DAZN - Tech Test

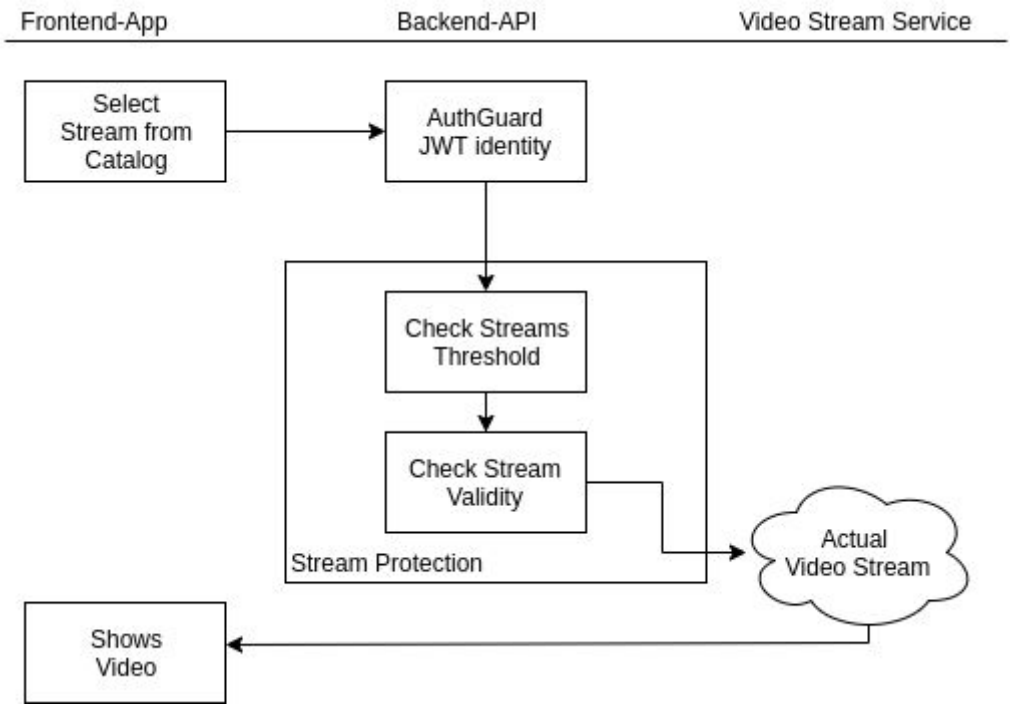
Functional Requirement

Build a service in your preferred language that exposes an API which can be consumed from any client. This service must check how many video streams a given user is watching and prevent a user watching more than 3 video streams concurrently.

My Solution

I decided to solve this problem with a simple service to be plugged on top of an endpoint to get the file stream for a logged in user.
I made the following assumptions:

- The **User** needs to be logged in to see the **stream**
- A **User maxStreams** configuration is dictated by the subscription **plan**. (I called **basic** the plan where you can see only 3 streams concurrently)
- The User can still see the **catalog** of live events even if is not logged in



Approach to the solution

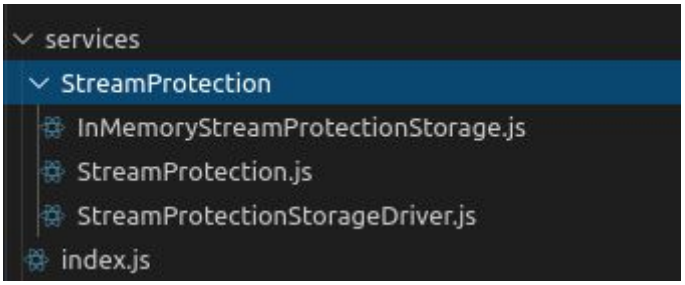
I initially thought to do something on a lower level, with an IP based download limit, but of course that won't work if multiple users will be streaming from the same IP, so I decided to go to a fully backend based protection logic.

1. I decided that I would recreate a fake DAZN website with Login and Stream Catalog. Capabilities, then I will implement the requested feature of the test in a TDD fashion.
2. I bootstrapped the BE with one of my personal boilerplate I use most of the time, instead of using express I used micro.
3. I adapted a simple login via JWT token I have used on other small side projects, custom written and quite small, just for the purpose of having the functionality of login up.
4. I bootstrapped the frontend part of the project with a simple create-react-app.
5. I designed the routes and protected routes that needs authentication with react-router.
6. Implemented and tested authentication FE side (used redux to store the global loggedIn state).
7. Implemented the catalog endpoint on the backend and linked it to the FE.
8. Implemented a fake Stream Protection Service to test the FE behaviour in both scenarios (allowed to stream or not).

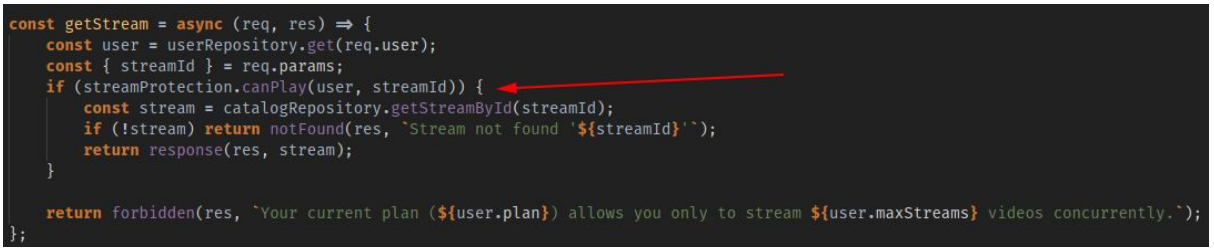
I used the codebase at this stage as a starting point for the functionality asked.
I created a suite of simple tests and then driven the development of the functionality required around those.

```
const { assert } = require('chai');
const StreamProtection = require('../../src/libs/services/StreamProtection/StreamProtection');
const InMemoryStreamProtectionStorage = require('../../src/libs/services/StreamProtection/InMemoryStreamProtectionStorage');

describe('Stream Protection tests', () => {
    const user = { id: 'userId', maxStreams: 3 };
    const streamId = 'someId';

    it('does allow a user to get a stream if he is not streaming anything', () => {
        const streamProtection = new StreamProtection(new InMemoryStreamProtectionStorage());
        assert.isTrue(streamProtection.canPlay(user, streamId));
    });

    it('does not allow a user to get a stream if he is streaming his threshold', () => {
        const streamProtection = new StreamProtection(new InMemoryStreamProtectionStorage());
        streamProtection.driver.registerStream(user.id, 'somestreamId');
        streamProtection.driver.registerStream(user.id, 'somestreamId1');
        streamProtection.driver.registerStream(user.id, 'somestreamId2');
        assert.isFalse(streamProtection.canPlay(user, streamId));
    });

    it('does allow a user to get a stream if he stopped streaming one after he reached his threshold', () => {
        const streamProtection = new StreamProtection(new InMemoryStreamProtectionStorage());
        streamProtection.driver.registerStream(user.id, 'somestreamId');
        streamProtection.driver.registerStream(user.id, 'somestreamId1');
        streamProtection.driver.registerStream(user.id, 'somestreamId2');
        assert.isFalse(streamProtection.canPlay(user, streamId));
        streamProtection.driver.removeStream(user.id, 'somestreamId2');
        assert.isTrue(streamProtection.canPlay(user, streamId));
    });
```

With the idea in mind of having this feature tech agnostic I created 2 main classes:



StreamProtection is the actual service that encapsulate the logic to check whether a given user is allowed to stream a certain video.
I plugged this into the endpoint to get the Stream like so:



```
const getStream = async (req, res) => {
    const user = userRepository.get(req.user);
    const { streamId } = req.params;
    if (streamProtection.canPlay(user, streamId)) {
        const stream = catalogRepository.getStreamById(streamId);
        if (!stream) return notFound(res, `Stream not found '${streamId}'`);
        return response(res, stream);
    }

    return forbidden(res, `Your current plan (${user.plan}) allows you only to stream ${user.maxStreams} videos concurrently.`);
};
```

StreamProtectionStorageDriver this class instead has the methods used by the protection system itself, and it will allow it to check in whatever type of storage we want to create driver about, if a user is streaming and how many other videos.

As I specified on the file **InMemoryStreamProtectionStorage** for the purpose of this test I decided to save those info in Memory, even though in a production situation I would have probably used **Redis** for an application of this type, as having this in the RAM of the single instance of the application wouldn't make this scalable, since the instance your client would be redirected too, if we have this app behind for example an ELB, would be always different and so this info will not be a single source of truth.

Frontend Integration

This implementation of the feature unfortunately is not completely transparent to the FE, which is the only one that can tell us whether the user is still streaming or not.
In the case scenario of him closing the App we would need to know that this stream is abandoned and so to release one of his slots.
For the purpose of the test I implemented this with a simple endpoint that will handle the events of the user leaving the stream page.

Problems and Solution

Of course it can always happen that the endpoint is not always called for whatever reason (browser crashing, computer powering off without calling the release endpoint), so the user, with this implementation, would have an occupied slot for no reason.
A possible solution for this problem would be to give the "slots" on the Storage a timeout, maybe twice or x1.5 the duration of the live event itself, and in case the user dropped with a cron job those expired locked slots would be cleaned up afterward.

Testing instructions

The app contains one single user with those credential *vince*:*password*

To run it you need to startup both FE and BE. Both of them needs an *npm install.*

FE can be run from the frontend/ folder, by just running *npm start.* This will launch the website on localhost:3000 on your default browser.

BE can be run either locally with *npm run dev* or *npm start* or on a docker instance:
docker-compose up -d
In both cases the endpoint would be running at localhost:3001

To check whether the BE api are working you can simply GET /ping to check the API status.