ASSIGNMENT NOTES

Initial and personal comments (not to justify myself, but to explain what happened)

- This is my first time using NestJS. As all frameworks, it probably presents several ways of configuring routes, and there will be an optimal way of doing it, but of course I went to the most straightforward way I found.
- I haven't used this techstack in more than a year, so I was super rusty and I had to spend some time figuring out syntax and common methods available on objects.
- I chose Intellij Community as an IDE, because I'm very familiar with it, but after a while I figured out it has no support for Javascript and Typescript, so no code completion and function navigator. I could have switched to another IDE (like VScode), but I thought that installing and configuring it would take away precious time from the task. In hindsight, I should have done it anyway, because I wasted a lot of time trying to figure out which functions are available on a certain object/variable, which would have been smooth if the IDE had supported Typescript. Valuable lesson learned.
- Because of unsupportive IDE, there might be unoptimized imports and such. In real life, I would spend time configuring the IDE to support all my needs.
- I preferred using the last half hour available to write this document, to be sure you guys have more context to judge on my work, instead of spending it to develop an extra step, but leaving you clueless as to why I have written what I have written.

Code comments

- To mock the API transaction call, I used Mock Service Worker, again, not because it's necessarily the best tool, but because it's the first one I found. In a real life scenario of course I would take my time to figure out which mocking tool is best, if I had to mock calls and objects in my tests.
- I didn't have time to make a dynamic mocked call, based on the query parameters shown in your example. So my mock is static, and it always returns the same object. A more complex mock could return different objects based on the query parameters (not only the start and end date, but more). The mocked call should also return status codes.
- The getUser endpoint makes a call to the mocked API to fetch the list of users. Clearly this is not sustainable in a real life scenario with limitations on the amount of calls I can perform, etc.
- The code could be structured in a nicer way following a MVC pattern. I am not familiar with how NestJS splits the code into controller/module/service source files, but I feel that it is something along the same lines. So the business logic on filtering the user list by user ID, for example, could be moved into the service file, and the UserAggregator DTO class could be moved into the module file (provided that it makes sense, I'm just guessing).
- I forgot to run the linter and formatter, but they would have been useful in making the code more readable.
- There is no error handling in the route or in the mocked call. In real life, endpoints should cover as much as possible not only the happy path, but the scenarios where something goes wrong:

- the data doesn't have the proper structure or some fields are null
- the mocked API returns an error and the list is unavailable
- A promise cannot be fulfilled and the appropriate error must be propagated

Git comments

- The repo history is not great, the last two commits could have been squashed into one, and so would all the "update readme" commits. I would have used git rebase -i to make the history more linear.
- I went straight into coding, but of course good practice would demand that I create a new branch to work with.
- It would have been nice to set up a GitHub Actions pipeline, so that whenever I pushed new commits, the pipeline could at least run linter, formatter and unit tests.
- I forgot to update the .gitignore with at least the .idea files that belong to Intellij and nobody cares about.

Testing comments

- I just added an example of how an endpoint test could work, because of time constraints. A real life scenario would imply mocking test objects, instead of using the mocked API call, which would allow to cover edge cases.
- If the business logic was moved into the service module, then there could be intermediate unit testing of, for example, the function that filters data by user ID (not necessarily 100% test coverage is to be targeted, in this case endpoint tests might be more useful in covering the whole endpoint flow, including edge cases).
- I would add tests to check the format of the data (in case the compiler doesn't already take care of this) and the behaviour in case of null fields.
- I would also cover expected behaviour in case of errors being returned (for example, test that an invalid/nonexistent user ID returns the appropriate error code).
- Regarding the TDD question, I'm not sure that would be my approach. But in that case, TDD means that you start writing tests that cover possible (edge) cases, and then develop the actual code based on making the tests succeed (a red-green-refactor approach could be used). So you start with simple tests (checking the endpoint works in case of a short list of users with all properties set correctly) and expand towards more complex cases (can an earned amount be negative? What can the API do in case the transaction API list fails to be fetched?).
- It's important to predict the API behaviour in case of unhappy paths and unexpected results, so I would focus towards that.

Challenge comments

- I wasn't sure about the second endpoint description and what it meant, so I didn't spend time figuring it out. In real life I would ask for clarifications, and hopefully there would be more than two lines documenting the endpoint design and its requirements.
- The metadata field in the transaction payload also confused me and I wasn't sure how to interpret it, so I left it out of the picture (again, I would have asked for clarifications).

- If I'm being honest, a strictly timed challenge is not an approach I like. I understand the reasons behind it (you want to see how the candidate performs when time-pressured, and how they react when they have limited time to figure out stuff they have no idea about), but it doesn't reflect a real life situation. If I ever happened to find myself having very few hours to develop a new app feature, for example, it wouldn't be just me working on it, and the question should more be: why did the company allow itself to be found in this situation? I don't work well under pressure, and I don't think a healthy work environment should require this from its developers. So having a flexible time range (a few days?) to work on this could also allow people to show more creativity and out-of-the-box thinking, when doing the assignment. I have very strong performance anxiety, so that didn't help at all, I fully panicked when I saw the task (maybe it's just me). I feel like it's important for me that you guys know all of this, when reviewing the code, because I think with more time and less pressure, I could have done a much better job, reading and investigating which approach to go for, before starting implementing, etc. If you decide to proceed with the in person interview, you will see that I am straightforward and honest, and I don't hide my opinion and when I disagree with someone. This said, I'm happy to discuss any of these points face to face!

Challenge constraints

- Millions of requests per day: my service might need to run on a distributed system approach. I could Dockerize the app and then instantiate Kubernetes clusters (or some other orchestrator service), where several nodes would run my containerized app on a distributed network of servers. It could be done locally, by owning myself a server farm, or the best approach would probably be renting cloud services. AWS allows to run Kubernetes on the cloud, and also offers an alternative to it, with an AWS-provided orchestration service.
- Data up to date within 2 minutes: this constraint pais up with the following ones, but also when taken by itself, it imposes a limitation on the amount of time it can take the aggregator app to fetch the transaction data. The list of users returned by the transaction API grows over time, and it could potentially take non-negligible time to fully fetch. A solution, in case we also managed the transaction API and could expand its functionalities, could be to connect it to a broker/messaging service, and push to it the list of users, whenever the list changes. No need to push the full list, but only the users that have been updated/added/deleted since last time the broker service received the list. The broker would then notify the aggregator API (hitting a specific endpoint) with the changed user list, and that could be stored somewhere in the aggregator API, so that we are sure the list of users we manipulate is always up to date and we fetch only the necessary data over the network. In case the transaction API is external or cannot be modified, even more so, it would be imperative to only fetch the data that has changed in the last few seconds, and not the full list. So the transaction API could be hit with query parameters that cover the last N seconds that passed from the last request.
- Maximum 5 requests per minute: the distributed system approach would be an advantage to increase the amount of allowed requests per minute, but it wouldn't be ideal: it would be very expensive to have multiple servers executing one request every few seconds, and being idle the rest of the time. So probably not the best

solution. Maybe a better solution would be to have a module (a separate microservice?) that only takes care of requesting data from the transaction API every 12 seconds, saturating the 5 requests per minute constraint. That way, we would have 12 seconds to fetch only the data that has changed over the last 12 seconds. This microservice could then save the fetched data into a database, so that the aggregator API never directly hits the transaction API, but it reads data from the database. Any relational database would suffice in this case, because the amount of data to be written and read wouldn't be so massive as to give performance concerns over time (in the database, there would be one row of aggregated data per each user, and the amount of new data being written would be limited by the transaction API constraints anyway).

- Maximum 1000 transactions per request: the way I interpret it, it means that the transaction API returns the data in chunks, with a maximum of 1000*chunk-size capability. So if each chunk is of 3 data as shown in the example payload, the transaction API could return a maximum of 3000 data per request, and my retrieving microservice could retrieve a maximum of 15000 new data per minute per server. We would need to investigate how often it happens that the retrieving service falls behind, and it cannot fetch all the new data within 12 seconds, because there's more. In that case it might be worth adding extra servers to run the retrieving service.
- To summarize, considering all constraints, to me a reasonable approach would be:
    - A microservice distributed system with a retrieving API, an aggregator API and a database. The retrieving API calls the transaction API every 12 seconds, fetching only the new data that was added/deleted/updated in the last 12 seconds. The retrieving API then writes the new data into our own database. The aggregator API fetches the data from the database, and manipulates it to return the most updated results to the client.
    - If scaling was necessary, it would probably be focused on the retrieving side, because the most stringent constraints are on the amount of requests and data that can be fetched per minute from the transaction API.
    - One database instance would probably be enough, but if the list of users grows fast over time, then indexing might be necessary to speed up reading operations (the ones that would happen more often than writing). Sharding could also be a solution, in case indexing wasn't enough.