Gestión de Tecnologías Informáticas en las Organizaciones

EventPlanner

Memoria de Proyecto

Paul Vega

Sonia Elizondo

Sonia Elizondo

Xabier Dendarieta

Índice

[Alcance 3](#_Toc41557314)

[Características técnicas 4](#_Toc41557315)

[Recursos 5](#_Toc41557316)

[Tiempo 6](#_Toc41557317)

[Estimación inicial 6](#_Toc41557318)

[Estimación final 6](#_Toc41557319)

[Metodología 7](#_Toc41557320)

[Requisitos de componentes lógicos 8](#_Toc41557321)

[Anexo 10](#_Toc41557322)

[ADR Base de Datos 10](#_Toc41557323)

[Contexto 10](#_Toc41557324)

[Procesos 10](#_Toc41557325)

[Opciones consideradas 10](#_Toc41557326)

[Decisión 11](#_Toc41557327)

[Estado 11](#_Toc41557328)

[Implicaciones 11](#_Toc41557329)

[ADR Gestión de la Configuración 13](#_Toc41557330)

[Contexto 13](#_Toc41557331)

[Procesos 13](#_Toc41557332)

[Opciones consideradas 14](#_Toc41557333)

[Decisión 14](#_Toc41557334)

[Estado 15](#_Toc41557335)

[Implicaciones 15](#_Toc41557336)

[ADR Integración Continua 16](#_Toc41557337)

[Contexto 16](#_Toc41557338)

[Procesos 16](#_Toc41557339)

[Opciones consideradas 17](#_Toc41557340)

[Decisión 19](#_Toc41557341)

[Estado 19](#_Toc41557342)

[Implicaciones 19](#_Toc41557343)

# Alcance

La idea de este proyecto se basa en el desarrollo de una aplicación web para **gestionar la asistencia a eventos sociales** de pequeñas proporciones, entre los que se encuentran comidas, cenas, reuniones familiares y ocio similar.

El funcionamiento es sencillo: cada usuario puede **crear** los eventos que considere necesarios y enviar una **invitación** del evento a los usuarios que considere oportuno. Éstos, por su parte, pueden indicar si **asisten o no asisten** a cada evento, de forma que se almacenen los **asistentes en forma de listado**. De este modo, el creador de dicho evento podrá gestionar de forma muy sencilla el **número de asistentes** con los que contará y la **lista de sus nombres**.

Para ello necesitaremos tres **componentes** principales: el **cliente web**, el **servidor** y la **base de datos**. El cliente estará pensado para ejecutarse en el lado del usuario, y tanto el servidor como la base de datos, se alojarán en la máquina virtual de la que disponemos en la asignatura.

# Características técnicas

* El nombre de la aplicación web elegido es **EventPlanner**.
* Para el diseño del portal web se utilizará Bootstrap junto con HTML/CSS y JavaScript. Se han decidido estas tecnologías por las siguientes razones:
  + **Bootstrap:** es una herramienta de prototipado rápido para clientes web, por lo que nos ahorrará una cantidad considerable de tiempo a la hora de implementar nuestro cliente. Entre sus ventajas, nos permite crear proyectos receptivos y móviles de forma sencilla y completamente integrada con HTML y es de código abierto. Su uso genera resultados rápidos y visualmente elegantes.
  + **JavaScript:** es necesario en el lado del usuario para añadir funcionalidades en el cliente web sin tener que recargar la página completa. Hay un conjunto grande de librerías en este lenguaje que nos facilitarán el trabajo de implementación.
* El portal web se comunicará con el servidor web mediante una **API** (*backend*) desarrollada en el lenguaje de programación **Node.js**. Esta API será de tipo **REST**. Se ha decidido esta tecnología por lo siguiente:
  + **Node.js:** este *framework* es conocido por los desarrolladores y es una alternativa muy sencilla ante la mayoría de los lenguajes y *frameworks* disponibles. Se tuvo en consideración el uso de Python con la librería *Flask*, pero se descartó porque, a pesar de ser muy similar a la anterior, Node.js se escribe en JavaScript, al igual que nuestro cliente, y ofrece una API estable para la librería LevelDB.
* Toda la información sobre los usuarios y los eventos se gestionará mediante una base de datos llamada **LevelDB** en forma local en la máquina virtual. Se ha decidido esta tecnología por lo establecido en el [ADR sobre la Base de Datos](#_ADR_Base_de).

# Recursos

Para la implementación del sistema propuesto se hará uso de **una de las máquinas virtuales proporcionadas** para esta asignatura como principal medio de almacenamiento y despliegue. La máquina virtual citada también hará la función de servidor y contendrá la base de datos, como ya hemos comentado antes. Consideramos que son recursos suficientes y no disponemos de medios para obtener más (AWS u otras plataformas).

Además de lo anterior, necesitaremos diferentes **editores de texto** o **IDEs** para la programación del proyecto. Puesto que los lenguajes de programación que hemos propuesto son tan sencillos y populares, hay una variedad muy amplia de editores e IDEs disponibles y todos son compatibles entre sí. Con el fin de que los desarrolladores estén lo más cómodos posibles durante el desarrollo, se deja a su elección tal herramienta.

# Tiempo

## Estimación inicial

Estimamos una primera demostración de funcionalidad para el día 22 de abril, miércoles. Estimamos también la finalización del proyecto para la última semana de mayo. La fecha de finalización última queda pendiente de concretarse más, mediante una estimación de tiempo más detallada.

## Estimación final

Como seguimiento del proyecto se han realizado dos demostraciones de funcionalidad. En ellas, se ha expuesto la implementación conseguida en cada momento, incluyendo los avances conseguidos respecto a los procesos de integración continua acordados en el ADR correspondiente.

La demostración final se realiza el 1 de junio, es decir, es la fecha final de entrega. El proyecto se prevé que no estará completado con todas las funcionalidades que se quisieron prestar en un principio, pero se trabaja de forma continuada para lograr una versión básica estable que pueda utilizarse de la forma esperada (mediante un explorador web).

Los detalles considerados en la sección “[Requisitos de Despliegue](#_Requisitos_de_Despliegue)” pretenden ser conseguidos mediante la creación de una *pipeline* completa que guíen la disposición final de la aplicación, desde los tests que se consideren necesarios hasta el despliegue en contenedores Docker para proporcionar el servicio.

# Metodología

Seguimos una metodología basada en la **metodología ágil** junto con **Kanban**.

Estamos siempre en contacto mediante un **servidor de Discord** exclusivamente dedicado a este proyecto. En ese servidor hemos habilitado tres chats de texto y uno de voz. De este modo, la comunicación entre los integrantes del proyecto es cercana y continua. Hemos elegido Discord por ser una plataforma fiable, sencilla y con un amplio abanico de posibilidades para comunicarse.

El **chat de texto principal** (llamado simplemente chat) es el que se utiliza para comunicación general. El **chat de texto secundario** (llamado notificaciones) se utiliza con el único fin de que cada desarrollador notifique al resto del equipo los eventos que van ocurriendo, por ejemplo, el completar una tarea importante. El **tercer chat de** texto (llamado reuniones) se utiliza exclusivamente para dejar por escrito las fechas y horas de las reuniones planificadas por el equipo (a realizar en el propio Discord). El **chat de voz** se utiliza para realizar reuniones. Es posible, en caso de necesitarlo, activar webcam o compartir pantalla.

Las **reuniones** antes citadas son solicitadas cuando al menos uno de los desarrolladores lo ve necesario, para resolver cualquier duda o incidencia. Además de las convocadas de manera más improvisada, se realiza una reunión cada dos semanas (día y hora a concretar según disponibilidad) para hacer un seguimiento global del proyecto y poner al día al equipo, con intención también de asignar a cada uno nuevas tareas de desarrollo o documentación.

También incorporamos a nuestra metodología un **tablón de tareas**, extraído de la idea de Kanban. La herramienta utilizada es **Trello**, con la que podemos crear tarjetas y organizarlas de una forma muy intuitiva. Además, dicha herramienta es ya conocida por los desarrolladores, por tanto, no es necesario que inviertan excesivo tiempo en aprender a utilizar una diferente.

Para mantener el desarrollo lo más organizado posible en cuanto a los componentes programados y la documentación generada, se trabaja en la plataforma **GitHub**. Las especificaciones de la herramienta y la justificación de uso frente a otras alternativas se encuentran en el [ADR de Gestión de la Configuración](#_ADR_Gestión_de).

Con todo esto tenemos una metodología que refuerza la comunicación entre desarrolladores y ayuda a mantener el proyecto organizado en todas sus vertientes, tanto en el caso del código como en el caso de las tareas a realizar.

# Requisitos de Despliegue

Se necesita definir la forma en la que desplegar la aplicación EventPlanner. Para especificar los componentes lógicos que van a tomar parte en dicho despliegue, nos basamos en la arquitectura de la aplicación y en las características más importantes para mantener el servicio:

* La arquitectura planteada consta de tres servidores básicos: el interfaz web, la API y la Base de Datos.
* Estos servidores deben mantener su actividad de forma continua para poder prestar el servicio que se pretende, es decir, se requiere alta disponibilidad.
* Además, la aplicación debe ser escalable: continuar prestando el servicio a pesar de que haya una alta carga de peticiones por parte de los usuarios.
* Siendo el objetivo de la aplicación gestionar asistentes a eventos sociales, utiliza de forma continua datos de usuarios y de eventos. Dichos datos deben permanecer almacenados esté o no el servicio en ejecución.

Para lograr las pautas anteriores, se plantea el siguiente modo de despliegue:

* Utilización de Docker como herramienta para el despliegue de los componentes en contenedores.
* Realización de una imagen de contenedor distinta por cada tipo de servidor de la aplicación (por tanto, tres tipos distintos).
* Para asegurar la alta disponibilidad:
  + Empleo de mínimo dos contenedores por cada componente de despliegue.
* Para asegurar la escalabilidad:
  + Alternancia de peticiones entre los contenedores disponibles, de modo que se reparta la carga de trabajo entre varios servidores en ejecución al mismo tiempo.
  + Es necesario contar con un balanceador de carga que distribuya de forma eficiente la carga, ya que no basta con un elegir un contenedor aleatoriamente (por código sencillamente), sino que se debe fijar mejor ese reparto (teniendo en cuenta, por ejemplo, que el tiempo de resolución de las peticiones *get* suele ser más elevado que el de las *put*).
* Para mantener los datos de forma persistente:
  + Creación y asignación de un volumen a los contenedores dedicados al servidor de Base de Datos.
* Para evitar posibles pérdidas de datos:
  + Presencia de varias réplicas de la Base de Datos en otras máquinas.
* Para la comunicación entre los componentes del servicio:
  + Configuración de una red *network* en Docker que permita la conexión entre ellos del modo ya establecido:
    - Base de Datos y API por sockets TCP (librería ‘zmq’).
    - Cliente web y API por protocolo HTTP.

Los anteriores requisitos se establecen para un entorno de producción, digamos final. Pero también es necesario definirlos para las pruebas a realizar antes de desplegar la aplicación como tal:

* Para los tests unitarios de cada componente, es suficiente con generar una imagen del tipo que corresponda y lanzar los tests desde dentro del propio contenedor; de modo que las conexiones no supongan un problema.
* Para los tests de rendimiento sí que se establece una composición de contenedores como la prevista para producción, debido a que es importante conocer el comportamiento que va a tener el sistema de la forma más parecida a la realidad posible.
* Para los tests de carga, indicamos lo mismo que anteriormente, sólo que prestando más atención al modo en el que se reparten dicha carga los contenedores del mismo tipo conforme aumenta el número de peticiones.

# Problemas encontrados

Es importante resaltar aquellas incidencias surgidas durante el desarrollo del proyecto que han podido causar retrasos en el cumplimiento de hitos estipulados por el equipo de forma interna en sus reuniones para la obtención de una aplicación funcional para inicios de junio.

## Instalación de librerías Node.js

En primer lugar, para el desarrollo del código propiamente dicho, es imprescindible que Node.js esté instalado correctamente en la máquina virtual (elegida para el despliegue de la aplicación), para probar que funcionen correctamente los componentes basados en tal lenguaje y, en caso contrario, corregir los posibles errores de implementación.

Uno de los elementos cruciales respecto a librerías es la Base de Datos, puesto que se construye por completo sobre ‘LevelDB’, la librería para manejo y almacenamiento de datos. Además, también requiere ‘zmq’ para la conexión con la API.

A la hora de instalar las citadas librerías, se encontraba una excepción poco habitual y a la que no se sabía poner solución en un primer momento. Por lo tanto, se necesitaba buscar información al respecto para paliar el error y poder hacer uso de dichas librerías, ya que no se podía trabajar sin ellas. Tras mucho tiempo de búsqueda, siendo un tiempo dedicado a resolución en lugar de a implementación, se resolvió el problema con los siguientes comandos:

sudo apt-get update -y

sudo apt-get install -y pkg-config

Aunque se invirtió tiempo importante para el desarrollo, se llegó a una solución satisfactoria. Es preferible dejarla plasmada en este documento para posibles situaciones similares en otros entornos de despliegue.

## Comprobación automática de Jenkins

Jenkins es la herramienta elegida para llevar a cabo los procesos de integración continua del desarrollo de la aplicación.

Para ello, se dispuso un pipeline en la que realizar desde los tests unitarios de los componentes acordados hasta el despliegue en contenedores de Docker.

La primera aproximación realizada fue la de comprobar cada 15 minutos si el repositorio de GitHub que albergaba el código había registrado algún cambio en el mismo, de modo que se clonase dicho repositorio al proyecto de Jenkins al cargo.

Dicha funcionalidad básica, fue sustituida por una *pipeline* al uso, la cual quería ser utilizada para los primeros pasos de la integración: comprobar cambios en el repositorio, clonarlo, instalar las dependencias necesarias y ejecutar los tests unitarios de la Base de Datos. Dichos pasos se establecieron en formato código en un Jenkinsfile, establecido posteriormente en el propio repositorio, para enlazarlos.

El proceso de configuración parecía satisfactorio debido a que se producían los pasos correctamente y, además, los tests indicaban que los datos de prueba se almacenaban correctamente, al igual que podían ser consultados.

Pero, se comprobó al poco tiempo que dicha *pipeline* presentaba un error de configuración, no en su código: la comprobación cada 15 minutos siempre llevaba a ejecutar el resto de pasos, hubiese o no cambios en el repositorio.

*Todavía se está buscando una solución a este problema, para poder dejar la pipeline completamente automatizada.*

# Anexo

## ADR Base de Datos

### Contexto

Es necesario definir qué tipo de Base de Datos se va a utilizar para almacenar y gestionar los datos dispuestos en la aplicación.

### Procesos

1. Interacción continua entre las tres componentes de la aplicación para que la Base de Datos realice su cometido (utilizando JSON para dar estructura a los datos en los envíos):
   1. El usuario realiza una operación desde el cliente web (registrarse, identificarse, crear un nuevo evento, indicar que se asiste a un evento, etc.).
   2. El cliente web manda la acción correspondiente a la API con los datos proporcionados por el usuario.
   3. La API se comunica con la Base de Datos para indicar la petición a realizar en ella (get o put, es decir, guardar o devolver datos) con los datos disponibles.
   4. La Base de Datos almacena datos nuevos o devuelve los almacenados según haya sido realizada la petición.
      1. Los datos críticos deben estar cifrados (en este caso, únicamente la contraseña del usuario se considera relevante).
      2. No se deben perder datos debido a errores de gestión.
   5. La Base de Datos devuelve información a la API, ya sea como confirmación de éxito de una operación como de los datos requeridos.
   6. La API envía la información obtenida al cliente web.
   7. El cliente web se actualiza para mostrarse del modo que corresponda (cambio de página, confirmación de operación realizada, etc.).

Cabe resaltar las características que se requieren para el sistema, de modo que, aunque no se trata de un proceso como tal, se tengan en cuenta para la decisión: alta disponibilidad, escalabilidad y almacenamiento persistente (lo cual conlleva a que no se produzcan pérdidas por fallos internos o de la red o por cualquier otra razón).

### Opciones consideradas

Aunque hay un amplio abanico de posibilidades para la implementación de una Base de Datos, incluyendo tanto locales como en la nube, se decide acotar la comparación a dos opciones: una muy completa pero complicada de dominar y otra muy básica pero sencilla de utilizar.

1. ElasticSearch:
   1. Pros:
      1. Permite escalabilidad horizontal.
      2. Está orientada a ficheros JSON.
      3. Incluye un sistema de *master-workers* distribuido en un clúster que facilita una alta disponibilidad con alta tolerancia a fallos.
      4. Proporciona un sistema de replicación automático.
   2. Cons: el equipo no está familiarizado con su uso.
2. LevelDB:
   1. Pros:
      1. Librería de Nodejs.
      2. Uso sencillo.
      3. Más de la mitad del equipo está familiarizado con su uso.
   2. Cons:
      1. Sólo permite almacenamiento de pares clave-valor.
      2. No facilita los requerimientos del sistema.

### Decisión

Se ha escogido **LevelDB** como Base de Datos para la aplicación.

La decisión ha sido tomada en base a que la aplicación ya se está implementando en JavaScript y Nodejs. Además, algunos de los desarrolladores ya tienen experiencia tratando con la librería, por lo que no va a ser necesario dedicar horas a formación del personal antes de comenzar la programación de la Base de Datos.

Los datos que se van a utilizar en la aplicación son muy básicos y no necesitan de estructuras relacionales muy complejas, por lo que la simpleza de esta librería es suficiente.

Por último, el alcance de este proyecto es reducido debido a los recursos tecnológicos y temporales de los que se dispone, por lo que conseguir un sistema con todas las características mencionadas anteriormente no es crucial en este momento; sino que, aun siendo básico, sea funcional.

### Estado

(Tachar las que no apliquen)

1. Aceptado
2. ~~Rechazado~~
3. ~~Reemplazado~~

### Implicaciones

1. Instalación de la librería LevelDB de Nodejs en la máquina virtual utilizada para el proyecto.
2. El sistema no cumplirá, en principio, las características requeridas.

Hay que destacar que, si el contexto fuese otro, ElasticSearch sería la mejor opción para la Base de Datos, dado que permitiría alcanzar las características principales para el sistema de forma satisfactoria y medianamente automática.

#### Fecha

21 de mayo. [La decisión no había sido plasmada como ADR]

## ADR Gestión de la Configuración

### Contexto

Se requiere de una herramienta donde almacenar los archivos relacionados con el proyecto que se vayan desarrollando (código fuente, documentación, etc.), así como una organización concreta para su orden y claridad.

### Procesos

* Creación de un repositorio disponible para todos los miembros del equipo.
  + Acceso a la nube por medio de navegador, línea de comandos o interfaz gráfico.
* Organización del repositorio.
  + Utilización de árbol de directorios para ubicar rápidamente los archivos.
    - Directorio “doc”: almacenar toda la documentación referente al proyecto, por ejemplo, los propios ADRs. El directorio constará de la siguiente estructura interna:
      * Directorio “actas”: en él figurarán las actas de reunión.
      * Directorio “ADR”: para almacenaje de los ADRs, junto con la plantilla.
      * Directorio “anotaciones”: en él se guardarán apuntes y notas arbitrarias.
      * Directorio “protocolo”: en él se almacenarán los documentos sobre los protocolos de comunicación acordados entre las diferentes partes del proyecto.
      * Además de todo esto, en el directorio “doc” también va a figurar el documento de la memoria completo.
    - Directorio “src”: almacenar todo el código desarrollado.
    - No se descarta la creación de más directorios conforme las necesidades del equipo así lo requieran.
  + Utilización de ramas (*branches*) para disponer el código:
    - Rama *master*: rama principal y única.
* Política de nombres en los archivos:
  + ADRs: deben comenzar con ADR y tras un ‘\_’ especificar con sustantivo (o varios en caso de ser necesario) el objetivo principal del mismo o tema principal tratado.
  + Pudieran añadirse más normas si así se requiriese (siempre dejando constancia).
  + Actas de reunión: deben comenzar con el número de acta que tiene asignado (secuencialmente) y debe estar representado en formato de dos cifras, seguidos de un ‘\_’, y finalizado por el día y mes de realización, ambos con formato de dos cifras también. Las actas estarán escritas en Markdown, por tanto, la extensión será *“.md”*.
  + El documento de la memoria estará en formato DOCX y llevará de nombre “memoria\_proyecto”.
* Acciones pertinentes a cambios en archivos:
  + Siempre que se realice un cambio, incluir un comentario breve con la descripción o motivo de lo realizado.
  + Realizar peticiones de cambios (*pull* *requests*), no actualizar sin control.
* Realización de pruebas sobre el código desarrollado:
  + Lanzar desde el acceso al repositorio (rama *master*) los tests pertinentes (tanto de integración como unitarios) para la comprobación del código desarrollado cada vez que se genere un cambio significativo.

### Opciones consideradas

1. GitHub:

* Pros:
  + Permite almacenar todos los tipos de archivos y de la forma establecida.
  + El equipo está familiarizado con su uso.
  + Permite realizar *pull requests*.
  + Se puede integrar con Trello (herramienta de gestión del equipo).
  + Disponemos de “Paquete Education”:
    - GitHub Pro gratuito: repositorios ilimitados, colaboradores ilimitados, etc.
  + Posibilidad de integración con Jenkins (herramienta para integración continua).
  + El equipo tiene cuentas registradas.

1. Bitbucket:

* Pros:
  + Gratuito para equipos de hasta 5 usuarios: repositorios privados ilimitados.
  + Se puede integrar con Trello (herramienta de gestión del equipo).
  + Permite realizar *pull requests*.
  + Posibilidad de integración con Jenkins (herramienta para integración continua).
* Cons:
  + El equipo no está familiarizado con su uso.
  + No se tienen cuentas ya registradas.

### Decisión

Se utiliza la herramienta **GitHub**.

No se han planteado más alternativas debido a que hay múltiples herramientas disponibles con características muy similares (Bitbucket es un ejemplo).

Se ha decidido en favor de GitHub no sólo por lo que ofrece (que, como se dice, es bastante común al resto) sino debido a que es completamente gratuito por el “Paquete Education” del que ya disfruta el equipo. Además, ya se tienen cuentas abiertas y utilizadas en la plataforma, por lo que su uso, al menos básico, es conocido.

### Estado

(Tachar las que no apliquen)

1. Aceptado
2. ~~Rechazado~~
3. ~~Reemplazado~~

### Implicaciones

Todos los miembros del equipo deben tener una cuenta registrada en la herramienta (ya se cuenta con ello) y acceso (incluyendo permisos de edición) al repositorio utilizado para el desarrollo del proyecto.

El uso de la herramienta seleccionada debe ser ágil y preciso ya que será utilizada a diario para la actualización del trabajo.

Además, se deben tener en cuenta las normas establecidas en este ADR en cuanto a la disposición del repositorio y sus componentes.

#### Fecha

27 de marzo de 2020

## ADR Integración Continua

### Contexto

Se necesita definir una serie de herramientas capaces de proporcionar una forma ágil de integrar los cambios en el proyecto. Asimismo, deben posibilitar que una vez integrados, todo funcione correctamente.

### Procesos

1. Instalación de requisitos previos para la integración y despliegue:
   1. Instalación de herramientas: Git, Docker, Jenkins. (Véanse apartados de “Opciones consideradas” y “Decisión”)
   2. Creación de un pipeline en la herramienta Jenkins. (Véase Proceso 3, “Integración de cambios en el entorno”)
2. Integración de cambios en el repositorio (git push):
   1. Todos los cambios deberán aplicarse a la rama *master* del repositorio, siguiendo la filosofía de integración continua.
   2. Los cambios deberán ir acompañados de un breve comentario que indique a grandes rasgos qué cambia.
3. Integración de cambios en el entorno (Pipeline de Jenkins):
   1. Cualquier cambio realizado en la aplicación se actualiza en la rama *master*. No se realizan ramificaciones.
   2. Tests unitarios: definir scripts simples para comprobar el correcto funcionamiento del código implementado. Se debe realizar un test por cada nuevo cambio en el código desarrollado y lanzarlos sobre rama Master.
   3. Si los test unitarios SÍ son satisfactorios:
      1. Se genera un ejecutable
      2. Se genera el correspondiente contenedor con Docker
   4. Si los test unitarios NO son satisfactorios:
      1. Revisa los test unitarios paras aplicar las respectivas correcciones.
   5. Tests de integración: comprobar que los cambios realizados no entren en conflicto con partes ya realizadas. Se debe realizar un test por cada nuevo cambio en el código desarrollado.
   6. En caso de que todo funcione correctamente, se procederá al despliegue. (Véase Proceso 4, “Despliegue de la aplicación”)
4. Despliegue de la aplicación (Docker):

(Aunque consten una serie de requisitos de despliegue, el tiempo y los recursos disponibles no permiten una preparación tan completa. Por ello, se especifica en este documento, el despliegue real que se pretende conseguir para el final del proyecto).

* 1. Realización de una imagen de contenedor para el servicio global.
  2. Creación de un volumen para mantener los datos de la base de datos de forma persistente.
  3. Asignación del volumen al contenedor creado.
  4. Conexión del contenedor a la red para poder servir el cliente web y recibir las interacciones producidas por un usuario.
  5. Puesta en marcha del contenedor.

### Opciones consideradas

1. Integración continua

Las herramientas consideradas para la integración continua no sólo se diferencian por sus características, sino que también por sus precios y licencias.

* Jenkins
  + Escrito en Java
  + Se ejecuta en un contenedor EJB
  + Más de 1.000 plugins
  + Asiste también en la entrega y el despliegue continuo
  + Compatible con muchos sistemas de control de versiones
  + Controles mediante GUI (basados en web), API REST o línea de comandos
  + Alojamiento opcional en la nube
  + Gratuita
  + De código abierto (licencia MIT)
* Travis CI

Trabaja en estrecha relación con el popular software de control de versiones. Puede configurarse con un sencillo archivo YAML. GitHub informa a Travis CI de todos los cambios efectuados en el repositorio y mantiene el proyecto actualizado.

* + Programado en Ruby
  + Multiplataforma
  + Funciona con GitHub
  + Se configura con un archivo YAML
  + Gratuita para proyectos de código abierto
  + Precio para proyectos comerciales: entre 69 y 489 dólares/mes
  + De código abierto (licencia MIT)
* Bamboo

Esta herramienta no solo sirve de ayuda en la integración continua, sino también para funciones de despliegue y gestión de lanzamientos. Funciona a través de una interfaz web.

* + Escrito en Java
  + Multiplataforma
  + Fácil integración de otros productos Atlassian
  + Gran cantidad de addons
  + Realización de varias pruebas al mismo tiempo
  + Interfaz web y API REST
  + Gratuita para proyectos de código libre, ONG y centros escolares
  + De lo contrario, pago único de entre 10 y 126 500 dólares, dependiendo del número de servidores utilizados
* GitLab CI

Forma parte del conocido sistema de control de versiones GitLab. Además de integración continua, GitLab ofrece despliegue y entrega continua. Al igual que con Travis CI, la configuración de GitLab CI se lleva a cabo con un archivo YAML. Por lo demás, su utilización es sencilla.

* + Forma parte de GitLab
  + Programado en Ruby y Go
  + Configuración con un archivo YAML
  + Asiste también en la entrega y el despliegue continuo
  + Open Core
  + Alojamiento propio o en la nube
  + Versión gratuita con pocas funciones
  + Precio para otras versiones, entre 4 y 99 dólares/mes por usuario.
* Resumen de características orientada a la toma de decisión:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Jenkins** | **Travis CI** | **Bamboo** | **GitLab** |
| Entrega continua | ✓ | ✗ | ✓ | ✓ |
| Alojamiento en la nube | ✓ | ✓ | ✓ | ✓ |
| Licencia | MIT | MIT | De propietario | MIT/EE |
| Precio versión comercial | - | 69-489 $/mes | 10 - 126 500 $ (pago único) | 4-99 $/mes |
| Versión gratuita | ✓ | ✓ | ✓ | ✓ |
| Particularidades | Numerosos plugins | Conexión directa con GitHub | Conexión directa con otros productos Atlassian |  |

1. Despliegue de la aplicación

* Docker

Sistema que nos permite construir, transferir, desplegar y ejecutar los contenedores con nuestras aplicaciones dentro de una manera muy sencilla y confiable, garantizando un despliegue escalable de forma eficiente sin importar el sistema operativo anfitrión.

* Kubernetes

Sistema que se encarga de gestionar todo el clúster de servidores, distribuye los contenedores a través del sistema según los recursos disponibles en el clúster, además de crear, ejecutar, vigilar, medir, destruir y relanzar los contenedores, debe mantener y controlar en todo momento cada aspecto relevante de los contenedores y su estado.

|  |  |  |
| --- | --- | --- |
|  | **Docker** | **Kubernetes** |
| Escalado | Sin escalado automático | Autoescalado |
| Configuración de clúster | Desafiante y complicado | Simple (con dos comandos) |
| Instalación | Fácil y rápido | Complicado y lento |
| Volumen de datos | Comparte entre varios contenedores el mismo Pod | Comparte con cualquier otro contenedor |
| Balance de carga | Autobalanceo de carga | Configuración manualmente |
| Relación de tolerancia | Alta tolerancia a fallos | Baja tolerancia a fallos |
| Límite de contenedores | 95.000 | 300.000 |
| Multiplataforma | ✓ | ✓ |
| Tiempo de creación | Pocos segundos | Minutos |
| Tiempo de arranque | Milésimas de segundo o unos pocos segundos | Minutos |

### Decisión

1. Para la **integración continua**, se ha decidido usar **Jenkins** porque es una herramienta gratuita (en comparación a las otras tres aplicaciones), es compatible con muchos sistemas de control de versiones (en nuestro caso usaremos GitHub) y dispone de numerosos plugins. Además, existe la disposición de *Pipelines* para los tests de integración.

Se podría haber decidido usar **Travis CI** por la conexión directa con GitHub, pero la condición para ser una herramienta gratuita es que el proyecto sea de código abierto y el nuestro no lo es.

1. Para el despliegue de la aplicación se ha decido usar **Docker** porque es una herramienta que todo el equipo conoce y la instalación del mismo es fácil y rápida.
2. Se decide que no es necesario realizar tests de rendimiento (comprobaciones de que el sistema mantiene el servicio en momentos de gran afluencia de usuarios o peticiones).

* Se consideran actualmente omisibles debido a que el sistema no se encuentra en situaciones de tal envergadura, en principio, luego el tiempo dedicado a diseñar los tests y probarlos con cada cambio significativo en el servicio, no parece razonable. Esta decisión podría cambiar si se dispusiera de tiempo suficiente al final del proyecto para realizar un estudio rigurosa de la escalabilidad de nuestro sistema, aunque por ahora, como se ha dicho, no es lo más relevante.

### Estado

(Tachar las que no apliquen)

1. Aceptado
2. ~~Rechazado~~
3. ~~Reemplazado~~

### Implicaciones

La utilización de las herramientas elegidas supone que el equipo de desarrollo deba aprender a manejarlas, lo cual conllevará un tiempo considerable a tener en cuenta en la gestión inicial del proyecto.

#### Fecha

27 de marzo de 2020