Case Study: Designing A Finance Application

Martin Joo

Stock Pickr was my first $1B SaaS idea. The only problem was that it never made a single $ of revenue. As a good "developer who wants to be a founder" I put 99.9% of my energy into coding, building, and architecture.

However, the gains of this app are much bigger than $1B. I truly believe that this project helped me become a software engineer from a software developer.

Before we begin there are two notes on the source code:

- In this project, I used the term *Container* instead of *DataTransferObject*. So if you see a Container it's the same as a DTO explained in the book.
- I use service and repository classes instead of actions and/or query builders. I wasn't using actions at the time of this project, so that's the only reason.

The application I want to show you in this case study is similar to Seeking Alpha or Morningstar. You can browse publicly traded companies and check their:

- Income statement
- Balance sheet
- Cash flow
- And dozens of other financial metrics calculated from these financial statements

It seems pretty straightforward, however, we can quickly create 8-10 services from these features.

First, let's discuss the basic workflow of the application. There are tens of thousands of publicly traded companies and their financial statements are public so we obviously don't want to create these records by hand. We need a 3rd party service that provides us with this data. Luckily there are a lot of them. But as I said there are tens of thousands of companies, so we probably don't want to create these records at once. Just imagine if a company has 150 financial data points and we calculate 50 other metrics, and also fetch 20 or so market-related real-time data (such as stock price) and prepare some other numbers. The number of SQL queries will quickly reach tens of millions. So we probably need a way to schedule the creation of companies. For example, the app will import 1000 of them each day or something like that.

The next problem is that these statements are not static. Every quarter these companies issue a new income statement, balance sheet, and cash flow. So we need to repeat the steps above all the time. However, in the app, we only process and show yearly data but the process remains the same since every company has a

different fiscal year. For example, 2022 March 31th can be the end of 2021 for a given company and they issue their statements for 2021.

So these processes will be automated and the app is read-only. So users come to the site to check out the financial information of Apple and Microsoft. Just like Seeking Alpha, Morningstar, or Hypercharts.

StockPickr has another important feature. It can score companies based on their financial performance. There are some important metrics such as:

- Average yearly revenue growth %
- Current net income margin %
- And so on

The app will calculate the average of these metrics for a given sector. For example, what's the average yearly revenue growth for software companies? Let's say it's 20%. If company XYZ has 23% it will get a score of 4. If company ABC has only 12% it'll get a 1. So the scale goes from 1 to 4 where 4 is the best and 1 is the worst.

Now that we understand the very basics of StockPickr let's take a closer look at the data model.

Data Model

Company data

These are the fundamental financial data of every company.

Company

- Ticker
- Sector
- Industry
- Description
- Number of employees
- CEO

Income Statements

- Company ID
- Year
- Total revenue
- Cost of revenue

- Net income
- ...And other income statement items

Balance Sheet
- Company ID
- Year
- Cash
- Current cash
- Total assets
- Long term debt
- ...And other balance sheet items

Cash Flow
- Company ID
- Year
- Operating cash flow
- Cash from investing
- Free cash flow
- ...And other cash flow items

All of this information comes from a 3rd party provider. This is the bread and butter of the application. From these data, it can calculate a bunch of other metrics.

These are also some market-related information about a particular company.

Company share
- Ticker
- Stock price
- Market capitalization
- Shares outstanding
- Beta

Analyst-related data
- Number of buy recommendations
- Number of hold recommendations
- Number of sell recommendations
- Price targets (low, high, average)

This data also comes from a 3rd party provider.

Metrics

I'm not gonna go through all the metrics since there are many.

Metrics

- Company ID
- Year
- Gross margin
- Net margin
- Revenue growth
- Return on assets
- Cash to debt ratio
- Debt to capital ratio
- ...And other metrics calculated from financial statements

This table looks like this:

company_id	year	gross_margin	net_margin	revenue_growth
1	2022	0.60	0.23	0.18
1	2021	0.58	0.21	0.17

The app scores every possible metric on a 1..4 scale. However, it scores every metric in two ways:

- Compared to every other company's metric
- Compared to one sector

So if Apple has revenue growth of 15% it gets two points:

- A 4 compared to every other company's average
- And a 2 compared to IT companies' average

So we need two tables:

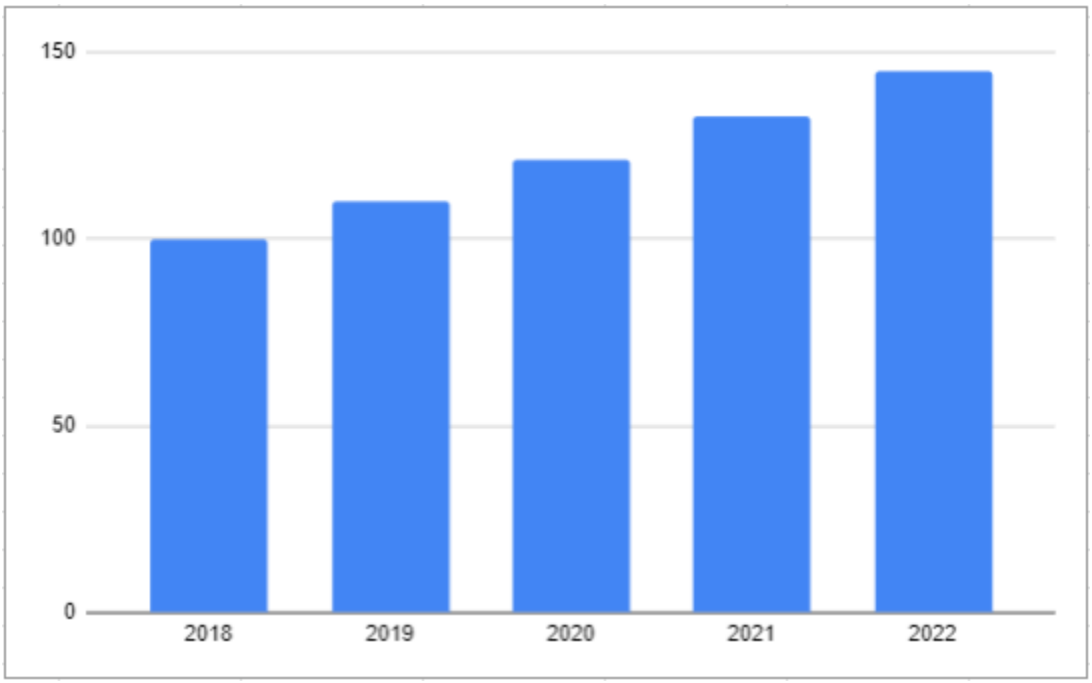
- company_scores
- company_sector_scores

These two tables will have a column for every metric. They look like this:

company_id	year	gross_margin	net_margin	revenue_growth
1	2022	4	4	3
1	2021	4	3	2

Charts

Users love fancy charts and reports. They just do. Especially in a financial app. You can imagine some pretty basic chart such as revenue that looks like this:



Let's think about the size of the tables for a minute. Let's say we store 50 000 companies. Each has 5 years of financial data and metrics. This means the income_statements table will have around 250 000 rows. What happens if we need to handle quarterly data? It'll have 1 000 000 rows. The same goes for balance_sheets, cash_flows, metrics, and the scores table.

To show a single chart we need to query in millions of rows. To make things worse the app also has charts that show information from multiple tables. For example, revenue (income_statements) and free cash flow (cashg_flow) are on one chart. This means we have to join tables with millions of rows.

In order to avoid slow queries and possible N+1 problems, we can store the chart data in a separate table. For example, the above chart can be stored such as:

company_id	chart	data	years
1	revenue	[100, 110, 121, 133, 145]	[2018, 2019, 2020, 2021, 2022]

When the chart needs to show multiple numbers for the same year, such as the free cash flow to revenue, we can use a data structure like this:

company_id	chart	data	years

1	cash_flow_to_revenue	{
		[100, 110, 121, 133, 145],
		[10, 11, 12, 13, 14]
		}
			[2018, 2019, 2020, 2021, 2022]

We can insert these charts when creating or updating a company. This way the query will be super fast. We don't need to search data in millions of rows, no data processing, no data mapping, no for loops, no N+1 problems. Just a simple select query and that's it. It's also super easy to cache these charts.

I'll talk more about charts later because it's pretty interesting.

Scheduler-related tables

Earlier I told you that a scheduled process will create and update companies. This process also requires some tables. At least we need to keep track of what companies we have tried to update and when. If we know that information we also know that we don't have to try for at least a few weeks. We can update the other 49 999 companies. But of course, these tables are not essential to the application domain.

This is the bare minimum data model we need in order to implement stock-pickr.

Services

Now that you have a basic understanding of the data model we can talk about services. What services does this app need?

The first step should be to collect the main features and try to group them. These groups would give us a good idea about the services.

The user-facing parts of the application:

- **Login and register.** Users need to be logged in to access the app.
- **Company list**. This is the index page where users can see every company sorted by their total score. They can also search for specific companies and sectors.
- **Company profile.** This page shows the basic profile of a company with the ticker symbol, market capitalization, current price, analyst recommendation, etc.
- **Financial statements**. The last five years of income statements, balance sheets, and cash flows.
- **Charts.**
- **Metrics.** All the calculated metrics are shown on a dedicated page for a given company.
- **Scores**. This is a different page where users can see the company's scores vs the average and the sector.

And some "stuff" users cannot see:

- **Company provider**. As I said earlier we need to integrate with a 3rd party data provider. Generally, it's a good idea to develop this feature in a way that provides us with an easy way to switch between providers. This means we don't want to hardcore one particular provider's API endpoints, method names, etc. We also want to avoid copying their data structure into our own database. For these reasons, the integration needs some abstractions and extra code.
- **Scheduler**. This is the component that takes care of creating new companies and updating existing ones.
- **Gateway**. As I wrote in the book you can use two kinds of gateways: a plain nginx reverse proxy, or a custom service that will proxy requests and implement some extra behavior. In this app, I went with the second option because I need to communicate with the auth service and I also use cache.
- **Backup**. There are tons of solutions to back up your databases and user-uploaded files. In this application, I decided to write some Nodejs code

to the job and sync everything to S3. I just love hacking around with node or python.

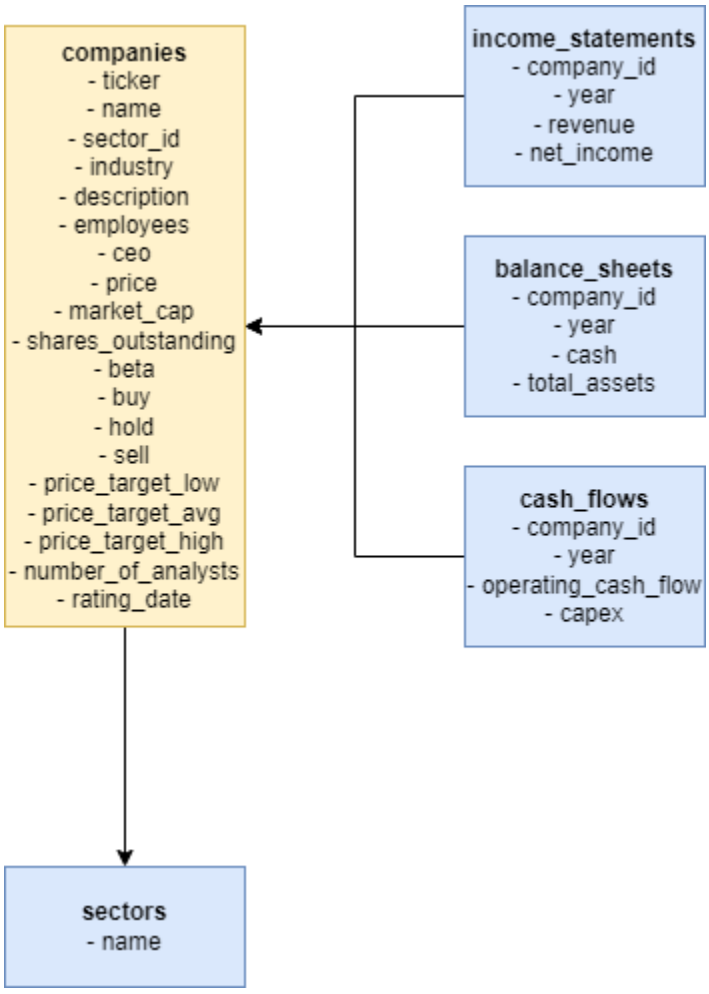
The second half of this list is easy. We need a dedicated service for every item:

- **Company provider** will be a wrapper around Finnhub or other finance APIs.
- **Scheduler** is a service that will update every company on a scheduled basis, and also create new ones.
- **Gateway** is the only service the frontend knows about.
- **Backup** is a service that helps us sleep peacefully.

Company service

Now let's think about the first half of the list. I think it's clear that the app needs a **company** service. This is the bread and butter of StockPickr. But in my opinion, it's pretty easy to fall into the trap of thinking "this feature is related to companies so I put it into the company service." If you take a look at the list, you can see that **everything is related to companies.** So we need a better way to draw the boundaries.

Let's model the "core" part of the database. This is what I think is "the company service" right now:

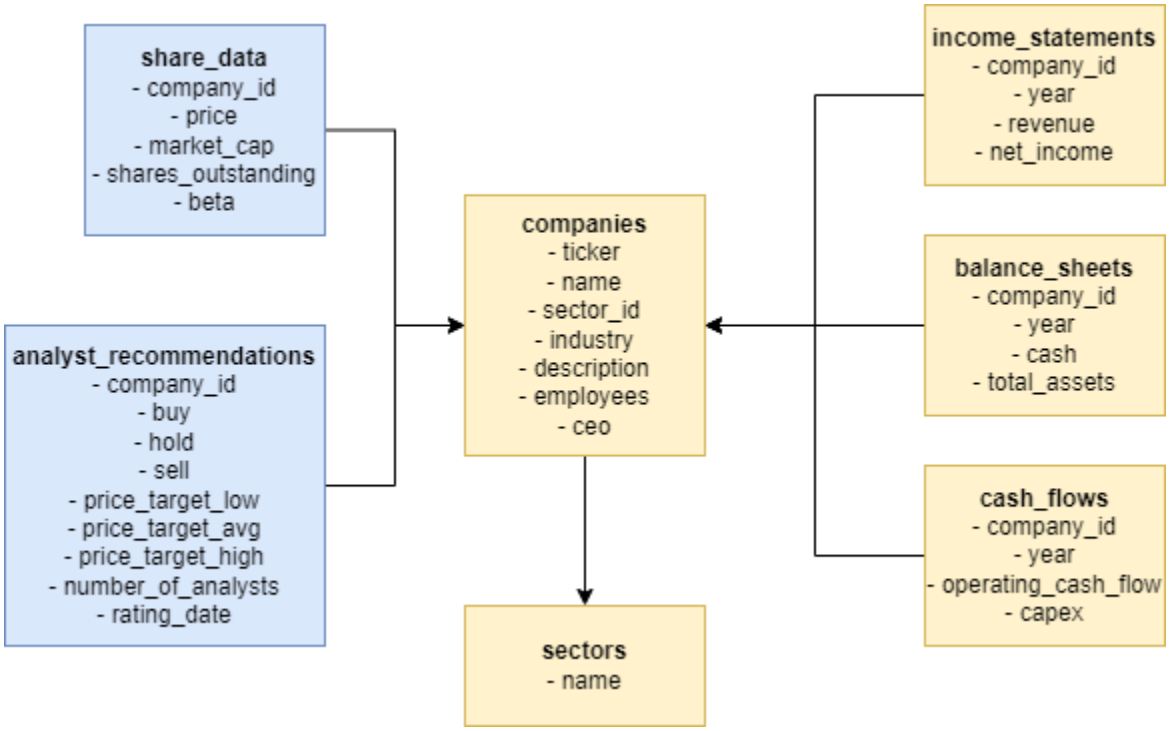


First impressions? The companies table is big. And it'll only get bigger as time goes by. What kinds of columns does it have?

- Fundamental company data
- Price and share-related data

- Analyst recommendations. Buy, for example, means the number of analysts who think the company is a good buy.

These are different from each other and probably require different business logic and features. So let's separate them:



Now that we normalized the DB a little bit, what does it have to do with microservices at all?

First of all, in my opinion, the tables highlighted in yellow are the company service. The core of this application is companies and their financial statements. Company service is the most fundamental one, and in this service, the term "company" means a company and its statements. So my first thought is to build this service as small as possible.

All right, but why not just store the blue tables in the company service as well? Why build a separate service? Well, there's no straight answer to that. You can definitely put these tables and APIs in the company service. However, let's think about possible feature requests regarding share data and analyst recommendations:

- Right now, we only store the current price. What if we need to store historic prices? What if we need features that involve more complex calculations with these historic prices? For example, comparing 52-week moving averages and deciding if a company is a good deal or not.

- Other valuation methods based on market data can also be interesting features.
- What about real-time prices and notifications to users?
- I can see tons of features based on only prices.
- What if we need more analysts from other sources? What if we need to collect, let's say articles or videos about these recommendations? I can see some integration with financial news services as well.

Right now, these two tables look pretty innocent, but after six months you can easily have a company service that:

- Stores and serves companies and financial statements via API
- Integrates with CNBC
- Evaluate companies based on past prices and other market factors
- Has a websocket server to provide real-time prices

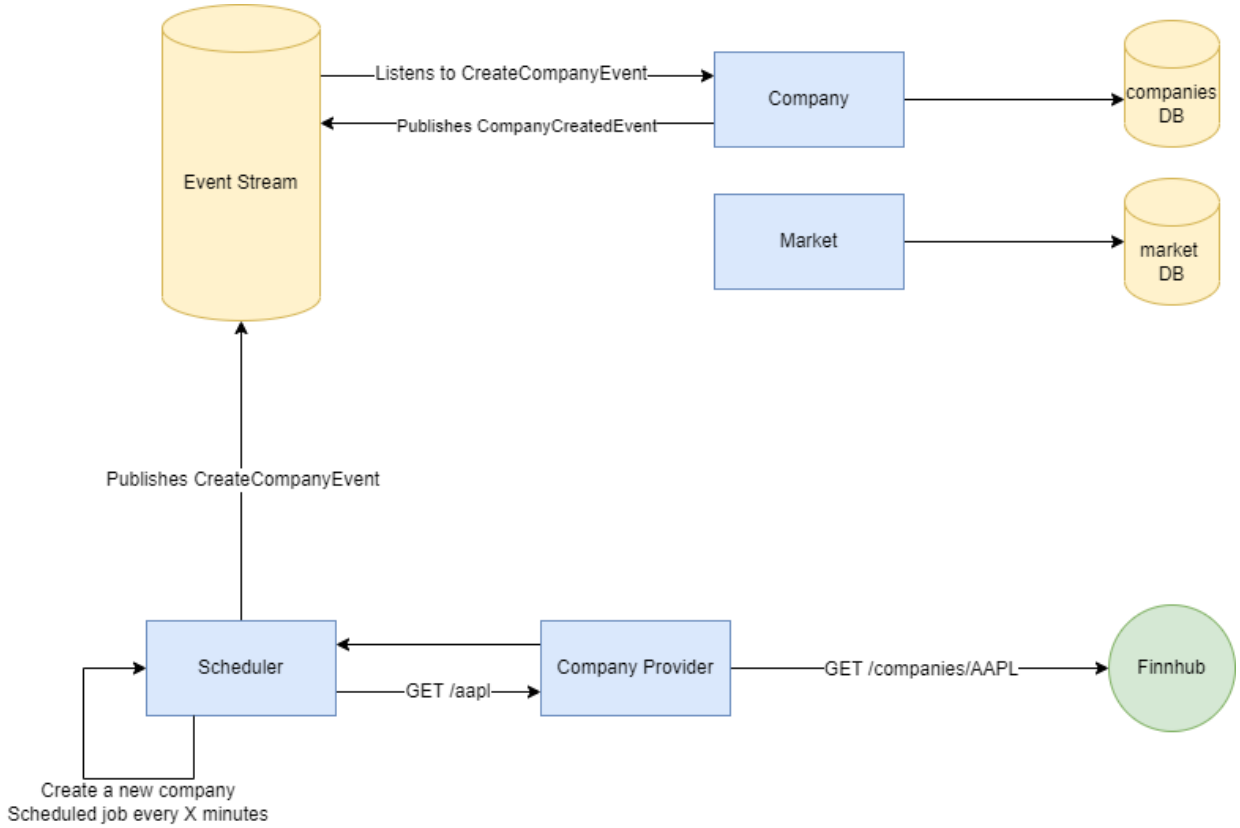
But the real reason I decided to have a dedicated market service is this: **fundamental company data and market-related information are two different things.** Easy as that.

It's important to note that this process has no right or wrong answers. It's all subjective. Only time will tell what's the better option. Either you think it should be one service, or you're with me, you take some risks:

- If you decide to build one company service that is also responsible for market-related features, you take the risk of having a bigger, more complex service.
- If you decide to build two separate services, you take the risk of having more complexity in your overall architecture. One more service, one more repo, one more pipeline, one more docker image, one more running container, etc. If the feature set doesn't grow over time, it's overkill.

Company Provider and Scheduler

Earlier we said that we need some kind of scheduler and also a company provider that integrates with Finnhub. It might be confusing that the app has a company service but also a company provider so let's pause for a minute and think about the "flow" of the application:



Here's what's happening step-by-step:

- The Scheduler runs commands on a scheduled basis. *CreateCompanyCommand* is one of them. There's also a *UpdateCompanyCommand.*
- The Scheduler sends an HTTP request to the Company Provider with a ticker symbol.
- The Company Provider calls Finnhub and gets the company data. This includes basic information and financial statements.
- When the Scheduler gets a response from the Company Provider it publishes an event to the Redis events stream. This is the *CreateCompanyEvent.* Technically it's an event, but we can call it a command as the name suggests.

- The Company service listens to this event and inserts the company and its financial statements into its database.
- After it's done it publishes a *CompanyCreatedEvent*. This is the event other services will listen to. We'll talk about it later.

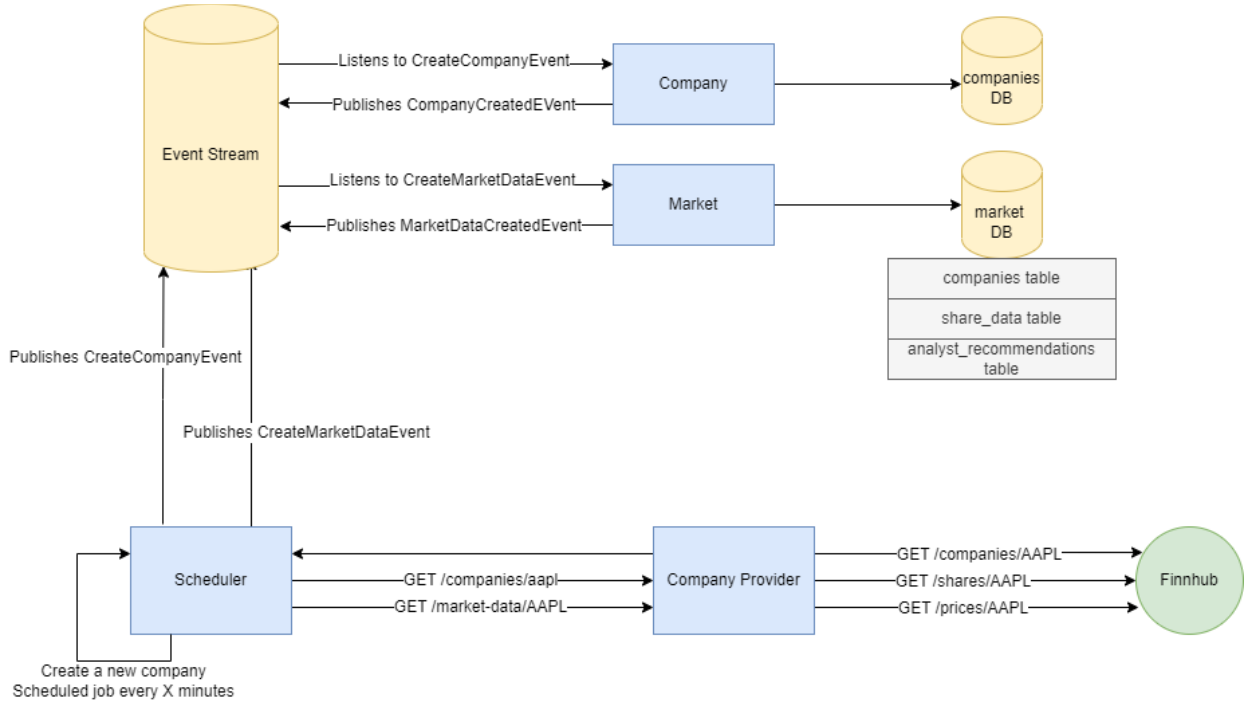
So what's the difference between the Company Provider and the Company service?

- Company Provider is a generic API that provides an abstraction over Finnhub and other financial providers.
- Company service receives this data processes it and stores it in a way that fits our application.

Can they be one service? Yeah, of course, we can merge the provider into the company service. However, it has some disadvantages:

- As you already guessed, the company service is the most crucial service in the app. It's always a good idea to build these crucial services in an independent way. So what happens if Finnhub is down? It may take down your entire company service. What happens when they change their APIs and you're not aware of these changes yet? It may take down your entire company service. What happens if the company service is down? Your FE will not be able to show companies to the users. I know these events are unlikely, but they can happen, and it's a valid risk.
- Finnhub is a big service. It has a lot of different APIs. For example, market news, prices, trading history of the company owners, etc. If we use it from the company service it's easier to mix responsibilities over time.
- Over time we might need to use more financial APIs at the same time. And once again, it gets more and more complex.

What about the Market service? How does it build its own database? I wasn't completely honest, because there are two events involved in this process:



There are some additional steps here:

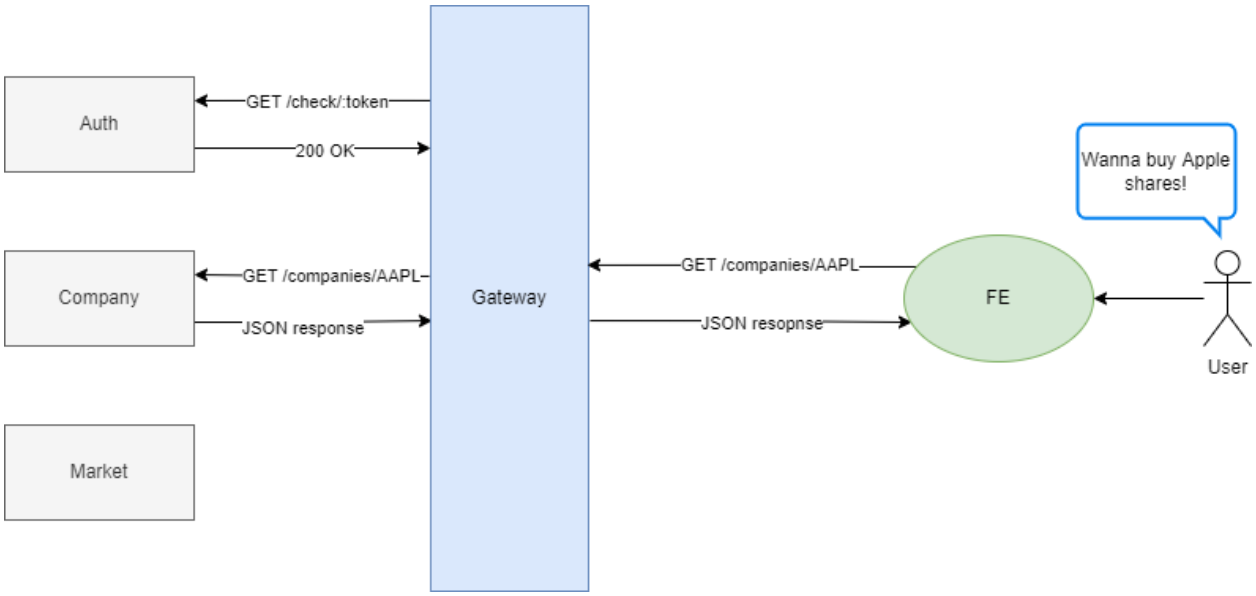
- The Scheduler also sends a request to get the market data of the company
- The Company Provider will collect every data using multiple APIs of Finnhub
- The Scheduler publishes an additional *CreateMarketDataEvent*
- And finally the Market service listens to this event. The *CreateMarketDataEvent* contains everything that is inserted into the *share_data* and *analyst_recommendations* tables.
- After it's done, the Market service publishes a *MarketDataCreated* event. Other services can consume this event.

The important thing here is that the Scheduler publishes "commands." It tells the Company service "Hey, here's some new data, create a company." And it does the same with the Market service. But other services don't listen to these commands. If they need a company, they will listen to the *CompanyCreatedEvent* published by the Company service. These commands are commonly used in systems where scheduling is important.

Gateway and Frontend

Before moving on to discuss other services we can take a pause and introduce a gateway and the frontend. What happens if a user wants to check out the income statement of a company? The frontend will send a request to the company service. What if they want to see what analysts are saying about the company? The frontend sends a request to the market service. Both of these requests require a logged-in user or in other words, a valid token in the request headers.

This is they fit into the picture:



Step-by-step:

- A user clicks on the company profile
- The Frontend sends a request to the Gateway. And this is important (and discussed in the book). The Frontend only knows about the Gateway.
- The Gateway first sends a request to the Auth service. This is a simple GET request and its headers contain the user's token. The Auth service checks its database and if it's a valid token it sends a 200 OK response. Otherwise, the response will be 401 Unauthorized and the gateway sends it back to the frontend. So unauthorized requests didn't make it through the Gateway.
- If the token is valid, the Gateway redirects the original request to the Company service.
- The Company service then serves the request and sends its response to the Gateway.
- As the last step, the Gateway returns the Company service's response to the Frontend and we have a happy user.

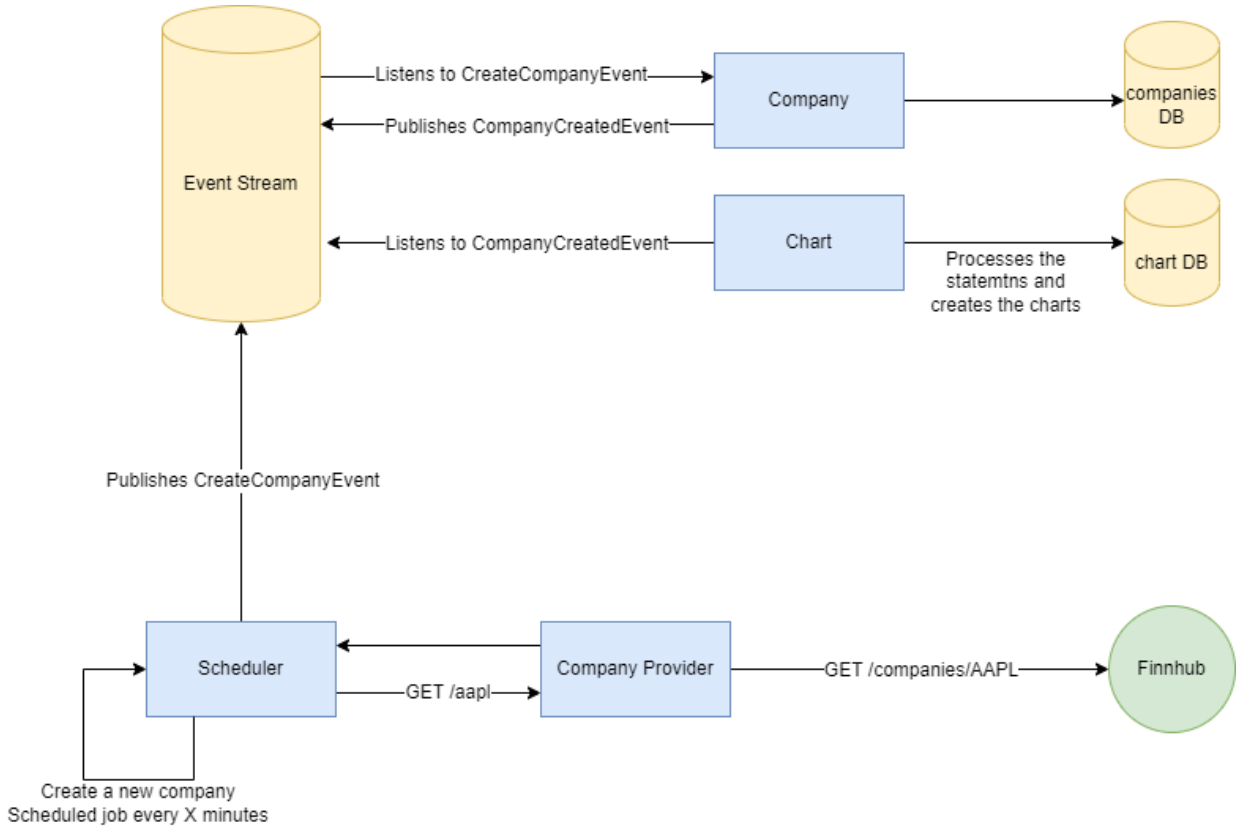
If the user wants to see the analyst recommendations the whole process is the same, but the Gateway will interact with the Market service instead of the Company.

This flow might give you a better picture of the difference between the Company and the Company Provider services. The first one is a user-facing API and the second one is a “back-office” service that is only used internally.

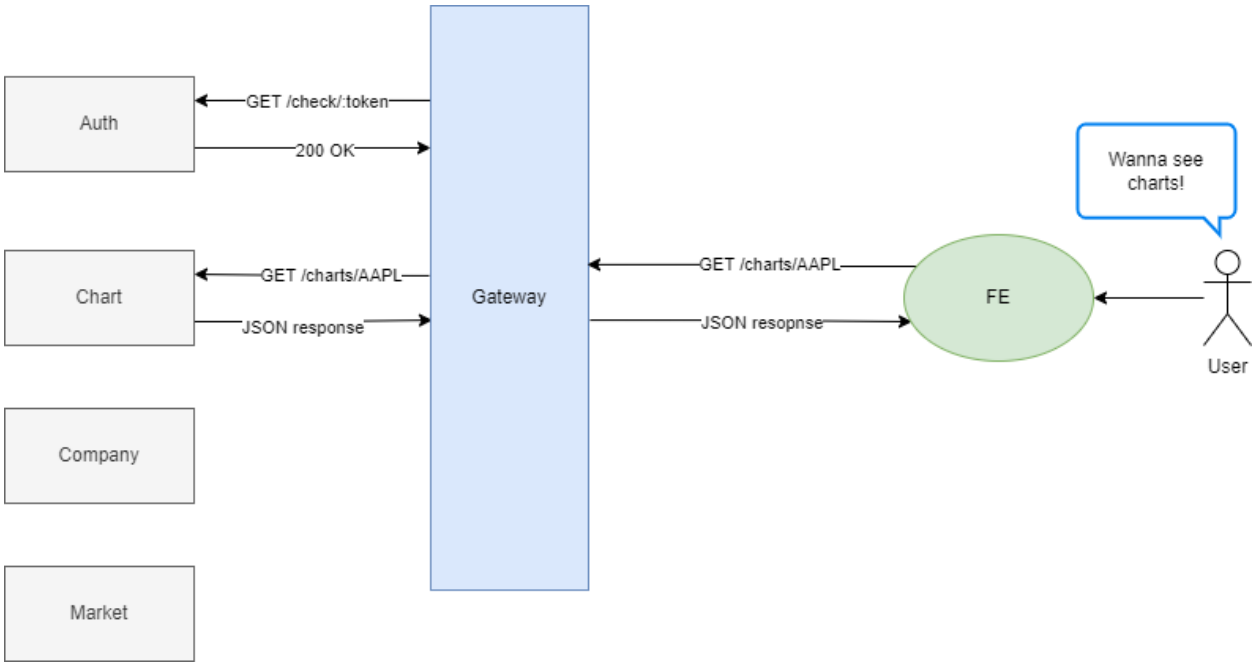
Chart service

Now users can check out income statements and analyst recommendations on the site. Great! But they want fancy charts.

If you think about it, the Chart service is the most straightforward one so far. Previously we discussed that the Company service will publish a *CompanyCreatedEvent* after it creates a new company. This event will contain financial statements. As you might already have guessed, the Chart service will listen to this event, collect the important data, and saves it in a “chart-friendly” way, and that’s it.



And it also serves user requests via an API:



I think you can see a pattern emerging here:

- Whenever a service needs to communicate with another service it listens to events.
- If a user needs to interact with a service it exposes an API.

This is almost always the case in event-driven systems. However, there are some exceptions. For example, the Company Provider service. It doesn't publish or listen to events. It exposes an API and the Scheduler uses it.

Metrics service

We know that the app needs to calculate tons of financial metrics such as revenue growth, gross margin, etc. These metrics are calculated from financial statement items, for example, gross margin is gross profit divided by revenue. Both of these values are on the income statements. Since these values can be found in the Company service it seems logical that we implement metrics in this service. However, I think we can benefit from a dedicated Metrics service for these reasons:

- This service will be calculation heavy. We have something 25 financial metrics. Most of them have some edge cases or require some special calculation. For example, negative values in growth calculations, zero values in ratios and so many other edge cases. Working with numbers is hard.
- What about scores? Since scores are calculated from these metrics it seems logical to have them in the same service. I think it's a good idea. Metrics and scores and closely related to each other.
- Even more logic and calculations. If we decide to implement scoring in the metrics service it requires even more code.

For these reasons, I think metrics should be a dedicated service. The event flow is the same here as it was earlier. Whenever the Company service creates a new company it publishes a *CompanyCreatedEvent,* which, the Metrics service listens to and it calculates the metrics.

Now let's think about scoring companies. Here's the basic logic:

- The Metrics service keeps track of all the median values. For example, if the are 3 companies in the service with 10%, 12% 16% revenue growth, then 12% is the median revenue growth across all companies.
- Whenever a new company comes in, we need to recalculate these median values.
- After that, we need to re-score every company in the database. A company's score is dependent on every other company in the database.

Here's a quick example. Let's assume we have only one company in the database:

ticker	**revenue_growth**	**net_margin**
AAPL	12%	20%

The median values are the same:

- Revenue growth median: 12%
- Net margin median: 20%

So Apple is a perfectly average company and it gets these scores:

ticker	revenue_growth	net_margin
AAPL	3	3

Remember scores go from 1 to 4 so 3 means ‘average’. What happens when a new company comes in?

ticker	revenue_growth	net_margin
AAPL	12%	20%
MSFT	18%	25%

Now we need to re-calculate the medians:

- Revenue growth median: 15%
- Net margin median: 22.5%

Since the median values have changed, all the scores are out-dated. We need to re-calculate them:

ticker	revenue_growth	net_margin
AAPL	2	3
MSFT	4	4

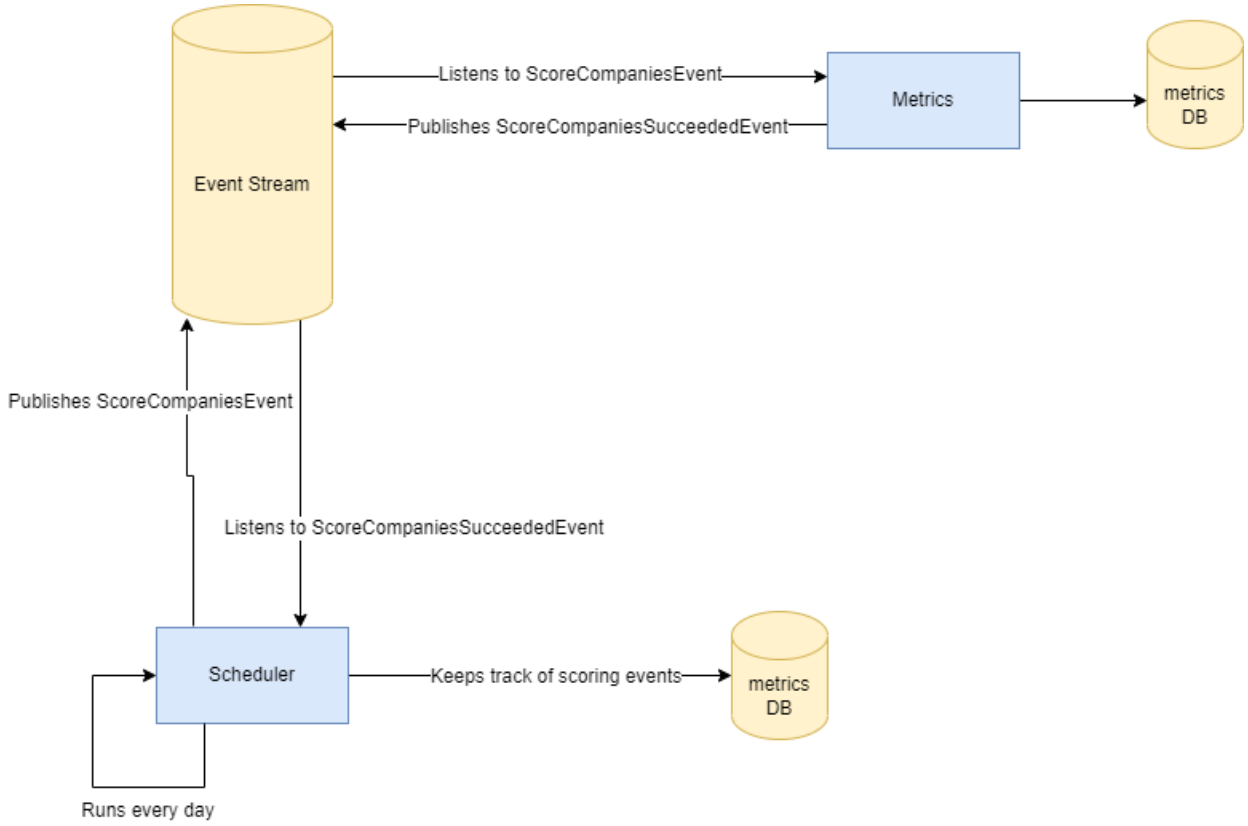
Since Microsft has better than average metrics it gets better than average scores. The same process happens when a company gets updated. I didn’t mention it, but the scheduler will schedule *UpdateCompany* commands on a regular basis. When a company gets updated it means they have new financial statements so we need to re-calculate all the metrics and scores. And this happens quite frequently.

We have two choices here:

- Every time a new company is created, or an existing one is updated we re-calculate tens of thousands of companies.
- Or we just schedule it, for example, once a day.

I vote for the second option. The first one is pretty resource-demanding. It needs to recalculate everything for tens of thousands of companies multiple times a day. I’d say it’s a little overkill.

The implementation of the second option is pretty simple. We just need to publish a command from the scheduler:



Step-by-step:

- The Scheduler publishes a *ScoreCompaniesEvent* once a day. By the way, it's a totally arbitrary number.
- The Metrics service listens to this and scores every company.
- If everything went well, the Metrics service publishes a *ScoreCompaniesSucceededEvent*
- The Scheduler service listens to that event and keeps track of it in its database. This step is totally optional but it's pretty useful when debugging. And by the way, the scheduler keeps track of everything. We'll look at it later.

Why doesn't the Metrics service start this process as a scheduled command? Why do we need to involve the Scheduler? It's perfectly fine to implement this process in the Metrics service without the need for an event stream or the Scheduler. However, since we already have a dedicated Scheduler service I think it's a good idea if it handles every scheduled command. This way the whole thing is centralized and easy to understand or debug.

Before moving on, I want to note that this scheduled approach has a flaw compared to the non-scheduled alternative. Now the scores are not up-to-date. They are delayed because of the *ScoreCompaniesEvent* and its hourly or daily

schedule. In this application, it's perfectly acceptable, but you need to consider this delay effect every time you want to schedule something like this.

Company Query service

If you take a look at the source code, you'll find a service called Company Query. I built this application a few years ago and looking back, I think the query service was a poor design decision.

First, let's discuss what does it do?

- When a new company is created the Company Query service creates its own version of the company.
- At this point, it only has basic company information. Since the scoring happens on a scheduled basis, the new company doesn't have scores just yet.
- When a company gets scores the query service also listens to these events and saves every score-related property. There are six of them:
    - *total_scores*. It's a single number that represents the sum of scores this company has.
    - *total_score_percent*. If three are 25 metrics and the maximum score is 4, and this company has 82 points this value is 82%
    - *total_sector_score* and *total_sector_score_percent*. Same as the previous ones, but for sector scores only.
    - *position*. Represents the current ranking of the company. For example, there are 15000 companies and Company A is number 1 based on its scores.
    - *position_percentile*. It tells you that a company is in the top 10% percent of every company based on its scores.

So this service stores every score-related property. It also exposes three APIs:

- *GET /leaderboard*. This is the index page of the site. It returns the 50 top companies based on their positions.
- *GET /search*. You can search companies by their names or ticker symbols. It returns only those that already have a *position*.
- *GET /count*. Simply returns the count of companies but only those that already have a *position*.

All of these APIs care about scores and position. And this is important. This is the number one reason they are in a separate service. If it was a simple search API it would be in the Company service. But remember, the Company service doesn't care about scores or positions.

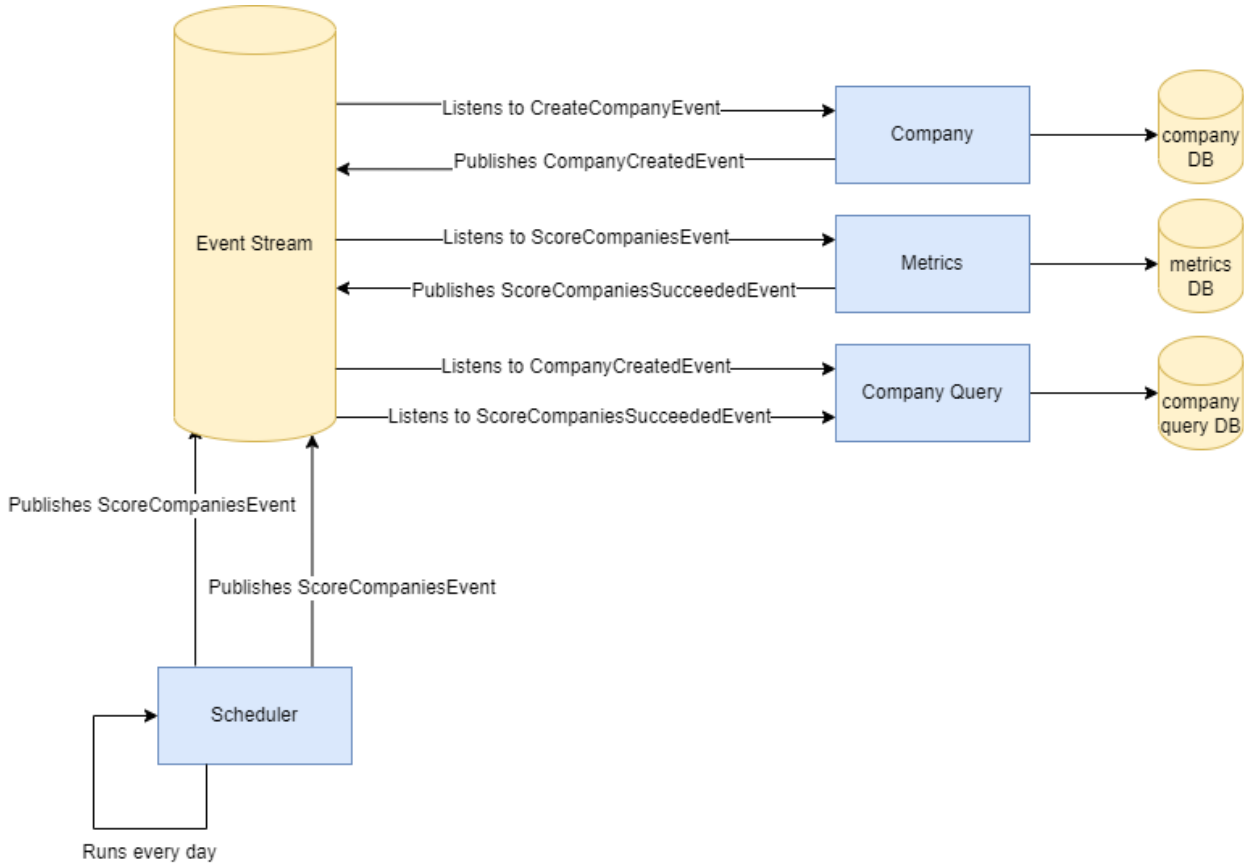
So the question is: do we want to build a separate service that stores companies with score information and exposes some APIs? Or do we want to introduce a few extra columns in the Company service and add these endpoints to its API?

Having a separate service introduces more complexity and cognitive load. And it's kind of weird, to be honest. Just imagine a conversation with a new developer.

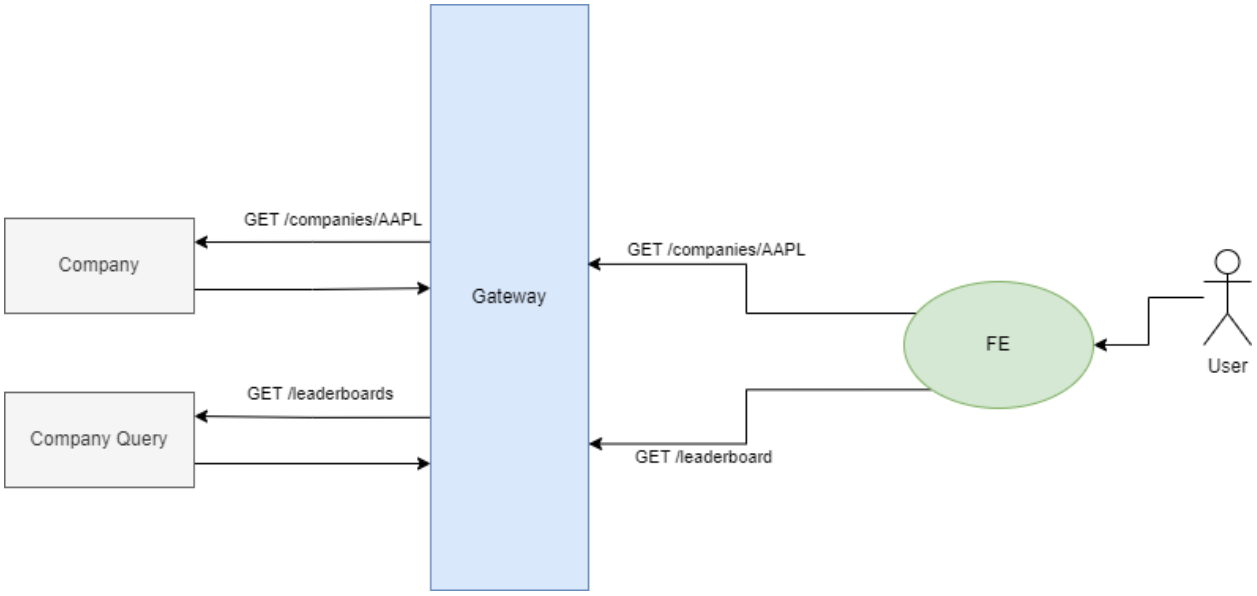
Me: "So there's the Company service where you can query all the companies. But there's also a company query service which can run score-related queries."
New developer: "All right. So I need a list of companies. What do I do?"
Me: "It depends. Do you need every company regardless of the scores and positions?"
New developer: "Yep"
Me: "Then you need to use the Company service"
New developer: "What if I also need scores?"
Me: "Then you need to use the Company Query service"
New developer: "Wait... Why?"
Me: "Because we are building microservices"

So if my best response is "because we're building microservices" it means I overdid it and it makes no sense. It happens whenever we're trying out new technologies new architectures or just new stuff in general. We are overdoing it. It's a weird thing because looking back I think it's simply a terrible solution, why didn't I see it when I was building it? And the answer is simply: I was in love with my small, little services. How to avoid these kinds of mistakes? **You won't.** The best you can do is to ask for other people's opinions and maybe someone will point out your mistakes.

So let's see how it works now and how it should be working:

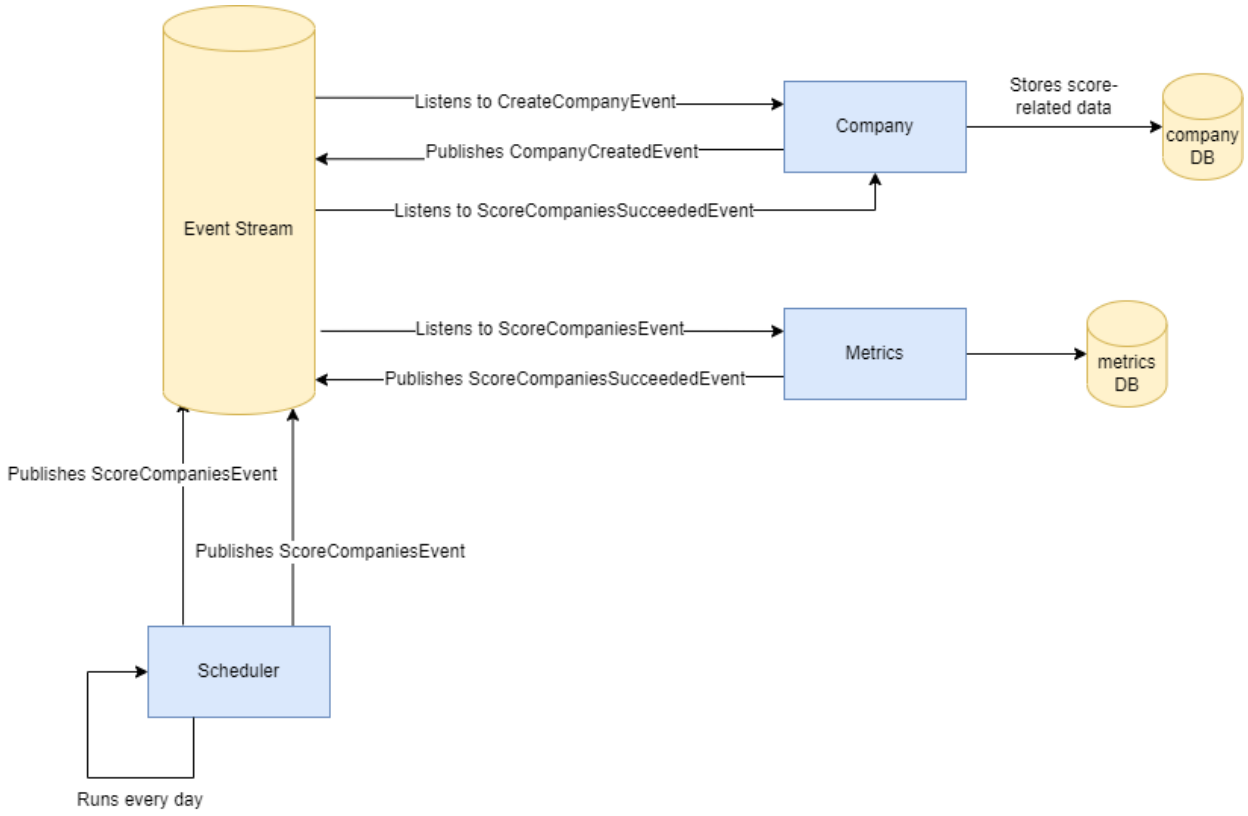


And this happens at the API level:



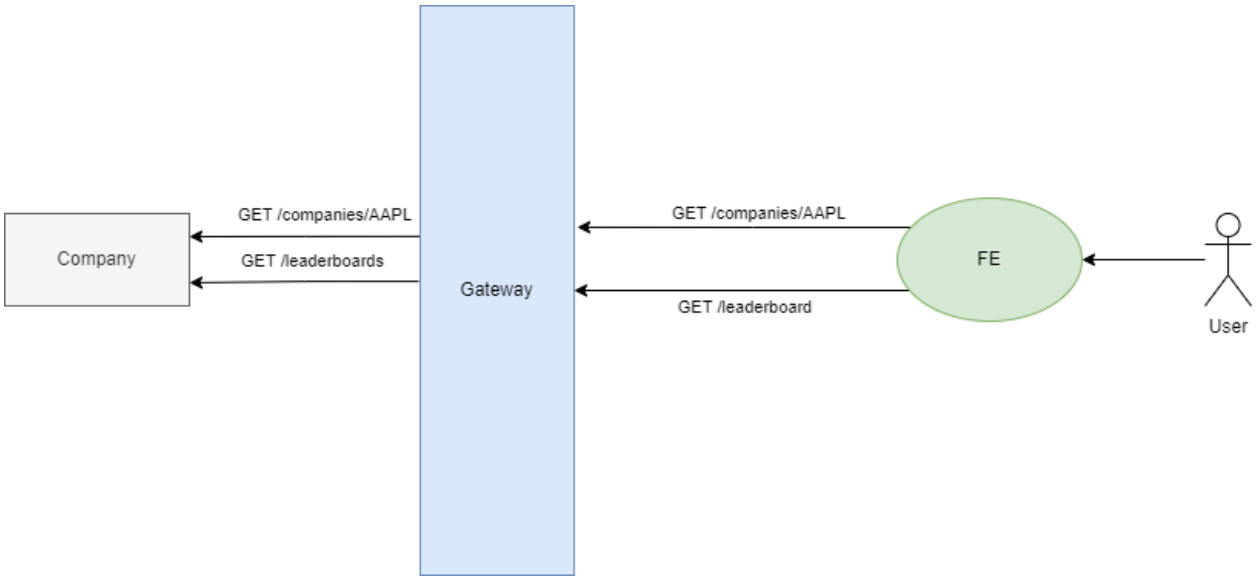
As we discussed earlier a simple *GET /copmany/AAPL* request gets served by the Company service, but a *GET /leaderboard* request is handled by the Company Query service. It's not a big problem from the FE point of view, since it only knows about the gateway, but it's a problem for backend developers, for sure.

This is how I'd do it today:



One less service, one less database. The Company service's database simply has an extra table that stores score-related data.

And the API would also be simpler:



In my opinion, this is one of the major flaws of this application. However, I didn't want to refactor it and represent the "good" solution for you. I think this way you can learn more from my mistakes. I wanted to add a shiny new service so much that I forgot about simplicity and developer experience.

Auth service

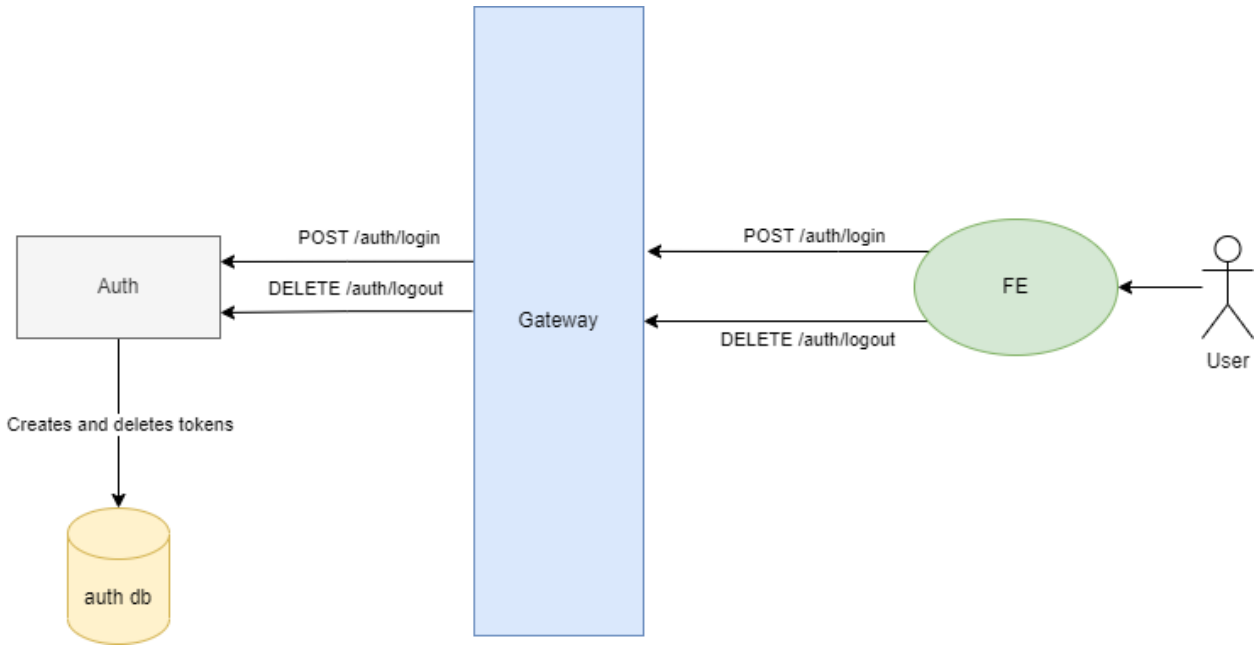
We already talked about how the Gateway calls the Auth service everytime a request comes in. Let's take a closer look at the Auth service.

This is the service that stores user information, so it has the usual APIs:

- *POST /users* to register new users.
- *PATCH /users/:id* to update existing users.
- *DELETE /users/:id* to delete users.
- *POST /auth/login* to login users with e-mail and password.
- *DELETE /auth/logout* to logout users.
- *GET /auth/check* to check a token if it's valid or not.

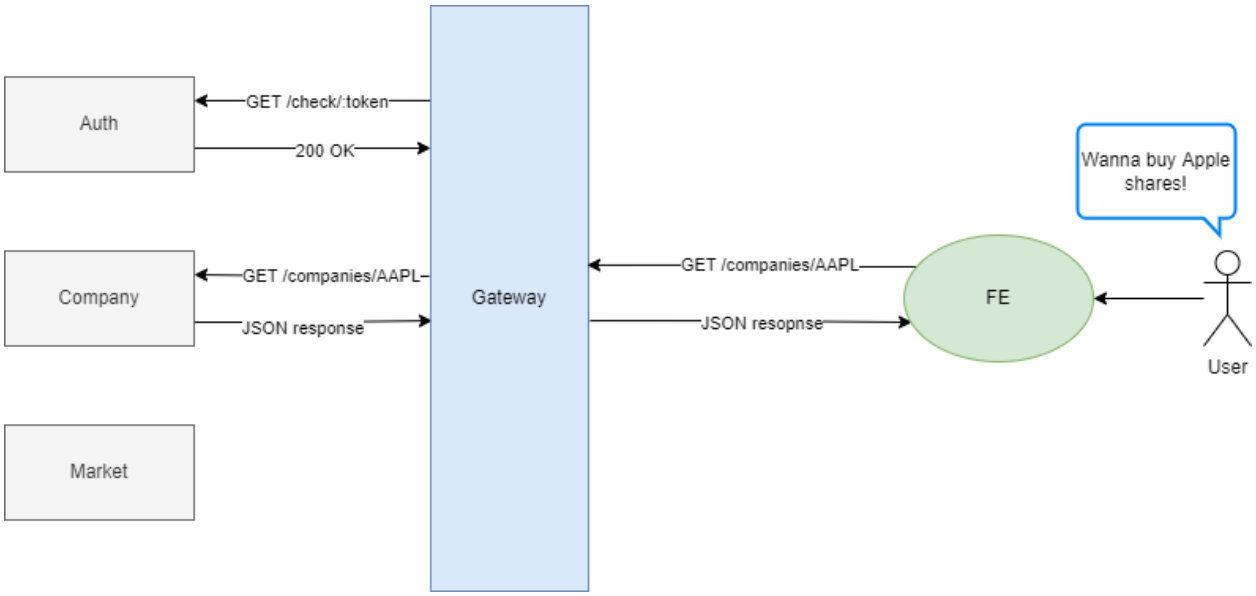
This service implements a pretty simple JWT token based authentication system. It's simple because users can only have one token, there are no refresh tokens or any other "advanced" feature.

The basic flow is pretty straightforward:



This is the same logic as if it was a monolith application. There's only one important difference: it's a Nodejs application. I simply choose Node and Express over Laravel because it's much faster. When I say much faster I'm referring to 5x or even better performance. Laraven Octane is certainly a competitor to Node's performance, so you can give it a try.

The other difference compared to a monolith is the *GET /check* API. We already talked about this, but here's a quick overview:



Step-by-step:

- Whan a request comes in the Gateway automatically sends it to the Auth service
- The Auth service has a *GET /check* API which accepts a token in the header. If this token is in the database and it represents a valid user it returns a 200 OK response.
- The Gateway receives the response and sends a request to some other service.
- Then it responds to the frontend.
- If the given token is invalid then the Auth service responds with a 401 status code and the Gateway immediately responds to the frontend. So in this case the original request never touches the Company service, for example.

Scheduler logic

Creating copmanies

We have talked about the scheduler quite a bit and you can see it's a crucial part of the entire application. For this reason, we can take a closer look at it.

There are four scheduled commands running in this service:

- *company:create* will create a new company.
- *company:update* will update an existing one.
- *company:update:market-data* will update the market data of an existing company.
- *company:score* will start the scoring process.

You've already seen the *company:create* but how does the Scheduler know which company to create? The Company Provider service has a *GET /available-tickers* API which will return every ticker symbol we can play with.

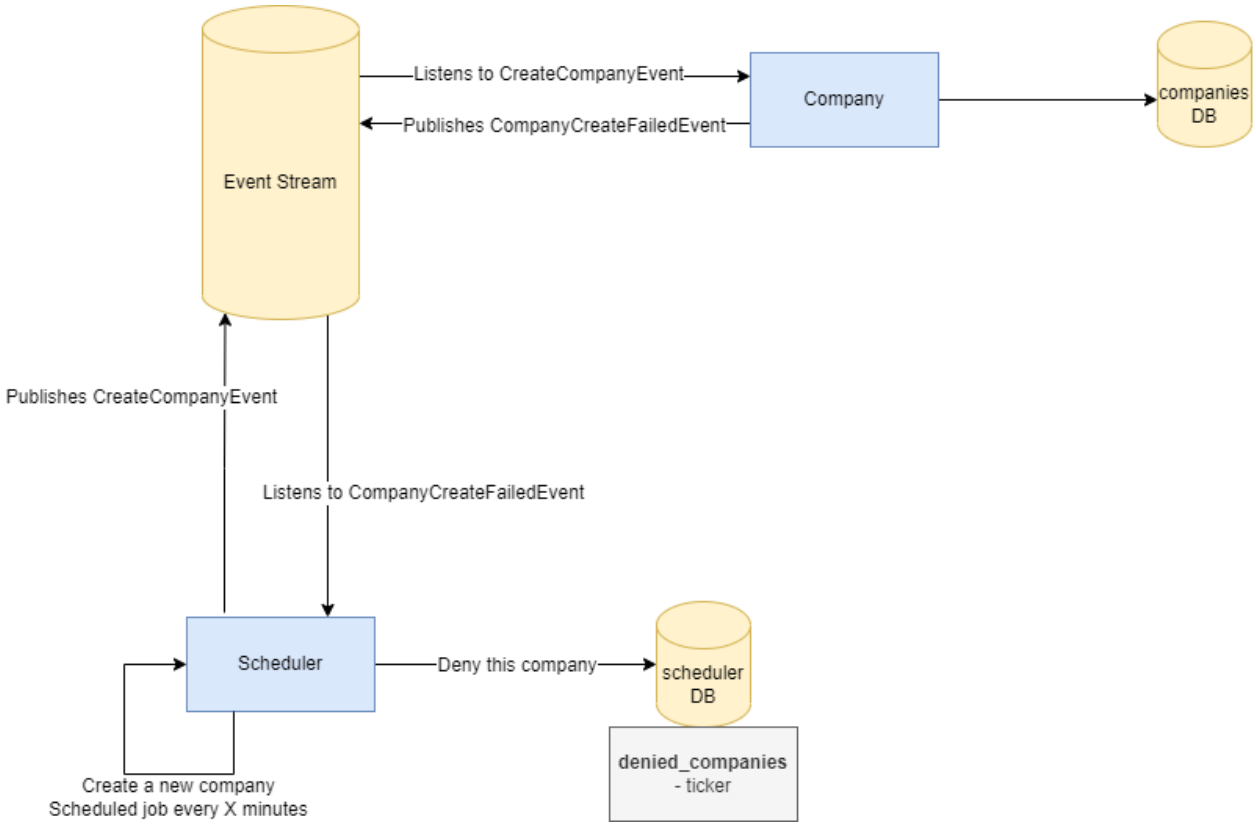
After the Scheduler has the available tickers it will reject the following tickers:

- Companies that are already in the application.
- Companies that cannot be created for some reason.

The first one is quite simple. The Scheduler knows which companies have already been created. The second one is a bit "trickier." Any time the Scheduler publishes a *CreateCompanyEvent* it'll listen to a responding event. It can be either:

- *CompanyCreatedEvent*
- *CompanyCreateFailedEvent*

Whenever the Company service publishes a *CompanyCreateFailedEvent* the Scheduler will save this ticker in the *denied_companies* table:



So the *denied_companies* table contains companies that cannot be created for some reason. This reason is mostly missing information in their financial statements. For example, Finnhub doesn't return revenue or net income numbers.

After the Scheduler rejects existing and denied companies, it just picks a random ticker from the list given by Finnhub and publishes a *CreateCompanyEvent*.

Before it publishes the event it creates a new record in the *company_schedules* table. This table contains a nice log of every schedule happening in the application:

event	ticker	started_at	finished_at	state
create-company	AAPL	2022-09-06 10:50:13		in_progress
update-company	MSFT	2022-09-06 10:47:31	2022-09-06 10:48:01	succeeded
create-company	ABC	2022-09-06 09:33:54	2022-09-06 09:33:56	failed

This table and the Redis event stream is extremely useful when it comes to debugging. It can also help us with maintenance:

- What happens if the first job gets stuck? It happens if the Scheduler never receives a *CompanyCreatedEvent*. In this case, the will be *in_progress* forever. With the help of this table, we can easily clean up these stuck jobs.
- We can also retry failed jobs. Maybe the last job failed because the Company service was down at the moment. We can write a simple command that retries these failed jobs.
- We have a clear history of the entire flow of the application.

Updating companies

Updating companies is pretty similar to creating them. The Scheduler will run a command on a scheduled basis and publishes a *UpdateCompanyEvent* which the Company service listens to. It'll update the company with the new financials and publishes a *CompanyUpdatedEvent*.

So how the Scheduler knows which companies to update? As I said, the app has tnes of thousands of companies, so it's quite important to have some kind of order. I chose the most simple approach: we always update the oldest companies. So the Scheduler will pick companies that were updated weeks or months ago.

We can query this information from the *company_schedules* table, however, I decided to create a separate table that only tracks company updates. This table looks like this:

ticker	market_data_updated_at	financials_updated_at
AAPL	2022-08-01 13:32:11	2022-08-02 18:19:43
MSFT	2022-09-03 15:43:12	2022-08-02 17:31:27

Updating market data (such as stock price, market capitalization, analyst recommendations) and financial statements are two different jobs. Mainly for two reasons:

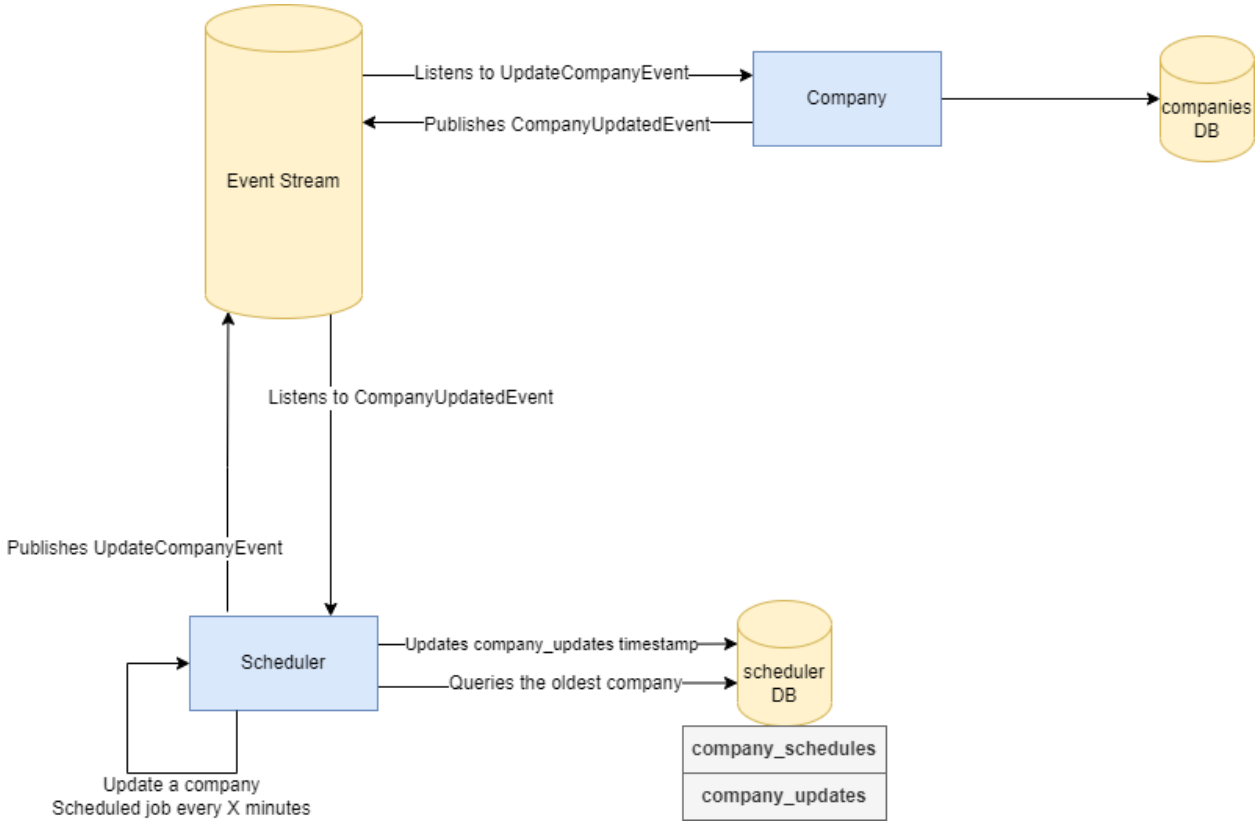
- It makes sense to update market-related information frequently, like every hour or so. But companies only issue new financial statements every quarter and the app cares only about yearly statements so it's not that frequent.
- Market data and financials statements are being stored in different services.

For these reasons, it makes sense to separate these two processes from each other.

This table contains every company we have, and it's pretty easy to query the oldest one. Here's the whole process:

- The Scheduler will run *company:update* and *company:update:market-data* commands on a regular basis.
- It queries the oldest updated company from the *company_updates* table.
- It also creates a new record in the *company_schedules* table.
- It publishes an *UpdateCompanyEvent*.

- The Company service listens to this event, updates the company, then publishes a *CompanyUpdatedEvent*.
- The Scheduler listens to that event and it updates the *company_updates* records with the current timestamp.



Scoring companies

We have already talked about scoring companies but I left out some important details.

Scoring companies is a "batch process." What I mean by that is that the Metrics service scores 20 000 companies in one process. If everything went well it notifies the Company Query service so it can update its database. Here's the problem we need to solve:

- If the Metrics service publishes one *ScoreCompaniesSucceededEvent* event after it scored 20 000 companies, how does the Company Query service get the data? We cannot push 20 000 objects into one huge event. I mean it's possible, but it's probably not a good idea.
- If the Metrics service publishes one small *CompanyScoredEvent* after each company and the Company Query service updates companies one-by-one it also has some problems. Inconsistency, for example. What happens if the process fails after 5 000 companies? Then 5 000 companies are updated in the Company Query database and 15 000 are not.

Here's what we want:

- Small events.
- Consistent database. The whole scoring process should be one "transaction." Even if we are talking about multiple services and databases.

In order to achieve this, we can use the best of both worlds. Here's a possible solution:

- The Metrics service scores all companies in one batch.
- After it's done **with every company**, it publishes one small event for **every company**. This is the *CompanyScoredEvent* and it contains the data for one company.
- The Company Query service "collects" these small events but it doesn't do anything just yet. I'm gonna explain what "collect" means soon.
- After all 20 000 events were published successfully and everything is fine in the Metrics service it publishes a *ScoreCompaniesSucceededEvent* which doesn't contain any data. It's just a signal that says "everything's fine."
- Finally, when the Company Query service gets the *ScoreCompaniesSucceededEvent,* it starts to process the previously collected 20 000 events and updates its database.

I said that the Company Query "collects" these events. This can mean a bunch of different things, but point is that it needs to store these events until it receives a *ScoreCompaniesEvent*. And this storage can be:

- A table in its database
- Or some in-memory storage

Since there's no need to store these events for the long term, I decided to go with an in-memory data storage. These events are temporary. We only need them for *5* minutes or so. One of the best ways of solving this problem is a Redis queue. I know, if I'm talking about a Redis queue in a Laravel-related book everyone will think about jobs, workers, Horizon, and so on. But the truth is, a queue in its original form is just an array. A simple array with a specific rule. This rule in our case is "first in first out." So we'll have an array in memory (in Redis), but we cannot just randomly access it. We are only able to access the "oldest" element in this array. It special array has two methods:

- enqueue
- dequeue

You can imagine a "first in first out" queue like this:

We can implement this queue with ~20 lines of code:

```
class ScoreQueue
{
    public const QUEUE = 'score-queue';

    public function enqueue(string $json): void
    {
        Redis::lpush(self::QUEUE, $json);
    }

    public function dequeue(): string
    {
        if ($this->isEmpty()) {
            throw new EmptyQueue();
        }

        return Redis::rpop(self::QUEUE);
    }

    public function isEmpty(): bool
    {
        return Redis::llen(self::QUEUE) === 0;
    }
}
```

There are two important Redis commands:

- *lpush*. It stands for “left push.” It simply pushes an item to the beginning of the array.
- *rpop*. It stands for “right pop” and it pops an item from the end of the array.

This is what you can see in the figure above, but you can implement the same functionality with *rpush* and *lpop*. They do the opposite as *lpush* and *rpop* but the result will be the same.

And here's how this queue can be processed:

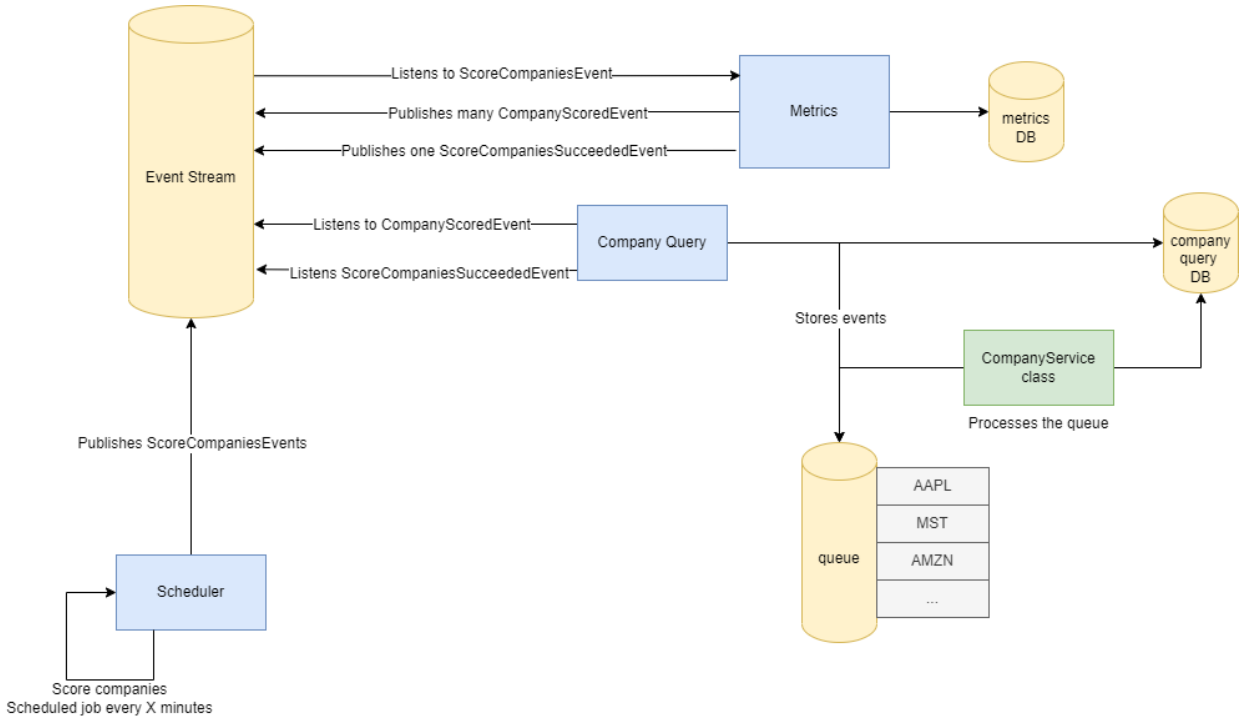
```
while (! $this->scoreQueue->isEmpty()) {
   $json = $this->scoreQueue->dequeue();

   $container = CompanyScoredContainer::fromJsonArray(
      json_decode($json, true)
   );

   $this->updateScores($container);
}
```

So this queue will collect 20 000 small events and after the Company Query service receives the *ScoreCompaniesSucceededEvent* it starts to process the queue.

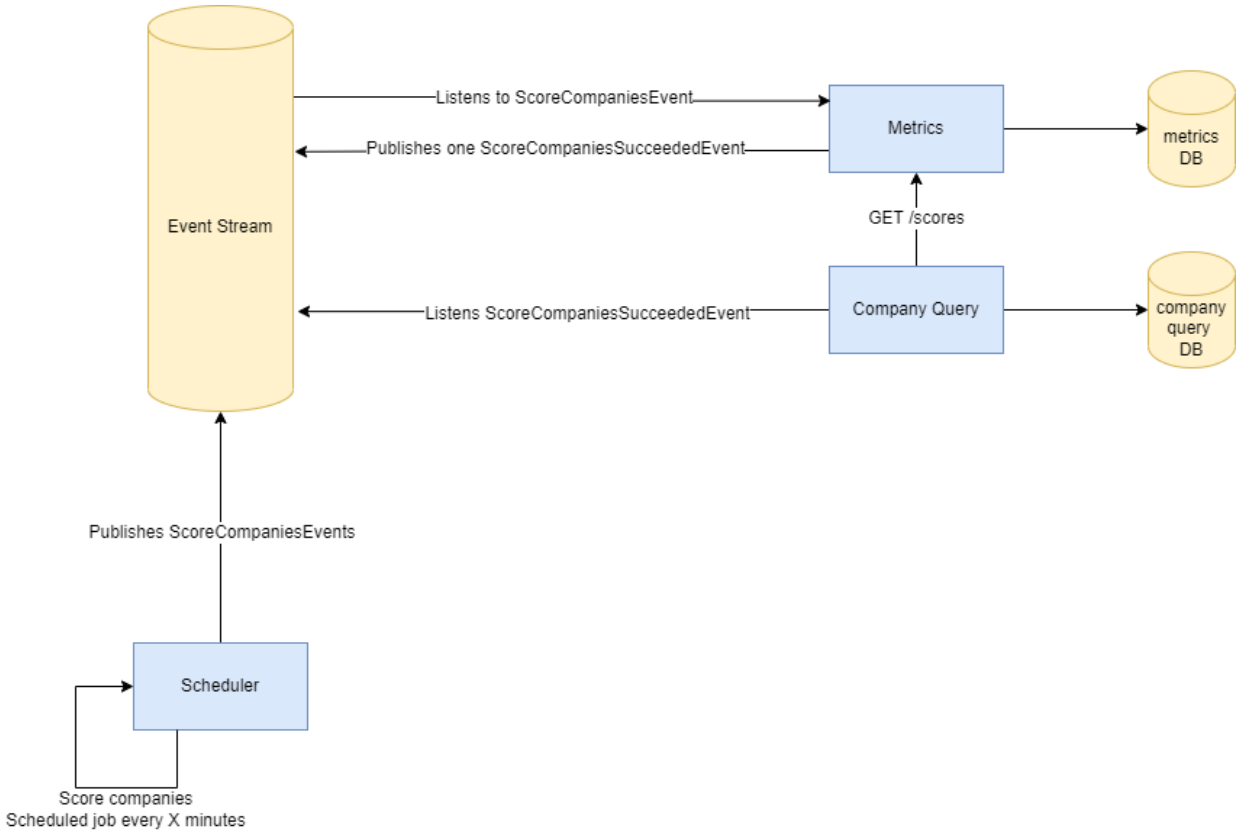
Here's the whole process:



The green box represents a class inside the Company Query service. It's the one that runs the *while* loop and processes the items in the queue.

You might be asking: why do we use a queue and all these events? Why don't we call the Metrics service's API from the Company Query service?

This can simplify the whole architecture. We can get rid of the queue and one of the events:



It's certainly easy to understand and a perfectly good solution, however, it has some disadvantages:

- The Metrics service has to return 20 000 companies from its API. It's a huge payload and causes a lot of memory consumption, and probably will be slow.
- The other option is to have a smaller API. An endpoint that has pagination and returns only 1 00 or 1 000 companies at once. In this case, the Company Query service will send hundreds or thousands of API requests so it's still resource intensive.

Of course, these problems are solvable, and we also have some load with Redis, but I found Redis to be easier to optimize.

Navigating the source code

Each service follows the same pattern. There's a *RedisConsumeCommand* in the *app/Console/Commands* directory. This gives a good idea about the flow of events.

This command also follows a given pattern. When a Redis event comes in, the command will fire an in-app Laravel event. In order to understand what happens inside a specific service always check out the *RedisConsumeCommand* and the *EventServiceProvider* classes. In the latter, you can see which listener will run after a certain event happens.

In the *routes/api.php* you can see what API endpoints a service exposes. All services follow a service-repository structure. You can often see two classes, for example, *CompanyService and CompanyRepository.* The responsibilities of these classes are:

- A repository only contains database queries.
- A service does everything else. For example, publishing events into the Redis event stream. Or sending e-mails, etc.
- Controllers mostly use service classes, and they use repositories.
- You can see a file called *depfile.yaml* in each service. It defines the exact rules of what class can use what other classes. For example, a repository can never use a service.
- Each service contains feature and unit tests, so you can see how to deal with Redis events, for example.

There are some performance tests in the root *tests* folder. They were written with Apache Jmeter. A pretty great tool when it comes to performance testing.

You can also find a ton of deploy scripts in the *bin* folder. This app was built before I knew about k8s or Elastic Beanstalk so I wouldn't do the same nowadays. However, I thought I kept them in the package. There are also some load balancer and fluentd configs in the *config* and *fluentd* folder. You can also find some fluentd, elasticsearch, and kibana containers in the prod compose file.