**Requisitos de componentes lógicos**

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 varios contenedores de cada tipo.
* 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.

Cabe destacar que, de haberse tratado de un proyecto con mayores recursos y disponibilidad de tiempo, la Base de Datos podría haberse realizado con ElasticSearch en vez de LevelDB, por las siguientes razones:

* Permite escalabilidad horizontal, lo cual supone un bajo coste en caso de necesitar escalar nuestro sistema.
* Al ser una base de datos orientada a ficheros JSON, se amolda perfectamente a nuestras necesidades.
* Incluye un sistema de *master-workers* distribuido en un clúster que nos facilita una alta disponibilidad con alta tolerancia a fallos.
* También proporciona un sistema de replicación automático, por lo que la herramienta por sí sola ya contiene todos los aspectos que necesitamos de una base de datos.