Prueba desarrollador Back End

Diego Manjarrés

Coding challenge.

La solución fue desarrollada en JavaScript en el framework NodeJs.
Es necesario tener instalado NodeJs y para iniciar la solución solo es necesario ejecutar npm install y luego npm start , esto inicia un pequeño servidor que expone un archivo html para interactuar con un servicio POST que maneja el requerimiento.

Las pruebas se ejecutan con el comando npm run test.

Las capas son:

Vista

Esta capa se encuentra en la carpeta /routes y contiene la relación de las rutas expuestas por el microservicio, con la lógica de negocio correspondiente.

Negocio

Esta capa se encuentra en la carpeta /business y contiene la lógica de negocio necesaria para responder los llamados de la capa de vista, aquí se encuentran los archivos cube-logic que representa las acciones de negocio que se realizan con los cubos, y cube-business-object que representa un prototipo(objeto) de cubo con los métodos correspondientes a un cubo, según su abstracción en OOP.

Persistencia

Esta aplicación no requiere persistencia, pero de ser así, existiría otra capa que se encarga de almacenar los objetos de negocio.

Aparte de estas capas existe la carpeta static, que permite exponer un archivo html que consiste en una aplicación Angular compacta que sólo se encarga de hacer la consulta al servicio del requerimiento. Este archivo está completamente desacoplado y existe solo con el objetivo de hacer fácil la revisión de la funcionalidad. Este desacoplamiento permite que el back end de la solución pueda ser probado por aplicaciones de front-end totalmente independientes.

Refactoring Challenge

```
public function post_confirm() {
    $id = Input::get('service_id');
    $servicio = Service::find($id);
    //dd($servicio);
    if ($servicio != NULL) {
        if ($servicio->status_id == '6') {
            return Response::json(array('error' => '2'));
        }
        if ($servicio->driver_id == NULL && $servicio->status_id == '1') {
            $servicio = Service::update($id, array(
                        'driver_id' => Input::get('driver_id'),
                        'status_id' => '2'
                            //Up Carro
                            //,'pwd' => md5(Input::get('pwd'))
            ));
            Driver::update(Input::get('driver_id'), array(
                "available" => '0'
            ));
            $driverTmp = Driver::find(Input::get('driver_id'));
            Service::update($id, array(
                'car_id' => $driverTmp->car_id
                    //Up Carro
                    //,'pwd' => md5(Input::get('pwd'))
            ));
            //Notificar a usuario!!
            $pushMessage = 'Tu servicio ha sido confirmado!';
            /* $servicio = Service::find($id);
              $push = Push::make();
              if ($servicio->user->type == '1') {//iPhone
              $pushAns = $push->ios($servicio->user->uuid, $pushMessage);
              } else {
              $pushAns = $push->android($servicio->user->uuid, $pushMessage);
              } */
            $servicio = Service::find($id);
            $push = Push::make();
            if ($servicio->user->uuid == '') {
                return Response::json(array('error' => '0'));
            }
            if ($servicio->user->type == '1') {//iPhone
                $result = $push->ios($servicio->user->uuid, $pushMessage, 1, 'honk.wav', 'Open', array('serviceId' => $servicio->id));
            } else {
                $result = $push->android2($servicio->user->uuid, $pushMessage, 1, 'default', 'Open', array('serviceId' => $servicio->id));
            }
            return Response::json(array('error' => '0'));
        } else {
            return Response::json(array('error' => '1'));
        }
    } else {
        return Response::json(array('error' => '3'));
    }
}
```

Aunque no voy a mostrar una nueva versión del código, a primera vista se observan algunas malas prácticas.

En el código se evidencian varios errores comunes que se pueden catalogar como malas prácticas de programación.

Verde: ifs anidados, no se recomienda profundizar más de 3 niveles y en el código hay 4, es preferible evaluar condiciones de error primero y hacer el código más plano.

Azul: Mezcla de idiomas, el código debería ser en inglés para poder ser mantenido por más personas pero si se escoge otro idioma debería ser uniforme en todo el código.

Cyan y Amarillo: No se usan constantes, este tipo de strings deberían declararse como constantes para hacer más fácil el cambio, de ser necesario. No usar constantes también hace más propensos los errores.

Magenta: Longitud de las líneas. Las líneas más largas deberían ser de alrededor de 80 o 100 caracteres. Esto se puede lograr encapsulando parámetros o declarando variables en diferentes líneas.

Preguntas

Preguntas

1. ¿En qué consiste el principio de responsabilidad única? ¿Cuál es su propósito?

Consiste en que, en un producto de software, cada clase, módulo, archivo… debe ser responsable de una única parte de la funcionalidad e implementarla correctamente.
El uso de este principio no tiene impacto sobre el desempeño del producto, pero es muy importante para la mantenibilidad y escalabilidad del mismo.
Cuando se aplica correctamente, y se usan convenciones de nombres coherentes, se logra que los desarrolladores que trabajen sobre el producto, logren encontrar más rápidamente la lógica para cada funcionalidad y sean más eficientes al hacer cambios o diagnosticar errores.

2. ¿Qué características tiene según tu opinión “buen” código o código limpio?

Para mí, la característica más importante de un código limpio, es que, un grupo significativo de desarrolladores experimentados en las herramientas utilizadas, logren entenderlo. Actualmente, gracias a los compiladores modernos, el desempeño del software es muy similar a pesar de diferencias en la implementación. Por esta razón la calidad del software depende más de la mantenibilidad, qué tan fácil es de escalar, mantener o depurar.