https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 153 - 168

Estrategias y herramientas para depuración de código en el back-end

Luz María Ramírez Romero
Dirección General de Cómputo y de Tecnologías de Información y Comunicación
Universidad Nacional Autónoma de México
luzrr@unam.mx
ORCID: 0000-0002-8867-9374

Información del reporte:

Licencia Creative Commons

El contenido de los textos es responsabilidad de los autores y no refleja forzosamente el punto de vista de los dictaminadores, o de los miembros del Comité Editorial, o la postura del editor y la editorial de la publicación.

Para citar este reporte técnico:

Ramírez Romero, L. M. (2023).Estrategias y herramientas para depuración de código en el back-end. *Cuadernos Técnicos Universitarios de la DGTIC, 1* (1), páginas (153 - 168). https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Resumen:

El desarrollo de módulos y componentes libres de errores o bugs es uno de los objetivos medulares de la construcción de sistemas. Esto forma parte de entregar y/o liberar en ambientes de producción, sistemas de calidad, libres de problemas o vicios ocultos que impidan cubrir los requerimientos de los usuarios y que eviten cumplir los objetivos del sistema. Se exponen algunas estrategias y herramientas que el programador puede utilizar para depurar los sistemas de información, basados tanto en cambios de metodologías o paradigmas, como es la aplicación de *Scrum*, como en el uso de la Inteligencia Artificial para encontrar rutinas de código probadas, o incluso para la refactorización del código. Asimismo, se ha observado útil la integración de técnicas como el *Test Driven Development* (*TDD*), que propone automatizar las pruebas de módulos individuales, además del uso de herramientas propias de *Laravel/Livewire*, como sus archivos de depuración, la herramienta *Tinker* y la barra de depuración de *Laravel*, o bien el uso de extensiones integradas al editor de código Visual Studio Code.

Palabras clave:

Herramientas de depuración, bug, error de codificación, asistentes de codificación basados en Inteligencia Artificial, pruebas unitarias.

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 154 - 168

1. INTRODUCCIÓN

Los programadores introducen defectos en el código que desarrollan o que actualizan y esto ocurre de manera involuntaria porque errar y equivocarse es una situación inherente a la naturaleza humana. El programador al cometer un error, introduce defectos en el código. Estos defectos o desperfectos harán que el sistema falle eventualmente, producirán malestar en los usuarios, problemas en el dominio o contexto donde se ejecuta el sistema y generarán un costo para la organización.

El especialista en TI José M. Huerta, menciona en su *blog Gestión de TI*, las siguientes causales de defectos en el *software*, mismas que propone analizar para intentar reducir los defectos y fallos en las aplicaciones desarrolladas (Huerta, J., 2028): Los programadores introducen defectos por no comprender los requerimientos de los usuarios, o al encontrarse con errores de diseño que llevan a un planteamiento equivocado o confuso.

Otros defectos son los directamente atribuibles al código por sí mismo, consistentes en los llamados *errores de dedo*, que agregan caracteres extras o equivocados al escribir el código, o que son ocasionados por no conocer con precisión la sintaxis del lenguaje, o *framework*, o marco de trabajo de programación

También se producen defectos con los consecuentes fallos en el software por cambios de versión de la tecnología de soporte o tecnología subyacente, con la que se conforman los sistemas.

Hay varias estrategias para contrarrestar y disminuir la introducción de defectos en los sistemas de información, tales como:

- El uso de herramientas que ayuden al programador a detectar defectos de sintaxis de manera temprana.
- Codificar utilizando un conjunto de buenas prácticas, configuradas a través de la experiencia acumulada en el mundo de la ingeniería de software, que permitan hacer más fácil la depuración del código y que además faciliten su actualización y evolución posterior.
- Contar con rutinas, funciones y componentes ya probados, que resuelvan problemas menores y que puedan reutilizarse.
- Disponer de herramientas que permitan detectar puntos de mejora y refactorización[1] del código.

En este reporte técnico se tratará el tema de la producción de *software* de calidad, dentro de la metodología y enfoque ágil de desarrollo de sistemas *Scrum*. También se hablará sobre la construcción del *back-end*[2] de sistemas o aplicaciones desarrolladas bajo los marcos de trabajo o *frameworks Laravel-Livewire*, lenguaje de programación *PHP*, a través del editor de código *Visual Studio Code* (*VSC*)[3].

[1] *Refactorización* consiste en re-estructurar o re-escribir algunas partes de código para hacerlo más comprensible, mejorar su estructura, eliminar código muerto, con el fin de hacerlo más mantenible y para no introducir defectos durante su actualización posterior.

[2] *Back-end* es la programación del lado del servidor para implementar la lógica de negocio (Luna et al., 2018). En el *Modelo Vista-Controlador* (*MVC*) del *framework Laravel*, el *Back-end* lo constituyen los controladores de la aplicación.

[3] *Visual Studio Code* (*VSC*) es uno de los editores de código fuente, ampliamente conocido y utilizado por programadores en todo el mundo. Es una herramienta gratuita

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 155 - 168

2. OBJETIVOS

- Revisar cómo contribuye la metodología del enfoque ágil *Scrum* en la obtención temprana de los resultados de pruebas sobre el *software* en desarrollo para aplicar las técnicas y herramientas de depuración del código.
- Compartir estrategias para la depuración de código en el *back-end* de una aplicación bajo el paradigma orientado a objetos de *PHP-Laravel*.
- Interpretar la información que proporcionan las herramientas de depuración de código en los *frameworks* o marco de trabajo *Laravel-Livewire*.
- Proponer nuevos esquemas para producir código con menos defectos basados en tecnologías actuales como las propuestas por los modelos de Inteligencia Artificial (*IA*).

3. CICLO DE VIDA CLÁSICO DE DESARROLLO DE SOFTWARE VS. EL ENFOQUE ÁGIL DE SCRUM.

La construcción de aplicaciones de *software* ocurre dentro de un proceso metodológico y estructurado, denominado *Ciclo de Vida Clásico* del desarrollo de *software* (*Ingeniería de Software*). Este proceso inicia con la fase de *Análisis*, que indica que la etapa más importante del desarrollo de sistemas, no es la programación, sino que consiste en comprender el objetivo del sistema y las necesidades o requerimientos del usuario.

Partiendo de ese *Análisis*, continúa el *Diseño del software*, que puede ser conformado a través de diagramas y prototipos basados en satisfacer las necesidades del usuario. A continuación del *Diseño*, sigue la etapa de *Programación*, en la que se implementan los prototipos que responderán a las necesidades del usuario. Posteriormente, está la etapa de *Pruebas* que consiste en que un equipo de personas diferentes al equipo de desarrolladores, revise el sistema y sus módulos para asegurar que todas las piezas desarrolladas cumplan con los requerimientos del usuario. Finalmente, llega la etapa de *Liberación* o *Despliegue* del *software*, que colocará el sistema en un ambiente de producción, lo que permitirá a los usuarios ocupar el sistema para lograr sus objetivos iniciales.

A pesar de que el desarrollo de *software* ocurre dentro de este *Ciclo de Vida Clásico*, metodológico y estructurado, en 1968, durante la primera conferencia sobre desarrollo de *software* de la *OTAN*, se definieron una serie de problemas denominados como la *Crisis del Software*, que es un término que se refiere a que comúnmente los sistemas a la medida se concluían mucho después del tiempo planeado y que al final los sistemas no cubrían las necesidades de los usuarios (Noriega, M., 2015).

Por lo tanto, en 1995 un grupo de ingenieros de *software*, entre los que destaca Jeff Sutherland, crean el *Manifiesto Ágil*, a partir del cual se desarrolla la *Metodología Scrum*, cuya principal aportación es la creación del *software* en varios ciclos cortos. En cada ciclo, se elige desarrollar algunas de las funcionalidades que requiere el usuario, de tal manera que al finalizar cada uno, se obtiene una parte del sistema probada y funcionando, lista para que el usuario la revise y determine si es lo que esperaba.

El otro gran cambio que introduce *Scrum* es la realización de las pruebas en cada ciclo de desarrollo, con lo cual se obtiene rápidamente retroalimentación sobre los componentes desarrollados en el ciclo, para

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 156 - 168

que el código sea depurado, a diferencia del *Ciclo de Vida Clásico* de desarrollo de *software*, en el que las pruebas comienzan hasta que es concluido todo el sistema.

Las técnicas y herramientas documentadas en este texto dan soporte a la etapa de *Desarrollo* y *Pruebas* del *software*.

En la tarea de depuración de *software*, no se ha encontrado en la literatura de ingeniería de *software*, ni en la de métodos ágiles, alguna metodología definida para llevar a cabo la depuración del *software* (Trigás, G. (2012). Dependiendo de las herramientas de desarrollo seleccionadas, se utilizan diversas herramientas y técnicas de depuración.

A continuación, se abordarán las que son aplicables al desarrollo *back-end* (es decir, de la implementación de las reglas de negocio de las aplicaciones de software a la medida), para los *frameworks Laravel/Livewire*, lenguaje *PHP* y para el editor de código *VSC*.

4. USO DE ASISTENTES DE CODIFICACIÓN BASADOS EN INTELIGENCIA ARTIFICIAL (IA)

Estas herramientas surgen a partir del año 2021, y son las más novedosas y recientes para ayudar al desarrollo y depuración de código. Estas piezas de *software* inteligentes, sugieren a los programadores humanos, desde simples líneas de código individuales, hasta pequeñas rutinas que resuelven problemas sencillos, específicos y granulares (*IA para desarrolladores*).

El primer asistente de código conocido es ***Codex***, que fue desarrollado por la empresa *OpenAI,* que de acuerdo con la definición que brinda su sitio web, *Codex* es capaz de traducir el lenguaje natural a líneas de código (OpenAI, 2023). Este es un hecho importante en el mundo del desarrollo de aplicaciones porque con ayuda de estos asistentes, los programadores pueden crear módulos y componentes con menos defectos, además de que estos asistentes ayudan a desarrolladores tanto principiantes, como expertos.

A partir de *Codex*, se desarrolló el complemento *GitHub Copilot*, que se puede instalar en varios editores de código tales como *VSC, Vim, JetBrains* o *Neovim*, potenciando estos editores con el soporte mencionado para los programadores. Otros asistentes de código son *Blackbox AI Code Generator* y *GitHub Copilot*.

Blackbox AI Code generator. Se trata de un asistente de programación de uso gratuito, que se le instala al *VSC*. El programador le puede preguntar al *Blackbox* por rutinas de programación específicas, por ejemplo, cómo conectarse a un servidor de base de datos específico o cómo ordenar una lista de números de mayor a menor, tras lo cual propondrá el código correspondiente. *Blackbox* realmente buscará en diversas fuentes de Internet dichas rutinas de código, tales como *Stack Overflow* y brindará los resultados de la búsqueda como propuestas. El programador revisará y adaptará la rutina que mejor convenga al problema a resolver (Blackbox AI, s.f.). Un ejemplo de funcionamiento de Blackbox se muestra en el Anexo A.

GitHub Copilot. Esta extensión también está potenciada con *IA*, y basa su núcleo en *OpenAI Codex* que es capaz de sugerir líneas de código, construir expresiones regulares, funciones completas y proponer comentarios para documentar los programas. *Copilot* realiza sus búsquedas de código en los repositorios de *GitHub*. El programador se encargará de revisar cuál opción conviene más al problema y adaptará el código a su módulo. *Copilot* es una extensión de pago, ya que sus funciones de *IA* le permiten proponer código más adaptado al contexto que tiene en el editor de *VSC* en ese momento (GitHub, 2022).

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 157 - 168

5. INSTALACIÓN DE OTROS COMPLEMENTOS EN EL EDITOR DE CÓDIGO VISUAL STUDIO CODE (VSC)

El *VSC* cuenta actualmente con la función de detectar el lenguaje de programación de cada archivo, a través de la extensión del mismo. Por ejemplo, los archivos con extensión *.blade.php* son asociados a código *HTML* con directivas del motor blade de Laravel. Los archivos con extensión .php son asociados con código *PHP*. Una vez detectado el lenguaje en el que está escrito cada archivo, *VSC* marca de manera gráfica con color rojo, los errores de sintaxis para que el programador los identifique y los pueda corregir rápidamente.

Asimismo, *VSC* es capaz de identificar cuando se abre una función, o ciclo, o bucle de programación y marca en rojo cuando no se detecta el cierre correspondiente para su corrección. Además de esta funcionalidad propia del *VSC*, se pueden agregar complementos o extensiones para apoyar la tarea de programación de componentes y sistemas, tales como los siguientes:

Run on Save. Revisa estilos de codificación y eficiencia de las líneas codificadas en el momento de guardar el archivo del módulo, de tal manera que elimina cosas innecesarias, como líneas en blanco excesivas y propone cierto grado de refactorización en el código, cambiando las líneas de código, o eliminando variables innecesarias, por ejemplo.

PHP Intelephense. Realiza autocompletado de código, mostrando en una pequeña ventana emergente diferentes opciones de selección, ante una instrucción PHP. Lo anterior apoya al programador para tener a la mano la estructura y sintaxis de funciones básicas de PHP.

Laravel Extra Intellisense. Esta extensión proporciona un autocompletado de las rutas de Laravel.

Laravel Goto Controller. Permite abrir controladores de Laravel desde el archivo de rutas web.php.

6. EL ANÁLISIS DE LA BITÁCORA DE ERRORES DE LARAVEL

Los proyectos del marco de trabajo *Laravel*, cuentan con archivos de bitácora de errores, dentro de la carpeta *storage/logs* del proyecto, en dónde típicamente se registran los errores ocurridos en tiempo de ejecución. Estos archivos de errores son *laravel.log, errors.log* y *debug.log*.

El programador de *back-end* debe revisar estos archivos de bitácora, los cuales brindarán información más específica de los errores producidos, junto con mensajes del *framework*, que le ayudarán a determinar la causa de la falla (problemas en la conexión con la base de datos, nombres de campos o de variables mal nombradas o que no existen, falta de instalación de componentes, entre otros).

Asimismo, el programador que utiliza *Laravel* puede enviar mensajes y el contenido de variables, arreglos, clases y colecciones con la directiva *Log::debug*, mismos que se registrarán en el archivo *laravel.log*, para su revisión y análisis correspondiente. (Aguirre, S., 2022) En el Anexo B se presenta un resumen del uso de la bitácora de *Laravel*.

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 158 - 168

7. DIRECTIVA DD (DUMP AND DIE) DE LARAVEL

La función dd ayuda al programador a revisar el contenido de la variable especificada en el navegador web y además detiene la ejecución. La siguiente figura muestra cómo imprimir el contenido de la variable $this->sesion y detener la ejecución del programa.

Figura 1

Código para imprimir el contenido

```
dd($this->sesion);
```

Se incluye en el Anexo C un ejemplo de la respuesta de la directiva *dd* de *Laravel*.

8. USO Y ANÁLISIS DE LA BARRA DE DEPURACIÓN DE LARAVEL

Este elemento se muestra en la parte inferior del navegador web, una vez instalado, y proporciona información valiosa sobre el tiempo de ejecución de las consultas a la base de datos por cada interfaz de la aplicación, así como el número y nombre de los modelos (tablas) consultadas de la base de datos.

Es un paquete que debe instalarse en el proyecto de *Laravel* para poder hacer uso de sus características y ocuparlas para la depuración del código.

La barra de depuración de *Laravel* se muestra en el Anexo D.

9. TINKER, LA CONSOLA DE LARAVEL

Dice la definición técnica que *Tinker* es una consola de *Laravel* (Blastcoding , s.f. y Aguirre, S. 2022), que permite revisar de manera inmediata y directa los resultados a obtener de una instrucción o conjunto de instrucciones y así poder integrar el código a los controladores y componentes del *back-end*, con la certeza de no haber insertado defectos en el código, puesto que el fragmento de código ya fue probado en la consola *Tinker.*

Usos recomendados de la consola *Tinker*:

- Probar algún método o clase propios de Laravel, a fin de incluirlos en los controladores que se estén desarrollando.
- Interactuar con cualquier clase que se haya programado.
- Revisar funcionalidades de librerías que se hayan integrado a *Laravel*, como el manejo de fechas a través de la librería (Carbon, s.f.).
- Revisar la interacción entre los controladores de *Laravel* con la base de datos, ejecutando las sentencias de su *ORM* (*Object Relational Mapping*) *Eloquent, SQL* nativo o a través del constructor de consultas *Query Builder.*

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 159 - 168

10. PRUEBAS UNITARIAS

Otra estrategia para la depuración de código es que el programador desarrolle código para probar de manera automática su programación.

Esta es una estrategia denominada *Test-Driven Development* (*TDD*) (Paradigma, s.f.), que permite ejecutar las pruebas automatizadas cuantas veces sea necesario, cada vez que el código es re-factorizado, mejorado, modificado o actualizado. Cabe mencionar que las pruebas unitarias no reemplazan otro tipo de pruebas como las funcionales de caja negra, pruebas de integración o pruebas de regresión y tampoco sustituyen a los probadores o *testers*.

El enfoque *TDD* también propone alterar el *Ciclo de Vida tradicional* de desarrollo de *software*, indicando que primero se deben definir y automatizar las pruebas de un módulo y después el desarrollador o programador debe construir su módulo, haciendo que éste vaya cumpliendo con las pruebas definidas para él. Una vez que se ha terminado de codificar el módulo, se ejecuta la prueba automatizada, que indicará si se está cumpliendo con los objetivos para los que fue construido (Beck, K, 2003).

Los programas correspondientes a las pruebas unitarias deben colocarse en el directorio *tests/Unit* de la estructura del proyecto de *Laravel*.

Para una referencia rápida sobre las pruebas unitarias en *Laravel* se sugiere revisar el Anexo E.

11. RESULTADOS

Las herramientas y estrategias revisadas se han aplicado en el desarrollo de sistemas y aplicaciones a la medida por el grupo de desarrolladores de la Subdirección de Sistemas Integrados de la DGTIC. Se han obtenido resultados de mejora en los tiempos de desarrollo y codificación, así como la reducción del número de incidencias atribuidas a errores por código.

Uno de los resultados más importantes es el cambio de metodología para el desarrollo de *software*. La experiencia obtenida brinda como resultado que es mejor el desarrollo de aplicaciones bajo el enfoque ágil, que enmarcar el proyecto en el *Ciclo de Vida Clásico*.

Otro hallazgo importante, es la aplicación de herramientas de Inteligencia Artificial en el desarrollo y depuración de código. Mediante el uso de estas herramientas, se puede incluir en el código de los sistemas a la medida, la implementación de algoritmos genéricos en el lenguaje de programación elegido. Otra ventaja adicional es que esas implementaciones ya han sido utilizadas y probadas en otros sistemas de código abierto publicados en repositorios tales como *GitHub*. Entre los algoritmos genéricos nombrados se puede mencionar los de ordenamiento y las rutinas para quitar espacios en blanco, entre otros.

Tradicionalmente primero se construía el *software* y después se hacían las pruebas al mismo. Actualmente es más útil involucrar al equipo de pruebas desde el inicio del proyecto, de tal manera que se diseñan las pruebas, junto con la etapa de análisis. Posteriormente, se construyen los módulos de la aplicación; y una vez construidos los módulos, se pueden probar de manera inmediata con las pruebas unitarias previamente construidas. Con ello, se sabe rápidamente si los módulos de manera individual, cumplen con los objetivos y requerimientos del usuario. Además, este enfoque de TDD permite que cada vez que se realicen cambios en los módulos de *software*, se pueda correr de manera automática las pruebas unitarias para saber si los módulos afectados siguen brindando los resultados esperados.

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 160 - 168

Asimismo, durante el proceso de codificación se ocupan herramientas como *Tinker*, que permiten al desarrollador probar paso a paso el código escrito y saber si se están obteniendo los resultados esperados.

Todas estas técnicas apoyan un proceso eficiente de desarrollo de *software*, dentro de un círculo virtuoso que tiene como propósito final, la producción de *software* de calidad en tiempos competitivos.

Adicionalmente, se observa que la elección de un editor de código no es una tarea trivial, ya que actualmente existen editores muy especializados, con extensiones que pueden agregarse a la funcionalidad básica del editor y que permite ir analizando el código, al tiempo en que es escrito, con la finalidad de detectar de manera pronta y expedita la introducción de defectos sintácticos en el código. Incluso hay extensiones que ofrecen cierto grado de refactorización del código, proponiendo líneas más eficientes de código.

12. CONCLUSIONES

Debido a que se sabe de antemano que todo código tiene defectos, que pueden traducirse en fallas de los sistemas, es obligatorio que todo grupo de desarrollo de aplicaciones cuente con una serie definida de estrategias y herramientas que apoyen la depuración del código, además de contar con un equipo de personas especializadas en pruebas de *software* que trabaje de la mano con ingenieros de requerimientos y desarrolladores de *software*, para apoyar la labor de depuración del código.

Dentro de la actividad de depuración de código, es importante integrar los cambios de metodologías y de paradigmas, tales como el enfoque ágil desde la organización del proyecto, la obtención de resultados de pruebas y detección de defectos de forma pronta, dentro del ciclo de vida del desarrollo de *software*.

Una de las principales ventajas que se ha detectado en el uso del enfoque ágil es la obtención y entrega al usuario final, de módulos que funcionan en tiempos muy cortos de hasta 1 o 2 semanas, lo cual refuerza la credibilidad en el equipo de desarrollo de *software* y permite que el usuario brinde retroalimentación sobre dichos módulos.

Otra técnica que ha resultado importante integrar en la depuración de código es el *TDD*, que permite probar automática y rápidamente los módulos del sistema de manera individual, cuantas veces sea necesario, lo que resulta útil cada vez que se cambia algo en el *software*, debido a alguna actualización, mejora o por atender nuevos requerimientos de los usuarios.

Asimismo, es importante considerar el probar e incorporar de manera continua y permanente nuevas herramientas en la construcción y depuración del código fuente de módulos y aplicaciones, en el marco del lenguaje de programación y *frameworks* seleccionados para el desarrollo del *software*.

Un ejemplo de este análisis y búsqueda constante de herramientas y técnicas de depuración, es la aplicación de herramientas en el campo de la *IA,* la cual está apoyando varios campos de las Tecnologías de Información y uno de ellos es el desarrollo de *software*. Resulta muy útil contar con este tipo de soporte tecnológico, ya que con ello se acortan los tiempo de análisis y depuración de código, pues estas herramientas novedosas proponen rutinas sencillas de código ya probado en otros sistemas de *software* abierto, o incluso sugieren la refactorización o re-escritura de código, en aras de lograr eficiencia.

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 161 - 168

Asimismo, no se debe soslayar las herramientas tradicionales de depuración que también resultan útiles en la detección de defectos y que permiten a los desarrolladores revisar los resultados de líneas de código, de manera parcial, para integrarlas al código, una vez que se ha comprobado que logran objetivos atómicos o más granulares, que se suman al objetivo global del módulo y del sistema en su conjunto.

Todo paradigma, técnica y herramienta enfocada en la depuración de código, debe ser considerada para lograr la confiabilidad en el sistema desarrollado, especialmente para asegurar que el sistema brindará los resultados requeridos por los usuarios, para superar la llamada *Crisis del Software*.

REFERENCIAS BIBLIOGRÁFICAS

Aguirre, S. (2022). *Laravel Curso completo*. Ra-Ma Editorial.

Beck, K. (2003). *Test-Driven Development by Example*. Addison-Wesley Signature Series.

Blackbox IA, (s.f.). *Blackbox*. Recuperado 21 de septiembre de 2023, de https://www.useblackbox.io/

[B]lastcoding.com (s.f.). *Qué es y cómo usar Laravel Tinker.* (1556204739). Blastcoding. https://blastcoding.com/?post_type=post&p=2163

Carbon (s.f.). *A simple PHP API extension for DateTime*. Recuperado 22 de septiembre de 2023, de https://carbon.nesbot.com/docs/

Huerta, J. (2028). *Coste de los errores en proyectos de software*. Recuperado 21 de septiembre de 2023, de https://josehuerta.es/gestion/proyectos/calidad/coste-de-los-errores-en-proyectos-de-software

Gamarra, F. (2023). *IA para desarrolladores*. Editorial RedUsers.

GitHub (Director). (2022, noviembre 15). *What is GitHub Copilot?* https://www.youtube.com/watch?v=IqXNhakuwVc

Luna, F., Millahual, C. P., & Iacono, M. (2018). *PROGRAMACION WEB Full Stack 13 - PHP: Desarrollo frontend y backend* - Curso visual y práctico. Editorial RedUsers.

Noriega, M. (2015). *El proceso de desarrollo de software*. Editorial IT Campus Academy.

OpenAI. (2023). *OpenAI Codex*. Recuperado 21 de septiembre de 2023, de https://openai.com/blog/openai-codex

Pantaleo, G y Rinaudo, L. (2015). *Ingeniería de Software.* Alfaomega editores.

Paradigma (s.f.). *TDD como metodología de diseño de software*. Recuperado 22 de septiembre de 2023, de https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/

Trigás, G. (2012). *Metodología Scrum: El proceso de desarrollo de software*. Editorial Universitat Oberta de Catalunya.

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 162 - 168

ANEXO A. EJEMPLO DE FUNCIONAMIENTO DE BLACKBOX

En las imágenes 2 y 3, se muestra como ejemplo una pregunta que se le hizo a la extensión de *Blackbox* que se instaló en el *VSC*. Se le pidió el código para ordenar una lista de número de mayor a menor en lenguaje *PHP* y la respuesta fue una explicación de cada paso que debe realizar el programador para lograr el resultado deseado, así como el código completo.

Figura 2

Ejemplo de forma de consultar a Blackbox y primera parte de la respuesta del software

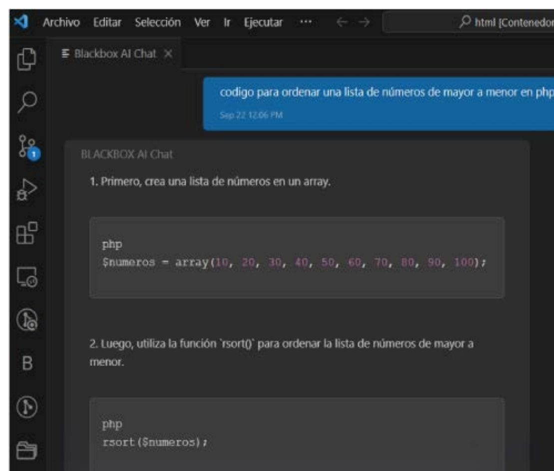
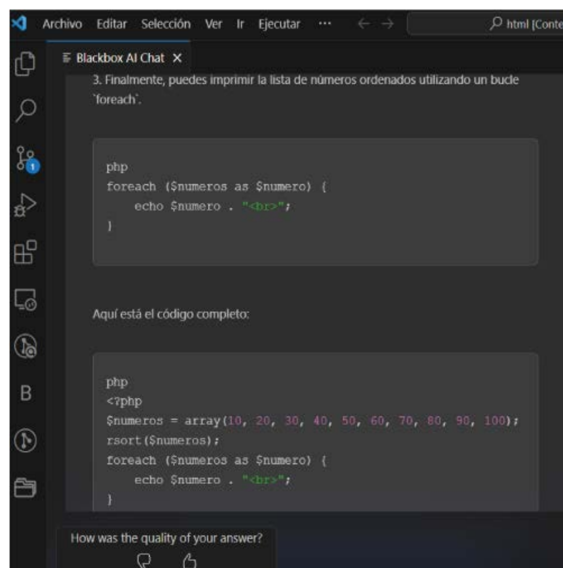


Figura 3

Parte 2 de la respuesta de Blackbox



https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 163 - 168

ANEXO B. RESUMEN DE USO DE LA BITÁCORA DE LARAVEL

Para utilizar la directiva *Log::debug* primero hay que importar la clase Log de la siguiente manera:

```
use Illuminate\Support\Facades\Log;
```

Un ejemplo de uso de *Log::debug* es el siguiente: imprimir el mensaje "*Id de sesión* =" y enseguida se imprimirá el contenido de la *variable $id_sesion*:

```
Log::debug("Id de sesion = " . $id_sesion);
```

En la siguiente figura se muestra un ejemplo de un error registrado en la bitácora de *Laravel*, ocasionado por la falta de instalación del paquete *DomPDF* que ayuda a generar archivos en formato *PDF* desde *Laravel*.

Figura 4

Ejemplo de error en la bitácora errors.log de Laravel

```
storage > logs > errors.log

1  [2023-09-13 18:53:44] local.ERROR: Class "Barryvdh\DomPDF\ServiceProvider" not found {"exception":"[obj
2  [stacktrace]
3  #0 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/ProviderRepository.php(142): Illumi
4  #1 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/ProviderRepository.php(61): Illumin
5  #2 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/Application.php(745): Illuminate\\F
6  #3 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/Bootstrap/RegisterProviders.php(17)
7  #4 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/Application.php(261): Illuminate\\F
8  #5 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/Http/Kernel.php(186): Illuminate\\F
9  #6 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/Http/Kernel.php(170): Illuminate\\F
10 #7 /var/www/html/vendor/laravel/framework/src/Illuminate/Foundation/Http/Kernel.php(144): Illuminate\\F
11 #8 /var/www/html/public/index.php(51): Illuminate\\Foundation\\Http\\Kernel->handle(Object(Illuminate\\
12 #9 {main}
13 "}
```

ANEXO C. EJEMPLO DE LA RESPUESTA DE LA DIRECTIVA DD DE LARAVEL

Como ejemplo de uso de dd se tiene la siguiente línea de código, en la cual se pide a *Laravel* imprimir el contenido de la variable *$this->sesión*, que contiene el resultado de consultar el registro que corresponda al *$this->id_sesion de la tabla Sesion*.

```
$this->sesion = Sesion::find($this->id_sesion);

dd($this->sesion);
```

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 164 - 168

La respuesta que se obtiene en el navegador web se muestra en la figura 5. En dicha salida se observa información diversa, como la base de datos a la que se está conectando (*Postgresql – pgsql*), la tabla sesión, la llave primaria de la tabla y de manera especial se muestra el contenido de los atributos que conforman la variable *$this->sesion*, que consiste en un arreglo de 8 atributos (*id_sesion, id_tipo_sesion, fecha, id_sede, codigo, activa* y dos atributos propios de *Laravel*: *created_at* y *updated_at*, que son la fecha y hora de creación y actualización del registro).

Figura 5

Respuesta del dump and die (dd) de Laravel

```
App\Models\Sesion {#658 ▼ // app/Http/Livewire/Consultar
  #connection: "pgsql"
  #table: "sesion"
  #primaryKey: "id_sesion"
  #keyType: "int"
  +incrementing: true
  #with: []
  #withCount: []
  +preventsLazyLoading: false
  #perPage: 15
  +exists: true
  +wasRecentlyCreated: false
  #escapeWhenCastingToString: false
  #attributes: array:8 [▼
    "id_sesion" => 2
    "id_tipo_sesion" => 1
    "fecha" => "2023-10-09"
    "id_sede" => 1
    "codigo" => "555555"
    "activa" => true
    "created_at" => "2023-09-12 12:42:21.075096"
    "updated_at" => null
  ]
  #original: array:8 [▶]
  #changes: []
  #casts: []
  #classCastCache: []
  #attributeCastCache: []
```

ANEXO D. BARRA DE DEPURACIÓN DE LARAVEL

Para la instalación se debe ejecutar el siguiente comando en la consola:

```
composer require barryvdh/laravel-debugbar
```

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 165 - 168

En la siguiente figura se muestra la barra de depuración de *Laravel*.

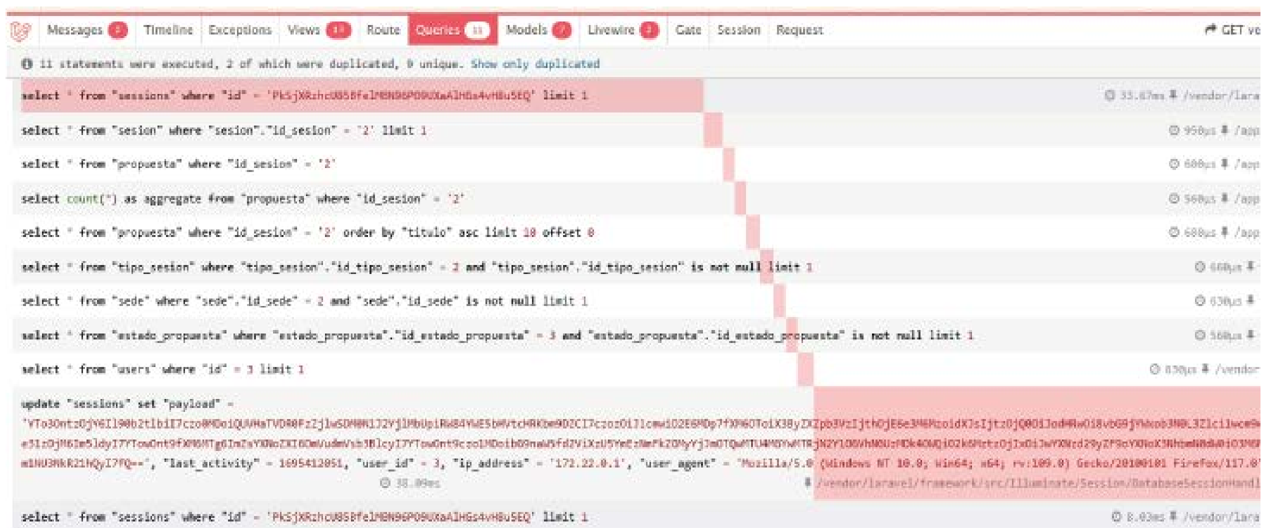
Figura 6

Barra de depuración de Laravel

La barra de depuración abierta y con la pestaña de "*Queries*" seleccionada se pueden apreciar en la siguiente figura, donde se observan 11 consultas a la base de datos y sus correspondientes sentencias *SQL* y del lado derecho de cada renglón, se aprecia el tiempo de ejecución de cada consulta. De esta manera, el programador se puede dar cuenta si necesita re-escribir algún *query* para optimizar su tiempo de respuesta y hacer más eficiente su aplicación.

Figura 7

Barra de depuración abierta con la pestaña Queries



ANEXO E. PRUEBAS UNITARIAS EN LARAVEL

A continuación, se muestran una serie de comandos para la consola relacionados con las pruebas unitarias.

Para crear una prueba unitaria se ejecuta el comando siguiente de *php artisan*:

```
PS C:\xampp\htdocs\peliculas\peliculas> php artisan make:test AcercadeTest --unit
```

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 166 - 168

Para ejecutar todas las pruebas automatizadas que se encuentren en la carpeta tests de *Laravel* se ejecuta el siguiente comando:

```
PS C:\xampp\htdocs\peliculas\peliculas> php artisan test
```

Para ejecutar solamente las pruebas unitarias que se encuentren en el directorio *tests/Unit* de *Laravel*, se corre el siguiente comando:

```
PS C:\xampp\htdocs\peliculas\peliculas> ./vendor/bin/phpunit
```

Para ejecutar únicamente una prueba (en este caso la prueba llamada “acercade”), se ejecuta el siguiente comando:

```
PS C:\xampp\htdocs\peliculas\peliculas> php artisan test --filter acercade
```

En la figura 7 se muestra el código de una prueba unitaria básica. En este ejemplo, se observa el uso de las clases *Tests\TestCase*, que contienen una serie de métodos para probar funciones específicas de *Laravel*.

Después se aprecia cómo se pide la carga de la página con la ruta */usuarios* con la directiva:

$this->get('/usuarios')

Una vez solicitada la carga de la página, con el método *assertStatus*, se revisa si el código que devuelve la carga de la página es un código 200, que equivale a la carga exitosa. Si se puede afirmar que el código devuelto es igual a 200 (código de éxito), entonces la prueba es exitosa y comprueba que se pudo cargar correctamente la página solicitada.

Después se comprueba con el método *assertSee* que la página incluya en su contenido la palabra ‘*Usuarios*’.

Figura 7

Ejemplo del código de una prueba unitaria básica en Laravel

```
tests > Unit > UsersModuleTest.php > UsersModuleTest > test_carga_un_nuevo_usuario
1  <?php
2
3  namespace Tests\Unit;
4
5  use Tests\TestCase;
6
7  class UsersModuleTest extends TestCase
8  {
9      public function test_carga_pagina_usuarios(): void
10     {
11         $response = $this->get('/usuarios')
12             ->assertStatus(200)
13             ->assertSee('Usuarios');
14     }
15
```

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24

Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 167 - 168

GLOSARIO

Back-end. En el *Modelo Vista-Controlador* (*MVC*) se le conoce como *back-end* a todos los complementos y funciones que se colocan en el servidor y que se encargan de implementar la lógica de negocio. Su contraparte lo constituye el *front-end* o interfaz de usuario. Parte del *back-end* son los manejadores de bases de datos que almacenan los datos de las aplicaciones.

Blackbox AI Code generator. Es un asistente de codificación de uso libre que permite codificar 10 veces más rápido. Permite convertir cualquier pregunta del lenguaje natural a código de programación.

Carbon. Librería para la manipulación y operaciones con fechas.

Codex. Asistente de codificación basado en Inteligencia Artificial, desarrollado por la empresa *OpenAI*.

Complementos/extensiones de Visual Studio Code. Son unidades de *software* que se instalan al editor de código (*Visual Studio Code*) para agregar funcionalidad diversa, apoyando la labor de programación.

Defecto. Desperfecto en un componente que puede causar que el mismo falle en sus funciones.

Depuración. Acción de diferentes modalidades de revisión para la detección de defectos con el propósito de efectuar su corrección.

Eloquent. Es el *ORM* (*Object-Relational Mapping*) de *Laravel* que permite la interacción entre el sistema desarrollado en *Laravel* y la base de datos.

Error. Acción humana que produce un resultado incorrecto.

Falla. Manifestación visible de un defecto.

Framework de programación. Es un conjunto de herramientas para el desarrollo de sistemas o aplicaciones de manera más rápida y eficiente.

Front-end. En el *Modelo Vista-Controlador* (*MVC*) se le llama *front-end* a todos los elementos que se encuentran en el cliente y que son el punto de contacto con el usuario final. A través del *front-end*, el usuario ingresa datos para obtener información por parte del *back-end*.

GitHub. Es un repositorio del código fuente de diversas aplicaciones que es publicado por la comunidad *GitHub* con el propósito de compartirlo (si se coloca como público) y/o para obtener aportaciones en funcionalidad, en documentación y que además permite controlar las versiones de las aplicaciones que ahí se alojan.

GitHub Copilot. Es un asistente para desarrollo de código cuyo núcleo está basado en *OpenAI Codex* y que busca rutinas de código en el repositorio *GitHub*.

Inteligencia artificial. Es una disciplina tecnológica cuyo propósito es desarrollar e implementar algoritmos que permitan emular algunas capacidades intelectuales humanas, a partir de mecanismos de entrenamiento de los algoritmos y con el objetivo de resolver problemas.

Laravel. Es un *framework* o marco de trabajo de código abierto, basado en el *Modelo Vista-Controlador*, y soportado por el lenguaje de programación *PHP*, que brinda organización, estructura y ciertas funcionalidades para el desarrollo eficiente de sistemas y aplicaciones informáticas para la web.

https://doi.org/10.22201/dgtic.ctud.2023.1.1.24
Vol. 1, núm. 1. Octubre-diciembre 2023, págs. 168 - 168

Postgresql. Es un manejador de bases de datos basado en el modelo relacional, en el cual los datos se estructuran en entidades y tuplas, que se relacionan a través de llaves foráneas que permiten estas relaciones.

Pruebas funcionales de caja negra. Son pruebas que no toman en cuenta cómo está construida una aplicación, sino que se basa solamente en que a ciertos datos de entrada, se deben obtener un determinado conjunto de datos de salida.

Pruebas de integración. Son pruebas en las que se revisa que los diferentes módulos trabajen correctamente en su conjunto, satisfaciendo los requerimientos del usuario.

Pruebas de regresión. Estas pruebas se realizan sobre todo el sistema cada vez que alguna parte del código se modifica para aumentar o modificar la funcionalidad.

Query. Se refiere a las sentencias que se utilizan para consultar o interrogar a la base de datos.

Query Builder. Es una forma alternativa de métodos que ofrece *Laravel* para crear y ejecutar consultas a la base de datos.

Stack Overflow. Se trata de un foro en internet en el que los programadores publican problemas de programación y que otros programadores solucionan. Todas las conversaciones y código quedan disponibles para toda la comunidad y usuario interesado, lo que va conformando una base de conocimientos sobre programación en diferentes lenguajes y marcos de trabajo.

Tester (probador). Es un rol informático dedicado a planificar y llevar a cabo pruebas de *software* para comprobar que las aplicaciones cumplan con los requerimientos funcionales y no funcionales.

Visual Studio Code. Aplicativo especializado para escribir y editar código de *software*.