Elixir

INTRODUCCIÓN PARA ALQUIMISTAS

MANUEL ÁNGEL RUBIO JIMÉNEZ

Elixir

Introducción para Alquimistas

Manuel Angel Rubio Jiménez

Elixir
Introducción para Alquimistas

Manuel Angel Rubio Jiménez

Resumen

Elixir nació como la solución a un gran problema que recaía sobre BEAM de ser una gran plataforma pero con un lenguaje ***raro***. Con influencia de otros lenguajes y una gran ejecución por parte de José Valim comenzaron a salir las primeras versiones del lenguaje en 2011 y muchos entusiastas comenzaron a dedicarse a mejorar y hacerlo crecer.

Este libro te ayuda a adentrarte en el mundo de Elixir. Cómo nació, cómo es su comunidad, su ecosistema y por supuesto el lenguaje. Las ventajas y lo que lo hacen único e incluso sus debilidades. Todo para ayudarte a dar tus primeros pasos en Elixir o aprender toda la base en caso de tener ya conocimientos previos.

Aprenderás cómo crear aplicaciones cliente-servidor y las mejores prácticas para desarrollar con Elixir, crear proyectos, integrar tu código con el de otros a través de la instalación de dependencias, publicar tus propias librerías y desplegar tus proyectos en producción.

Este libro cubre todos los aspectos principales de Elixir 1.7.

Depósito legal CO-2236-2018.

ISBN 978-84-945523-4-2

[1] http://altenwald.com/book/elixir
[2] https://creativecommons.org/licenses/by-nc-sa/3.0/deed.es

Capítulo 1. Lo que debes saber sobre Elixir

Todo gran logro requiere tiempo.
—Maya Angelou

Elixir se ha expandido muy rápidamente desde sus inicios. Los servicios en la nube requieren de soluciones de alta capacidad como Erlang o Elixir para soportar la inmensa cantidad de peticiones a un solo sistema llegando desde distintos puntos del planeta. Al principio solo las empresas de telecomunicaciones junto con la creciente demanda por parte de startups dedicadas a la mensajería instantánea parecían emplear este tipo de soluciones. Poco a poco otro tipo de empresas están usando soluciones en sectores como el de las apuestas deportivas, el sector financiero, los videojuegos online o publicidad son claros ejemplos de sectores con estas necesidades.

En este capítulo vamos a dar un repaso rápido a la historia de BEAM para continuar en el punto en el cual José Valim comenzó a elaborar Elixir, las decisiones de diseño que decidió adoptar y finalmente casos de uso que respaldan las buenas elecciones tomadas.

Comenzaremos antes definiendo cada elemento para no confundirnos cuando nos adentremos un poco más en el mundo de Elixir. Enumeraremos las características de su arquitectura, del lenguaje, sus ventajas y sus debilidades.

1. ¿Qué es Elixir?

Podríamos definir Elixir como un lenguaje funcional con capacidad de meta-programación. La mayor parte del lenguaje Elixir está definido en Elixir. El lenguaje permite construcciones muy flexibles como veremos a lo largo de los siguientes capítulos.

Junto con otros lenguajes como Ruby se dice que Elixir tiene mucho azúcar sintáctico[1]. Este término fue acuñado por Peter J. Landin en 1964 para referirse a todas esas construcciones del lenguaje que no aportan ninguna nueva funcionalidad pero ayudan enormemente en la facilidad de escribir código.

El azúcar sintáctico presente en Elixir es muy abundante. Como dijimos antes, la mayor parte de Elixir está escrito en Elixir y es gracias a su sistema de meta-programación. Todo gira en torno a las funciones, las macros y los flujos de datos.

[1] https://es.wikipedia.org/wiki/Az%C3%BAcar_sint%C3%A1ctico

En otros lenguajes estructuras condicionales como ***if*** pertenecen al propio lenguaje. En Elixir esta estructura condicional está definida a través del propio lenguaje. Veremos esto en profundidad en el Capítulo4, ***Las funciones y los módulos***.

Los lenguajes reservan una serie de palabras clave. No podemos usar estas palabras reservadas como nombres de variables, funciones, clases u otro tipo de construcción que nos permita el lenguaje. En lenguajes como Cobol podemos encontrar hasta 400 palabras reservadas, C++ dispone de 84 palabras reservadas, Java y Rust tienen 52, Ruby 41, Erlang tan solo 26 pero uno de los lenguajes con menos palabras reservadas es precisamente Elixir con tan solo 8.

Al igual que Erlang en Elixir podemos emplear los procesos de BEAM. Esto dota al lenguaje de capacidad para el desarrollo de soluciones de alta concurrencia, alta capacidad para aceptar gran cantidad de peticiones simultáneas, tolerancia a fallos y además nos permite emplear código realizado en Erlang/OTP dada la gran interoperatividad que tiene Elixir con Erlang/OTP.

En resumen podemos decir que Elixir es un lenguaje orientado a la meta-programación, con base funcional, diseñado para construir aplicaciones a nivel de servidor tolerantes a fallos, concurrentes y de alta capacidad. Pero aún hay mucho más porque como veremos más adelante una de las potencias de Elixir es su extensibilidad y esto nos proporciona una versatilidad a la hora de desarrollar increíble.

2. Características de Elixir

Para definir Elixir hemos nombrado muchas de las características de que dispone el lenguaje. En este sentido me gustaría separar lo que nos permite realizar el lenguaje de lo que nos permite realizar su máquina virtual (BEAM) y es común a todos los lenguajes compilados y que se ejecutan sobre ella.

2.1. Características de BEAM

Comenzando desde la base podemos decir que BEAM es una máquina virtual desarrollada y mantenida en producción por muchas empresas desde hace muchos años. Nació para dar soporte a sistemas de telecomunicaciones con unas características basadas en la tolerancia a fallos, alto nivel de concurrencia, alta capacidad, tiempo real blando y actualización en caliente para evitar la parada de los sistemas.

Vamos a explicar un poco más qué significa cada una de estas características:

Tolerancia a fallos

El código se aisla completamente en unidades mínimas de ejecución llamadas procesos. Un proceso es iniciado y debido a un fallo puede ser terminado. Al mantener cada proceso aislado y sin memoria compartida BEAM nos garantiza la consistencia en la memoria interna de cada uno de los procesos no afectados por el error. De esta forma la parte afectada por el error puede ser desechada y/o reiniciada y el sistema seguir funcionando correctamente.

Alto Nivel de Concurrencia

Al no existir memoria compartida los recursos no son compartidos en ningún momento. El modelo de protección de sección crítica se basa en mantener la unidad de ejecución conjutamente con los datos que maneja y ningún otro código puede acceder a esa información directamente. Solo a través de ese código. Eso nos garantiza un control de acceso a información compartida y evita situaciones de volatilidad de datos.

Alta Capacidad

BEAM define unos procesos internos mucho más ligeros que los proporcionados por el sistema operativo. Esto nos permite generar tantos procesos como necesitemos. De hecho la mayoría de sistemas operativos limita el número de procesos a 65536 (16 bits) mientras que los procesos que pueden ser creados en BEAM son un poco más de 134 millones. Esta característica dota a BEAM de la capacidad para poder atender a miles y millones de peticiones con un coste computacional menor.

Tiempo Real Blando

Al no contener memoria compartida el recolector de basura de BEAM no necesita bloquear ningún proceso para poder actuar. Esto se traduce en la eliminación de cortes en el funcionamiento normal de la máquina virtual y poder trabajar en tiempo real blando. Además veremos más adelante cómo mide el tiempo BEAM para poder proporcionar mediciones reales de tiempo y no basadas en los continuos cambios de hora que realizan los sistemas informáticos.

Actualización en Caliente

BEAM es de las únicas máquinas virtuales que dispone de un sistema seguro para el cambio de código en caliente. Permite modificar el código de un módulo aún manteniendo una copia

antigua para los procesos que aún se mantienen en ejecución e ir migrando de forma segura a la nueva versión del módulo sin tirar el sistema[2].

2.2. Características de Elixir

En la definición adelantamos muchas de las características de Elixir como lenguaje. Una de sus grandes potencias es la meta-programación que nos ayuda a evitar repetir una y otra vez los mismos códigos y reduce al mínimo las típicas plantillas de código a completar cuando se usan frameworks. Parte del código se ejecuta en tiempo de compilación con objetivo de generar otro código más completo o preciso y así mejorar el rendimiento en tiempo de ejecución.

En palabras de José Valim el lenguaje Elixir es un lenguaje dinámico enfocado en la productividad, en la extensibilidad y en la concurrencia.

Lenguaje Dinámico

Es de tipado dinámico y no es fuertemente tipado. Los tipos son opcionales y con fines de comprobación y documentación. Al igual que en Erlang podemos definirlos como una especificación adicional. Veremos esto más adelante.

Productividad

Al escribir código que genera código permite eliminar la necesidad de escribir muchas partes repetitivas y agregar construcciones específicas en lógica de negocio facilitándonos y reduciendo la generación de código. Pero con esta declaración José va aún más allá e incluye la necesidad de tener documentación de primera clase y un conjunto de herramientas como **mix** para la creación, administración y construcción de los proyectos, ***ExUnit*** para la generación de pruebas unitarias, **iex** y el repositorio de paquetes *Hex*.

Extensibilidad

A través de la definición de macros, guardas, funciones, sigilos y sobretodo protocolos nos permite agregar muchas más construcciones al lenguaje. Además José hace hincapié en las estructuras y el polimorfismo como otra piedra angular para la extensibilidad. Este aspecto es muy importante porque el foco de

[2]El cambio en caliente no es una característica que se use mucho actualmente pero para sistemas ***non-stop*** puede eliminar muchos dolores de cabeza y permitirnos cumplir mejor los SLA si diseñamos bien el sistema.

Elixir está en proveer a los programadores de las herramientas necesarias para adaptar el lenguaje a su modelo de dominio[3].

Concurrencia

Dada por BEAM. A través de comportamientos definidos desde la base a través de OTP[4] el lenguaje construye elementos permitiendo mantener la base de paso de mensajes, mantener la sección crítica y facilitar la generación de miles y millones de procesos.

Por mi parte considero la interoperatividad con Erlang otra de las grandes características del lenguaje. Aún teniendo muchas más características que el primero es posible sin mucho problema agregar código desde Erlang y emplearlo desde Elixir de forma fácil.

Además otra de las grandes características facilitadas en el lenguaje Elixir es la evaluación perezosa y/o la creación de generadores. Mientras que otros lenguajes requieren de obtener toda la información y todos los datos para poder proceder al siguiente paso del algoritmo, Elixir nos da la posibilidad de evaluar los elementos uno a uno en la cadena de proceso del algoritmo ahorrando así memoria y tiempo de procesador.

3. Historia de BEAM

Joe Armstrong asistió a la conferencia de Erlang Factory de Londres, en 2010, donde explicó la historia de la máquina virtual de Erlang. En sí, es la propia historia de Erlang/OTP. Sirviéndome de las diapositivas[5] que proporcionó para el evento, vamos a dar un repaso a la historia de Erlang/OTP.

La idea de Erlang surgió por la necesidad de Ericsson de acotar un problema que había surgido en su plataforma AXE, que estaba siendo desarrollada en PLEX, un lenguaje propietario. Joe Armstrong junto a dos colegas, Elshiewy y Robert Virding, desarrollaron una lógica concurrente de programación para canales de comunicación. Esta álgebra de telefonía permitía a través de su notación describir el sistema público de telefonía (POTS) en tan solo quince reglas.

A través del interés de llevar esta teoría a la práctica desarrollaron modelos en Ada, CLU, Smalltalk y Prolog entre otros. Así descubrieron que el álgebra telefónica se procesaba de forma muy rápida en sistemas de alto nivel, es decir, en Prolog, con lo que comenzaron a desarrollar un sistema determinista en él.

[3] El modelo de dominio es el ***lenguaje*** propio de negocio hablado por la empresa o programadores. Puedes encontrar una definición un poco más extensa aquí: https://altenwald.org/2010/04/26/modelo-de-dominio-la-importancia-de-los-nombres/.
[4] OTP son las siglas para ***Open Telecom Platform*** construido como framework de Erlang y provee la base para supervisores, servidores, máquinas de estados, generadores de eventos y aplicaciones.
[5] http://www.erlang-factory.com/upload/presentations/247/erlang_vm_1.pdf

La conclusión a la que llegó el equipo fue que, si se puede resolver un problema a través de una serie de ecuaciones matemáticas y portar ese mismo esquema a un programa de forma que el esquema funcional se respete y entienda tal y como se formuló fuera del entorno computacional, puede ser fácil de tratar por la gente que entiende el esquema, incluso mejorarlo y adaptarlo. Las pruebas realmente se realizan a nivel teórico sobre el propio esquema, ya que algorítmicamente es más fácil de probarlo con las reglas propias de las matemáticas que computacionalmente con la cantidad de combinaciones que pueda tener.

Prolog no era un lenguaje pensado para concurrencia, por lo que se decidieron a realizar uno que satisficiera todos sus requisitos, basándose en las ventajas que habían visto de Prolog para conformar su base. Erlang vio la luz en 1986, después de que Joe Armstrong se encerrase a desarrollar la idea base como intérprete sobre Prolog, con un número reducido de instrucciones que rápidamente fue creciendo gracias a su buena acogida. Básicamente, los requisitos que se buscaban cumplir eran:

- Los procesos debían ser una parte intrínseca del lenguaje, no una librería o framework de desarrollo.
- Debía poder ejecutar desde miles a millones de procesos concurrentes y cada proceso ser independiente del resto, de modo que si alguno de ellos se corrompiese no dañase el espacio de memoria de otro proceso. Es decir, el fallo de los procesos debe ser aislado del resto del programa.
- Debe poder ejecutarse de modo ininterrumpido, lo que obliga a no detener su ejecución para actualizar el código del sistema. Recarga en caliente.

En 1989, el sistema estaba comenzando a dar sus frutos, pero surgió el problema de que su rendimiento no era el adecuado. Se llegó a la conclusión de que el lenguaje era adecuado para la programación que se realizaba pero tendría que ser unas 40 veces más rápido como mínimo.

Mike Williams se encargó de escribir el emulador, cargador, planificador y recolector de basura (en lenguaje C) mientras que Joe Armstrong escribía el compilador, las estructuras de datos, el heap de memoria y la pila; por su parte Robert Virding se encargaba de escribir las librerías. El sistema desarrollado se optimizó a un nivel en el que consiguieron aumentar su rendimiento en 120 veces de lo que lo hacía el intérprete en Prolog.

En los años 90, tras haber conseguido desarrollar productos de la gama AXE con este lenguaje, se le potenció agregando elementos como distribución, estructura OTP, HiPE, sintaxis de bit o compilación de patrones para ***matching***. Erlang comenzaba a ser una gran pieza de

software, pero tenía varios problemas para que pudiera ser adoptado de forma amplia por la comunidad de programadores. Desafortunadamente para el desarrollo de Erlang, aquel periodo fue también la década de Java y Ericsson decidió centrarse en ***lenguajes usados globalmente*** por lo que prohibió seguir desarrollando en Erlang.

Nota

HiPE es el acrónimo de ***High Performance Erlang*** (Erlang de Alto Rendimiento) que es el nombre de un grupo de investigación sobre Erlang formado en la Universidad de Uppsala en 1998. El grupo desarrolló un compilador de código nativo de modo que la máquina (BEAM) virtual de Erlang no tenga que interpretar ciertas partes del código si ya están en lenguaje máquina mejorando así su rendimiento.

Con el tiempo, la imposición de no escribir código en Erlang se fue olvidando y la comunidad de programadores de Erlang comenzó a crecer fuera de Ericsson. El equipo OTP se mantuvo desarrollando y soportando Erlang que, a su vez, continuó sufragando del proyecto HiPE y aplicaciones como EDoc o Dialyzer.

Antes de 2010 Erlang agregó capacidad para SMP y más recientemente para multi-core. La revisión de 2010 del emulador de BEAM se ejecuta con un rendimiento 300 veces superior al de la versión del emulador en C, por lo que es 36.000 veces más rápido que el original interpretado en Prolog. Cada vez más sectores se hacen eco de las capacidades de Erlang y cada vez más empresas han comenzado desarrollos en esta plataforma por lo que se augura que el uso de este lenguaje siga al alza.

4. Comenzando la idea de Elixir

El nombre de Elixir no tiene un origen definido. José Valim dijo en la Elixir Conf de 2014[6] que no tuvo una razón específica para elegir el nombre. No obstante y gracias a su símil con la mezcla de elementos para la fabricación de elixires los programadores se atribuyen el nombre de alquimistas, la herramienta de construcción el nombre de ***mix*** (mezclar) y el gestor de paquetes ***hex*** (maleficio). Además del nombre de conocidas librerías como ***poison*** (veneno) para el tratamiento de JSON o ***distillery*** (destilería) para el empaquetado de un lanzamiento.

El autor del lenguaje ha concedido cientos de entrevistas y ha hablado sobre el lenguaje, cada liberación y cada nueva característica de forma repetida en varios medios de comunicación. Sobre todo en charlas como la Erlang User Conference de 2013[7] y más recientemente las ElixirConf[8].

[6] https://www.youtube.com/watch?v=aZXc11eOEpI
[7] https://www.meetup.com/es-ES/Stockholm-Erlang-Meetup/events/117945802/
[8] https://www.elixirconf.eu/elixirconf2015

José Valim ha sido uno de los contribuidores más activos a Rails y ante los problemas existentes para poder hacer funcionar Rails con websockets o la seguridad entre hilos al ejecutar Rails en un entorno con múltiples núcleos se decidió a probar nuevas vías. Nuevas máquinas virtuales. Ruby es un lenguaje que desde hace bastante tiempo tiene varias implementaciones y sus usuarios según sus necesidades y gustos pueden elegir una u otra base para desarrollar sus aplicaciones.

José Valim mencionó en su charla un estudio de Herb Sutter que acuñó la famosa frase ***se acabó la comida gratis***[9] haciendo mención al hecho del cambio de paradigma al crear los nuevos procesadores donde no se incrementaría más la velocidad del procesador sino el número de núcleos.

José Valim pertenecía al núcleo de desarrollo de Ruby on Rails por esa época y se encontró de frente con el problema. Ruby no estaba preparado para emplear diferentes núcleos en el sistema operativo y las soluciones desarrolladas hasta 2009 eran simples parches para intentar solventar el problema sin mucho éxito.

Al principio la idea de Elixir fue construir un Ruby sobre BEAM. Tal y como había intentado hacer Reia[10] anteriormente. José Valim se basó en Reia para iniciar Elixir[11] y cuando Elixir comenzó a incrementar la velocidad de su desarrollo fue cuando el desarrollador principal de Reia consideró oportuno abandonar su proyecto y dar su apoyo a Elixir.

Hay que considerar importante el apoyo de la empresa Plataformatec[12] donde José Valim trabajó durante la gestación del lenguaje. Según cuenta José Valim en charlas como la mencionada Elixir Conf de 2014 la empresa financió su trabajo en el lenguaje durante dos años hasta convertirlo en un lenguaje de alto interés y muy aceptado por muchos programadores del sector.

Desde mi punto de vista considero que hubo un antes y un después de la lectura del artículo de Joe Armstrong[13] en mayo de 2013 hablando de la excitación de Dave Thomas por Elixir y la gestación del primer libro sobre Elixir[14]. En el mes siguiente O'Reilly a través de Simon Saint Laurent publicó Introducing Elixir[15]. El lenguaje comenzó a ganar masa crítica y comienzan a aparecer muchos más eventos sobre el lenguaje.

[9] Traducido de la frase en inglés original: ***free lunch is over***.
[10] http://reia-lang.org/
[11] http://www.unlimitednovelty.com/2011/06/why-im-stopping-work-on-reia.html
[12] http://plataformatec.com.br/
[13] https://joearms.github.io/published/2013-05-31-a-week-with-elixir.html
[14] https://pragprog.com/book/elixir/programming-elixir
[15] http://shop.oreilly.com/product/0636920030584.do

5. Desarrollos con Elixir

A nivel empresarial Elixir es usado por empresas tan grandes como Adobe, AdRoll, Lexmark, PagerDuty, Pinterest o Slack[16].

Dado su origen el principal uso de Elixir está en la elaboración de servicios web y websocket para servicios en Internet. Uno de los grandes frameworks que posibilita y facilita esta adopción es Phoenix Framework[17]. Quizás por su influencia este framework tiene mucha semejanza con Ruby on Rails[18] pero difiere en muchos aspectos y ha conseguido hitos bastante notables como veremos en la siguiente sección.

6. Soportando 2 Millones de Usuarios Conectados

A finales de 2015 Chris McCord[19] comenzó a publicar una serie de tuits donde mencionaba cómo estaba probando e intentando alcanzar 2 millones de usuarios concurrentes[20] conectados por websocket en Elixir y más específicamente en Phoenix Framework.

Todo comenzó cuando Gary Rennie[21] estaba haciendo pruebas sobre cuántos canales simultáneos podría soportar Phoenix Framework. Conseguía un máximo de mil (1000) en su máquina local. Al comentarlo vía IRC y ante la falta de bancos de pruebas los desarrolladores del núcleo de Phoenix Framework quisieron arrojar un poco de luz sobre cómo realizar las pruebas y aportar números.

Rackspace aportó tres máquinas de 15GB I/O v1 con 15GB de RAM y 4 núcleos cada una. También tuvieron acceso a un OnMetal I/O de 128GB con 40 núcleos.

Las pruebas para generar el número crítico de clientes se realizaron a través de Tsung[22], un sistema para realizar bancos de pruebas en diferentes protocolos y generar una alta carga. La primera configuración probada fue con un servidor configurado para aceptar las peticiones y los otros dos para generar el tráfico cliente usando Tsung.

Se alcanzaron 27 mil conexiones simultáneas. José Valim hizo una corrección en el código de Phoenix para acelerar la entrada de usuarios.

[16] http://elixir-companies.com/
[17] http://phoenixframework.org/
[18] https://rubyonrails.org/
[19] https://github.com/chrismccord
[20] http://phoenixframework.org/blog/the-road-to-2-million-websocket-connections
[21] https://github.com/Gazler
[22] http://tsung.erlang-projects.org/

Se repitieron las pruebas y esta vez se alcanzaron 50 mil conexiones simultáneas.

A través de una visualización de los procesos mediante ***observer***[23] Chris pudo detectar otro cuello de botella. Chris eliminó el sistema de heartbeat de la conexión de websocket al ver que se duplicaba el número de temporizadores en uso. Tanto el sistema de websocket como cowboy implementaban un temporizador. Dejó únicamente el provisto por cowboy en una nueva modificación de Phoenix Framework. De nuevo ejecutaron las pruebas y esta vez llegaron a 100 mil conexiones simultáneas.

Debido a las limitaciones de tener solo dos máquinas para clientes y solo poder generar 40 mil y 60 mil conexiones simultáneas respectivamente, emplearon otra máquina de 128GB disponible para conseguir más clientes y seguir las pruebas. Esta vez consiguieron 330 mil conexiones simultáneas.

Gabi Zuniga[24] echó un vistazo y agregó una corrección al sistema. Empleando otro tipo de dato para las tablas ETS consiguieron llegar a 450 mil conexiones simultáneas.

La empresa Live Help Now[25] aportó a la prueba 45 máquinas en Rackspace para proseguir gracias a la participación de Justin Schneck[26] en las pruebas. Configuraron Tsung en unas cuantas máquinas y lo limitaron a 1 millón de conexiones simultáneas para hacer las pruebas de nuevo. Terminaron de configurar el resto de las 45 máquinas para probar si era posible llegar a 2 millones de conexiones en una sola máquina. Desafortunadamente se quedaron en 1,3 millones de conexiones. Nada mal no obstante.

Pero algo no estaba bien. A esos niveles de carga los mensajes para los suscriptores se demoraban más de 5 segundos. Tras varios análisis Chris tuvo la idea de generar un concentrador (pool) para pubsub y Justin la de fraccionar (shard) la tabla ETS donde se almacenaba todo. Esta combinación permitió reducir el tiempo de difusión a un segundo y alcanzar 2 millones de conexiones simultáneas.

Cada escenario concreto tiene sus propias limitaciones y como hemos visto pasar de mil conexiones simultáneas a dos millones ha tenido una travesía de análisis y correcciones permitiendo a los desarrolladores ir mejorando cada vez más el sistema. José, Chris, Gary y todos los implicados quedaron satisfechos con la cifra alcanzada aunque comenta

[23] Una aplicación de Erlang/OTP que permite obtener información en tiempo real del sistema en ejecución a través de una interfaz gráfica de usuario (GUI).
[24] https://twitter.com/gabiz
[25] http://www.livehelpnow.net/
[26] https://twitter.com/mobileoverlord

también que tras la publicación siguieron encontrando otras mejoras para poder aplicar.

La buena noticia para todos es que estas mejoras y las futuras que se han ido descubriendo han ido formando parte de Elixir y/o Phoenix Framework y eso nos garantiza un sistema robusto y con un alto grado de escalabilidad horizontal.