EventPlanner

Memoria de Proyecto

Gestión de Tecnologías Informáticas en las Organizaciones

Xabier Dendarieta
Sonia Elizondo
Paul Vega

Índice

Alcance 3
Características técnicas 4
Recursos 5
Tiempo 6
    Estimación inicial 6
    Estimación final 6
Metodología 7
Protocolo interno 8
    Protocolo Cliente – API 8
    Protocolo API – BD 8
    Puertos utilizados 9
    Entidades almacenadas en BD 9
Requisitos de Despliegue 10
Plan de Sistemas 12
    AWS 12
    CPD 14
    Comparación 16
Instrucciones de utilización 17
Problemas encontrados 21
    Instalación de librerías Node.js 21
    Comprobación automática de Jenkins 21
    Pipeline para despliegue en Docker 22
    Excepciones en API 24
    Conexión Cliente-API 25
Anexo 26
    ADR Base de Datos 26
        Contexto 26
        Procesos 26
        Opciones consideradas 26
        Decisión 27

Estado....27
Implicaciones....27
ADR Gestión de la Configuración....29
Contexto....29
Procesos....29
Opciones consideradas....30
Decisión....30
Estado....31
Implicaciones....31
ADR Integración Continua....32
Contexto....32
Procesos....32
Opciones consideradas....33
Decisión....35
Estado....35
Implicaciones....35
Presupuestos....37
Presupuesto AWS....37
Presupuesto CPD....38

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 necesario, que serán visibles por todos los usuarios. 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 API** 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.

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 3 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" 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.

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.

Protocolo interno

Protocolo Cliente – API

Llamadas tipo GET:

- GET("api/user/:username") => devuelve OK o ERR
- GET("api/event/:eventid") => devuelve JSON del evento o ERR
- GET("api/list ") => devuelve JSON con la lista (puede estar vacía) de todos los eventos

Llamadas tipo POST:

- POST("api/add/event", JSON con todos los datos de event menos eventid) => devuelve eventid o ERR
- POST("api/remove/event", JSON con :eventid) => devuelve OK o ERR
- POST("api/add/assistant", JSON con :username y :eventid) => devuelve OK o ERR
- POST("api/remove/assistant", JSON con :username y :eventid) => devuelve OK o ERR
- POST("api/add/user", JSON con :username) => devuelve OK o ERR

Protocolo API – BD

- Conexión mediante sockets TCP con la librería 'zmq' de Nodejs.
- Se envían *strings* en los mensajes que parten de formato JSON.
- Si se trata de mensaje de API a BD, se indican los siguientes parámetros importantes:
    - 'component': indica si se trata de una petición de ‘user’ o ‘event’.
    - 'id': identificador del elemento que quiera introducirse o consultarse.
    - 'body': cuerpo de la petición. Se forma de parámetros que detallan el elemento concreto y de ‘op’ (pudiendo ser únicamente *get* o *put*).

- Si se trata de mensaje de BD a API, se establece respuesta por cada tipo de petición:
    - Para 'put' de 'user': *Done* si la operación se ha realizado correctamente y *Failed* en caso contrario.
    - Para 'put' de 'event': 'id' aleatorio generado para el nuevo evento registrado (de ser correcto) y 'Failed' en caso contrario.
    - Para 'get' de 'user': *OK* si existe el usuario y *Failed* en caso contrario.
    - Para 'get' de 'event': los detalles que lo definen y 'Failed' en caso contrario.

Puertos utilizados

- Se sirve el cliente web en el puerto 4000.
- Se establece la API en el puerto 3000, de modo que el cliente se comunique con ella a través de dicho puerto.
- Se establece el servicio de Base de Datos en el puerto 1234, de tal forma que la API envíe las peticiones a través de este puerto.

Entidades almacenadas en BD

- User
    - Name (string)

- Event
    - ID (string)
    - Name (string)
    - Datetime (string)
    - Description (string)
    - Organizer (string)
    - Assistants (list of string)

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.

Plan de Sistemas

Se pretende definir la arquitectura necesaria para que la aplicación EventPlanner pueda prestar su servicio web a cualquier usuario.

Para ello, se toman dos puntos de vista distintos para estructurar dicho despliegue:

- **Amazon Web Services** (AWS): por medio de la contratación de una serie de servicios en línea que conformen una Cloud privada.
- Un entorno físico, como pueda ser un laboratorio o una empresa, conformando un **Centro de Procesamiento de Datos** (CPD).

Esto es, suponiendo que somos una pequeña empresa, buscamos la mejor solución para poder desplegar nuestra aplicación, comparando un servicio en la nube con un servicio local.

Para empezar, se sientan las bases de la arquitectura lógica y física a definir por medio de la arquitectura ya presentada al comienzo del proyecto, así como por los requisitos de despliegue explicados en el apartado anterior.

Además, se tiene en cuenta que, debido al presupuesto que una pequeña empresa de reciente formación pueda manejar, se deben ajustar los precios de ambas posibilidades lo máximo posible. Es decir, conseguir equipamiento lo más barato posible, pero sin descuidar los requerimientos mínimos establecidos, para asegurar la alta disponibilidad y la escalabilidad (sin olvidar la persistencia de los datos).

Tras un análisis exhaustivo de los componentes necesarios, se presentan los siguientes presupuestos:

AWS

Mostramos los componentes elegidos para conformar la arquitectura en AWS, destacando de cada uno sólo aquellas características por las que nos hemos decantado por cada uno de ellos:

Comenzando por el despliegue del cliente web, es necesario contar con:

- Route 53: servicio web DNS escalable y de alta disponibilidad en la nube.
    - Conecta efectivamente las solicitudes de los usuarios con la infraestructura desplegada en AWS, sean cuales sean sus tipos de instancias.

- CloudFront: permite servir el cliente web.
- S3: almacenamiento de hasta 10GB para contener todo el HTML, CSS y JavaScript para dar dinamismo a la hora de servir la página web.

En cuanto al resto de la arquitectura:

- T3.a.:
  - Indicadas para un uso general de la máquina.
  - Diseñado para optimizar el rendimiento de sistemas con utilización media en los que se producen picos eventuales de carga.
  - Ahorro en coste respecto a otras opciones similares.
  - Para la API se incorporan 2 instancias, de modo que cada una pertenezca a una zona de disponibilidad (referenciadas a continuación).
  - Para la BD se utilizan otras 2, con la misma intención de zonas diferenciadas.

- 2 balanceadores de carga (Elastic Load Balancing) para distribuir las peticiones tanto a la API como a la BD de forma eficiente.

- La Base de Datos, por su parte, requiere persistencia de datos:
  - EBS:
    - SSD de 100GB.
    - 2 instancias, una principal y otra que actúa como réplica de los datos presentes en la primera.

Se requiere de dos zonas de disponibilidad para asegurar la prestación del servicio en caso de caída de suministros. Se han escogido las siguientes:

- Dentro de la región “Irlanda EU-WEST-1”:
  - Se puede disponer de hasta 3 zonas distintas, por lo que hacemos uso de 2 de ellas.
  - Es una región sostenible: precio abaratado por la disponibilidad de energías renovables.

La arquitectura final a desplegar en AWS puede visualizarse en el siguiente diagrama:

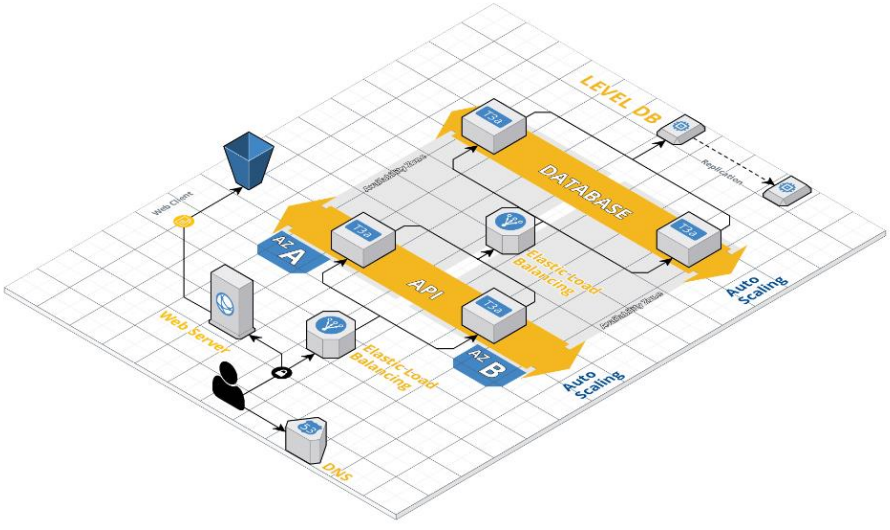


Ilustración 1 Arquitectura de AWS

CPD

Queremos disponer una arquitectura similar a la presentada en el diagrama anterior, sólo que por medio de componentes físico constituyendo un CPD. Los elementos elegidos para tal fin y sus propiedades más sugerentes son:

Empezando por los servidores requeridos para la puesta en marcha del servicio:

- **Servidor** (unidades de procesamiento):
  - HPE ProLiant DL20 G10:
    - Cuenta con procesador Intel Xeon E-2134.
    - **Se necesita disponer de 4 de ellos**, de modo que la capa de API y la capa de BD cuenten con 2 cada uno: uno principal y otro de soporte frente a fallos en el sistema.
    - Escogido por tratarse de una marca confiable con un procesador estándar que ofrece una buena relación calidad-precio.
      - Otras opciones: Dell PowerEdge R230, Lenovo ThinkSystem SR630 7X02A0CJNA

Para el almacenamiento persistente de los datos de la Base de Datos:

- **Servidor de almacenamiento**:
  - NETGEAR ReadyNAS 2304:
    - Servidor NAS para albergar hasta 4 discos duros.
    - **Son necesarios 2 componentes**, de modo que las peticiones sean servidas por uno y el otro haga las veces de réplica de datos.
    - Como se necesita replicación rápida entre dispositivos similares, es una buena elección. Además, es compatible con RAID 10 (sistema de replicación de discos antipérdidas).
      - Otras posibilidades: QNAP TS-453BU-8G-US, Synology RS819 Diskless System.

- **Discos duros sólidos**:
  - Vaseky 2.5’’ SATA 3 III SSD MLC:
    - Capacidad de 120 GB.
    - **Necesarias 8 unidades**.
    - En este caso, nos interesa más comprar discos baratos puesto que contamos ya con el sistema RAID 10.
      - Comparable, por ejemplo, con: Kingston SSDNow UV400

En cuanto al servicio en la red y la distribución de peticiones, se encuentra solución en un mismo componente:

- **Router** (con balanceador de carga):

- UTT ER4240G Business Gigabit Router:
    - Router NAT para configuración de VPN.
    - 4 puertos WAN y 4 puertos LAN.
    - **Necesarias 2 unidades**.
    - Seleccionado debido a que incluye el balanceador de carga.
        - Otras opciones: CISCO2821 Int. Services Router, Mikrotik Cloud Core Router CCR1009-7G-1C-PC Network Router PoE

Para mejor disposición de los anteriores componentes, en cuanto a orden y también facilidad de transporte, se considera:

- **Armario Rack**:
    - Tripp Lite 18U Wall-Mount Rack Enclosure Cabinet:
        - Capacidad de 18U.
            - La capacidad es superior a la actualmente necesaria, pero se considera una inversión por posible crecimiento del servicio.
        - Los componentes previamente detallados pueden acoplarse a un Rack.
        - **2 unidades**: una por cada zona de disponibilidad (comentadas a continuación).
        - Escogido debido a su capacidad y a que puede cerrarse por todos los lados, a diferencia de otros modelos como:
            - iStarUSA WX-228 22U 4-Post Open Frame Rack – OEM, Tripp Lite SR4POST 45U 4-Post SmartRack Open Frame Rack

Cabe destacar que la arquitectura de parte del cliente web, se virtualizaría incluso en el caso del CPD. De modo que no necesitamos especificar más componentes físicos que los ya indicados.

En cuanto a las zonas de disponibilidad, se pretende contratar los servicios de red a dos proveedores de Internet distintos, de modo que no se pierda la conexión del servicio por completo en caso de caída de alguna de las compañías. Con el mismo propósito, pero en cuanto a caídas de la red eléctrica, se quiere disponer de dos circuitos eléctricos diferenciados.

En definitiva, la arquitectura podría plasmarse de la siguiente forma:

Aunque no están presentes en el presupuesto actual, no hay que olvidar que una partida es necesaria íntegramente a la compra de cables de alimentación, cables de red y demás hardware básico para el montaje del CPD en su totalidad.

Ilustración 2 Arquitectura de CPD

Además, debería contemplarse el gasto económico que pudiera derivarse de la refrigeración de dicho CPD, así como de las medidas de prevención de riesgos, como medidas antiincendios.

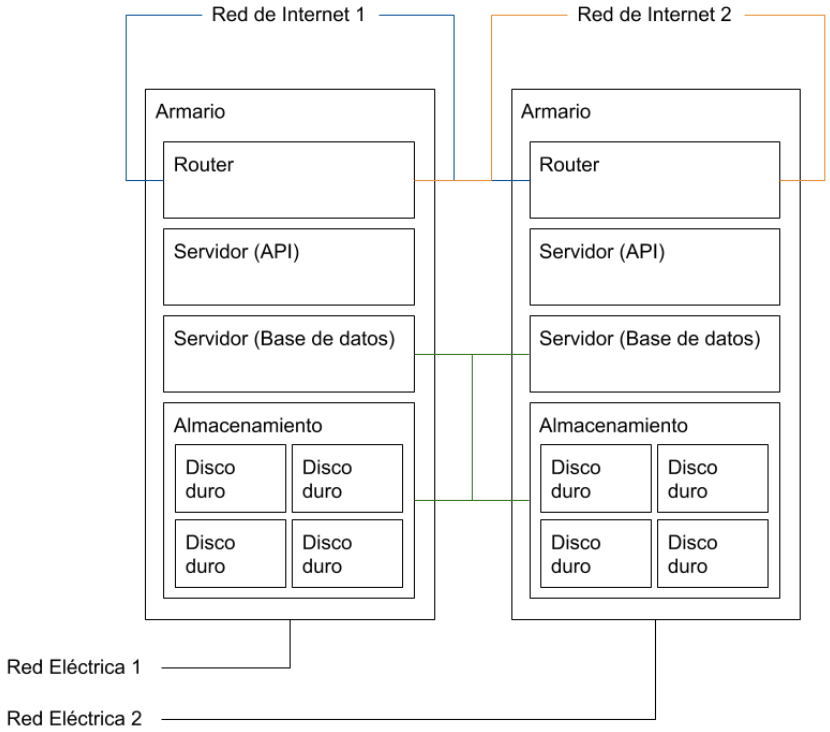


Ilustración 2 Arquitectura de CPD

Comparación

Las dos posibilidades de despliegue estudiadas conllevan un presupuesto de: 277,35€/mes, en el caso de AWS, y 5.367,50€, en el caso del CPD. (Para ver el desglose de los mismos, véase Presupuestos).

Tratándose de un servicio a prestar completamente nuevo, que en un principio no espera crecer de manera demasiado acelerada, y que consideramos que somos una pequeña empresa de nueva creación, la opción más factible es la de desplegar en AWS, debido al presupuesto que sostiene, al igual que por todas las facilidades que presta para dicho despliegue.

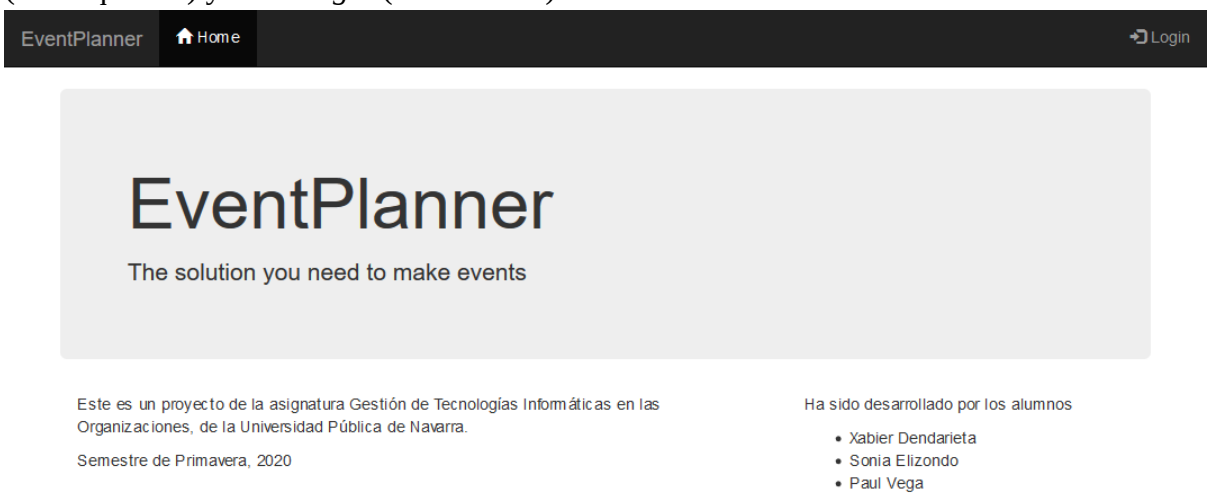
Pero, también hay que tener en cuenta que, de quererse mantener el servicio durante largo tiempo siendo propietarios de todo el equipamiento del que se requiere para prestarlo, la inversión en el CPD se amortizaría en aproximadamente 19 meses de contratación de AWS.

Es decir, si se pretende que la aplicación tenga una vida útil de más de año y medio, el montaje de un CPD debería considerarse como una opción factible.

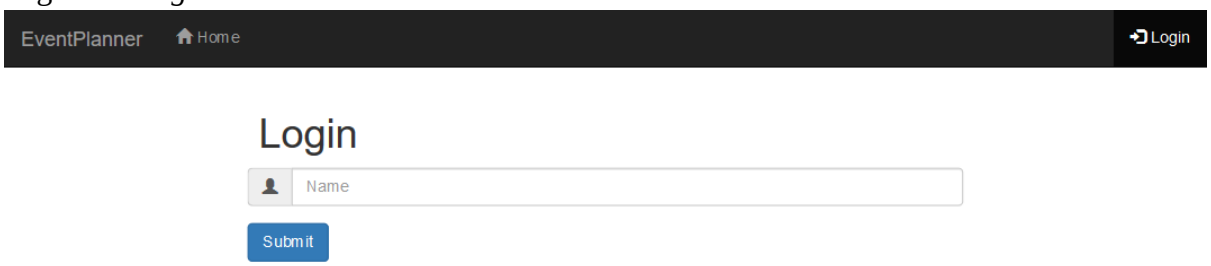
Instrucciones de utilización

El cliente web presenta una interfaz sencilla e intuitiva. Aquí se pretende plasmar una pequeña guía de uso.

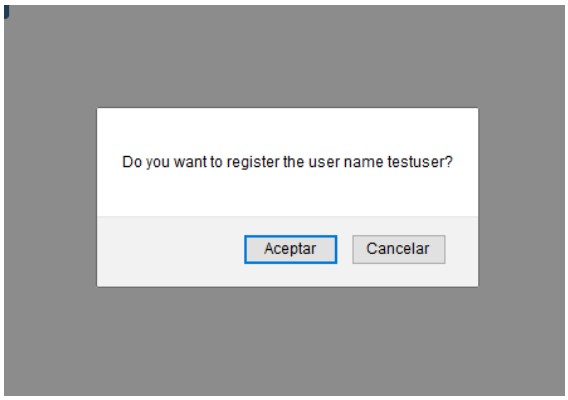
- Página principal: en el menú superior podemos encontrar habilitada la pestaña de *Home* (a la izquierda) y la de *Login* (a la derecha).



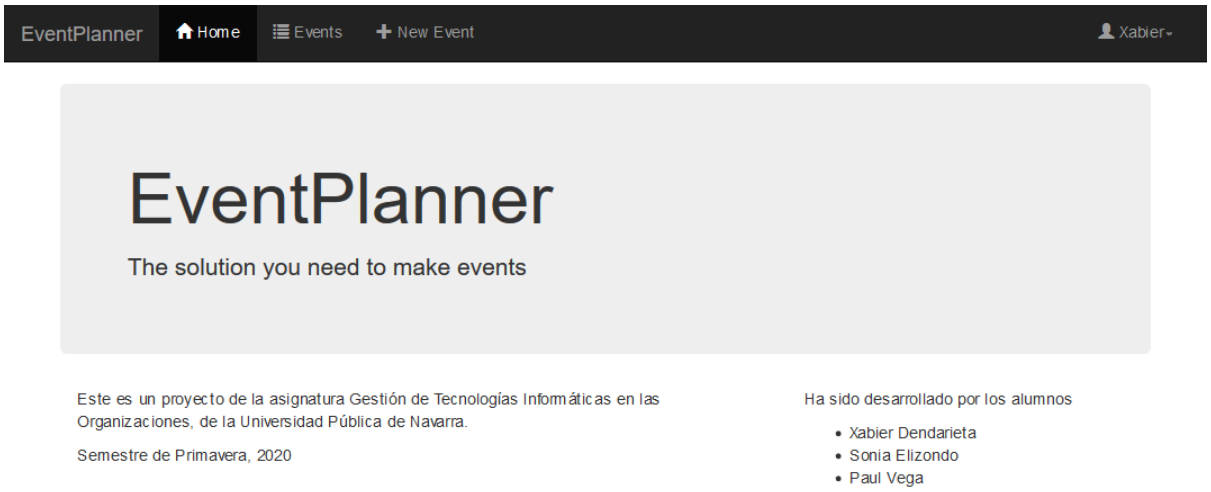
- Página de *Login*.



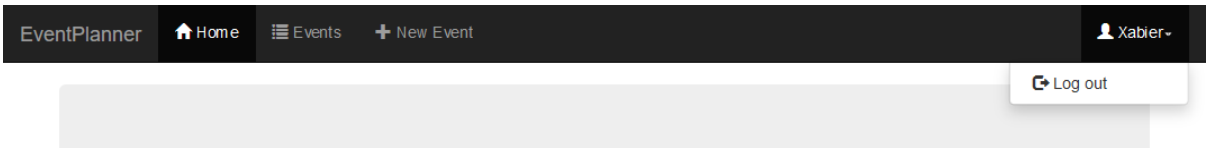
  - Mensaje de *register*: cuando el usuario no figure en la base de datos se le dará opción de registrarse.



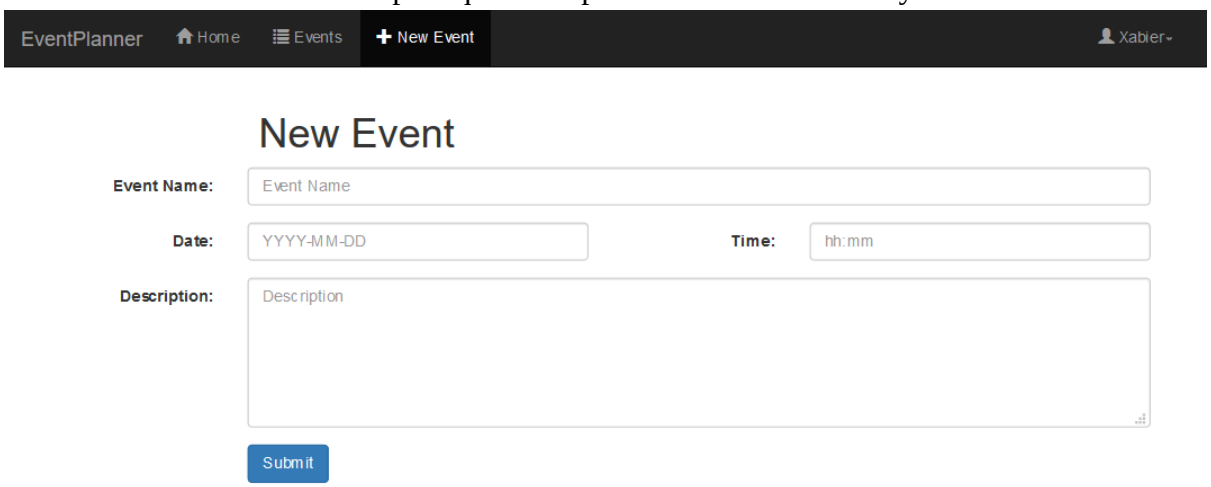
- Página principal *Logged*: al identificarse un usuario, se activarán las pestañas de lista de eventos (menú superior, izquierda), de nuevo evento (menú superior, izquierda) y de usuario (menú superior, derecha, desplegable).



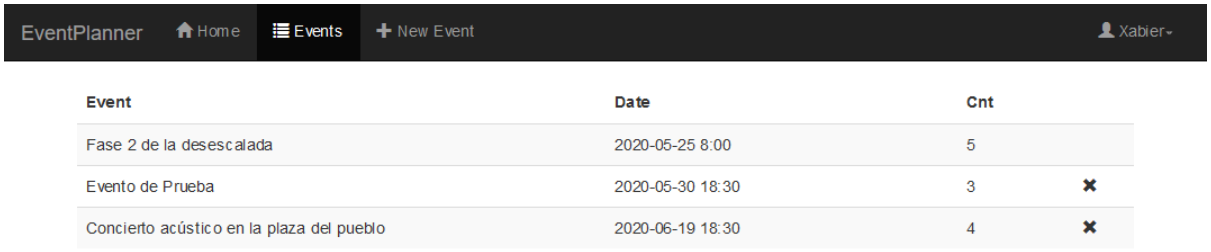
  - Despliegue de menú para *Logout*.



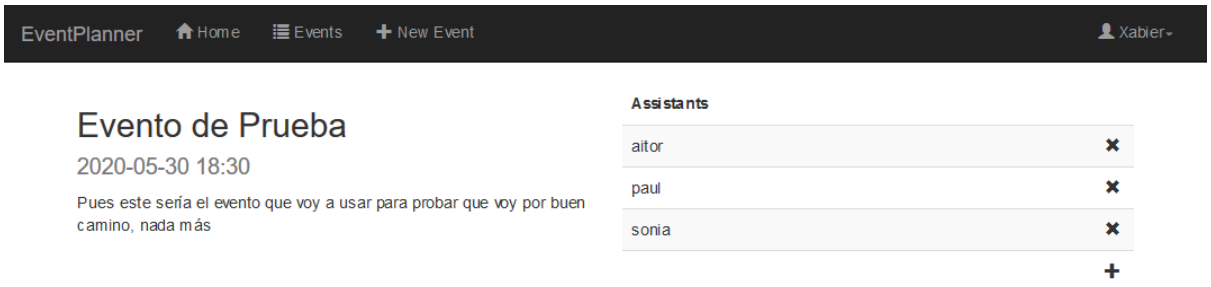
- Página de *New Event*: el evento no podrá ser almacenado si el usuario deja algún campo sin rellenar correctamente. Se pide que se respete el formato de fecha y hora.



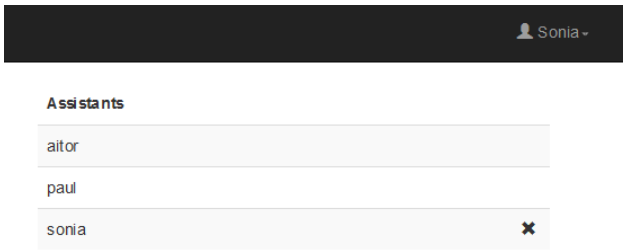
- Página de *Events*: corresponde al listado de eventos. En la celda del nombre del evento se puede hacer click para acceder a su página concreta. A la derecha de cada fila vemos el botón (en forma de equis) para borrar el evento, que solo estará disponible para los eventos que haya creado el usuario identificado en ese momento.



- Página de un evento: aquí se podrá ver toda la información sobre el evento y sus asistentes.

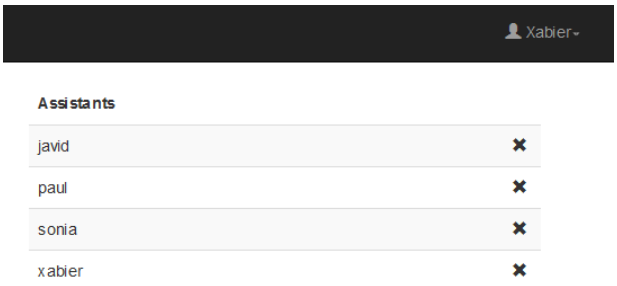


  - Asistente: si el usuario está en la lista de asistentes, aparecerá al lado de su nombre el botón (con forma de equis) para que pueda borrarse de dicha lista si lo desea.



  - No asistente: si el usuario no está en la lista de asistentes, aparecerá el botón (en forma de plus) para que se añada si lo desea.

- Organizador: si el usuario identificado es el organizador del evento, podrá borrar a quien crea conveniente.



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
```

A pesar de esto la librería 'zmq' seguía dando problemas en la mayor parte de las instalaciones, por lo que se optó por sustituirla por su análoga 'zeromq', la cual tuvo más tiempo de mantenimiento y no parece dar problemas actualmente. Queda decir que su funcionamiento interno es exactamente igual al de 'zmq', por lo que no supone ningún problema extra.

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.

Pipeline para despliegue en Docker

Con la intención de cumplir los requerimientos del proyecto en cuanto a integración continua, se continuó avanzando en perfilar y añadir funcionalidades a la *pipeline* antes comentada.

Hasta el momento sus tareas eran las de clonar el repositorio, instalar las dependencias que necesitase la aplicación para ejecutarse correctamente y realizar un test unitario sobre la base de datos. Ahora, lo que se pretendía era añadir de forma básica la utilización de Docker para esos mismos pasos. El trabajo realizado sobre Jenkins, y más concretamente sobre el *Jenkinsfile* alojado en el repositorio, ha conllevado varios errores (en algunos casos subsanados) por falta de experiencia en la definición de dicho archivo.

Al principio, siguiendo información relacionada, se intentó construir una imagen de Docker a la medida de las necesidades del proyecto; es decir, con Node.js instalado en la versión necesaria. Lo cual implicaba que Jenkins pudiera acceder posteriormente a DockerHub, donde debería estar la imagen alojada, y realizar un *build* con la misma.

Para acelerar dicho proceso, ya que no se necesitaban características muy particulares, se optó mejor por utilizar una imagen oficial de Node.js de la versión apropiada y utilizarla como agente para la *pipeline*.

Una vez se llevó a cabo esta decisión, empezaron a surgir errores de ejecución en la *pipeline* por distintos motivos, que tuvieron que ir resolviéndose conforme aparecían.

En la primera ejecución de la *pipeline* un error comentaba que Jenkins no tenía permisos para ejecutar Docker. Para solucionarlo se incluyó este primer programa en la lista de privilegios para ejecución de Docker mediante el comando:

```
sudo gpasswd -a Jenkins docker
```

Tras lo cual hubo no sólo que reiniciar Docker sino la máquina virtual completa, para que el cambio se procesase y quedase permanente. Al volver a iniciarse, el contenedor ya se generaba correctamente en Jenkins.

A continuación, se mostraba un error de que no se tenían permisos para utilizar npm y el comando *sudo* no es una opción dentro del Jenkinsfile como se pudo comprobar (aunque por línea de comandos nunca se tuvo que utilizar *sudo* para *npm*).

Tras varios intentos fallidos de encontrar una solución *online*, como pudiera ser cambiar los privilegios de escritura del *workspace* donde se instalaban las dependencias, se soluciona mediante el siguiente código en el Jenkinsfile:

```
environment { HOME = '.' }
```

Ya que anteriormente estaba intentando instalar las dependencias en un directorio no conocido, y ahora se ciñe a lo establecido en Jenkins.

Ahora que ya puede instalar las dependencias surge un inconveniente con la librería zmq (que ya dio problemas al inicio del proyecto). Pero esta vez, los comandos utilizados en aquella ocasión no surten efecto. Los problemas con zmq también han empezado a entorpecer el desarrollo y comprobación de la API, por lo que se termina utilizando la librería zeromq. Lo cual no conlleva ningún cambio significativo en el código, porque funciona de forma similar, y se establece dicha librería en el *package.json*.

La *pipeline* se ejecuta correctamente en toda su extensión: generar el contenedor Docker, clonar el repositorio, instalar las dependencias, realizar el test unitario sobre la base de datos y cerrar adecuadamente el contenedor. Pero, en la salida por comandos, se aprecia una advertencia:

```
Warning: JENKINS-30600: special launcher
org.jenkinsci.plugins.docker.workflow.WithContainerStep$Decorator$1@
491e104b; decorates hudson.Launcher$LocalLauncher@7a0b66c4 will be
ignored (a typical symptom is the Git executable not being run
inside a designated container)
```

Esta traza deja claro que, aunque todo se ejecuta correctamente, realmente los comandos presentes en los *steps* del Jenkinsfile, no se están realizando dentro del contenedor Docker provisto para ello; sino que se ejecutan como en la versión anterior de la *pipeline*.

Por ahora no se ha encontrado solución a este problema. Si bien se ha dedicado tiempo y esfuerzo en entender Docker dentro de Jenkins, no se ha conseguido utilizarlo de manera satisfactoria.

Pero, al menos, el proceso de integración continua de test unitario sigue siendo funcional y nos da la capacidad de comprobar que la base de datos es funcional y realiza las operaciones y devuelve mensajes con los resultados obtenidos según lo esperado. El añadido de Docker optimizaría más dicha integración, sólo que necesita más tiempo de aprendizaje del actualmente disponible.

Excepciones en API

El desarrollo de la API ha conllevado bastante más tiempo del esperado por tener que seguir un estricto protocolo de comunicación tanto con la base de datos como con el cliente web, como se puede observar en el apartado dedicado a detallarlos.

En la versión de código considerada final, por tratarse de la obtenida hasta el momento de entrega del proyecto, existen dos excepciones no solucionadas en la API.

Cuando se conectan los componentes de la base de datos y la API, al realizar peticiones de pruebas a la primera, se obtiene respuesta correctamente, salvo en algunas ocasiones en las que el *socket* de API queda escuchando indefinidamente, a la espera de un mensaje de respuesta a su petición.

No se ha conseguido conocer el motivo de esta situación, que no ocurre de forma habitual, pero sí se repite en algunas ocasiones específicas, pero sin aludir a una misma causa.

Otro problema encontrado es el hecho de que, al realizar una misma petición de forma reiterada, se encuentra una excepción cuya traza parece comenzar en la propia librería *express*. La descripción de la excepción es:

```
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are
                        sent to the client
```

Con ello se supone que se quiere indicar que, una vez realizada la petición, los datos introducidos por la ruta de un explorador conectado a la API dejan de ser válidos.

Conexión Cliente-API

En el último momento, a la hora de conectar la API y el cliente web, ha surgido el problema del bloqueo de peticiones por cross-domain. Este bloqueo lo crea el propio navegador, actualmente todos los navegadores están configurados de esta manera, y la causa es que la petición está destinada u originada en un dominio diferente al actual.

En nuestro caso, a pesar de estar siempre en el mismo dominio, lo que genera este bloqueo es el cambio de puerto, que es interpretado de la misma forma. Es decir, a la hora de tratar con este problema, el dominio no es únicamente el host, sino el host con el puerto.

El navegador devuelve por consola un error de este tipo al intentar hacer un GET sobre el usuario *usertest*:

```
Solicitud desde otro origen bloqueada: la política de mismo origen impide leer el recurso remoto en http://localhost:3000/api/user/usertest (razón: falta la cabecera CORS 'Access-Control-Allow-Origin').
```

```
Solicitud de origen cruzado bloqueada: La misma política de origen no permite la lectura de recursos remotos en http://localhost:3000/api/user/usertest. (Razón: Solicitud CORS sin éxito).
```

Desde la petición Cliente-API es posible modificar las cabeceras, pero requiere también hacer que el navegador permita el mensaje de vuelta. Se sigue buscando una solución factible a esto.

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):
    a. El usuario realiza una operación desde el cliente web (registrarse, identificarse, crear un nuevo evento, indicar que se asiste a un evento, etc.).
    b. El cliente web manda la acción correspondiente a la API con los datos proporcionados por el usuario.
    c. 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.
    d. La Base de Datos almacena datos nuevos o devuelve los almacenados según haya sido realizada la petición.
        i. Los datos críticos deben estar cifrados (en este caso, únicamente la contraseña del usuario se considera relevante).
        ii. No se deben perder datos debido a errores de gestión.
    e. La Base de Datos devuelve información a la API, ya sea como confirmación de éxito de una operación como de los datos requeridos.
    f. La API envía la información obtenida al cliente web.
    g. 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:
    a. Pros:

   i. Permite escalabilidad horizontal.
   ii. Está orientada a ficheros JSON.
   iii. Incluye un sistema de *master-workers* distribuido en un clúster que facilita una alta disponibilidad con alta tolerancia a fallos.
   iv. Proporciona un sistema de replicación automático.
  b. Cons: el equipo no está familiarizado con su uso.

2. LevelDB:
  a. Pros:
   i. Librería de Nodejs.
   ii. Uso sencillo.
   iii. Más de la mitad del equipo está familiarizado con su uso.
  b. Cons:
   i. Sólo permite almacenamiento de pares clave-valor.
   ii. 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.
2. 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)

4. Aceptado
5. ~~Rechazado~~
6. ~~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:
   1.1. Instalación de herramientas: Git, Docker, Jenkins. (Véanse apartados de "Opciones consideradas" y "Decisión")
   1.2. 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):
   2.1. Todos los cambios deberán aplicarse a la rama *master* del repositorio, siguiendo la filosofía de integración continua.
   2.2. 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):
   3.1. Cualquier cambio realizado en la aplicación se actualiza en la rama *master*. No se realizan ramificaciones.
   3.2. 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.
   3.3. Si los test unitarios SÍ son satisfactorios:
      3.3.1. Se genera un ejecutable
      3.3.2. Se genera el correspondiente contenedor con Docker
   3.4. Si los test unitarios NO son satisfactorios:
      3.4.1. Revisa los test unitarios paras aplicar las respectivas correcciones.
   3.5. 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.
   3.6. 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).

   4.1. Realización de una imagen de contenedor para el servicio global.
   4.2. Creación de un volumen para mantener los datos de la base de datos de forma persistente.
   4.3. Asignación del volumen al contenedor creado.

4.4. Conexión del contenedor a la red para poder servir el cliente web y recibir las interacciones producidas por un usuario.
4.5. 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	

2. 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.
2. 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.
3. 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)

7. Aceptado
8. ~~Rechazado~~
9. ~~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

Presupuestos

Presupuesto AWS

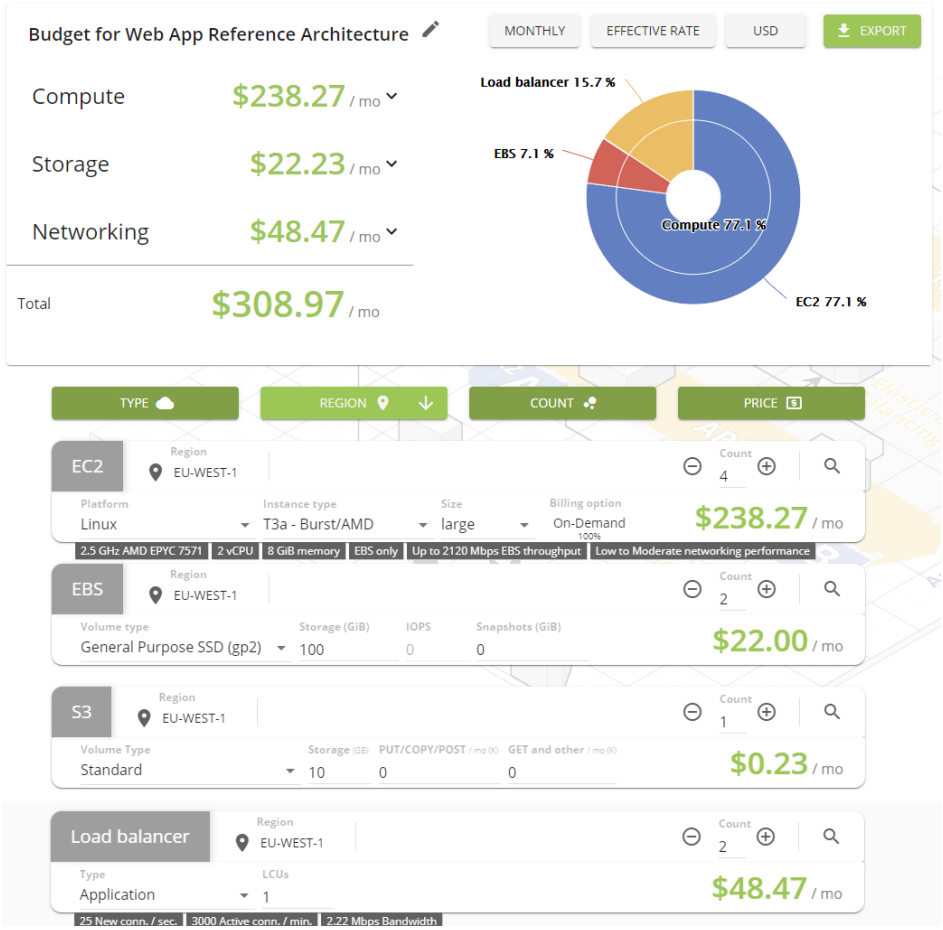


Ilustración 3 Captura de Presupuesto en CloudCraft

Muestra dos formas de leer el presupuesto.

En la parte superior (y también en forma de gráfico) lo computa en base al tipo de servicio que se contrata: cómputo, almacenamiento o red.

En la parte inferior, en cambio, despliega los componentes adquiridos en base al tipo escogido, las características del mismo (por ejemplo, los GB de memoria) y las unidades requeridas de cada uno.

En definitiva, el presupuesto asciende a 308.97$ al mes. Cifra equivalente en la actualidad a 277.35€/mes.

Presupuesto CPD

Nombre Componente	Tipo	Unidades	Precio/unidad	Coste
HPE ProLiant DL20 G10	Cómputo	4	$ 831,11	$ 3.324,44
NETGEAR ReadyNAS 2304	Almacenamiento	2	$ 438,99	$ 877,98
Vaseky 2.5'' SATA 3 III SSD MLC	Almacenamiento	8	$ 22,99	$ 183,92
UTT ER4240G Business Gigabit Router	Red	2	$ 239,00	$ 478,00
Tripp Lite 18U Wall-Mount Rack Enclosure Cabinet	*Soporte Físico*	2	$ 557,54	$ 1.115,08
			Coste Total	**$ 5.979,42**

Ilustración 4 Tabla de Presupuesto CPD

En base a los componentes escogidos y el número de unidades de cada tipo, se presenta un coste total de 5.979,42$ (como se ha indicado en el apartado correspondiente, realmente quedarían más elementos por considerar, así que el presupuesto será más alto que el actual). Equivale en la actualidad a 5.367,50€.

Nota: no se han tenido en cuenta posibles gastos de envío de los proveedores del equipamiento indicando.

Si se desea consultar todos los componentes comparados para el proceso de selección final, están disponibles en la web: https://www.newegg.com/.

Los enlaces de los componentes escogidos son:

- HPE ProLiant DL20 G10:

  https://www.newegg.com/hpe-proliant-dl20-gen10-p06479-b21-rack/p/3C6-000J-00GC0?Description=server%20processor%20rack&cm_re=server_processor_rack-_-3C6-000J-00GC0-_-Product

- NETGEAR ReadyNAS 2304:
  https://www.newegg.com/netgear-rr230400-100nes/p/0E6-0019-000Z5?Description=nas%20rack&cm_re=nas_rack-_-0E6-0019-000Z5-_-Product

- Vaseky 2.5'' SATA 3 III SSD MLC:
  https://www.newegg.com/p/0D9-00HS-00003

- UTT ER4240G Business Gigabit Router:
  https://www.newegg.com/utt-er4240g-10-100-1000mbps/p/0XK-00BE-00004?Description=vpn%20%2b%20load%20balancer&cm_re=vpn_%2b_load_balancer-_-9SIAE8J7603734-_-Product

- Tripp Lite 18U Wall-Mount Rack Enclosure Cabinet:
  https://www.newegg.com/tripp-lite-srw18us-wall-mount-cabinet/p/N82E16816228064?Description=rack%20cabinet&cm_re=rack_cabinet-_-16-228-064-_-Product