archivo de configuración de Pa11y sobre Marcotran

En la figura XXXII se puede observar el siguiente fragmento del artefacto HTML que genera.

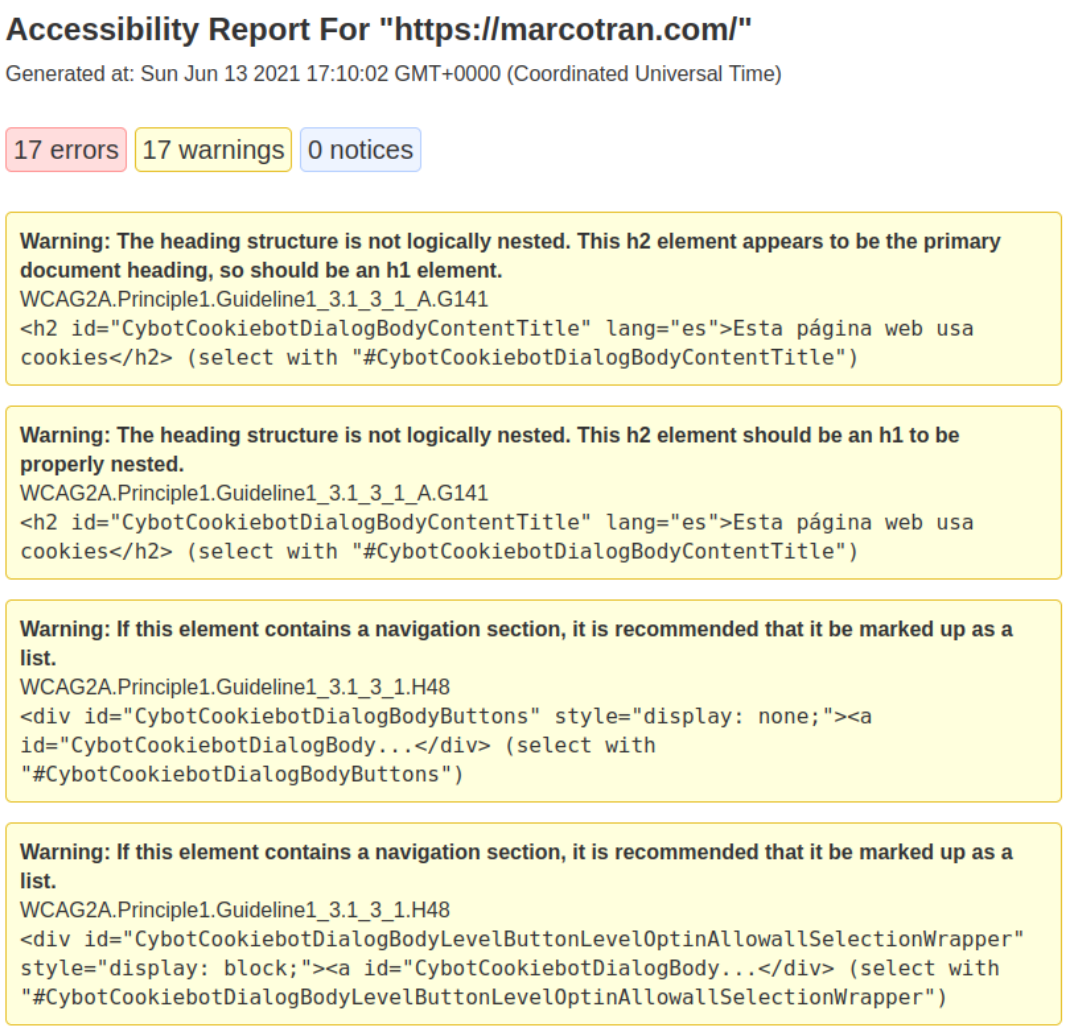


Figura XXX - Ejemplo de salida en formato web de Pa11y

7. Configuración de k6

Se ha realizado un script básico, y su configuración asociada, para ejecutar k6. Tal como se muestra en la figura XXXIII, este script se carga la ruta base y puede indicarse límites para los cuales se considera que falla la prueba de carga mediante thresholds.

```
import { sleep } from 'k6';
import http from 'k6/http';
export let options = {
    duration: '1m',
    vus: 10
    // thresholds: {
    //     http_req_duration: ['p(95)<500'], // 95 percent of response times must be below 500ms
    // },
};
export default function () {
    http.get('http://marcotran.des.demohiberus.com');
    sleep(3);
}
```

Figura XXXI - - Ejemplo de prueba en JavaScript para k6 (Proyecto Marcotran)

En la figura XXXIV se muestra la salida de la herramienta de prueba de rendimiento k6 aplicada sobre web en producción de Marcotran.

```
data_received..................: 9.1 MB 145 kB/s
data_sent......................: 14 kB  226 B/s
http_req_blocked...............: avg=315.97µs min=3.57µs   med=4.99µs  max=30.38ms p(90)=10.64µs  p(95)=1.58ms
http_req_connecting............: avg=236µs    min=0s       med=0s      max=28.93ms p(90)=0s       p(95)=535.42µs
http_req_duration..............: avg=1.18s    min=55.14ms  med=78.06ms max=15.96s  p(90)=731.02ms p(95)=15.94s
  { expected_response:true }...: avg=1.18s    min=55.14ms  med=78.06ms max=15.96s  p(90)=731.02ms p(95)=15.94s
http_req_failed................: 0.00%  ✓ 0    ✗ 150
http_req_receiving.............: avg=1.29ms   min=985.82µs med=1.19ms  max=3.17ms  p(90)=1.45ms   p(95)=1.86ms
http_req_sending...............: avg=1.74ms   min=15.04µs  med=23.42µs max=28.8ms  p(90)=63.99µs  p(95)=28.15ms
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s      p(90)=0s       p(95)=0s
http_req_waiting...............: avg=1.18s    min=54.07ms  med=76.72ms max=15.93s  p(90)=729.24ms p(95)=15.91s
http_reqs......................: 150    2.384856/s
iteration_duration.............: avg=4.18s    min=3.05s    med=3.07s   max=18.96s  p(90)=3.73s    p(95)=18.94s
iterations.....................: 150    2.384856/s
vus............................: 10     min=10 max=10
vus_max........................: 10     min=10 max=10
```

Figura XXXIIV - Ejemplo de salida de k6

ANEXO II – Documentación en inglés para el proyecto Marcotran

En este anexo se incluye la impresión del archivo README.md que se ha facilitado dentro del repositorio de Marcotran para documentar como se aplica la metodología de entrega continua para este proyecto sobre GitLab. La impresión es idéntica a la que se muestra en el repositorio de GitLab, salvo la sección "Table of content" que no se genera correctamente al realizar la impresión para incluirse en este anexo. En este archivo README.md se incluyen también las incidencias típicas provocadas por las herramientas y cómo solucionarlas.

Marcotran

Gitlab Pipeline (develop)

Features

- Static Analysis tool for PHP
  - Support for PHP Code Sniffer.
  - Support for PHPStan.
- Static Application Security Testing (**SAST**)
  - Support for local-php-security-checker
  - Gitlab SAST templates
  - Gitlab Secret Analyzer
  - Gitlab Accesibility Testing
  - Custom pa11y accessibility testing
- e2e Integration tests
  - Support for Cypress
  - Support for Lighthouse CI
- Load testing
  - K6

Table of Contents

[[*TOC*]]

Pipeline initial configuration

We define the following pipeline stages: - build - e2e-local-tests - tests - sast-tests - accesibility - e2e-tests - load_tests

Global .gitlab-ci.yml variables:

`COMPOSER_HOME: "/home/wodby/.composer"` - getComposer environment variables doc

Cache strategy

This is the cache strategy for the build job. We want to cache the following directories between jobs:

```
.cache_strategy_push:
  cache:
    paths:
      - vendor
      - web/core
      - web/libraries
      - web/modules/contrib
      - web/profiles/contrib
      - web/themes/contrib
      - drush/Commands
      - ./.lighthouseci/
```

To consume this cache we define a cache strategy pull the extends from the previous one:

```
# The cache strategy for the test jobs.
.cache_strategy_pull:
  extends: .cache_strategy_push
  cache:
    policy: pull
```

Test template

The next gitlab ci template extends from the cache strategy pull, defines the container image, services, dependencies and will be used by the PHP Code Sniffer, PHPStan and local-php-security-checker jobs.

```
.drupal8ci_test_template:
  extends: .cache_strategy_pull
  image: "wodby/drupal-php:7.4-dev-4.21.6"
  stage: test
  services:
    - name: mariadb:latest
      alias: db
  dependencies:
    - drupal8ci:build
```

Build stage

The build stage currently installs composer dependencies following a cache_strategy_push. First it uses `docker-compose up -d` to start all containers defined in *docker-compose.yml* After they are all up and running it installs composer dependencies via the just built php bash through docker:

```
    docker-compose exec -T php bash -c "composer install --no-interaction --no-progress"
```

And sleeps for a few seconds so the database is available.

e2e local tests stage and dump of the remote database

This stage makes use of both *docker-compose.yml* and *docker-compose.tests.yml* to build and run integration tests to the local environment throuhg the use of cypress and lightouseci, all traffic is redirected by traefik between Apache, Adminer, PHPMyAdmin, Portainer.

For this e2e local tests to work we need to dump the remote database for them to use it and deploy a local version of the project. To achieve this we dump everything from des-drupal3 into an .mysql file through SSH and store it temporarily in the local environment like:

```
# database dump
    - $SSH_LOGIN -T "$GIT_ROOT/vendor/bin/drush --root=$CI_DEVELOP_PATH sql:dump --result-file=/tmp/backupMarco.mysql --gzip" #--str
    - scp $SSH_OPTIONS $SSH_LOGIN_USER@$SSH_TARGET_SERVER:/tmp/$CI_DEPLOY_DATABASE_NAME $CI_PROJECT_DIR/
    - gunzip -f $CI_PROJECT_DIR/$CI_DEPLOY_DATABASE_NAME
```

Where each referenced local variable represents:

```
    GIT_ROOT: "/var/www/marcotran_des/"
    CI_DEPLOY_DATABASE_NAME: 'backupMarco.mysql.gz'
    SSH_LOGIN_USER: "root"
    SSH_TARGET_SERVER: "des-drupal3.hiberus.local"
    SSH_OPTIONS: "-oStrictHostKeyChecking=no"
```

After this process is completed we need to stablish the database stored in the .mysql file into the running php container like:

```
    - docker-compose exec -T php bash -c "drush sql:query  --file=/var/www/html/backupMarco.mysql" #-- builds database from backup f
    - docker-compose exec -T php bash -c "drush core:status"
```

And finally we can run the docker tests images (cypress and lighthouseci) using the proper database:

```
    - docker-compose -f docker-compose.yml -f docker-compose.tests.yml up -d
    - docker logs -f marcotran_lhci
    - docker logs -f marcotran_cypress
```

Tests stage

Tests stage make use of the docker deployed images set up of PHP Code Sniffer and PHPStan

Set up PHP Code Sniffer

For a local configuration we follow the official Drupal Documentation

After phpcs installation and Drupal /Drupal Practice standards registration we can verify the registered standards:

```
phpcs -i
```

You can test it in your local project with the following command:

```
phpcs --standard=Drupal,DrupalPractice web/core/modules/field/src/
```

For Gitlab CI/CD Pipeline first we define this job in the *.gitlab-ci.yml*:

```
drupal8ci:code_sniffer:
  extends: .drupal8ci_test_template
  stage: test
  services:
    - docker:dind
  before_script:
    - echo "estoy en before_script de code sniffer"
    - cd $CI_PROJECT_DIR
    - vendor/bin/composer global show -P
    - vendor/bin/phpcs --config-set installed_paths vendor/drupal/coder/coder_sniffer
    - >
      if [ ! -d "vendor/drupal/coder/coder_sniffer/Drupal" ]; then
        echo "Instalando Php Code Sniffer - Drupal standards"
        vendor/bin/composer global require drupal/coder:^8.3.1
        vendor/bin/composer require --dev dealerdirect/phpcodesniffer-composer-installer
        vendor/bin/phpcs -i
      fi
    - mkdir artifacts
    - mkdir artifacts/phpcs
  script:
    - cd $CI_PROJECT_DIR
    - echo "start running php code sniffer on custom modules"
    - vendor/bin/phpcs -v --standard=Drupal,DrupalPractice web/modules/custom/ --report=full --report-file=artifacts/phpcs/phpcs_mod
    - echo "start running php code sniffer on themes"
    - vendor/bin/phpcs -v --standard=Drupal,DrupalPractice web/themes/ --report=full --report-file=artifacts/phpcs/phpcs_themes.txt
  artifacts:
      when: always
      paths:
        - artifacts/phpcs/phpcs_modules.txt
        - artifacts/phpcs/phpcs_themes.txt
  allow_failure: true
```

Set up PHPStan

For a local configuration we follow the official PHPStan User Guide You can test it in your local project with the following command: `phpstan analyse web/core/modules/field/src/`

For PHPStan to check the Drupal standard properly it is needed to include the following dependencies in the *composer.json*:

```
    "require":{ ...
           "phpstan/phpstan-deprecation-rules": "^0.12.6",
           "phpstan/phpstan": "^0.12.85",
           "phpstan/extension-installer": "^1.1",
           "mglaman/phpstan-drupal": "^0.12.9",
           "phpstan/phpstan-phpunit": "^0.12.18",
           "drupal/upgrade_status": "^3.1"
           }
   "require-dev": {
               "dealerdirect/phpcodesniffer-composer-installer": "^0.7.1",
               "phpunit/phpunit": "^7",
               "drupal/core-dev": "8.9.13"
```

For Gitlab CI/CD Pipeline first we define this job in the *.gitlab-ci.yml*:

```
rupal8ci:phpStan:
  extends: .drupal8ci_test_template
  stage: test
  before_script:
    - >
      if [ ! -d "vendor/phpstan/phpstan" ]; then
        echo "Instalando PhpStan"
        vendor/bin/composer global require phpstan/phpstan
      fi
  script:
    - mkdir artifacts
    - mkdir artifacts/phpstan
    - vendor/bin/phpstan analyse web/modules/custom/ --error-format=table | tee artifacts/phpstan/PhpStanError.txt
    - cat PhpStanError.txt
  artifacts:
    when: always
    paths:
      - artifacts/phpstan/PhpStanError.txt
  allow_failure: true
```

Additionally PHPStan uses configuration format called NEON. A config file can be passed to the phpstan executable using the -c or –configuration option:

`vendor/bin/phpstan analyse -c phpstan.neon`

You can config it indicating level of analysis and specific path to be analyze like:

```
parameters:
    level: 6
    paths:
        - src
        - tests
```

For more information please refer to PHPStan Config Reference

Finally a piece of advice: To clear the current state of the result cache, for example if you're developing custom extensions and the result cache is getting stale too often, you can run the clear-result-cache:

`vendor/bin/phpstan clear-result-cache [options]`

As PHPStan by default caches the result if the source code has not changed. For more information please refer to PHPStan User Guide - Result Cache

Set up Cypress.io

We configure the cypress container inside *docker-compose.tests.yml* :

```
  cypress:
    image: cypress/included:$CYPRESS_TAG
    container_name: "${PROJECT_NAME}_cypress"
    depends_on:
      - $WEBSERVER_SERVICE
    environment:
      CYPRESS_baseUrl: $SIMPLETEST_BASE_URL
    working_dir: /e2e
    volumes:
      - ./tests/e2e/cypress/cypress.json:/e2e/cypress.json
      - ./tests/e2e/cypress:/e2e/cypress
    entrypoint:
      - cypress
      #- "--no-exit"
      - "run"
      - "--record"
      - "--key"
      - "6a70474c-def8-466a-a7a3-a4750ddb70a2"
```

Where $SIMPLETEST_BASE_URL is defined in the *.env* file. Interesting cypress environment variables:

> CYPRESS_TAG=3.2.0 WEBSERVER_SERVICE=apache SIMPLETEST_BASE_URL=http://apache SIMPLETEST_DB=mysql://drupal:drupal@mariadb/drupal
> BROWSERTEST_OUTPUT_DIRECTORY=/var/www/html/web/sites/simpletest/browser_output

Right now the cypress integration tests should be in the /tests/e2e/cypress , where the *cypress.json* config file and *integration* folder will be defined. Inside the integration tests right now there is a single demo test called *homepage.spec.js*

```
// tests/e2e/cypress/integration/homepage.spec.js
describe('The Home Page', () => {
  it('successfully loads', () => {
    cy.visit('/')
  })
})
```

We follow the official Cypress Doc to setu up a project record and connect it to cypress dashboard. After setting up the project to record we have to insert **projectId** into the *cypress.json* file, which now will look like this:

```
{
  "projectId": "6 character string",
  "baseUrl": "baseUrl considered for testing",
  "pluginsFile": false,
  "supportFile": false,
  "fixturesFolder": false,
  "video": false,
  "pageLoadTimeout": 30000
}
```

At this moment we hardcode the key to reference our cypress dashboard where records will be stored as well.

To check the result in the Gitlab CI/CD Pipeline:

> docker-compose -f docker-compose.tests.yml up -d docker logs -f drupal_demo_cypress

Set up Lighthouse CI

We configure the lighthouse ci container inside *docker-compose.tests.yml* :

```
lhci:
    image: pixboost/lighthouse-ci-cli:1.0.0-0.3.7
    container_name: "${PROJECT_NAME}_lhci"
    env_file:
      - .env
#    command: sh -c "npm install && npm install --save-dev install-chrome-dependencies && npm install -g @lhci/cli@0.7.x"
    depends_on:
      - $WEBSERVER_SERVICE
    volumes:
      - ./lighthouserc.json:/home/lhci/lighthouserc.json
      - ./.git:/home/lhci/.git
      - ./.lighthouseci:/home/lhci/.lighthouseci
    entrypoint:
      - lhci
      - "autorun"
```

It is neccesary to create a .lighthouseci folder in the project root directory and with write permissions so the docker container can map it and upload the results.

The *lighthouserc.json* config file in the root directory currently looks like this:

```
{
    "ci": {
        "collect": {
            "url": "http://apache",
            "settings": {
                "chromeFlags": "--no-sandbox",
                "preset": "desktop"
            }
        },
        "upload": {
            "target": "temporary-public-storage"
        }
    }
}
```

preset may take the values desktop to specify that the analysis runs on a desktop emulation evironment (default value is mobile)

Similar to cypress it can be executed via `docker-compose` :

```
docker-compose -f docker-compose.tests.yml up -d docker logs -f lhci
```

For each run lhci generates two type of files in the .lighthouseci/: - lhr-1617088238124.html - equivalent to the temporary public storage site. - lhr-1617088238124.json - all information in JSON type.

Finally add .gitignote inside .lighthouseci/ so it does not upload the content to git:

```
#.lighthouseci/.gitignore
# Ignore everything in this directory
*
# Except this file
!.gitignore
```

Set up local-php-security-checker

The Local PHP Security Checker is a command line tool that checks if your PHP application depends on PHP packages with known security vulnerabilities. It uses the Security Advisories Database behind the scenes.

Download a binary from the Releases page on Github, rename it to local-php-security-checker and make it executable.

For Gitlab CI/CD Pipeline first we define this job in the *.gitlab-ci.yml*:

```
drupal8ci:local-php-security-checker:
  extends: .drupal8ci_test_template
  stage: test
  script:
    - curl -L -sS --output local-php-security-checker https://github.com/fabpot/local-php-security-checker/releases/download/v1.0.0/
    - chmod +x ./local-php-security-checker
    - ./local-php-security-checker | tee localPhpSecurityCheckerAnalysis.txt
#    - vendor/bin/psecio-parse scan -v --format=json $CI_PROJECT_DIR > parseAnalysis.json
  artifacts:
    when: always
    paths:
      - localPhpSecurityCheckerAnalysis.txt
```

We define such artifact so the output can be downloadable through gitlab.

Set up Gitlab SAST, Secret detection and Accessibility

For Gitlab CI/CD Pipeline first we define this include template in the *.gitlab-ci.yml* and override the sast and .secrete-analyzer to specify the stage :

```
include:
 - template: Security/SAST.gitlab-ci.yml
 - template: Security/Secret-Detection.gitlab-ci.yml
 - template: Verify/Accessibility.gitlab-ci.yml
sast:
 stage: sast-tests
.secret-analyzer:
 stage: sast-tests
```

It is also needed for the accessibility job to define a `a11y_urls: "url to be analyzed"` variable.

Set up pa11y, custom Accessibility

This custom use of pa11y difference of the GitLab included accesibility tests in that a custom *.pa11yci.json* file can be included so pa11y tests can follow this configuration For Gitlab CI/CD Pipeline first we define this job in the *.gitlab-ci.yml*:

```
custom_pa11y_ci:
  image: registry.gitlab.com/gitlab-ci-utils/gitlab-pa11y-ci:latest
  stage: accessibility
  dependencies: []
  before_script:
    - mkdir pa11y
  script:
    # Expect this test to fail since it's scanning sites with known a11y issues.
    # So, this check the results and fails as appropriate.  If pa11y-ci passes
    # then exit 2. If it fails, then check the exit code ($?). Exit code 1 is a
    # pa11y-ci execution error, so if that occurs exit 1. If exit code 2
    # (accessibility error, which is expected), then exit 0.
    - pa11y-ci --json > pa11y/pa11y-ci-results.json
  after_script:
    - pa11y-ci-reporter-html -s pa11y/pa11y-ci-results.json -d pa11y/
  artifacts:
    when: always
    paths:
      - pa11y/
    reports:
      accessibility: pa11y/pa11y-ci-results.json
    expire_in: 1 week
  allow_failure: true
```

The config file in this case looks like:

```
{
  "defaults": {
    "chromeLaunchConfig": {
      "args": [ "--no-sandbox" ]
    },
    "includeWarnings": true,
    "standard": "WCAG2A"
  },
  "urls": [
    {
      "url": "https://marcotran.com/"
    }
  ]
}
```

Following the WCAG2A instead of WCAG2A by default (if standard not specified). This standard can be customized for each url in the case we are able to omit certain rules for specific urls.

This job also generates a full html report to benefit human readability.

Set up k6, Load test

k6 allows easy configuration through docker image in the pipeline. Allowing to carry out load tests and indicating, for example, the number of users who will attack the application and define criteria for the fulfillment of this test.

The possibility of establishing thresholds at the test level is valued as a key and there are several customization possibilities. Such as: - x% of the time the response time must be less than the indicated "y" milliseconds. - x% of failed requests or y% of successful requests - etc

For Gitlab CI/CD Pipeline first we define this job in the *.gitlab-ci.yml*:

```
loadtest-local:
  image:
    name: loadimpact/k6:latest
    entrypoint: ['']
  stage: load_test
  before_script:
    - ''
  script:
    - echo "executing local k6 in k6 container..."
    - k6 run ./tests/load/performance-test.js
  allow_failure: true
```

The JavaScript test file looks like:

```
import { sleep } from 'k6';
import http from 'k6/http';
export let options = {
    duration: '1m',
    vus: 10
    // thresholds: {
    //     http_req_duration: ['p(95)<500'], // 95 percent of response times must be below 500ms
    // },
};
export default function () {
    http.get('http://marcotran.des.demohiberus.com');
    sleep(3);
}
```

This job shows the metrics report in the actual GitLab pipeline bash.

Common errors

In the **build pipeline**:

> $ vendor/bin/composer install DrupalProject\composer\ScriptHandler::checkComposerVersion Loading composer repositories with package information Installing dependencies (including require-dev) from lock file Your requirements could not be resolved to an installable set of packages. Problem 1 - league/container 2.4.1 requires php ^5.4.0 || ^7.0 -> your PHP version (8.0.2) does not satisfy that requirement. - league/container 2.4.1 requires php ^5.4.0 || ^7.0 -> your PHP version (8.0.2) does not satisfy that requirement. - Installation request for league/container 2.4.1 -> satisfiable by league/container[2.4.1]. `

- Solved by editing *composer.json* dependency to:

```
"require": {
       "php": ">=7.0.8",
```

In the **lhci container**:

> Error: Unable to determine current hash with `git rev-parse HEAD`

- Solved by including the following *.env* variables > LIGHTHOUSE_VERSION=latest LHCI_BUILD_CONTEXT__CURRENT_BRANCH=$CI_COMMIT_REF_NAME LHCI_BUILD_CONTEXT__CURRENT_HASH=$CI_COMMIT_SHA GIT_BRANCH_NAME=master

composer.json and composer.lock differ in gitlab - Solved by updating the composer.lock in the local container like:

> docker-compose exec php bash composer update

```
Commit and push the changes
```

In the PHPStan job:

> RuntimeException thrown in /builds/drupal/projects/marcotran/drupal/vendor/mglaman/phpstan-drupal/src/Drupal/DrupalAutoloader.php on line 73 while loading bootstrap file /builds/drupal/projects/marcotran/drupal/vendor/mglaman/phpstan-drupal/drupal-autoloader.php: Unable to detect Drupal at

- This error occurs when it is a special project structure and Drupal Finder can not autodetect neither drupalRoot nor drupalVendorRoot in *DrupalAutoloader.php*
- To solve this, one solution is to hardcode this two variables with the proper directories. In this case the result is:

```
//      $this->drupalRoot = $drupalRoot;
        $this->drupalRoot = "/builds/drupal/projects/marcotran/drupal/docroot";

//      $this->autoloader = include $drupalVendorRoot . '/autoload.php';
        $this->autoloader = include "/builds/drupal/projects/marcotran/drupal/vendor" . '/autoload.php';

//      $this->serviceYamls['core'] = $drupalRoot . '/core/core.services.yml';
        $this->serviceYamls['core'] = "/builds/drupal/projects/marcotran/drupal/docroot" . '/core/core.services.yml';
        $this->serviceClassProviders['core'] = '\Drupal\Core\CoreServiceProvider';
```

Drupal Development project

Project scaffold base to start developing Drupal projects, modules and even core. The project scaffold sits on top Wodby's Docker4Drupal stack and Drupal Composer project.

Starting

Before starting the project copy de provided *env* file a *.env* and edit and change to meet your needs. Also if you have cloned this repository **delete** the .git folder on the project root to avoid targeting scaffold repository.

> **IMPORTANT!** MacOS users pay attention to the PHP tag since use a different image than Linux (default).
>
> **DON'T FORGET** To add the configured project URL in your OS hosts file.

Usage

Most common tasks are explained below, also for each task a GNU/Make target is provided.

Make	Docker	Task
make [up:start]	docker-compose pull	Start stack
	docker-compose up -d --remove-orphans	
make [down:stop]	docker-compose stop	Stop stack
make shell	docker-compose exec php sh	Enter PHP container
make composer "COMMAND [PARAMS]"	docker-compose exec php "composer COMMAND [PARAMS] "	Execute Composer command
make drush "COMMAND [PARAMS]"	docker-compose exec php "drush COMMAND [PARAMS] "	Execute Drush command
make logs [SERVICE]	docker-compose logs [SERVICE]	Check service logs
make prune [SERVICE]	docker-compose down -v [SERVICE]	Remove service (deleted all)

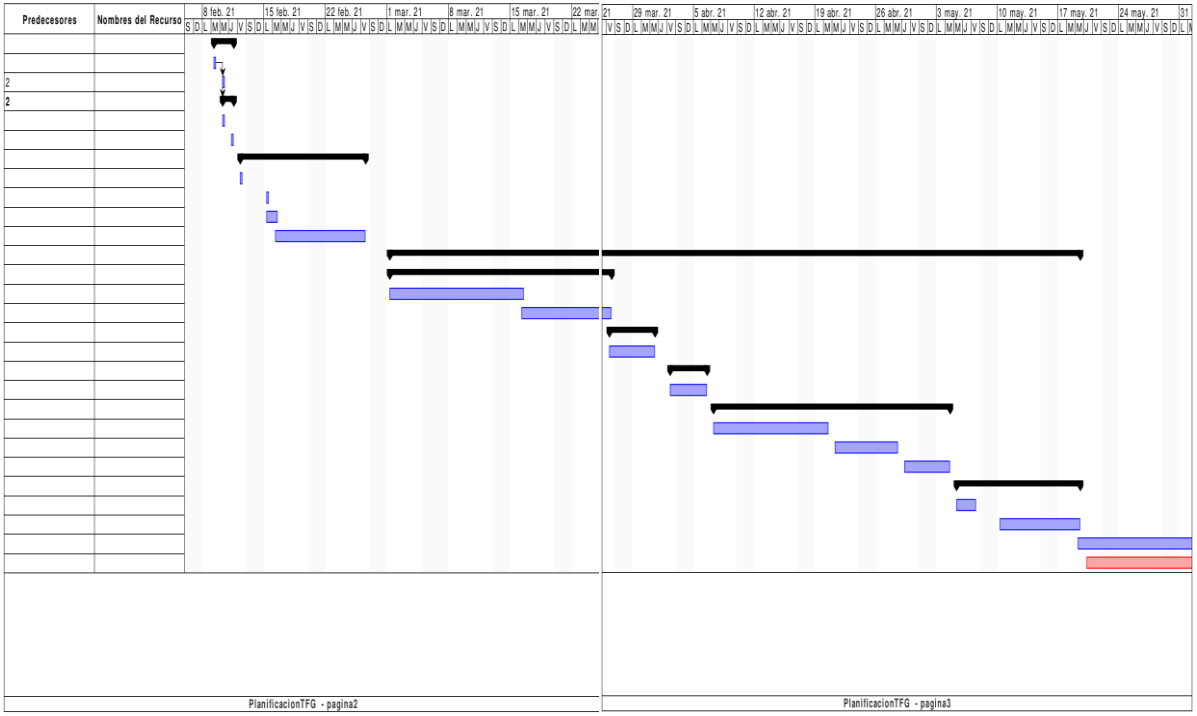
Further information is detailed here and here.

ANEXO III – Resultado de la organización del trabajo

Para documentar finalmente las tareas que se iban realizando conforme el desarrollo del proyecto, se decidió utilizar la herramienta ProjectLibre. Resultando en el siguiente cronograma.

	Nombre	Duracion	Inicio	Terminado
1	**Configurar Proyecto Drupal Local**	**3 days?**	**9/2/21 8:00**	**11/2/21 15:00**
2	Instalar Docker y drupal	1 day?	9/2/21 8:00	9/2/21 15:00
3	Instalar PHP Code Sniffer	1 day?	10/2/21 8:00	10/2/21 15:00
4	**Instalar y configurar PHPStan**	**2 days?**	**10/2/21 8:00**	**11/2/21 15:00**
5	Instalar PHP Stan	1 day?	10/2/21 8:00	10/2/21 15:00
6	Formatear salida PhpStan	1 day?	11/2/21 8:00	11/2/21 15:00
7	**Configurar Gitlab**	**11 days?**	**12/2/21 8:00**	**26/2/21 15:00**
8	Creación cuenta Gitlabdes	1 day?	12/2/21 8:00	12/2/21 15:00
9	Configurar VPN para trabajar con gitlabdes desde casa	1 day?	15/2/21 8:00	15/2/21 15:00
10	Estudio estructura gitlab-ci.yml	2 days?	15/2/21 8:00	16/2/21 14:59
11	Construir primera pipeline funcional (phpcs + phpstan)	9 days	16/2/21 8:00	26/2/21 15:00
12	**Gitlab + Pruebas en proyecto piloto**	**57 days?**	**1/3/21 8:00**	**19/5/21 15:00**
13	**Pruebas funcionales**	**19 days?**	**1/3/21 8:00**	**26/3/21 15:00**
14	Cypress + local drupal project	11 days?	1/3/21 8:00	16/3/21 15:00
15	Integración con Gitlab + Cypress	9 days?	16/3/21 8:00	26/3/21 15:00
16	**Análisis estático - calidad de código**	**4 days?**	**26/3/21 8:00**	**31/3/21 15:00**
17	Integración con Gitlab + PhpCS + PhpStan	4 days?	26/3/21 8:00	31/3/21 15:00
18	**Pruebas de seguridad**	**3 days?**	**2/4/21 8:00**	**6/4/21 15:00**
19	Gitlab SAST + Secret Detection	3 days?	2/4/21 8:00	6/4/21 15:00
20	**Pruebas de rendimiento**	**20 days?**	**7/4/21 8:00**	**4/5/21 15:00**
21	Lighthouse + local drupal project	10 days?	7/4/21 8:00	20/4/21 15:00
22	Integración Gitlab + Lighthouse	6 days?	21/4/21 8:00	28/4/21 15:00
23	Integración Gitlab + k6 (pruebas de carga)	4 days?	29/4/21 8:00	4/5/21 15:00
24	**Pruebas de accesibilidad**	**11 days?**	**5/5/21 8:00**	**19/5/21 15:00**
25	GitLab Accesibility	3 days?	5/5/21 8:00	7/5/21 15:00
26	Custom Pa11y	8 days?	8/5/21 8:00	19/5/21 15:00
27	Integración proyecto real (Marcotran): Metodología	15 days?	19/5/21 8:00	8/6/21 15:00
28	Redacción Memoria TFG	22 days?	20/5/21 8:00	18/6/21 15:00

PlanificacionTFG - pagina1

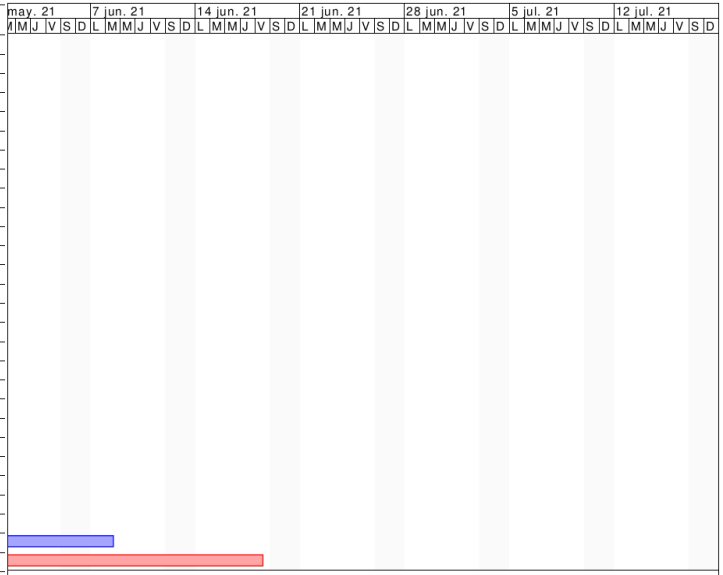


PlanificacionTFG - pagina2

PlanificacionTFG - pagina3

		Nombre	Duracion	Inicio	Terminado
1		**Configurar Proyecto Drupal Local**	**3 days?**	**9/2/21 8:00**	**11/2/21 15:00**
2		Instalar Docker y drupal	1 day?	9/2/21 8:00	9/2/21 15:00
3		Instalar PHP Code Sniffer	1 day?	10/2/21 8:00	10/2/21 15:00
4		**Instalar y configurar PHPStan**	**2 days?**	**10/2/21 8:00**	**11/2/21 15:00**
5		Instalar PHP Stan	1 day?	10/2/21 8:00	10/2/21 15:00
6		Formatear salida PhpStan	1 day?	11/2/21 8:00	11/2/21 15:00
7		**Configurar Gitlab**	**11 days?**	**12/2/21 8:00**	**26/2/21 15:00**
8		Creación cuenta Gitlabdes	1 day?	12/2/21 8:00	12/2/21 15:00
9		Configurar VPN para trabajar con gitlabdes desde casa	1 day?	15/2/21 8:00	15/2/21 15:00
10		Estudio estructura gitlab-ci.yml	2 days?	15/2/21 8:00	16/2/21 14:59
11		Construir primera pipeline funcional (phpcs + phpstan)	9 days	16/2/21 8:00	26/2/21 15:00
12		**Gitlab + Pruebas en proyecto piloto**	**57 days?**	**1/3/21 8:00**	**19/5/21 15:00**
13		**Pruebas funcionales**	**19 days?**	**1/3/21 8:00**	**26/3/21 15:00**
14		Cypress + local drupal project	11 days?	1/3/21 8:00	16/3/21 15:00
15		Integración con Gitlab + Cypress	9 days?	16/3/21 8:00	26/3/21 15:00
16		**Análisis estático - calidad de código**	**4 days?**	**26/3/21 8:00**	**31/3/21 15:00**
17		Integración con Gitlab + PhpCS + PhpStan	4 days?	26/3/21 8:00	31/3/21 15:00
18		**Pruebas de seguridad**	**3 days?**	**2/4/21 8:00**	**6/4/21 15:00**
19		Gitlab SAST + Secret Detection	3 days?	2/4/21 8:00	6/4/21 15:00
20		**Pruebas de rendimiento**	**20 days?**	**7/4/21 8:00**	**4/5/21 15:00**
21		Lighthouse + local drupal project	10 days?	7/4/21 8:00	20/4/21 15:00
22		Integración Gitlab + Lighthouse	6 days?	21/4/21 8:00	28/4/21 15:00
23		Integración Gitlab + k6 (pruebas de carga)	4 days?	29/4/21 8:00	4/5/21 15:00
24		**Pruebas de accesibilidad**	**11 days?**	**5/5/21 8:00**	**19/5/21 15:00**
25		GitLab Accesibility	3 days?	5/5/21 8:00	7/5/21 15:00
26		Custom Pa11y	8 days?	8/5/21 8:00	19/5/21 15:00
27		Integración proyecto real (Marcotran): Metodología	15 days?	19/5/21 8:00	8/6/21 15:00
28		Redacción Memoria TFG	22 days?	20/5/21 8:00	18/6/21 15:00

PlanificacionTFG - pagina1



PlanificacionTFG - pagina4