


# Breve tutorial de express.js + typeORM



Fernando García Hernández

Jul 17, 2018 · 6 min read

Follow



Recientemente he estado desarrollando *APIs* en **Node.js** y he querido usar bases de datos **MySQL**. Si bien es cierto que existen librerías para **node** que permiten hacer consultas a MySQL “a pelo”, no encontré muchos **ORM** que agilizaran el proceso y a la vez permitieran un diseño sencillo y eficiente.

Tras una breve búsqueda encontré *TypeORM*, que aparte de cumplir los requisitos anteriores, trabaja muy bien con **TypeScript**. ¡Genial! ¿Pero cómo se usa? *TypeORM* tiene una buena documentación, pero opino que ofrecen **demasiadas alternativas para realizar una misma tarea** y al final acaba siendo algo confuso. En este artículo explicaré cómo suelo implementarlo en mis proyectos. ¡Vamos allá!



Para empezar, crearemos un directorio nuevo para el proyecto, yo lo llamaré *tutorial\_typeorm*. En una consola de comandos, ejecutamos `npm init`

He creado un repositorio en *github* con el resultado final.



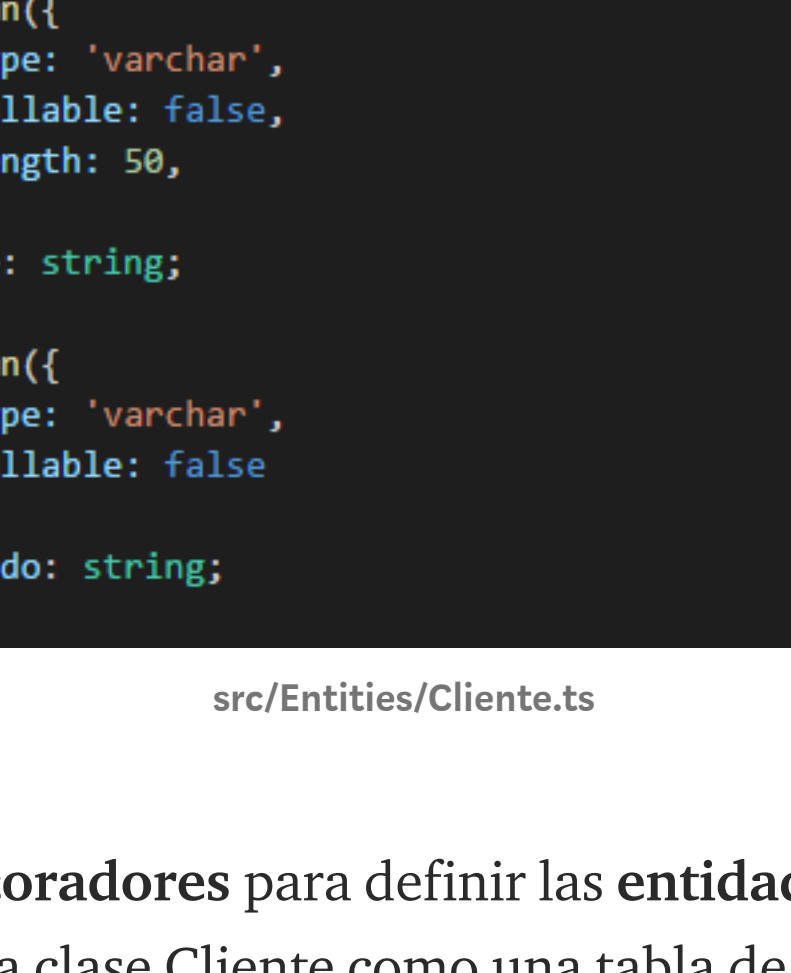
Es necesario instalar algunas dependencias. Para ello, abrimos consola de comandos y escribimos:

- `npm install --save typeorm express mysql2` con lo que instalaremos *TypeORM*, MySQL y Express.
- `npm install --save-dev typescript ts-node @types/express` para instalar TypeScript, TS-Node y el tipado de Express como dependencias de desarrollo.

Es necesario añadir en la raíz del proyecto un fichero `tsconfig.json`. En el repositorio hay uno que funciona bien con este proyecto, copiadlo de ahí o haced uno propio.



La estructura de carpetas será la siguiente:



Estructura

- En *Entities* crearemos las clases relacionadas con las tablas de la base de datos. En cada clase definiremos las columnas y sus tipos de datos.
- En *Repository*, crearemos una clase por cada entidad con los métodos necesarios para obtener, editar o eliminar registros de la base de datos.
- Por último, en *server.ts* escribiremos nuestra *API*. Empecemos:

Vamos a crear una *API REST* que gestione una tienda de música en la que manejaremos la información de los dependientes, los clientes y los álbumes de música de la tienda.

Para ello, necesitamos que nuestra base de datos tenga tres tablas, una para clientes, otra para los dependientes y otra para álbumes. Vamos a *Entities* y añadimos un nuevo fichero *Cliente.ts*



src/Entities/Cliente.ts

TypeORM utiliza **decoradores** para definir las **entidades** y las **columnas**. En este caso, define la clase *Cliente* como una tabla de la base de datos con el decorador `@Entity`. También tiene algunos decoradores para definir columnas de la tabla.

- `@PrimaryGeneratedColumn()` indica que la columna *id* es primaria y autoincrementa su valor. Además, el parametro *uuid* indica que no es un id numérico sino un **string aleatorio y único**.
- `@Column()` sirve para crear una columna en la tabla. **Puede recibir un objeto como parámetro** para definir el tipo de la columna, si puede tener valor nulo, la longitud máxima... más parámetros [aquí](#).
- `@PrimaryColumn()` sirve para indicar que la columna es **llave primaria**.
- [Aquí](#) hay más tipos de columnas.

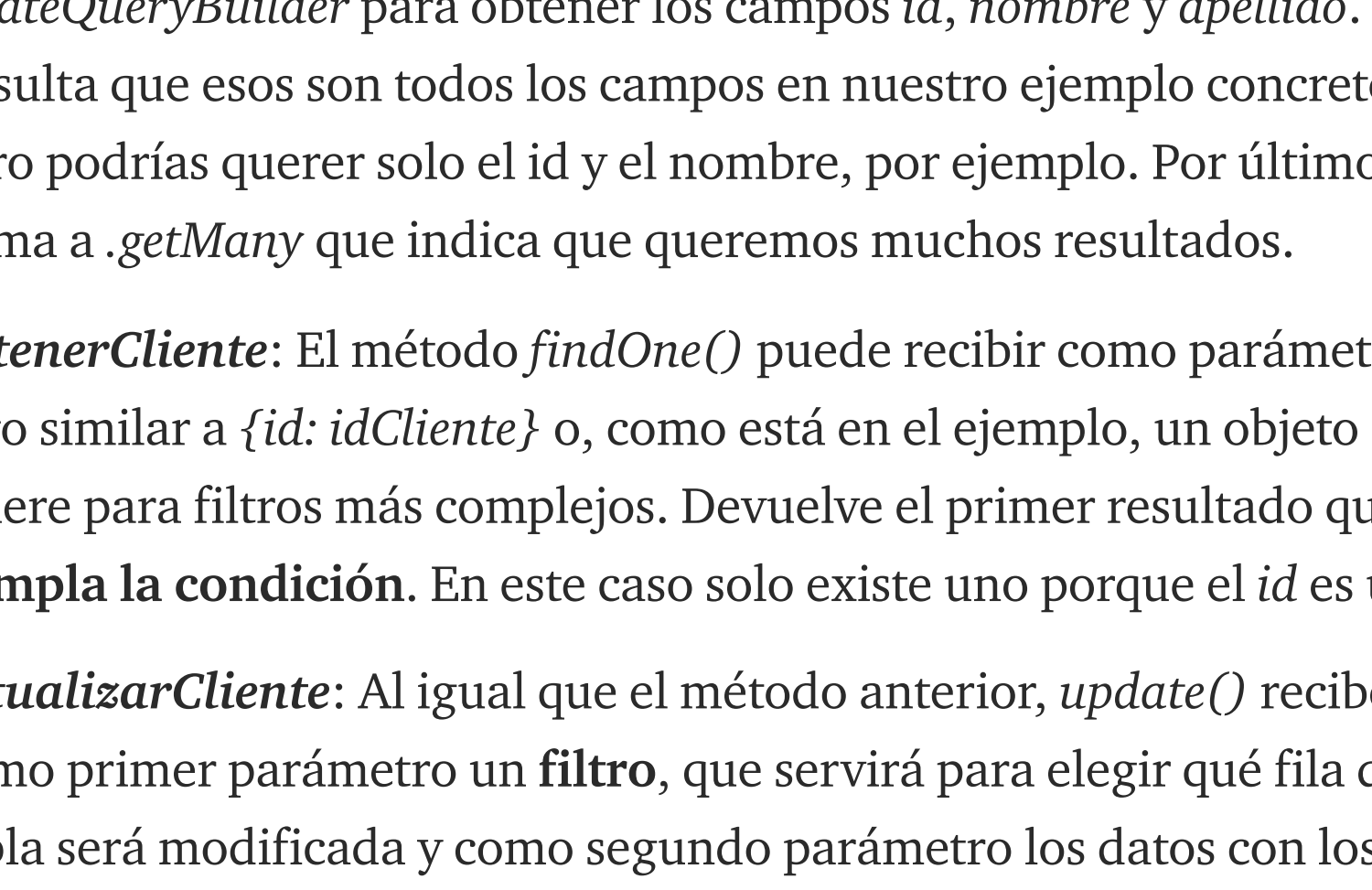
TypeORM, mediante herencia de clases, nos permite crear **entidades que comparten columnas** sin necesidad de definirlo explícitamente en cada entidad. Personalmente, prefiero que cada entidad defina sus columnas porque resulta más claro, pero hagamos uso de esta funcionalidad en la entidad *Dependiente.ts*:



src/Entities/Dependiente.ts

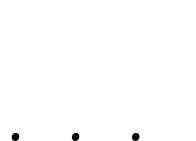
Con este código, tenemos todas las columnas definidas en *Cliente* y además la columna “salario”. ¡Así de fácil!

Por último, creemos *Album.ts*:



src/Entities/Album.ts

TypeORM nos permite además almacenar **arrays de primitivos** en la base de datos. Podéis verlo en la columna *tracklist*: *tracklist: String[]* será una **columna** que almacenará **arrays de strings**. De este modo nos ahorramos usar relaciones en casos sencillos.



Ya tenemos nuestras entidades bien definidas, ahora definamos los **repositorios** para gestionar las **transacciones** con la base de datos para cada entidad.

En *src/Repository*, creamos un nuevo fichero llamado *Cliente-Repositorio.ts*:



src/Repository/Cliente-Repositorio.ts

Detengámonos un momento. Aquí hay varias cosas en las que fijarnos. Primero, creamos una clase *ClienteRepositorio* cuyos métodos son las distintas transacciones con la base de datos. TypeORM funciona **asíncronamente** en este punto, de modo que todas estas operaciones devuelven una **promesa**. Vamos método a método.

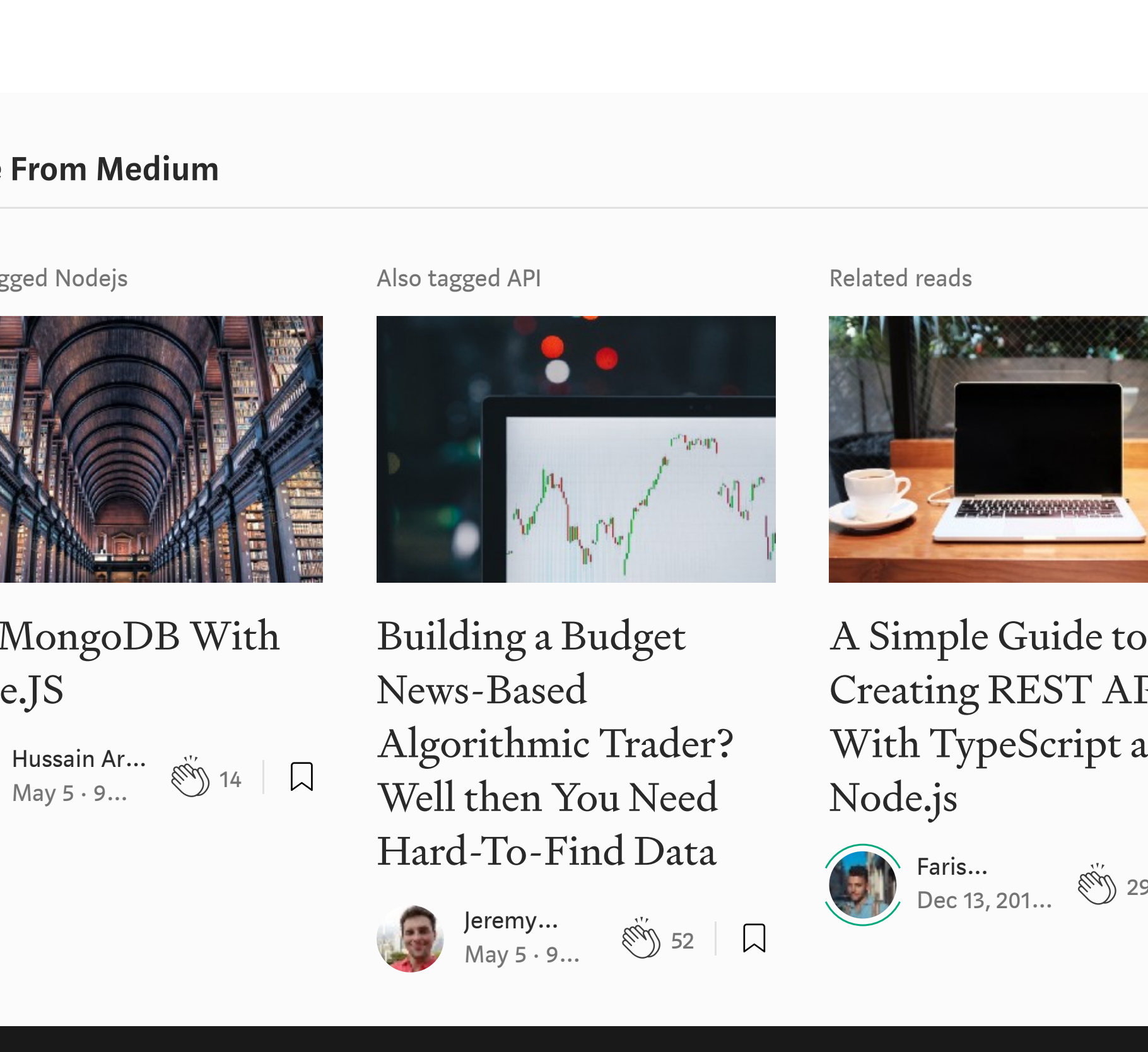
- **crearCliente**: recibe un objeto de tipo *Cliente* (definido en *Entities*) y devuelve una promesa que, al resolverse, da el cliente guardado.*getManager().getRepository(Entidad)* obtiene el **Repositorio** de la entidad con la que estamos trabajando, dicho repositorio tiene los métodos que acceden a la base de datos. En el caso de *crearCliente*, necesitamos el método **save**, que requiere la instancia del objeto de tipo *Cliente* que recibimos por parámetros.
- **obtenerListaClientes**: Devuelve una promesa que se resuelve en un **array de objetos de tipo Cliente**. He utilizado el método *createQueryBuilder* para obtener los campos *id*, *nombre* y *apellido*. Resulta que esos son todos los campos en nuestro ejemplo concreto, pero podrías querer solo el id y el nombre, por ejemplo. Por último, se llama a *.getMany()* que indica que queremos muchos resultados.
- **obtenerCliente**: El método *findOne()* puede recibir como parámetro algo similar a *{id: idCliente}* o, como está en el ejemplo, un objeto con un *where* para filtros más complejos. Devuelve el primer resultado que **cumpla la condición**. En este caso solo existe uno porque el *id* es único.
- **actualizarCliente**: Al igual que el método anterior, *update()* recibe como primer parámetro un **filtro**, que servirá para elegir qué fila de la tabla será modificada y como segundo parámetro los datos con los que se **sustituirá**.
- **eliminarCliente**: Solo necesita un **filtro** para determinar qué fila debe eliminar. Al igual que en *obtenerCliente* y *actualizarCliente*, este filtro podría ser un **objeto** con un *where*: *{}*.

- Aparte de *findOne()*, tenemos *find()* que devuelve todos los resultados que cumplan la condición.
- El método *createQueryBuilder* ofrece muchísimas opciones, puedes encontrarlas [aquí](#).

Tanto *Album-Repositorio* como *Dependiente-Repositorio* son muy similares a *Cliente-Repositorio*, por tanto puedes copiarlos del repo de GitHub o hacerlos por tu cuenta fijándote en *Cliente-Repositorio*.



Ahora que tenemos la parte de base de datos controlada, hagamos una *API* sencilla para gestionar los clientes. Editemos *server.ts*:



src/server.ts

Hacen falta algunas cosas para ponerlo todo en marcha.

- Primero, importar los *repositorios* que hemos creado para acceder a sus métodos.
- Segundo, crear la conexión. TypeORM tiene un método que se encarga de eso, se llama *createConnection*. Para usarlo podemos especificar como parámetro un objeto así:



método createConnection en server.ts

Alternativamente, y en mi opinión mucho mejor, podemos crear un fichero en la raíz del proyecto con el nombre ‘*ormconfig.json*’ que contenga ese objeto json que le pasamos por parámetro al método *createConnection*, con una salvedad.



ormconfig.json

El campo *entities* no tiene un objeto de cada entidad sino la **ubicación** de las entidades en tu proyecto. **¡Ojo al dato!** Una vez terminado el desarrollo y el proyecto haya sido **transpilado** (para pasar de *TypeScript* a *JavaScript*), es importante cambiarla ruta a la que apunta *entities* para indicarle que busque ficheros *.js* y no *.ts*. Ejemplo: `"entities": ["dist/Entities/*.js"]`



¡Eso es todo por ahora! En la próxima entrega del tutorial cubriré las **relaciones entre tablas** y cómo **organizar el código de la API** para que no se mezcle todo en *server.ts*



Nodejs

Expressjs

Typeorm

MySQL

API



157 claps

Twitter

LinkedIn

Facebook

Bookmark

WRITTEN BY



Fernando García Hernández

Follow

See responses (3)

## More From Medium

Also tagged Nodejs

Also tagged API

Related reads



Use MongoDB With Node.JS

Hussain Ar... May 5 · 9... 14

Building a Budget News-Based Algorithmic Trader? Well then You Need Hard-To-Find Data

Jeremy... May 5 · 9... 52

A Simple Guide to Creating REST APIs With TypeScript and Node.js

Faris... Dec 13, 201... 290

Discover Medium

Make Medium yours

Become a member

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just \$5/month. Upgrade