

## Senior developer test

The following test has been put together to try and test your skills has a SENIOR developer. These are some of the tasks that have already been done in our team.

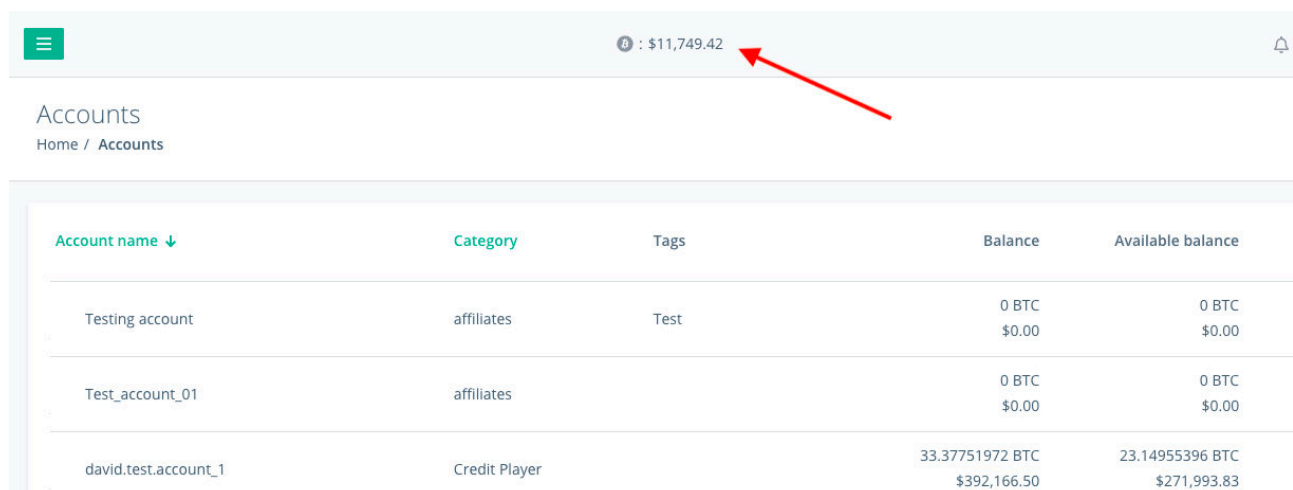
Our stack is Angular and NestJS, a NodeJS framework ( <https://angular.io/> & <https://nestjs.com/>). It's highly recommended that you try to resolve the test using these technologies but if you can't, please, change Angular for React and NestJS for pure NodeJS with express – for example. Although always try to use Typescript over Javascript – our language of choice.

### Part 1

Create a web application that displays a list of accounts (data table) showing the balance and available balance in both BTC (bitcoin) and the equivalent in Dollars. There is an example below (screenshot), where the column names are *Account Name*, *Category*, *Tag*, *Balance* and *Available balance*. *Category* and *Tag* are not important and return something at random.

The NestJS / NodeJS backend should return from a REST endpoint a minimum of 15 accounts (rows) so we have data to view. *Balance and Available Balance* are displayed both in BTC and Dollars. Your dataset should only contain BTC, the value of the Dollar is calculated using an exchange rate.

The backend should return the current exchange rate when the web page is first loaded and this value should be used to calculate the Balance and Available Balance. The current exchange rate should be displayed on the screen (like in the screenshot).



The screenshot shows a web application interface. At the top, there is a header bar with a green menu icon on the left, a Bitcoin icon followed by the text ': \$11,749.42' in the center, and a bell icon on the right. Below the header, the page title 'Accounts' is displayed, followed by a breadcrumb 'Home / Accounts'. The main content area features a table with the following columns: 'Account name' (with a dropdown arrow), 'Category', 'Tags', 'Balance', and 'Available balance'. The table contains three rows of data. A red arrow points from the exchange rate text in the header to the 'Balance' column of the table.

Account name ↓	Category	Tags	Balance	Available balance
Testing account	affiliates	Test	0 BTC \$0.00	0 BTC \$0.00
Test_account_01	affiliates		0 BTC \$0.00	0 BTC \$0.00
david.test.account_1	Credit Player		33.37751972 BTC \$392,166.50	23.14955396 BTC \$271,993.83

## Part 2

Now, let's make it more interesting. A new exchange rate should be pushed from the backend to the frontend every 30 seconds using websockets. The value of the new exchange rate is random but try to send within a realistic range (e.g from \$5000 to \$12000).

As new values arrive at the frontend the UI should update the exchange rate displayed at the top of the page (screenshot above) and also re-calculate all balance and available balances inside the table.

## Part 3

As users can send or receive BTC within an account, the balance and available balance can eventually change.

Each time the balance of an account changes the frontend should receive an update of the BTC balance and available balance. To simulate this changes just set a random interval (between 20 to 40 seconds) to send this update to the front end for the specific account (data table row).

To highlight the change, the background color of the specific row should flash one of the following colors:

- Red: if the new available balance is **lower** than the previous value.
- Green: if the new available balance is **higher** than the previous value.
- No flash color if the available balance does **not** change.

## Part 4

Finally clicking on an account row should open a new page with the account details (but keeping within the Single Page application). This is a pure Master / Detail implementation. Master being the account list and details being the information of the account the user clicked on.

The detailed page should show some transactions that belong to the specific account in a data table. Here is an example:

Account detail

Home / Accounts / Details

felixAccountTest

Active

Treasury account

1.05040000 BTC (\$10,359.99)

(includes 0 BTC (\$0.00) of unconfirmed funds)

Available balance: 1.05040000 BTC (\$10,359.99)

Account information

Statement

Show unconfirmed transactions

Select date range

Clear selected dates

Confirmed date	Order ID	Order code	Transaction type	Debit	Credit	Balance
02/11/2020 14:25	BXT7GU	SETTLEMENT	Payment received		0.00040000 BTC \$3.95	1.05040000 BTC \$10,359.99
02/11/2020 14:16	SQBRMS	ON RAMP	Payment received		0.05000000 BTC \$493.15	1.05000000 BTC \$10,356.05
02/11/2020 14:16	FF5XWX	DEPOSIT	Payment received		1.00000000 BTC \$9,862.90	1.00000000 BTC \$9,862.90

< >

Items per page: 10

The transactions table should show an Order Code and Order Id – these should be unique values and also an amount for the Debit or Credit columns (depending if the user sent or received BTC in this transaction). Use the example mock above. Again the transaction should be related to the account (do not show transactions that belong to one account in another).

As you can notice the Debit, Credit and Balance fields in the transactions displays BTC and Dollar. The Dollar values has to be calculated according the current exchange rate and re-calculated each time a new exchange rate is received in the front end.

Also if an update for the Balance and Available Balance for this specific account is received this should be highlighted with a flash color of the text for the account Balance and Available Balance displayed in the details page. Use the same colors flash conditions described in the part 3.

## Recommendations

Again we encourage you to use our stack technologies: Angular and NestJS.

You are free to use what you want for your datasource, an in-memory collection or even better something like Mongo. NestJS supports MongoDB out of the box – the choice is yours! If you use MongoDB then you should provide us with a seeding script.

The Typescript `<any>` type makes us sad, please avoid it and always create types where you can. Feel free to make good use of OOP. The use of abstract classes, interfaces, properties and generics are all pluses!

You should try and make use of good styling, using something like material design (angular material).

Finally we would like you to include some unit tests for Angular (1 method for 1 component and 1 method for 1 service) and Nestjs (1 method for 1 controller and 1 method for 1 service would be ideal).

All code should be manually tested and uploaded to a github repository with detailed instructions of how to execute it – step by step.