TeamOverFlow

Experimento 2

Construcción del FrontEnd y Seguridad

Introducción y comentarios

Este documento es una extensión del documento original presentado hace algunos meses de la arquitectura, el diagrama de despliegue y futuras decisiones.

Para el desarrollo de la capa de presentación web decidimos usar y extender los conocimientos de todos en el framework de Google AngularJS. Consideramos utilizar Angular2 para el desarrollo, pero dado que aún es demasiado nuevo y al ser un framework que abstrae tanto de la lógica para el manejo de aplicaciones single page, la curva de aprendizaje podría llegar a ser alta, por eso decidimos, al igual que muchos otros proyectos que están en proceso de desarrollo actualmente, seguir trabajando con Angular1 en su última versión, a la cual Google anunció que dará soporte por más años.

Consideramos inicialmente trabajar con una base de datos No relacional. Consideramos MongoDB, RethinkDB dada su facilidad para la construcción de aplicaciones web real time. Después de estudiar el problema y la solución que veníamos desarrollando, decidimos que al igual con Angular2, nos tomaría tiempo y esfuerzo aprender a usar estas tecnologías. Nos decidimos entonces por usar una base de datos Relacional conocida, sencilla de usar y rápida: PostgreSQL.

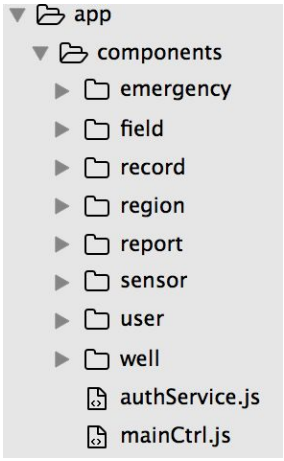
Nuevas tácticas y patrones Arquitecturales

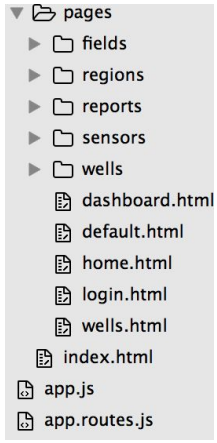
Dado que ya teníamos casi todo el backend hecho, excepto por unos servicios que fueron creados para poder desplegar información de cierta manera en la interfaz, sólo nos restaba construir una aplicación web segura, rápida y que consumiera los servicios que nuestra API ya expone.

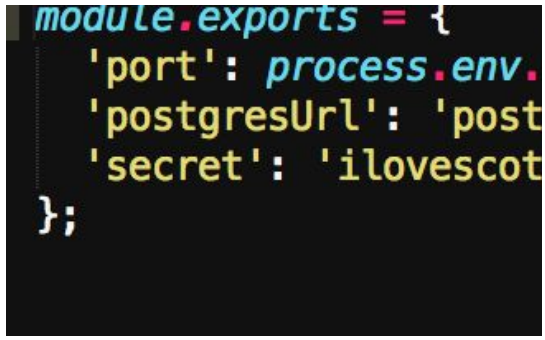
La seguridad la implementamos usando Json Web Tokens, pero dado que esta entrega se centra en el desarrollo de la capa web y en disponibilidad, hablaremos de eso en la siguiente entrega. Vale la pena mencionar sólo que dado el esquema de autenticación que tenemos, garantizamos la disponibilidad de las acciones y datos sólo a los usuarios autorizados.

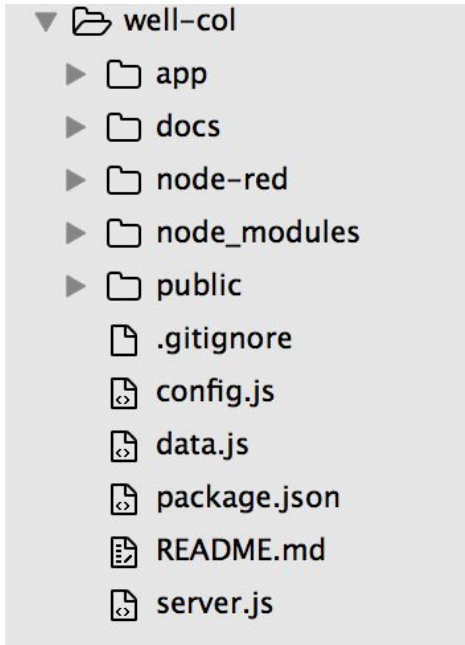
Aprovechamos el hecho de que Angular provee una manera sencilla de consumir servicios de otras aplicaciones para poder así consumir el API que ya teníamos y hacer que nuestro código fuera modular, cohesivo y con muy poco acoplamiento.

Esta modularización y separación de responsabilidades en el código tanto de javascript como de las vistas en html simplificó mucho el proceso de desarrollo, y nos permitió avanzar rápido hacia el objetivo final: Una aplicación web funcional.







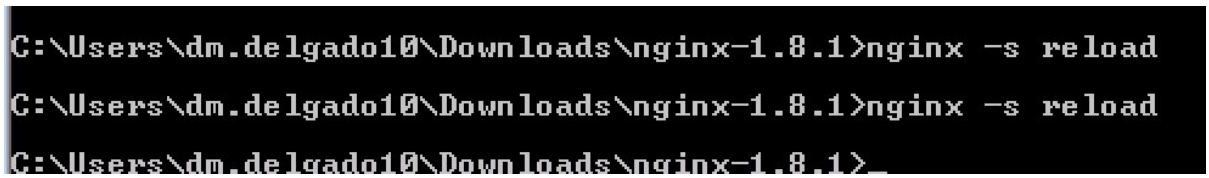


Ejemplo de código

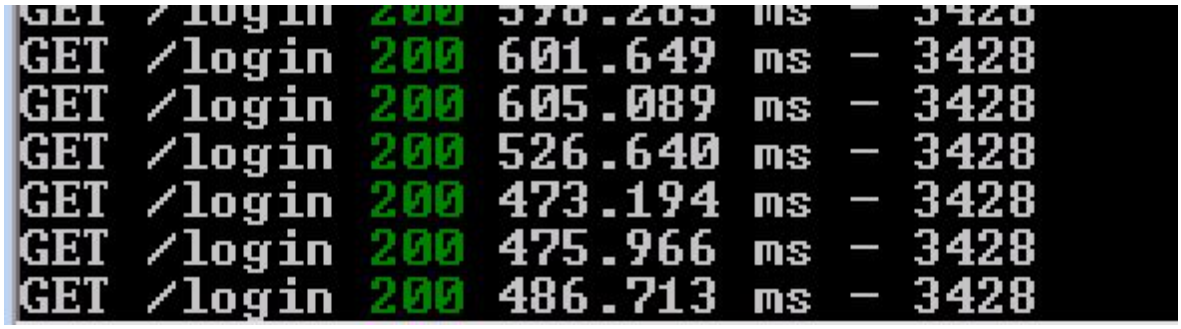
```
                templateUrl: 'app/views/pages/default.html'
            }
        }

})
.state('reports', {
  url         : "/reports",
  templateUrl : "app/views/pages/reports/reports.html",
  controller: "reportController",
  controllerAs: 'report'
})
.state('regions', {
  url         : "/regions",
  views       : {
    // the main template will be placed here (relatively named)
        '': { templateUrl: 'app/views/pages/regions/regions.html' },
       // the child views will be defined here (absolutely named)
       'map@regions': {templateUrl: 'app/views/pages/regions/map.html'},
         // for description column, we'll define a separate controller
        'description@regions': {templateUrl: 'app/views/pages/regions/description.html'}
       // for createRegion
```

Modularización y separación de responsabilidades en los controladores y vistas

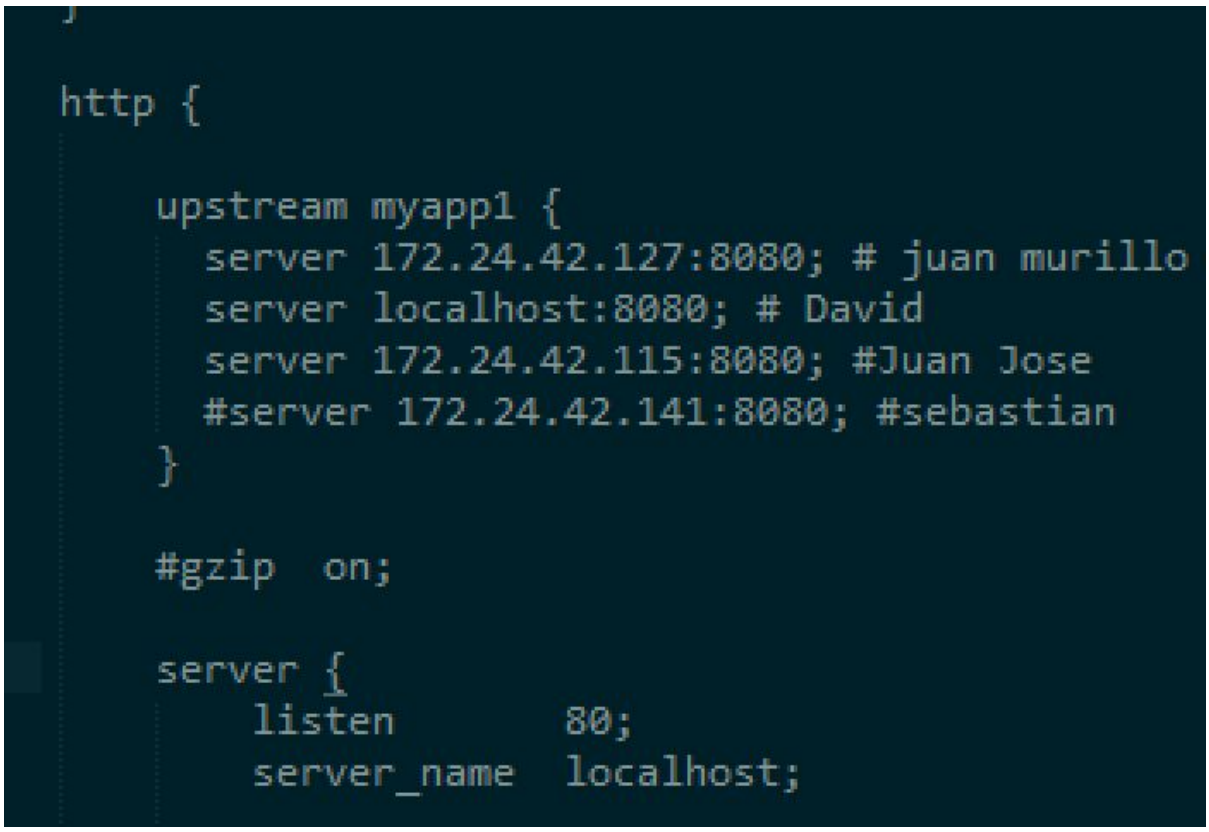
Implementación del balanceador de carga, pruebas de carga sobre el mismo y comparación



Reload del proxy de Nginx cuando se cambia la configuración



Una de las instancias de la aplicación aceptando peticiones del balanceador



Archivo de configuración del balanceador

Implementamos un balanceador de carga utilizando Nginx. Planteamos inicialmente que cada instancia a la cual el proxy podría reenviar la petición fuera una app desplegada en Heroku, sin embargo, debido a que, primero, Heroku obliga a usar una sesión segura ya sea SSL o TLS, lo que implicaba agregar una capa más de complejidad a la implementación del balanceador de carga. Lo intentamos, pero por falta de tiempo no logramos que se intercambiaran los certificados auto-firmados y decidimos entonces simplificar el modelo, a un balanceador desplegado en una de nuestras máquinas virtuales. Este balanceador repartiría las peticiones con un modelo round robin entre las máquinas de mis compañeros, agregando un poco de latencia, pero aumentando la disponibilidad.

Configuramos las pruebas con Jmeter locales en la misma máquina donde estamos corriendo el balanceador de carga Nginx.

Login:
1000 hilos, ramp up de 10 segundos

Podemos ver que a diferencia de correr la aplicación y golpear directamente un servidor desplegado en el puerto 8080, el simple hecho de tener el balanceador de carga a 3 máquinas distintas puede generar una latencia entre 4 y 460 ms extra para los usuarios que apenas se conectan para usar el servicio de la aplicación.

Esperamos que esta latencia tenga esta varianza constante, pues así nos aseguramos, dados los resultados de las pruebas de carga anteriores, que nuestra aplicación no superá tiempos de respuesta de 600 ms incluso para las operaciones más pesadas y concurrentes, como son por ejemplo los reportes continuos de los 4800 sensores de la red de campos de Well-col

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
login	1000	53	5	460	50,40	0,00%	94,1/sec	363,04	3951,0
TOTAL	1000	53	5	460	50,40	0,00%	94,1/sec	363,04	3951,0

Para 2500 usuarios en un ramp up de 10 segundos:
Aquí notamos que al aumentar el número de hilos la carga de la aplicación no es la misma, especialmente porque ahora está siendo desplegada en un computador local y no en un cloud escalable de pago (el cual estamos pagando) como Heroku. Nos damos cuenta que de manera inevitable el % error aparece mucho más pronto que antes (2380 threads activos). Ante esto, vemos la necesidad de resolver los problemas de implementación que tuvimos inicialmente.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
login	2380	3038	4	7186	2593,83	3,19%	136,0/sec	517,33	3895,2
TOTAL	2380	3038	4	7186	2593,83	3,19%	136,0/sec	517,33	3895,2

Esto no es nada con nuestras métricas pasadas, donde los primeros errores aparecían hasta los 50000 hilos concurrentes:

# Threads	Media (ms)	% Error	Rendimiento (Threads/ sec)
10	387	0	0,18
100	168	0	1,7
500	129	0	8,3
1000	132	0	16,6
5000	134	0	81,1
8500	187	0	129,6
10000	136	0	160
50000	416	2,19	354,5