Bienvenidos a la clase 32. Hoy vamos a hacer una introducción a testing, sobre como se implementan en el desarrollo de nuestra aplicación. Para después comprender la tecnica de TDD. Vamos a ver el concepto de mocking de datos y vamos a utilizar la librería de faker.js para generar datos falsos que nos sirvan para probar nuestra app.

Vamos con testing primero, a muchos les encanta el testing y a otros no les gusta, pero deberían, como desarrollador comprender los conceptos básicos porque alguna vez en la vida les va a tocar o pasar test o crear test y además esto les suma muchisimo en su curriculum.

En un equipo de desarrollo, la forma normal de trabajar es el siguiente: tenemos el código en producción,... todo está funcionando. Y se nos asigna una tarea, para modificar o agregar algo en ese proyecto, ya sea arreglar algo o implementar una nueva funcionalidad o quitar algo que ya no se usa. Entonces el desarrollador toma una copia del código en producción y empieza a trabajar en una rama nueva. Entonces, tomamos una copia del codigo que está productivo, que esta funcionando correctamente y de forma local, hacemos lo que nos pidieron, nuestros cambios, de ese modo nosotros podemos tocar lo que sea que el código de producción no se rompe. Entonces trabajamos en nuestra tarea hasta que terminamos, creemos que funciona todo, y solicitamos agregar ese cambio al codigo de produccion para fusionar los cambios de nuestra rama a la rama principal. Cuando aprueban el cambio se fusiona y el codigo de produccion se actualiza…. Entonces, ¿como demostras que el codigo esta bien y que no rompiste nada del codigo que estaba funcionando?.

Tenemos 2 caminos, uno, probar todo manualmente, por ej con postman, probamos lo que hicimos y documentamos las pruebas para que el que tenga que aprobar nuestro cambio tambien esté seguro de las pruebas que hicimos y tenga que hacer esas pruebas tambien. Ahora, que pasa si el proyecto es muy grande, como generalmente pasa, porque, nos puede tocar arrancar un proyecto de cero, pero casi siempre vamos a tocar un proyecto ya iniciado, que está funcionando. Por ej, tenemos, nose, 200 endpoints. Entonces, ahi lo optimo seria utilizar una herramienta de testing, donde la idea es que se hagan pruebas automaticas, que si dan como resultado que esta todo bien, es porque en nuestro código esta todo ok. Te ayuda a que te quedes tranquilo de que si subis algo al codigo principal, este funcionando todo como marcan los test. Se entiende hasta ahi?.

En lo que respecta a javascript tenemos un par de herramientas para implementar testing. Las dos más conocidas son jest y mocha, son 2 librerias que te ayudan a realizar testing automatico, por ahi jest es un poco mas popular porque se puede aplicar tanto en el back como en el front y mocha creo que es solo en el backend. Pero son muy similares. Vamos a un pequeño ejemplo:

((IR A 01-TESTS)) / calculadora.js

A grandes rasgos esto es lo que se hace con testing, mas adelante tenemos una clase exclusiva de esto y ahi vamos a ver un par de funciones mas, pero en esta clase nos toca esta mini introducción.

También queria mencionarles que tenemos 2 estrategias de testing, el test ***unitario*** en donde se testea un fragmento concreto de codigo, una sola funcionalidad... Por ejemplo: si testeamos un controlador que trae información de una base de datos y la devuelve. En un test unitario, esos datos lo tenemos que simular, no van a ser los datos que vienen de la db, porque estamos testeando la funcion. Se entiende?,
otro ejemplo seria si hacemos test unitario en un controlador que guarda usuarios, podemos hacer una validación si los datos son correctos, entonces ahí tenemos el test unitario, antes de hacer el registro en la base de datos.

La comunicación con la base de datos la puedo probar en otro tipo de estrategia que se llama test de ***integración***. Y es ahi donde se prueba un proceso o un flujo parcial,... por ej cuando se llama a una funcion que se comunique con la base de datos y que guarde, que elimine, que actualice...Se entiende la diferencia?

Osea...
En el unitario falsificas la comunicación con la base de datos, en el de integración no.

También tenemos el test funcional, que es más para el cliente que para nosotros, nos sirve para probar que el flujo de la aplicación funciona de punta a punta, sin dejar ningún cabo suelto.

Esta introducción minima de lo que es testing, no está en el contenido de la clase pero se las queria hacer para que tengan una idea antes de ver lo que es TDD.

--DIAP 6--

Bueno, entonces, ***TDD*** las siglas significan Test Driven Development – ***(Desarrollo dirigido por tests)***, es una estrategia de programación que consiste en escribir primero los tests, generalmente se usa con pruebas unitarias.. y después escribir el código que haga que esa prueba funcione y por último refactorizar el código. Con esta practica se consigue un código más robusto, más seguro y mantenible. Entonces, asi es el circuito --DIAP 13-- Escribo las pruebas, van a fallar todos los tests porque no hay código. Y después a medida que voy agregando las funciones, voy haciendo pasar los tests. Por ahi la ventaja seria, que si arrancás por las pruebas, ya sabes exactamente lo que queres que haga tu aplicación. Entonces... en ese primer intento para que pase el test, no vas a hacer el mejor código del mundo sino que vas a tratar de que pase el test correctamente y después una vez que ya estamos seguros, refactorizamos, mejoramos el código.
TDD, esta más pensado para test unitarios, si tenes que hacer un test de integración o funcional, no aplica tanto TDD. Igualmente,... la mayoría de las empresas trabajan de otra forma, te asignan una tarea, haces el trabajo, probas manualmente todo y: o te piden que hagas los test y no te aprueban los cambios, salvo que tengas los tests que verifiquen que lo tuyo funciona.

--DIAP 35--

Bueno y para simular esos datos que les decia,se aplica la técnica de MOCK, que son datos falsos que utilizamos justamente para hacer pruebas, porque, por ej, si quiero probar una función que borre todos los datos de una base de datos, no lo voy a hacer con la base de datos real, me echan en ese mismo instante.. se prueba todo con datos falsos.

--DIAP 36--

Acá nos dicen, podemos tomar **mock** como una “imitación” de un dato real. Nos resulta útil para poder crear datos “supuestos” con el fin de probar la funcionalidad de alguna función.
Un dato mock no debe comprometer jamás una estructura productiva, por lo que solo se usa en entornos de desarrollo.

Entonces, según la estrategia, unitaria, integral o funcional, vamos a mockear mas o menos datos.

La idea ahora es mostrarles un servidor que devuelva datos falsos, como para que el front tenga los datos que necesita.

--DIAP 21--

Supongamos que se reunen el equipo de backend y el de frontend para iniciar el desarrollo de una aplicación. Y bueno, se ponen de acuerdo sobre que datos necesita el front, pero el lider del equipo de back les dice a los de front, mira, por todo lo que piden, vamos a tardar 3 meses masomenos y el lider de front le dice: no, yo necesito tener los datos ya, para ir mostrando, viendo

como acomodar la informacion, etc. Entonces, ahí la solución es crear una API paralela, con datos falsos digamos. Para que el front ya pueda trabajar con esos datos falsos.
Entonces les decimos a los de front
aca estan los endpoints con los datos falsos, tomá hace tus pruebas con estos datos y dejame trabajar tranquilo a mi.

--DIAP 38--

Entonces ahi entra en escena una librería bastante útil que nos trae estos datos falsos de manera aleatoria solamente con ejecutar metodos que ya vienen implementados en esta librería. Se llama faker.js, podemos traernos nombres, emails, direcciones, cuentas bancarias, etc etc.

Bueno, hace unos meses, el creador de faker, se calentó con todas las empresas que utilizaban su librería y nunca le dieron un peso, entonces un dia agarró y borro todo de npm digamos. Cerró la librería. Pero, antes que se borrara, alguien copió la última versión de faker y creo una librería a partir de lo último que se subió de faker. Entonces podemos usar esa. Se llama faker-js/faker

https://www.npmjs.com/package/@faker-js/faker

https://fakerjs.dev/api/

Fijense como creció de golpe las descargas semanales, cuando la comunidad se enteró de esto.
Esto es legal porque es codigo abierto, digamos, esta copia fue de la version que era gratuita y funciona exactamente igual que lo que era faker originalmente. Entonces vamos a hacer esta api con faker.

((VS CODE 03-faker))