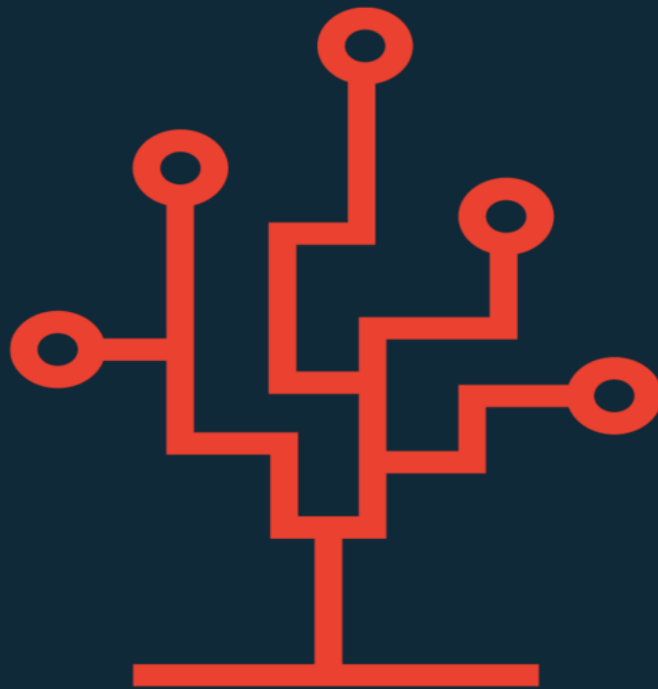


MICROSERVICES WITH LARAVEL



MARTIN JOO

Some theory	4
What's the problem with monoliths?	4
Loose coupling	5
High cohesion	5
What are microservices?	6
What's exactly a service?	8
What makes it micro?	8
Finding bounded context	9
Storing data in different services	10
Communication between services	13
Sync communication	13
Event-driven communication	15
Frontend	16
The promise of microservices	18
Building a microservice application	20
Requirements	20
Create products	20
Fill up inventory	21
Show the product catalog	21
Search in the catalog	21
Show the product's details	22
Rating a product	22
Buying a product	22
High-level overview	23
Installing the project	26
Products service	30
Redis event streams	33
Add a new entry	34
Read from the stream	34
Publishing events	36
Consuming events	37
Back to the Product service	41
Ratings Service	47
Warehouse service	56
Catalog service	68
Aggregate Catalog service	69
Proxy Catalog service	73

No Catalog service at all	76
How to choose?	76
Implementation	78
Warehouse	80
Warehouse service API	82
Ratings	85
Ratings service API	87
Products	89
Products service API	93
Put everything together	96
Orders service	102
Resources vs DTOs	109
Docker Containers	110
Dockerfile	111
Docker-compose	115
API gateway	121
Nginx reverse proxy	121
Custom gateway	122
Implementation	124
Logging	128
Frontend	131
Testing	137
Setup	137
API tests	138
CI/CD Pipeline	144
Final thoughts	150

Some theory

What's the problem with monoliths?

In order to understand what microservices are, first, we have to talk about monolith systems and their drawbacks.

Imagine Amazon as a Laravel project. It is one huge git repository, and it's made of classes like these:

- Product
- Inventory
- Catalog
- Rating
- Shipping
- Order
- OrderHistory
- Payment
- Discount
- Refund
- Recommendation
- Wishlist
- Account
- PrimeMember
- OneClickBuy

And so on. Just ask yourself: what does the one-click buy functionality have to do with shipping? One-click buy cares about credit card numbers, and user info, while shipping cares about product attributes, like weight and height, and addresses. Similarly refund logic has nothing to do with a wishlist.

Despite the fact that these modules, these classes are very different from each other, they live in the same repository, they can be accessed by the same API, they can call each other, and they can share some general functions or classes.

They are **tightly coupled** and have **low cohesion**. But in great architecture, we'd like to have **loose coupling** and **high cohesion**.

What the hell do those words mean?

Loose coupling

If a project is loosely coupled, a change in one place should not require a change in another place. Imagine you have Product, Order, Discount, and Payment in the same project. When you start the project you have some small, nice classes. The user creates an order with some product ids, you make a query from Product, calculate sum prices, and so on, nice and simple. After a while business wants discounts. So you call the Discount class from Order, and you pass the products. Nice and simple. One day, the business wants a discount based on the total amount of orders. No problem, you just pass the Order object to Discount. Next, you need to give a discount based on the user's order history and payments, so you pass more orders, and the User object to the Discount, maybe you also want to get something from the Payment module.

And here we are. The tightly coupled monolith. Now you have communication between:

- Order
- Discount
- Product
- Payment
- OrderHistory
- User

What happens next? You modify the *getNetPrice()* function in the Product class, and suddenly the Discount API call returns a 500 status code. Or even worse it calculates the wrong amounts. You change something in the User class and Order starts to behave differently. You modify something in OrderHistory and you get the wrong discounts.

This is a tightly coupled system. You make a change somewhere, and the application breaks in a different place. Oftentimes in a completely unrelated place.

High cohesion

High cohesion means that related behavior sits in one place, and other unrelated behavior sits in a different place. You can interpret this at the level of classes, for example, in the Order class, you don't want a method called *getProductPrice()*, you want it in the Product class. Soon, when we actually get to the point of microservices we will talk about cohesion at the level of product or services. But for now, the important part is we want related behavior in one place. And this is because, if we want to change that behavior we want to make sure to not break

anything else. So if you have high cohesion, you can make easier, more secure changes in your system.

In the world of monoliths, it is very easy to make high coupling and low cohesion. It comes from the nature of monoliths, you just put a ton of unrelated classes and methods next to each other. And as the number of features, the number of classes, methods, and the number of developers grow, it's just too easy to make the mistakes I mentioned above.

Don't get me wrong, you can write good, high-quality monoliths. And in my opinion, every system has some coupling and cohesion problems, no matter how well made it is.

What are microservices?

Imagine a restaurant where the waiter is using a mobile app to take your order. He takes your order, and the chef gets the information about what he has to cook. When the chef is done, the waiter delivers your meal, then you eat it. After that, the waiter prints your receipt via the mobile app. At the end of the day, the manager opens the same system on his laptop and sees the revenue, the net income, the cost of revenue and calls it a day. Every Friday he goes to his favorite supermarket and buys some ingredients, then he tracks them in the system. At the end of every month, he downloads an Excel about financials and e-mails it to the bookkeeper.

In this domain, we have models like:

- Product
- ProductGroup
- Ingredient
- Order
- OrderItem
- Receipt
- Discount
- Finance
- Storage
- Purchase

First, try to create some groups from these classes. In each group let's put similar classes, which are related to each other:

Groups:

- Product
- Order
- Discount
- Finance

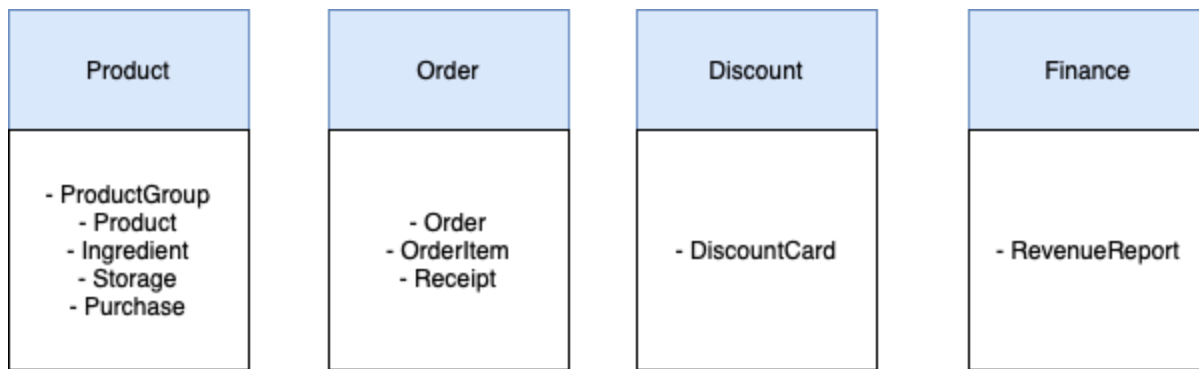
So we're grouping the classes based on their functionality. In our application, we have responsibilities, such as:

- Product management
 - Keeps track of our products and ingredients
- Order management
 - Stores the customers' orders
- Discount management
 - Handles discount cards and so on
- Finance
 - Creates reports about the business

Now let's put each class into one group:

- Product management
 - ProductGroup
 - Product
 - Ingredient
 - Storage
 - Purchase
- Order management
 - Order
 - OrderItem
 - Receipt
- Discount management
 - DiscountCard
- Finance
 - RevenueReport
 - CostOfRevenueReport

So, we basically namespaced our classes right? Yes, and in a classic monolith application, you create one repository and put all the classes in it. But in the microservices world, we create **a separate project for each namespace**.



What's exactly a service?

A service is a running Laravel instance, an application on its own, which usually has an API, and can communicate with the outside world. You create these groups, these boundaries around your functionality, and put each group into a different Laravel application with its own API.

It's not a composer package. It's an application. You can start it, stop it, deploy it at any time, call its API, run an artisan command in it, and so on.

What makes it micro?

In software development, size does matter. Usually the smaller, the better. The smaller your function, your class, the easier it is to maintain it. The same concept applies to services.

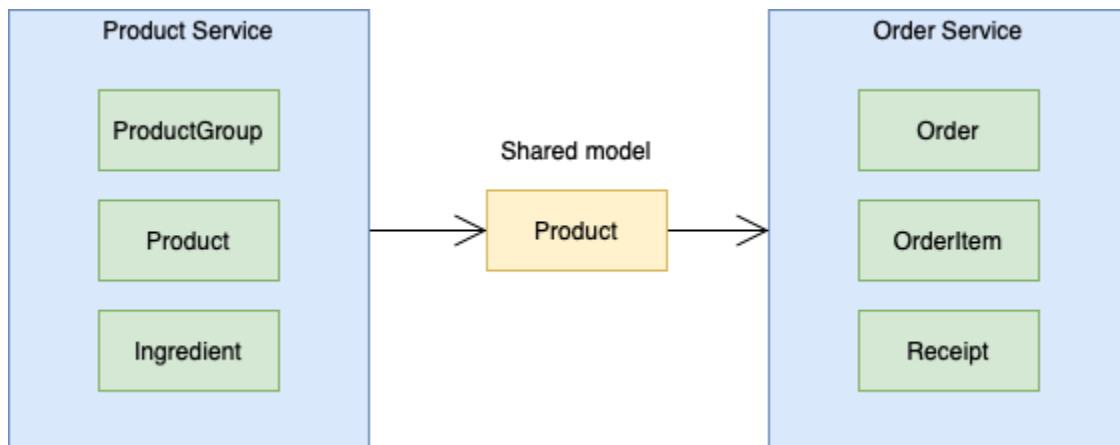
How small a service is? It's hard to tell exactly, but there's a good rule of thumb: it has to be small enough to completely rewrite it in one or two sprints. So if you have a team of 4 and you work in 2 weeks sprints, then you can rewrite the whole service in 2-4 weeks. By rewrite I mean, you delete the git repo and start from scratch. It really depends on the project, on the team, maybe it's 5000 lines of code for you, maybe it's 2000 for another team. By the way, the rewrite rule comes from Sam Newman's Building Microservices.

Alright, so we took our classes, put them into groups, and created 4 different services. How do they communicate with each other? How does the Finance service get the orders and products to calculate revenue? In the next few chapters we will talk about all of our options, but first, we need to talk about these groups a little bit.

Finding bounded context

On the previous pages, we talked about groups and grouping classes. But in the real world, we call these groups bounded contexts. The expression comes from Eric Evans' book called Domain-Driven Design. Every product has its own domain, and language, like the one in the restaurant example. Every domain has models in it, but you can always find a set of models that can be grouped together and you can draw a boundary around them. This boundary means that these models can live together without any other external dependency.

In every bounded context, there are some internal implementation details, and some internal models which you don't want to expose to the outside world. But it also has some shared models, which can be shared with the outside world. This is a very important concept. What it means is that you don't want to expose all of your Laravel Models from the Product service for example. You want to keep these models inside the product service because it has a lot of implementation details, and you want to create some leaner, transformed version for external usage. We will talk about it in more detail later in this book. For now, you can imagine something like this:



Where the green Product box is the internal model, a classic Laravel Model, and the yellow one is an external model, which can be shared with the order service, for example. In code, we'll use DataTransferObjects to represent these shared models.

Finding bounded context is the single most important thing when we talk about microservices. And unfortunately, it is the hardest skill to have. For example in the restaurant example, we have a model where:

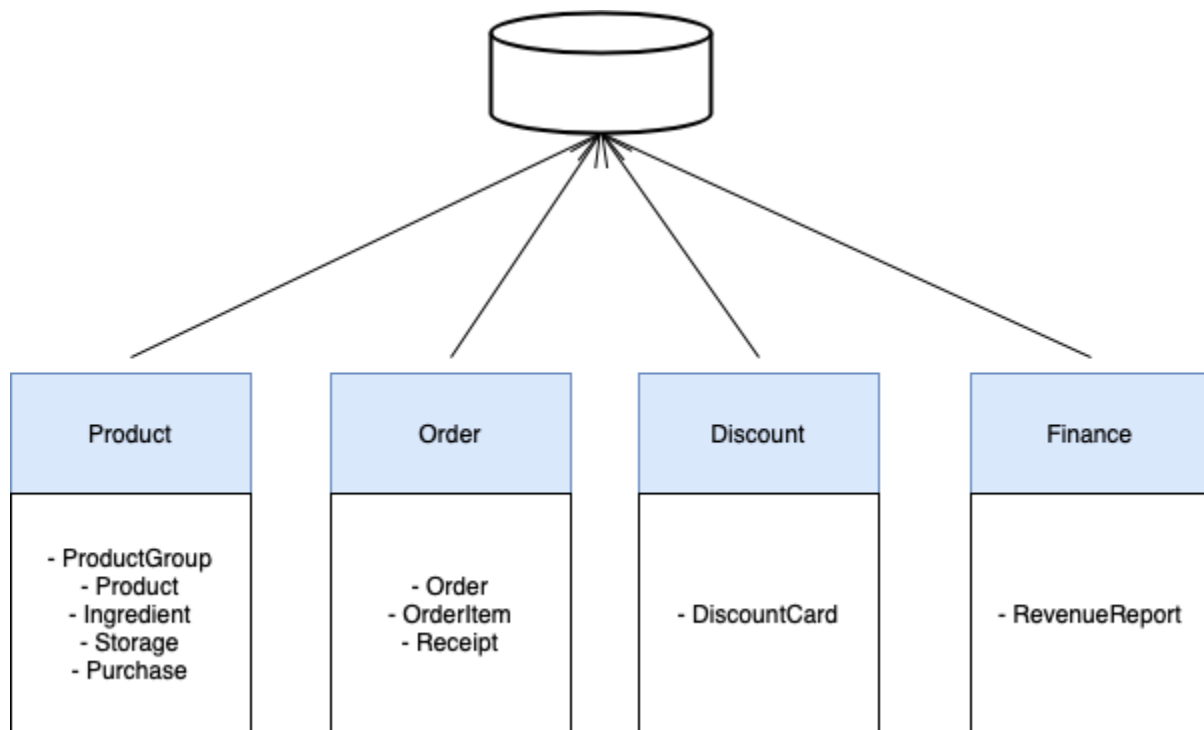
- ProductGroup contains products
- A Product is made of ingredients

- An Ingredient is stored in a Storage

And we have these models in the same service, in the Product service. But why don't we have a separate service for the whole storage management? And the answer is because this is just an example. Here's the thing: products and ingredients are closely related things, so it's logical to put them in one bucket. But in my experience warehouse management is complicated enough to have its own service. But maybe in this application, it's a simple thing so we don't want to complicate things. So it really depends on the product, the domain, and the functionality of our software.

Storing data in different services

The first big challenge that comes with microservices is storing data. This is harder than you think. The single most important thing is that you don't want to have only one database. So we won't do this:

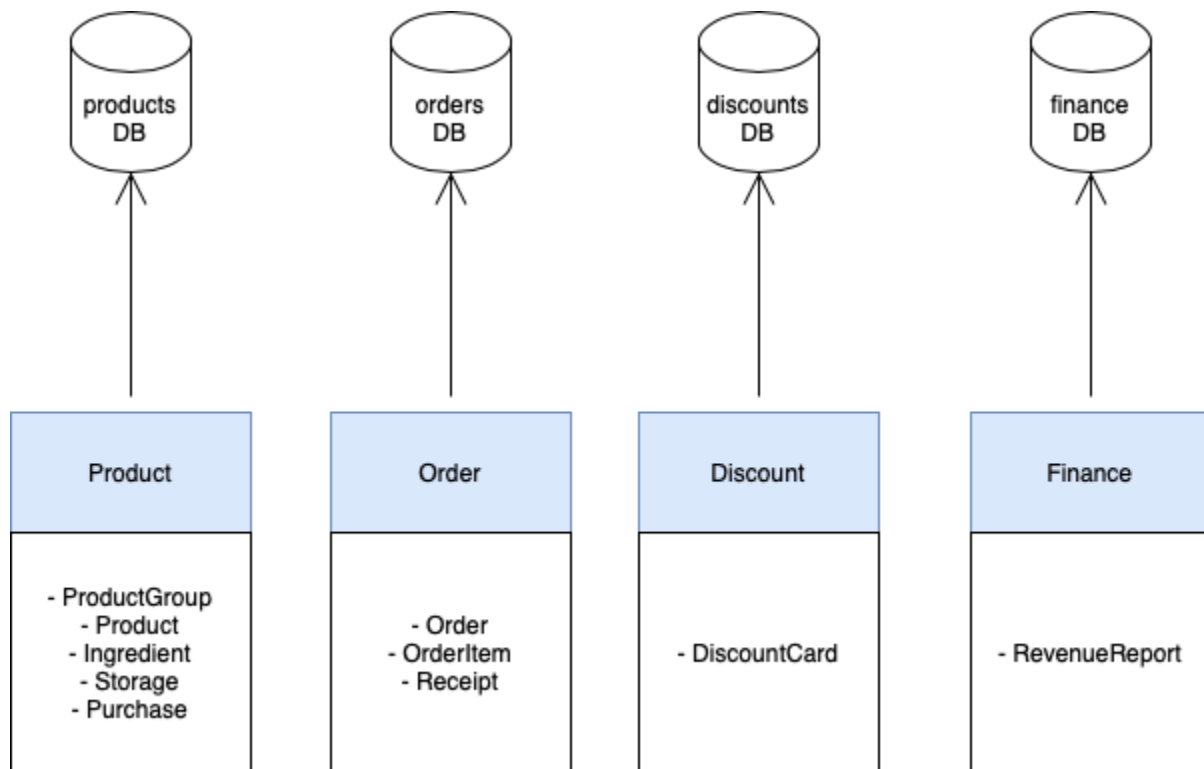


There are several problems here:

- **Tight coupling.** In this example, the Finance service needs to know every column and every detail about how a Product is stored in the DB. Finance is just a user of the product service, so it does not need to know if the ID is a string or an integer and so on.

- **Low cohesion.** Finance service will possibly break every time when someone changes the scheme of the products table. You make a change in the Product services and suddenly Finance stops working. This is bad.
- **Shared responsibility is bad.** Who's in charge of modifying the scheme? You can say it's easy, the scheme of products table is in the Product service, the scheme of orders table is in the Order service, and so on. It's easy to keep track when you have 4 services. But what if you have 132 services? In one database basically, there are no boundaries, no rules about how the data is related to services, and who is responsible to manage this data.
- **Too much dependency.** We want each service to run independently from other services. If we have a giant, shared database it's hard to achieve.
- And a bonus point: maybe we want a different type of database for a service, because it works better with NoSQL for example, meanwhile the rest of our services work better with SQL.

So here's what we want to do instead:



So every service has its own database (if it needs one). There are several benefits:

- **Loose coupling.** Much less coupling between services. Now Finance service does not depend on the scheme of products table. It only uses the shared models of the Product service.

- **High cohesion.** Now Finance service has no idea of the products table or products scheme. You can do whatever you want in the Product service and products DB, it will not break Finance.
- **Well-defined responsibilities.** Now the Product service is in charge of the products DB and no one else can touch it. Similarly, the Discount service is responsible for the discounts database and so on.
- **No dependencies.** Now every service runs independently from each other's data. They are just using the shared models and communicating with well-defined contracts (APIs and events).
- Now Finance can use MongoDB, while Product has a MySQL db, but Order has PostgreSQL. There are no technical limitations.

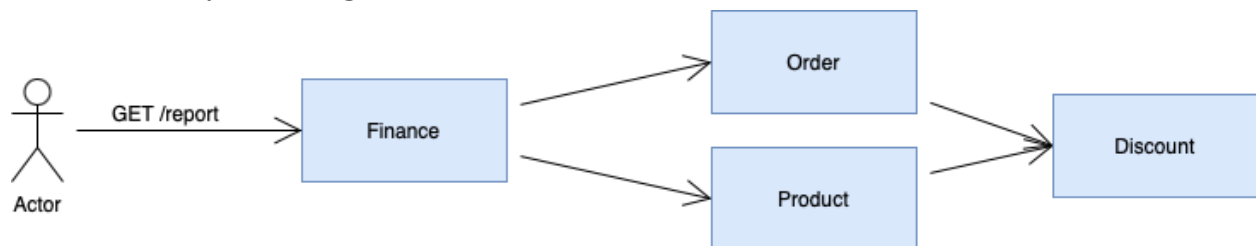
Communication between services

The other big challenge we have to solve is communication between services. So we split our application into different, small services. But we are no longer able to call functions between two services like we were in a monolith. In other words, how does the Finance service get the orders for a report? There are two ways to solve that problem (in fact there are more than two solutions, but we will focus on these two).

Sync communication

This is the easiest to understand and implement so we'll start here. We are using a request-response model to communicate between services. We are not talking about HTTP exclusively, but this is the most common one (other solutions can be RPC and protocol buffers).

Let me show you a diagram first:



Each arrow represents an HTTP API request. So whenever a service needs some data from another service it sends a request and gets back a response with the data. It's called sync communication because everything happens in order:

- The browser sends an HTTP request to the Finance service and waits for a response
- The Finance service sends a request to the Order service and waits for a response
- The Finance service sends a request to the Product service and waits for a response
- Both Order and Product service send a request to the Discount service

After every request is served, the Finance service does its calculations and responds to the user's request.

It has some advantages:

- Easy to understand. It's very similar to function calls in a monolith, except they are HTTP requests. But the flow of information is kind of the same.

- You can implement some services without a database. In the restaurant domain, the Finance service can be a good example. It does not necessarily require a database, because all the data that it needs is already there in some other services. So it's a possible solution to just request this data.

Besides, it is easy to understand and implement, but it has some major drawbacks:

- **Dependency** between services. This is an important one. We want to use microservices because we can develop services independently from each other, and we can deploy them whenever we want. If we couple them with HTTP calls it just makes life harder.
- **More sources of errors.** We make a lot of HTTP requests inside our services. What happens if the Order -> Discount request fails? Then the whole request chain fails. What happens if a service is down, and a timeout occurs? Then the whole request chain fails. This is gonna take us to our next drawback.
- **Latency.** We make **lots of requests**. Imagine we have just 30 services. We can easily make a request chain of 20-30 HTTP calls. It's just slow, way slower, than 20-30 function calls in a monolith.
- **Circular dependencies.** What happens if the Order service calls the Product service, then the Product service calls the Discount service, but for some reason, the Discount service also calls the Product service?



If you have a lot of services, and multiple teams working on them, it is very easy to do something like this. And now you have a circular dependency, and an out-of-memory or max execution time exceeded error message. These HTTP calls are very similar to function calls in a big monolith, where every module calls every other module.

Disclosure: Don't get me wrong, sync communication via HTTP is not evil by nature! You'll see later in this book, that we use this technique because it's simple and easy to use. In fact, sometimes you don't have another choice. Sometimes you just have to make an HTTP call, because it's synchronous, and you need an immediate response. But in my opinion, it's not a good solution, to build an entire system around it.

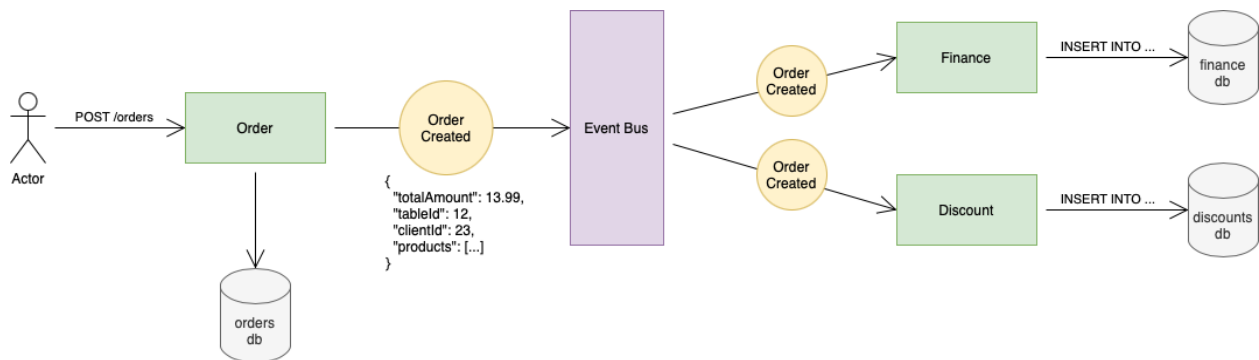
So what other alternative do we have?

Event-driven communication

In this communication model, we have to introduce a new component, called the event bus. We will talk about it in more detail later, and we will implement it, but for now, think about it as a broker who delivers messages between services.

In this model, we also rely on the concept that each service has its own database.

Let me show you an example:

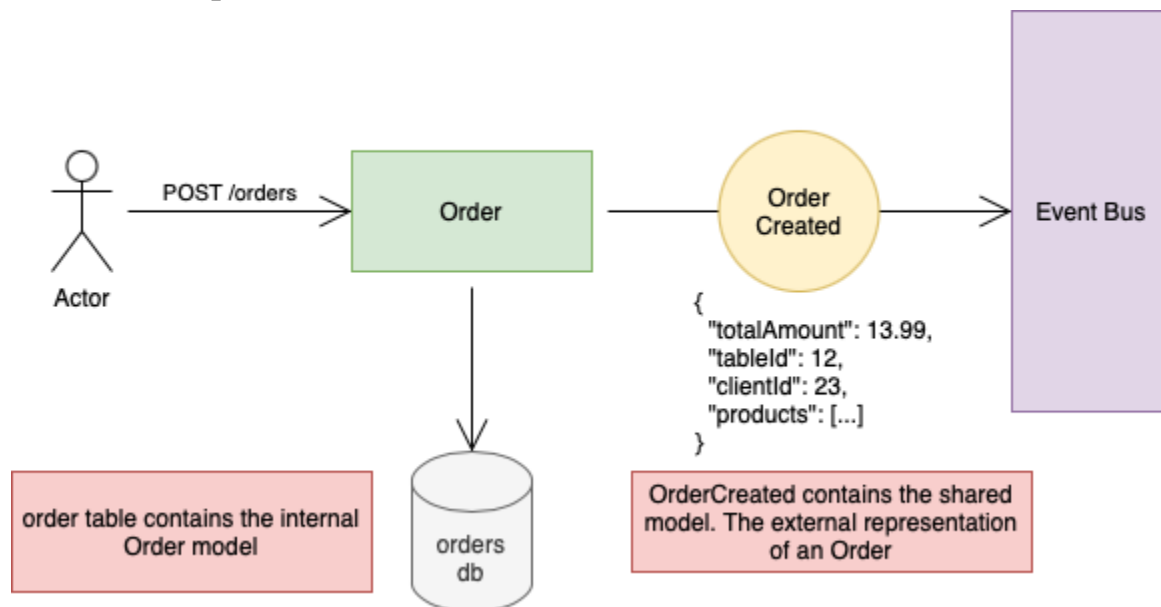


Let's review what is happening here:

- The user creates a new order
- The Order service receives an HTTP POST request with the order data
- It does its job, calculates some stuff, and writes it into the DB. So now the new order is created
- The Order service then **publishes** an event called *OrderCreated* to the Event Bus. It contains all of the important information about the order, such as the total amount, the table number, the customer, the product, etc.
- The Event Bus pushes the *OrderCreated* event to all the services that are interested in it. For example, the Finance service needs the newly created order to calculate the daily revenue. But maybe the Product service does not care about orders, so there is no need to push. (By the way, technically it can be a pull model also, where services pull the new events out of the event bus)
- The finance service gets the *OrderCreated* event, processes it, and makes an *insert* or an *update* on its own database.

So this is how every service builds its own database. By communicating through an event bus, and consuming interesting events. This way, every service has every important data it needs. No need to make an HTTP request from the Finance service to the Order service anymore. As the events come into the Event Bus, every service processes it and saves the important data.

One more important note:



When the Order API processes the POST request and saves an Order instance into the orders table, this is what we call the **internal Order model**. It's only visible to the Order service, and it may contain 20 columns.

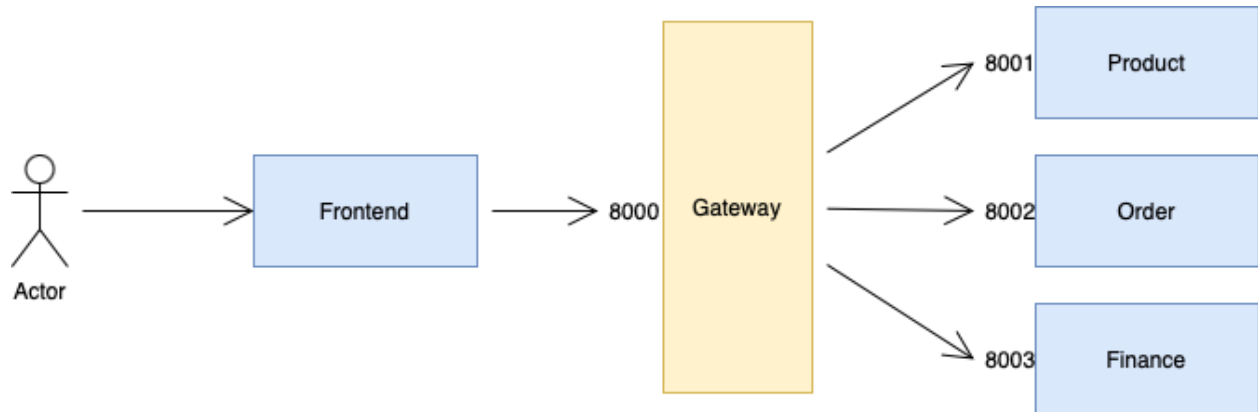
When the Order service publishes the *OrderCreated* event, it contains **the shared Order model**. It's visible to the whole system and may contain only 5-10 columns. Events and the Event Bus always contain language-independent formats, like JSON, so they can be processed by any language.

The important takeaway here is that only the shared, external model (DTO) can leave the service boundaries.

Frontend

So now we have a bunch of services running on a host on different ports. What about the frontend? How the frontend will know that when a user clicks on the "Create Order" button we need to send a request to localhost:8001/api/v1/posts, but when a user clicks on the "Create Product" button we need to use localhost:8002 because this is where the Product service listens?

The answer is that our frontend does not know anything about these different services and ports. We need a gateway between the FE and the services. Something like this:



So the gateway sits between the FE and the BE services. It acts as a reverse proxy. It takes requests from the FE, maps them to the right service, and sends the request to it.

You can implement an API gateway in (at least) two ways:

- Reverse proxy. You can use Nginx or haproxy. It's very simple and takes about 5 lines of config by service. It only proxies your requests.
- You can write your own gateway "service." It acts exactly like an Nginx reverse proxy, but it actually contains some code, so you can do extra tasks like:
 - Authenticate every request via an Auth service. And by the way, in the microservices world, you want to use stateless, jwt token-based authentication.
 - Serving the responses from the cache. If you have a gateway service, you can cache in one place, and you don't have to implement caching in every service. I'm talking about Redis or Memcached.

Each solution can work, but if you choose the second one, please don't use Laravel. Don't get me wrong, it's an excellent framework, but it's a behemoth for this job. It will be really slow compared to a Nodejs express or a Python flask gateway.

The promise of microservices

I hope all of this sounds good and architect-y, but there is one more important question. Why are we doing this?

So, here are some of the advantages of microservices:

- There is no huge codebase anymore. All of our services will be small, easy to reason about, and simple. This is a huge gain if you think about it.
- It's more resilient and reliable. Almost every error and bug has a scope of one service. So if something goes wrong in the Finance service, everything else will work properly. This is also a huge advantage. Basically, it's way harder to make a mistake that brings the whole application down.
- You can deploy small changes frequently. Because we have small services, it's very easy to deploy and ship new features. If you identify the right service boundaries, you can make new features and changes that only affect one or two services.
- Availability. You can use Docker Swarm or Kubernetes to orchestrate your services. This means that you can run mission-critical services in more instances, you can easily isolate worker processes, and you can auto-scale more effectively.
- You can use 4-5 member teams to develop and maintain two or three services. It's much easier to scale your organization.
- It's also easier to create autonomous and cross-functional teams. This means that every team can handle the whole lifecycle of a feature or a service, which includes:
 - Idea
 - Design
 - Development
 - QA
 - Deployment
 - Shipping to production
 - Maintenance
- The above point is possible because they only need to know a couple of services from bottom to top. It contains a few thousand lines of code. Not millions.
- You can use multiple languages, multiple frameworks, or technologies in your services. You can use Laravel in the Product service, but use Symfony in the Finance service. They are communicating via API calls and events, not function calls, so it doesn't matter what's inside a service. You can also write some Nodejs or Python services if you want. You can use MongoDB for a specific service. And so on.

- Fast and small pipelines. Each service is about a few hundred or thousand lines of code, so tests will be run fast, images will be built fast, and so on. There are no more 20, 30 minutes long pipelines.

But of course, microservices are no silver bullet, so there are also some drawbacks:

- Harder to understand. There are a lot of services communicating with each other. There are a lot of containers running, and multiple databases to migrate and maintain.
- Harder to debug. Let's say a user clicks a button, and the operation goes through 5 services. Now if something bad happens, you may have to debug 5 different Laravel projects. If you use event-driven communication it gets even worse, because it's asynchronous, so it's even harder to debug.
- Harder to monitor. You could have 30 services, and 75 running containers to monitor. You have to learn specific, and sometimes complicated tools to monitor your running application (like Prometheus).
- Harder to collect logs. Again, you have 75 running containers which dump logs to stdout. It gets even worse if you have more than one server (and you probably have more). Once again, you need to learn specific tools to see your logs (like ELK or EFK stack).
- You could run into problems that only come with distributed systems. For example, a service publishes a couple of events, and somehow it's processed in the wrong order, and now you have some data inconsistency in your databases. You have to debug multiple services, events, databases, and so on.
- Sometimes you have to make a change in every service. Let's say you have 14 services, and you found a great Laravel package, maybe some static code analysis package. You want to use it. But you want to use it in every service. So you have to install it in all 14 services. If you need this to run in your pipeline, you have to make changes to 14 pipelines.

As you can see there are some serious issues that come with microservices. Problems that you never have in a monolithic application. And you need to learn some new stuff to effectively operate this architecture.

But one thing is for sure: if you do it right, and you do it with the right application, it is worth it.

Building a microservice application

In this part, we are going to build a basic e-commerce application. Similar to Amazon, but a very simple one. It's gonna be simple in terms of features, but we will build everything that makes it a good example for a microservices project. The main features are:

- Creating products
- Filling up the inventory
- Browsing and searching the products
- Rating and buying the products

We are not gonna implement every CRUD (create, read, update, delete) functionality, we will focus on the create and read operations mostly, but the same concepts will apply to update and delete. Also, in this sample application, we won't implement authentication or authorization.

In the next chapter, we will specify all of the requirements.

Requirements

Create products

We need to create products in order to show a product catalog to the users, so they can search and buy stuff. Attributes:

- Category
 - Predefined values, for example 'book'. We won't implement a create page for categories, we'll just migrate some into our database.
- Name
- Description
- Price
 - Sale price for the products. We won't handle net and gross prices in this example. Price is always in dollars.

Events to publish:

- ProductCreated

Fill up inventory

A product can only be bought if we have it in our warehouse, so we want to fill up inventory. Attributes:

- Product
 - We can choose from the existing products.
- Warehouse
 - We can choose from the existing warehouses. In this example we won't create CRUD operations for the warehouse model, we'll just migrate some.
- Quantity
 - We specify how much we want to fill up

In this way, we have a basic inventory, like "12 of Product A in Warehouse A". A product can be in multiple warehouses, so we need to sum up the quantities for each warehouse in order to get the full inventory for a product.

Events to publish:

- InventoryUpdated

Show the product catalog

This is the homepage of our e-commerce application. Users can browse our products, they can click on them and buy them. Attributes to show in the catalog:

- Product name
- Product description
- Product price
- Average ratings for the product
 - Users can rate products from 1..5. This is the average of all of the ratings for this product

Only products with available quantities will show in the catalog.

Search in the catalog

We'll have a search box at the top of the page, where users can search for products. Attributes to search in:

- Product name
- Product description
- Product category
- Sort by:
 - Name

- Price

After the user clicks on the search button, we have to filter the catalog to only show the products that match up with the search term.

Show the product's details

When a user clicks on a product in the catalog we navigate him to the details page of a product. Attributes to show:

- Product name
- Product description
- Product price
- Average ratings for the product
- Available quantity
- All ratings for the product

Rating a product

On the product's details page, we can rate the product. Attributes:

- Rating from 1..5
- Comment

Events to publish:

- ProductRated

Buying a product

On the product's details page, we can buy a product. We won't integrate any payment solution. Buying a product means the user can click on the buy button, and after that, we create an order.

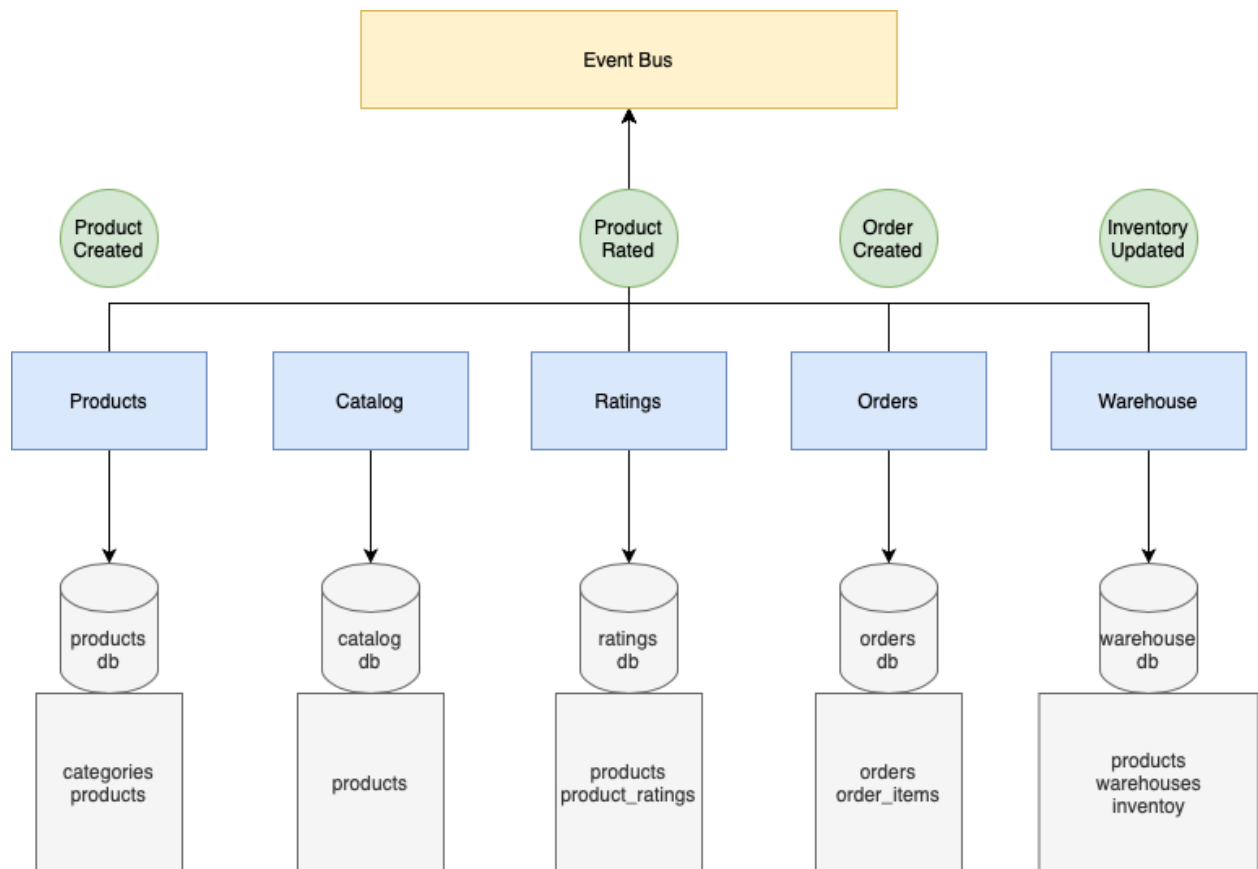
After we buy a product we have to decrease the inventory.

Events to publish:

- OrderCreated
- The Inventory service will listen to this event, decrease the inventory, and publishes an InventoryUpdated event

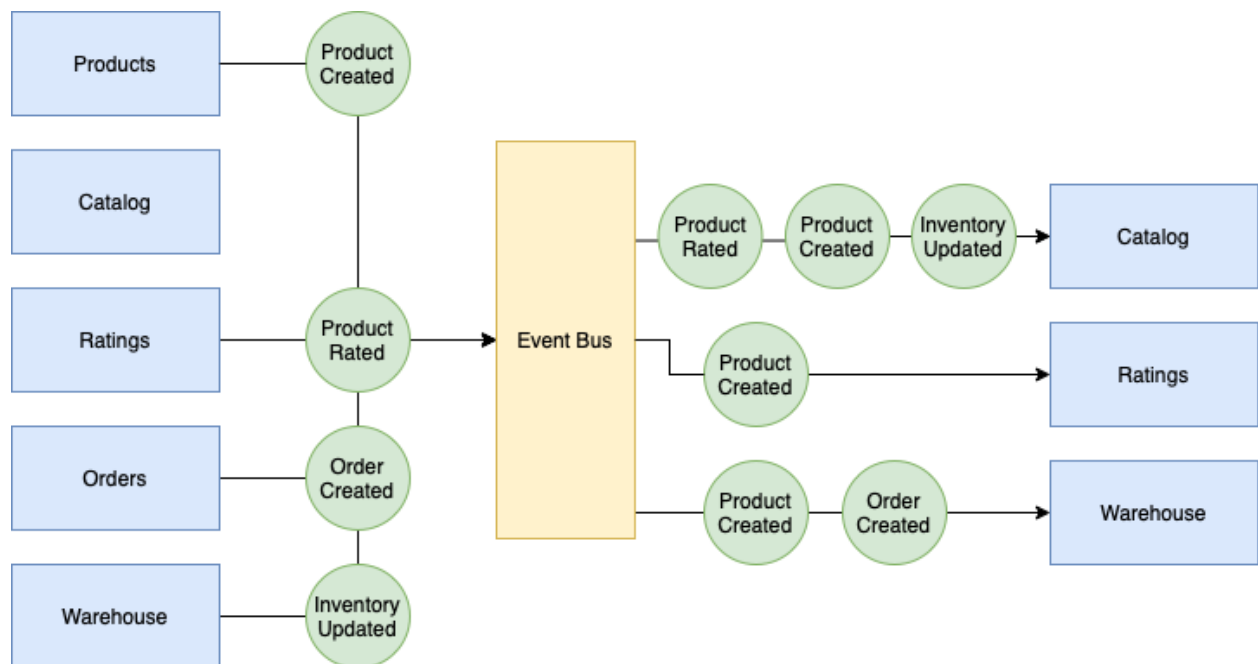
High-level overview

Here is the basic structure of our project:



I included the most important tables as well. As you can see our most important model is the Product. Almost every service has some idea about what a product is. But every service has a different definition for it. For example, in the Ratings service, we don't care how much a product costs, but in the Products or Catalog service this is an important attribute. In fact, price is maybe the most important thing for the Catalog service and for the users.

Here you can see the event flow:



This is what we talked about earlier. When we create a product in the Products service, it will publish a ProductCreated event, and three other services will listen to this event, and create their own version of the product.

Catalog service needs to listen to InventoryUpdated because only products with inventory will show up in the catalog. So when an InventoryUpdated occurs we'll update an is_visible flag.

Warehouse service needs to listen to OrderCreated because each time it will decrease the inventory for a product and publishes an InventoryUpdated event.

Right now no one listens to the ProductRated event, but the Ratings service will publish that. In my opinion, it's always a good idea to publish important events even if it's unnecessary right now. Imagine if we need a Recommendation service in the future. It needs all of the ratings because this is an important attribute for recommending products. If we have all of the ProductRated events in the event bus, then all that Recommendation service needs to do is to listen to those events, and it can build its own database from them. But if we don't have these events, then we have to write some extra code to migrate them into the recommendation database.

The last question is: where do we start? Which service to implement first? If you take a look at the event flow chart, you can see that the Product service does not listen to any event, it only publishes one. It means that we can implement it right

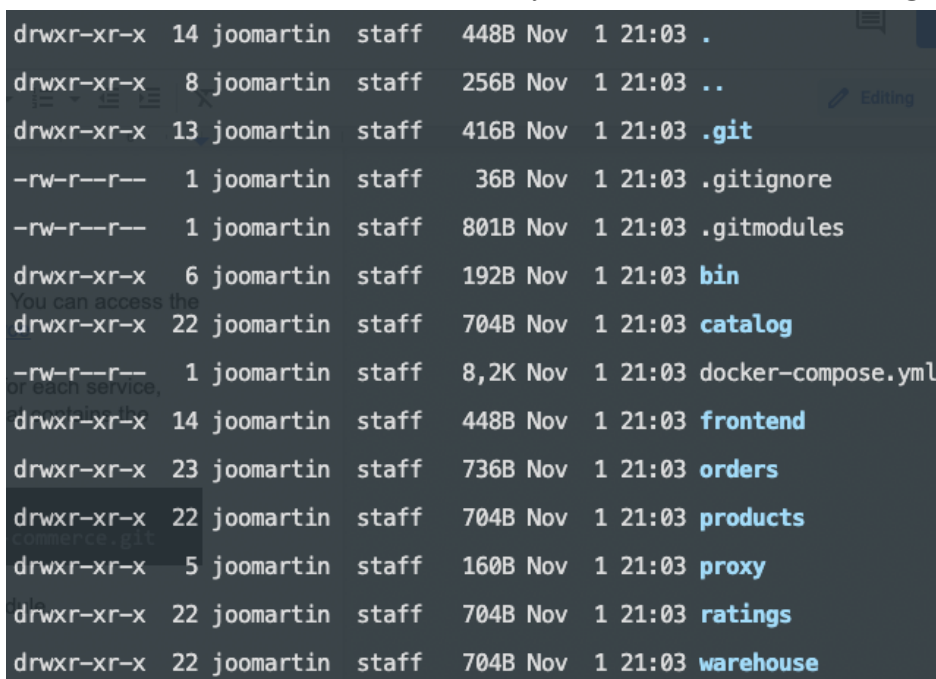
now, without any existing shared model, or service. So we'll start with the Products service.

Installing the project

You can try out the sample application [here](#). If you don't have the source code or don't want to install it for yourself feel free to check out this one. However, it's a bit slower since it's running on a pretty small server.

To install and run the application you need docker and docker-compose, so first please install them.

After you extract the source from the ZIP file you should see something like that:



```
drwxr-xr-x 14 joomartin staff 448B Nov  1 21:03 .
drwxr-xr-x  8 joomartin staff 256B Nov  1 21:03 ..
drwxr-xr-x 13 joomartin staff 416B Nov  1 21:03 .git
-rw-r--r--  1 joomartin staff  36B Nov  1 21:03 .gitignore
-rw-r--r--  1 joomartin staff 801B Nov  1 21:03 .gitmodules
drwxr-xr-x  6 joomartin staff 192B Nov  1 21:03 bin
drwxr-xr-x 22 joomartin staff 704B Nov  1 21:03 catalog
-rw-r--r--  1 joomartin staff 8,2K Nov  1 21:03 docker-compose.yml
drwxr-xr-x 14 joomartin staff 448B Nov  1 21:03 frontend
drwxr-xr-x 23 joomartin staff 736B Nov  1 21:03 orders
drwxr-xr-x 22 joomartin staff 704B Nov  1 21:03 products
drwxr-xr-x  5 joomartin staff 160B Nov  1 21:03 proxy
drwxr-xr-x 22 joomartin staff 704B Nov  1 21:03 ratings
drwxr-xr-x 22 joomartin staff 704B Nov  1 21:03 warehouse
```

Result of git clone

The next step is to run the project. You need to run the following command in the root directory (where docker-compose.yml lives):

```
docker-compose up
```

For the first run, it could take several minutes or so, because it's building images, initializing a MySQL server, and creating several databases. It waits for MySQL to be ready, and then it migrates all the databases. It also builds the frontend which is time-consuming.

If you're on an Apple M1 chip and see an error message "no matching manifest for linux/arm64/v8 in the manifest list entries" just add a *platform: linux/x86_64* config the following services in the docker-compose.yml:

- mysql
- redis
- mysql-test
- Redis-test
- phpmyadmin
- redis-commander

If everything went well you should be able to run the following:

```
curl localhost:8000/api/v1/catalog
```

The result should be an empty JSON. It's fine, all the databases are empty. If you want, you can seed some data by calling the run-seed script:

```
./bin/run-seed.sh
```

If you're on Windows and having trouble running the seed script you have to change the \$PWD variable in the run-seed script. Change to your absolute path, so the volume mount should look something like this: "c://projects/e-commerce:/work". If you're having trouble running sh files, just copy the content into your terminal (simple docker commands).

If you wish, you can reset all the databases by running:

```
./bin/reset-db.sh
```

If you run this, you probably want to clean Redis as well:

```
docker-compose exec redis redis-cli  
127.0.0.1:6379> FLUSHALL
```

The docker-compose includes a phpMyAdmin, which you can access on <http://localhost:8080>

It also comes with a redis-commander (Redis GUI) on <http://localhost:63791>

After running and seeding you should be able to access the frontend on <http://localhost:3000>

Here you can see what ports are being used:

Port number (on your host machine)	Service
3000	Frontend
8000	Proxy. Every API available via 8000
8001	Products API
8002	Ratings API
8003	Warehouse API
8004	Catalog API
8005	Orders API
33060	MySQL
63790	Redis
8080	PhpMyAdmin (MySQL GUI)
63791	Redis-Commander (Redis GUI)
<i>You can change any of these in docker-compose.yml</i>	

Here's a list of API endpoints:

Verb	API	Action
POST	127.0.0.1:8000/api/v1/products	Create a Product
POST	127.0.0.1:8000/api/v1/categories	Create a Category
GET	127.0.0.1:8000/api/v1/products	Get all Products
GET	127.0.0.1:8000/api/v1/products/1	Get a Product
GET	127.0.0.1:8000/api/v1/categories	Get all Categories
POST	127.0.0.1:8000/api/v1/products/1/ratings	Rate a product
GET	127.0.0.1:8000/api/v1/products/ratings?productIds[]=1&productIds[]=2	Get rating summaries for many Products
GET	127.0.0.1:8000/api/v1/products/1/ratings	Get all individual ratings for a Products
POST	127.0.0.1:8000/api/v1/warehouses	Create a Warehouse
POST	127.0.0.1:8000/api/v1/inventory	Create Inventory for a Product
GET	127.0.0.1:8000/api/v1/inventory/products/1	Get Inventory for a Product
GET	127.0.0.1:8000/api/v1/inventory/products?productIds[]=1&productIds[]=2	Get Inventory for many Products
GET	127.0.0.1:8000/api/v1/catalog	Get the Product Catalog
POST	127.0.0.1:8000/api/v1/orders	Create an Order

You have a file called “api-endpoints-postman.json” in the root folder, you can import it into Postman, and you’ll have a collection called “Microservices With Laravel” with all of the APIs above.

Products service

There are two important models in this service:

- Category
- Product

We won't implement any user interface for the Category model, we're just gonna seed some, so we don't need an API. But we need a product API, where we can create new ones.

And there is only one API endpoint for products right now which is *POST /products*. You can see in the *App\Http\Requests\StoreProductRequest* that it will take the following request data:

```
public function rules()
{
    return [
        'categoryId' => 'required|exists:categories,id',
        'name' => 'required|string|unique:products,name',
        'description' => 'required|string',
        'price' => 'required|numeric',
    ];
}
```

App\Http\Requests\StoreProductRequest

Creating a product is easy, we just run an insert query. You can see the logic in the *App\Actions\CreateProductAction* class.

The harder part is that we need to publish an event with the product data. For this purpose, I use to create a package called *common*. This is not a service, but a composer package, which contains the shared models and the events. Shared models are data shared between services. We will call Data Transfer Objects or DTO for short.

Each service will publish some DTOs, so it's a good idea to namespace these classes by services. For example, every DTO which comes from the Product service lives under the *Ecommerce\Common\DataTransferObjects\Product* namespace.

After all, DTOs are just plain old PHP objects. They contain some properties and (in most cases) a simple factory function:

```

namespace Ecommerce\Common\DataTransferObjects\Product;

class ProductData
{
    public function __construct(
        public readonly int $id,
        public readonly string $name,
        public readonly string $description,
        public readonly float $price,
        public readonly CategoryData $category,
    ) {}

    public static function fromArray(array $data): self
    {
        return new static(
            $data['id'],
            $data['name'],
            $data['description'],
            $data['price'],
            new CategoryData(
                $data['category']['id'],
                $data['category']['name'],
            )
        );
    }
}

```

Ecommerce\Common\DataTransferObjects\Product\ProductData

When the Product service creates a new product it will instantiate this class using the factory function and pass all the data. Other services which listens to this event will use the *fromArray* method that constructs a new *ProductData* from an array. We will talk about it later when we introduce our second service. The important thing is that the *ProductData* will be *transferred* between services using a Redis event stream (more on that later). Now you have a better idea why these classes are called data *transfer* objects.

As you can see a *ProductData* contains a *CategoryData* because a product needs a category. The *CategoryData* looks like this:

```

namespace Ecommerce\Common\DataTransferObjects\Product;

class CategoryData
{
    public function __construct(
        public readonly int $id,
        public readonly string $name,
    ) {}

    public static function fromArray(array $data): self
    {
        return new static($data['id'], $data['name']);
    }
}

```

Ecommerce\Common\DataTransferObjects\Product\CategoryData

So now we have our DTOs. The last thing we need is an Event. This is the thing we will publish into a Redis stream, and other services will listen to them. Basically, an Event is just a wrapper around a DTO. At the end of the day, when we create a new Product, we want to publish a data structure similar to this:

```

{
    "type": "product:created:v1",
    "data": {
        "id": 1,
        "name": "Product #1",
        "description": "My product",
        "category": {
            "id": 1,
            "name": "Books"
        }
    }
}

```

ProductCreated event in JSON

This is the JSON which we will publish into Redis. The outer object which has a *type* and *data* key is the Event. The object inside the *data* key is the *ProductData*.

So our event is really simple:


```

<?php

namespace Ecommerce\Common\Events\Product;

use Ecommerce\Common\DataTransferObjects\Product\ProductData;
use Ecommerce\Common\Enums\Events;
use Ecommerce\Common\Events\Event;

class ProductCreatedEvent extends Event
{
    public string $type = Events::PRODUCT_CREATED;

    public function __construct(
        public readonly ProductData $data
    ) {}
}

```

Ecommerce\Common\Events\Product\ProductCreatedEvent

Only a *type* and a *data* attribute. The base Event class only contains a *toJson()* method, which we will use later.

Now that we have our DTOs and Events, it's time to introduce Redis streams.

Redis event streams

If you haven't used Redis before, please, first go through the documentation.

We need a message broker or message queue to communicate between services. For example, the product service will publish a ProductCreated event in the form of JSON, the Ratings service will listen to it, and create its own version of the product. A message queue ensures communication between these services. There are multiple options like Apache Kafka, RabbitMQ, and Redis. For a bigger project maybe Kafka or RabbitMQ would be a better fit, but for our e-commerce app, Redis is more than enough. Don't get me wrong, Redis is a production-ready tool with very high throughput and good scalability, but Kafka or RabbitMQ offers more useful features. Also, Redis is pretty simple compared to the others.

Basically, a stream is a data structure in Redis which is similar to a big log. It's meant to be an append-only data structure. You can think of it like a big log file, where your application writes data. Only, in this case, services will write events to it.

Every stream entry has an ID field. By default, it's the current timestamp.

More on Redis streams: <https://redis.io/topics/streams-intro>

Add a new entry

We can use the *xadd* function to add an entry to our stream. It takes a key, an ID, and the field-value pairs, something like this:

```
XADD events * field1 value1 field2 value2
```

The *** means that we don't want to specify an ID, we let Redis use the current timestamp. We want to use this method in the Product service.

Redis doc: <https://redis.io/commands/xadd>

Read from the stream

We can read entries from the stream with the *xrange* function. It takes a key, a start, and an end ID.

```
XRANGE events 1633380851837-0 1633391951823-0
```

The start and the end IDs are just timestamps. So we want to get every entry from the stream called *events* between these two timestamps. The 0 suffix at the end of the timestamps is just a serial number, Redis will generate it. If you have more than one event at the exact same moment it will be 0, 1, 2, and so on.

More on xrange: <https://redis.io/commands/xrange>

As I mentioned before stream is an append-only data structure. That means it is a little bit different than a traditional job queue. When you add a job to a queue a worker will pick it up, so it's a one-time job. If you add a new entry to a stream, and a consumer reads it, the entry will remain in the stream. And that's a good thing for us for three reasons.

First, you have the whole history of your system in one place. Every user operation, every event ever happened, it's there. Imagine you have an application that is running in production for six months. You get a new feature request and you realize you need a new service to implement it. How your new service will build its database? You need to write some migrations, or commands that will collect all the data, process it, then insert it into the new database. But, you have 12 services, and 12 databases in your application, so it's maybe a little bit more complicated than you think. Maybe you need to implement some new API endpoint in other services, just to get the data that your new service needs. Maybe it's not viable, so you have to have direct access to your databases from your new service. These options can work, but neither of them is optimal. Obviously, you don't want to write new API endpoints because of data migrations. And having access to a database from multiple services is just a dangerous idea. But if you have a stream that contains all the events from the past six months, you just have all the data that your new service needs. You just have to write a stream consumer in your new service, which will consume and process all the interesting events. And your service will build its database from these events. All you have to do is to deploy your new service.

The other reason why the append-only structure is good for us is kind of the same as the first one, but it's a different scenario. Imagine your 12 services running, but something goes wrong and the Product service goes down. Maybe you don't use Kubernetes, so you notice it only after an hour. Meanwhile 20 events happened that the Product service needs to know about. But fortunately, the events are not lost. They are stored in the stream. All you have to do is to fix the Product service and deploy it.

And lastly, it's good for us, because one event can be consumed by many services. So we definitely don't want a queue-like one-time job behavior.

The main downside of Redis events is that you need to maintain a list for every service which contains the last processed events. The timestamp of the last processed event will be the start timestamp in the `XRANGE` command. Don't worry this is easy to implement.

Earlier in this book, we talked about communication between services via API. Now imagine what happens if your only communication channel is an HTTP API, and the Product service goes down. What happens to a request which was sent to the Product API meanwhile the downtime? It gets lost. You have to write some script or command to resend these requests.

Of course, there are some workarounds, but all of these problems are solved by an event stream, message stream, or message broker. You can call it whatever you like.

Now that we understand what are Redis events, let's implement a *RedisService* class in the *common* package which will be used by all of our services. You can see the class in *Ecommerce\Common\Services\RedisService*

Publishing events

```
public function publish(Event $event): void
{
    Redis::xadd(self::ALL_EVENTS_KEY, '*', [
        'event' => $event->toJson(),
        'service' => $this->getServiceName(),
        'createdAt' => now()->format('Y-m-d H:i:s')
    ]);
}
```

Ecommerce\Common\Services\RedisService

We're using the PHP extension called *phpredis* to interact with Redis. It's very simple, basically, you have all the Redis commands as a static method on the Redis class. You can read about it here: <https://github.com/phpredis/phpredis>

So we call the XADD with the following arguments:

- Key: it's gonna be the key in our Redis database. `self::ALL_EVENTS_KEY` is just a constant for 'events'.
- ID: this is the asterisk which means we rely on Redis to use the current timestamp as an ID
- Data: this is an array that holds the actual data
 - event: the event as JSON (remember this is why we have the `toJson` method on the base Event class)
 - service: the name of the service that published the event. It's good for debugging.
 - createdAt: a human-readable date, it also helps to debug.

And that's it. We have a method that can publish events into Redis.

Consuming events

This operation is a little bit more complicated:

```
public function getUnprocessedEvents(): array
{
    $fromTimestamp = $this->getLastProcessedEventId();

    $events = $this->getEventsAfter($fromTimestamp);

    return $this->parseEvents($events);
}
```

Ecommerce\Common\Services\RedisService

We get the id (timestamp) of the last processed event and get every event after that one. We will talk about how to do this a bit later. Now let's see how we can get events after a timestamp:

```
protected function getEventsAfter(string $start): array
{
    /** @phpstan-ignore-next-line */
    $events = Redis::xRange(
        self::ALL_EVENTS_KEY,
        $start,
        (int) Carbon::now()->valueOf()
    );

    unset($events[$start]);

    return $events;
}
```

Ecommerce\Common\Services\RedisService

We call XRANGE with the last processed event ID as the start and the current timestamp as the end. XRANGE intervals are inclusive, so we have to remove the start because it's already processed. As you can see XRANGE returns an associative array indexed by event IDs.

Finally, we need to parse the events:

```
/**
 * @return array{type: string, data: array, id: string}
 */
protected function parseEvents(array $eventsFromRedis): array
{
    return collect($eventsFromRedis)
        ->map(function (array $item, string $id) {
            return array_merge(
                json_decode($item['event'], true),
                ['id' => $id]
            );
        })->all();
}
```

Ecommerce\Common\Services\RedisService

Remember, when we publish an event we use the *toJson()* method, so now we have to decode the event, and we also add the event ID to the result array.

So, back to the last processed events. I mentioned earlier that all services will write the ID of the last processed event to *something* in Redis. That something is a simple list. A list is a pretty similar data structure to PHP arrays. You can read more about sorted sets here: <https://redis.io/commands/?group=list>

This is the method that adds a new entry to the list:

```
public function addProcessedEvent(array $event): void
{
    Redis::rpush(
        $this->getServiceName() . '-' . self::PROCESSED_EVENTS_KEY,
        $event['id'],
    );
}
```

Ecommerce\Common\Services\RedisService

The *R PUSH* command will add a new element at the end of the list. It does the same thing as *\$arr[] = 10;* in PHP. The list looks like this:

ID (Total: 18)	Value ↕
1	1661797743334-0
2	1661797743407-0
3	1661797743480-0
4	1661797743555-0
5	1661797758764-0

Event ids in a list

The application will push items in chronological order so the list will be sorted by default. For that reason, it's straightforward to get the last processed event's ID:

```
private function getLastProcessedEventId(): string
{
    $lastId = Redis::lindex(
        $this->getServiceName() . '-' . self::PROCESSED_EVENTS_KEY,
        -1,
    );

    return empty($lastId)
        ? (string) Carbon::now()->subYears(10)->valueOf()
        : $lastId;
}
```

Ecommerce\Common\Services\RedisService

The *LINDEX* command returns one item at a given index in the list. The first argument is the key (the name of the list) and the second one is the actual index. In this case, we use index -1 which means it'll return the last item from the list. This is the same command as *array_pop()* in PHP.

If there are no items in the list it'll return an empty string and in this case, we need a fallback value. I choose now minus 10 years. If the service has not processed any events yet it will call *XRANGE* with a very old timestamp and it will return every event in the stream.

These are the most important parts of the *RedisService* class. You can check out the whole class in the common module *Ecommerce\Common\Services\RedisService*.

Back to the Product service

Now, that we have a generic *RedisService* class we can extend it in the Product service:

```
<?php

namespace App\Services;

use Ecommerce\Common\Containers\Product\ProductContainer;
use Ecommerce\Common\Events\Product\ProductCreatedEvent;
use Ecommerce\Common\Services\RedisService as BaseRedisService;

class RedisService extends BaseRedisService
{
    public function getServiceName(): string
    {
        return 'products';
    }

    public function publishProductCreated(ProductData $data): void
    {
        $this->publish(new ProductCreatedEvent($data));
    }
}
```

App\Services\RedisService

We will create a lot of methods like this one. Each one follows the naming pattern *publishEventName* and has one parameter which is a DTO.

Let's see what our Controller looks like:

```
public function store(
    StoreProductRequest $request,
    CreateProductAction $createProduct,
) {
    $product = $createProduct->execute(
        Category::findOrFail($request->getCategoryId()),
        $request->getName(),
        $request->getDescription(),
        $request->getPrice()
    );

    return response([
        'data' => $product->toData(),
    ], Response::HTTP_CREATED);
}
```

App\Http\Controllers\ProductController

First, we query the Category, then the *CreateProductAction* creates the product:

```
class CreateProductAction
{
    public function __construct(
        private readonly RedisService $redis,
    ) {}

    public function execute(
        Category $category,
        string $name,
        string $description,
        float $price,
    ): Product {
        $product = Product::create([
            'category_id' => $category->id,
            'name' => $name,
            'description' => $description,
            'price' => $price
        ]);

        $this->redis->publishProductCreated(
            $product->toData(),
        );

        return $product;
    }
}
```

App\Actions\CreateProductAction

The action returns a *Product* model. But if you take a look at the controller we call a *toData* method before returning the response. This method is a simple factory function that creates a *ProductData* from a *Product* model:

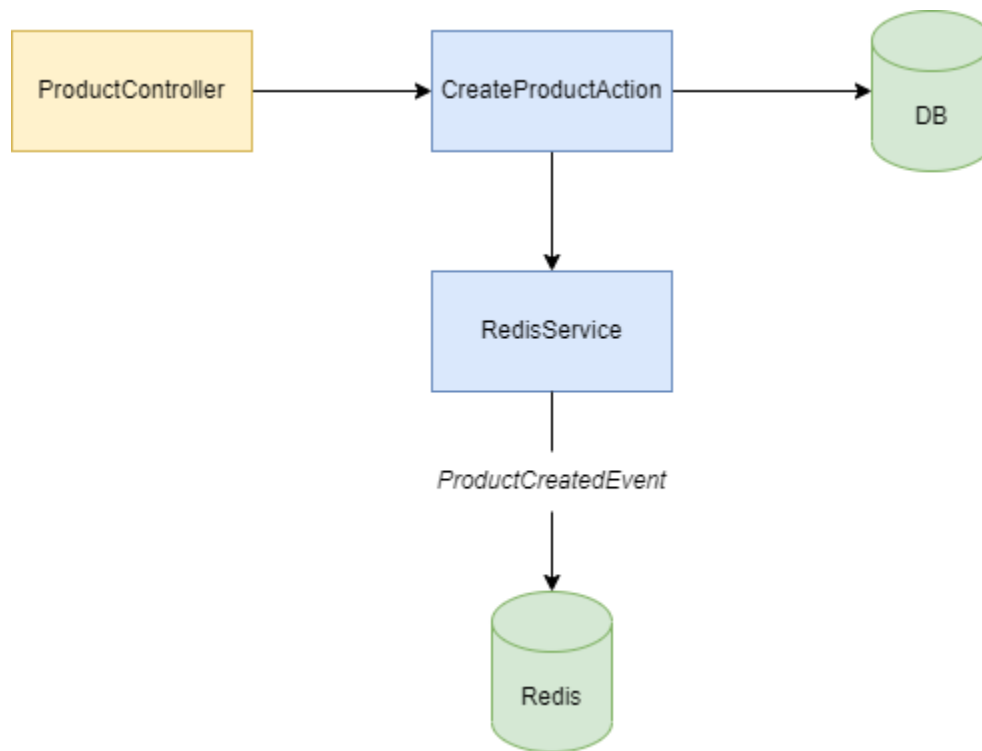
```
public function toData(): ProductData
{
    return new ProductData(
        $this->id,
        $this->name,
        $this->description,
        $this->price,
        new CategoryData(
            $this->category->id,
            $this->category->name,
        ),
    );
}
```

App\Models\Product

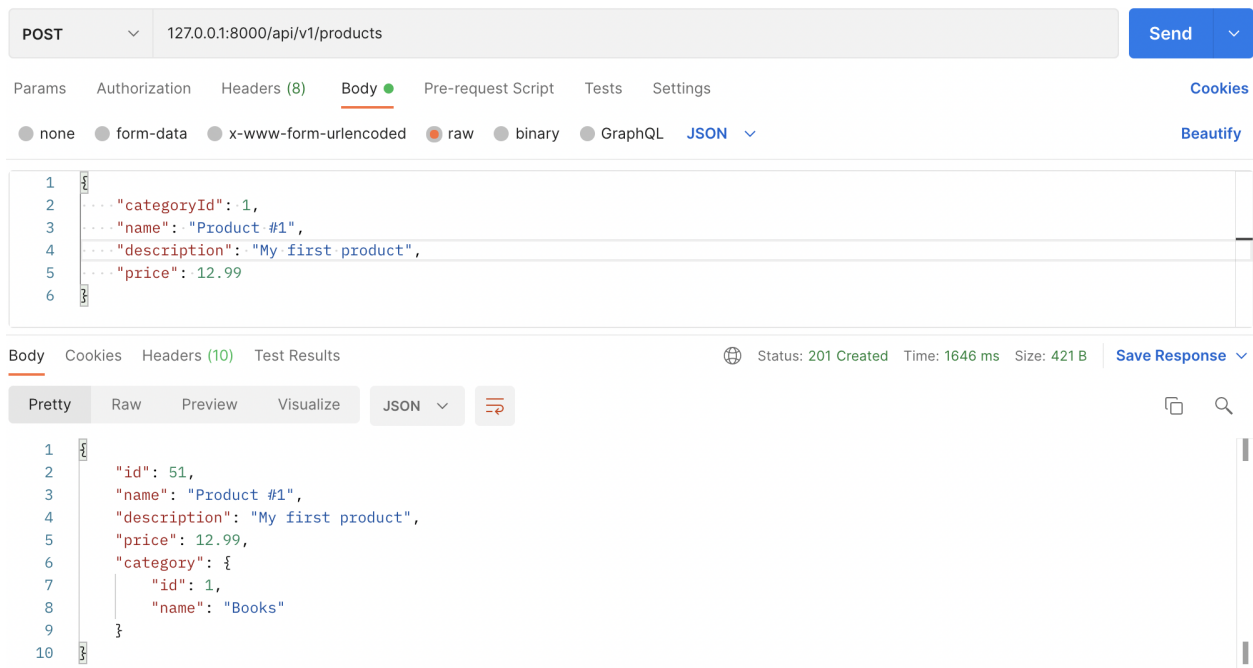
Another approach would be to add a *fromModel* method to the *ProductData* class. Both work fine.

If you're not familiar with action classes, please check out [this](#) and [this](#).

So let's see the whole flow of the Product service:



Finally, we can try it out with Postman:



After we send the POST request, we can see a new event in Redis, something like this:

```
event: {  
  "type": "product:created:v1",  
  "data": {  
    "id": 51,  
    "name": "Product #1",  
    "description": "My first product",  
    "price": 12.99,  
    "category": {  
      "id": 1,  
      "name": "Books"  
    }  
  }  
},  
service: products,  
createdAt: 2021-10-10 08:47:06
```

Ratings Service

The responsibility of the Ratings service will be to store products after they are created and expose an API. The API will be simple and consists of only one endpoint to rate the actual product.

The Ratings service will consume events from the stream. The best way to implement this feature is by creating a Command which will use our *RedisService* to retrieve new events. We will run this command in a scheduled way, like once every 30 seconds or so. We implement this Command in every service that needs consuming Redis events. We will call it *RedisConsumerCommand*. Here's how it looks like:

```

class RedisConsumeCommand extends Command
{
    protected $signature = 'redis:consume';

    protected $description = 'Consume events from Redis stream';

    public function handle(
        RedisService $redis,
        CreateProductAction $createProduct,
    ) {
        foreach ($redis->getUnprocessedEvents() as $event) {
            match ($event['type']) {
                Events::PRODUCT_CREATED =>
                    $createProduct->execute(
                        ProductData::fromArray($event['data'])
                    ),
                default => null
            };

            $redis->addProcessedEvent($event);
        }
    }
}

```

App\Console\Command\RedisConsumeCommand

First, we retrieve all unprocessed events by calling *getUnprocessedEvents()*, then we iterate through them, matching the type, and if it's a product:created, then we call the *CreateProductAction*. Finally, we add the event to the processed events set.

In the *CreateProductAction* class nothing magical happens, we only insert the Product. There is one important thing. Products are created in the Product service, which will assign a unique ID. We need a way to identify products across multiple services. So, if the products table in the Ratings service has an auto-increment primary key as well as in the Product service then it will cause problems. Just imagine the user calls the POST `/products/12/rate` endpoint. What does 12 mean in this context? This is the ID in the Ratings service? Or the ID in the Products service?

It's important that IDs need to come from one and only service. And these are global IDs, so if we call any endpoint in the system, or publish an event that takes a product ID, we need to make sure, that every service has this ID. We accomplish this by copying the ID from the event and storing it in our table:


```

class CreateProductAction
{
    public function execute(ProductData $data): Product
    {
        return Product::create([
            'id' => $data->id,
            'name' => $data->name,
        ]);
    }
}

```

App\Actions\CreateProductAction

So, in the Ratings service, we don't generate IDs, we just copy them from the event. This way, the ID 12 means that the Products service has a product with an ID of 12, and every other service will have the same ID in its database. In a real-world example, it's best to use UUIDs instead of integers.

Now let's see what our rating API looks like. Our API endpoint will be *POST /products/<id>/ratings*. We expect a *rating* and a *comment* key in the body. Rating contains a number between 1 and 5. Here's the Controller:

```
public function store(  
    Product $product,  
    RateProductRequest $request,  
    RateProductAction $rateProduct,  
) {  
    $productRatingData = $rateProduct->execute(  
        $product,  
        $request->getRating(),  
        $request->getComment(),  
    );  
  
    return response([  
        'data' => $productRatingData  
    ], Response::HTTP_CREATED);  
}
```

App\Http\Controllers\RatingController

The *RateProductAction* looks like this:

```
class RateProductAction
{
    public function __construct(private readonly RedisService $redis)
    {
    }

    public function execute(
        Product $product,
        float $rating,
        ?string $comment
    ): ProductRatingData {
        return DB::transaction(function () use ($product, $rating, $comment)
        {
            $averageRating = $this->rate($product, $rating, $comment);

            $data = new ProductRatingData(
                $product->id,
                $rating,
                round($averageRating, 2),
            );

            $this->redis->publishProductRated($data);

            return $data;
        });
    }
}
```

App\Actions\RateProductAction

Step by step:

- It stores the rating and calculates the new average rating of the product
- It creates a new *ProductRatingData* DTO
- It publishes a *product:rated* event with the DTO
- Then it returns the DTO

This is how it stores and calculates the average rating:

```
private function rate(
    Product $product,
    float $rating,
    ?string $comment
): float {
    ProductRating::create([
        'product_id' => $product->id,
        'rating' => $rating,
        'comment' => $comment,
    ]);

    return $this->updateAverageRating($product);
}

private function updateAverageRating(Product $product): float
{
    $product->average_rating = $product->ratings->avg('rating');

    $product->save();

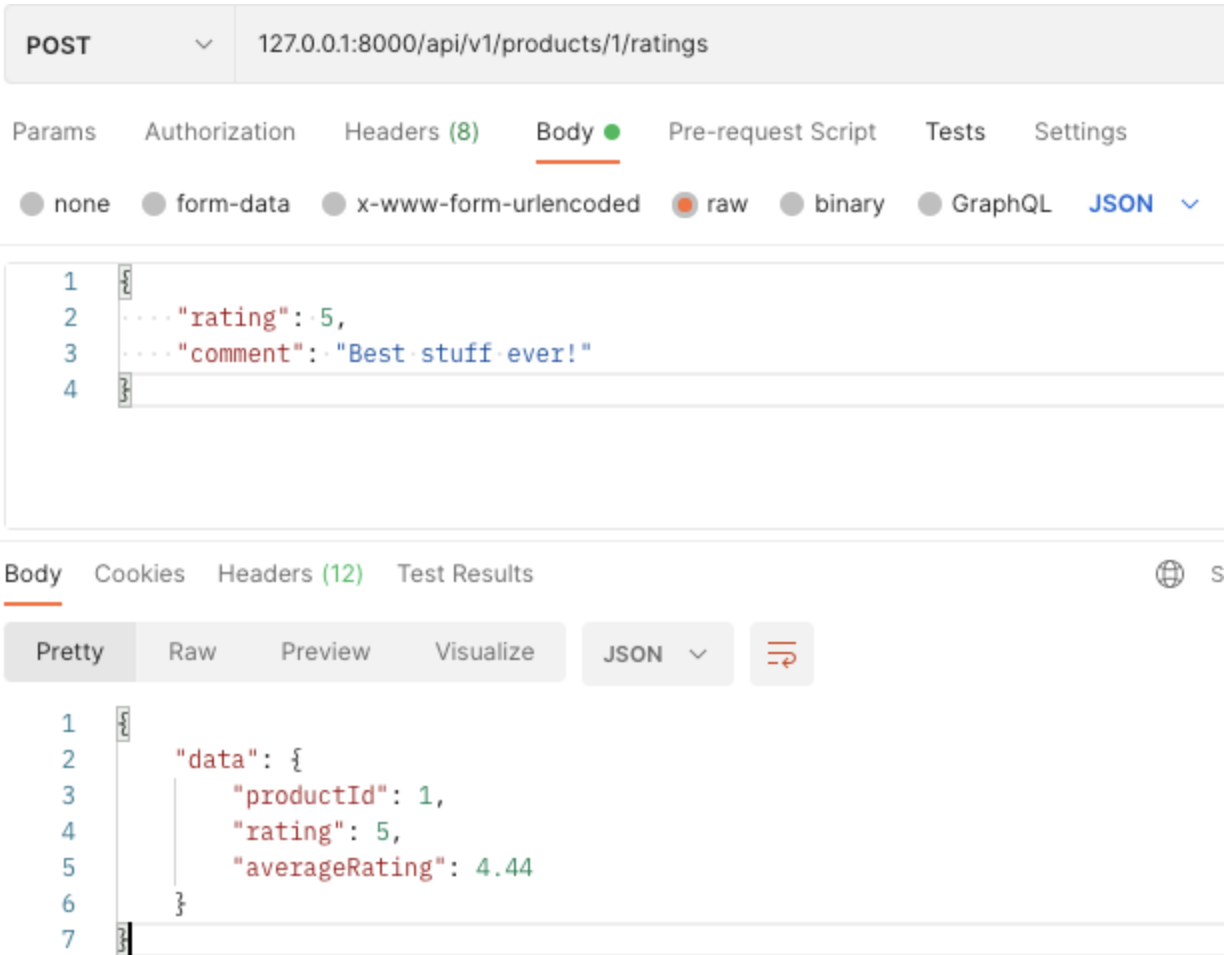
    return $product->average_rating;
}
```

App\Actions\RateProductAction

It creates a new record into the *product_ratings* table, then it calculate the average ratings based on these records and save it in the *products* table.

In the *RateProductAction* class, after we are done with database operations, we are instantiating a *ProductRatingContainer*, which is quite simple. This time we don't have a *toData()* method on the model, because we need additional information which we don't have in the *Product* model, so we are doing it in the Action.

After everything is done, we are publishing a *ProductRatedEvent* with a *ProductRatingData*. Now we can call the new API endpoint:



Rating API in Postman

If everything's fine, we have an event in Redis:

```
event: {  
  "type": "product:rated:v1",  
  "data": {  
    "productId":1,  
    "rating":5,  
    "averageRating":3.33  
  }  
},  
service: ratings,  
createdAt: 2021-10-10 12:02:06
```

The last thing we need to do is to run the *RedisConsumeCommand* regularly. We don't need any library (like supervisor), docker-compose is perfectly capable of doing things like that. We only need to add two properties to the container's config:

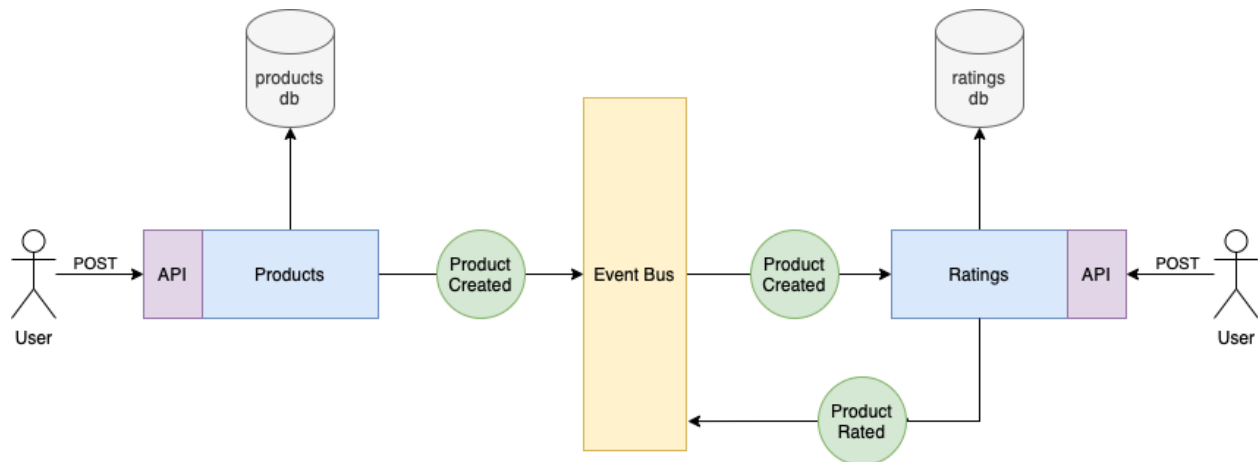
```
ratings-consumer:
  entrypoint: sh -c "sleep 10 && php
/usr/local/src/ratings/artisan redis:consume"
  restart: always
```

docker-compose.yml

Later, I will talk about Docker containers and docker-compose in more details.

I added an entry point which is a basic shell script. I also specified the restart policy to *always*. It means that docker-compose will spin up the container, it sleeps for 10 seconds, runs the `redis:consume` command, exits with 0 status code, and then it restarts the container. You can also achieve this with a while loop in a shell script, but I found this version a bit more clear. The only downside of this solution is that the container is always stopping and restarting, so if you need to run a *docker-compose exec*, for example, it will be very annoying because the container will just stop after 10 + x seconds, where x is the time needed by `redis:consume`.

So far we have these components in our application:



To sum it up:

- The user creates a product via API
- The products service inserts it into the database and publishes a ProductCreated event
- The ratings service consumes the event and creates its own version of the product
- The user rates a product via API
- The ratings service inserts it into the database and publishes a ProductRated event

Warehouse service

Now we are ready to build our next service, the Warehouse. It's gonna be very similar to the Ratings service. There are only three models in this one:

- Product
- Warehouse
- Inventory

The products table stores all the products that come from Redis. In the warehouses, we only need a name for the warehouse. Finally, the inventories table is a pivot table that connects products and warehouses. It looks like this:

id	product_id	warehouse_id	quantity	created_at	updated_at
1	53	4	57.00	2021-10-12 17:58:31 ↕	2021-10-12 17:58:31 ↕
2	54	5	46.00	2021-10-12 17:59:12 ↕	2021-10-12 17:59:12 ↕
3	55	6	34.00	2021-10-12 18:00:01 ↕	2021-10-12 18:00:01 ↕
4	1	1	11.00	2021-10-12 18:23:14 ↕	2021-10-12 18:24:00 ↕
5	1	2	89.00	2021-10-12 18:45:23 ↕	2021-10-12 19:10:41 ↕

Inventories table

We keep track of how many of a given product we have in stock. We will have only one API endpoint that inserts or updates a row in the inventories table.

Let's start with the *RedisConsumeCommand*:


```

class RedisConsumeCommand extends Command
{
    protected $signature = 'redis:consume';
    protected $description = 'Consume events from Redis stream';

    public function handle(
        RedisService $redis,
        CreateProductAction $createProduct,
        DecreaseInventoryAction $decreaseInventory,
    ) {
        foreach ($redis->getUnprocessedEvents() as $event) {
            match ($event['type']) {
                Events::PRODUCT_CREATED =>
                    $createProduct->execute(
                        ProductData::fromArray($event['data']),
                    ),
                Events::ORDER_CREATED =>
                    $decreaseInventory->execute(
                        OrderData::fromArray($event['data']),
                    ),
                default => null
            };

            $redis->addProcessedEvent($event);
        }
    }
}

```

App\Console\Commands\RedisConsumeService

It's only interested in the product:created and the order:created event. Here we use the ProductData::fromArray() method again, to create a DTO from an array, which comes from Redis as JSON. After we get the event, we just insert it into db:

```
class CreateProductAction
{
    public function execute(ProductData $data): Product
    {
        return Product::create([
            'id' => $data->id,
            'name' => $data->name,
        ]);
    }
}
```

App\Actions\CreateProductAction

The warehouse service only needs an ID and a name for the product. The next thing is the API. It's only one endpoint: *POST /v1/inventories*. You can send a product, warehouse, and quantity you want to add.

Here is the request for the API:

```

public function rules()
{
    return [
        'productId' => 'required|exists:products,id',
        'warehouseId' => 'required|exists:warehouses,id',
        'quantity' => 'required|numeric',
    ];
}

```

App\Http\Requests\StoreInventoryRequest

The Controller is fairly simple:

```

class InventoryController extends Controller
{
    public function store(
        StoreInventoryRequest $request,
        CreateInventoryAction $createInventory,
    ) {
        $inventoryData = $createInventory->execute(
            Product::findOrFail($request->getProductId()),
            Warehouse::findOrFail($request->getWarehouseId()),
            $request->getQuantity(),
        );

        return response([
            'data' => $inventoryData,
        ], Response::HTTP_CREATED);
    }
}

```

App\Http\Controllers\InventoryController

It queries the product and the warehouse, and calls the *CreateInventoryAction::execute* method which looks like this:

```

public function execute(
    Product $product,
    Warehouse $warehouse,
    float $quantity
): InventoryData {
    Inventory::create([
        'product_id' => $product->id,
        'warehouse_id' => $warehouse->id,
        'quantity' => $quantity,
    ]);

    $totalQuantity = Inventory::totalQuantity($product);

    $inventoryData = new InventoryData(
        $product->id,
        $totalQuantity,
    );

    $this->redisService->publishInventoryUpdated(
        $inventoryData
    );

    return $inventoryData;
}

```

App\Actions\CreateInventoryAction

After the inventory has been created, we have to publish an event. This information will be needed in the catalog service. When a user is browsing the products we want to show the available quantity. But we don't want to show it by warehouse, we only need the total quantity of a product. So, in the event, we need the total quantity of a product in all warehouses, not just the quantity from the POST request. This is why we call the *Inventory::totalQuantity* method:

```
namespace App\Builders;

use Illuminate\Database\Eloquent\Builder;

class InventoryBuilder extends Builder
{
    public function totalQuantity(Product $product): float
    {
        return Inventory::select('quantity')
            ->where('product_id', $product->id)
            ->sum('quantity');
    }
}
```

The query is pretty straightforward. The more interesting thing is that it's a custom query builder class. In Laravel, we can override the base Builder class for each model. This is a class where we can write our database queries. This way, your models remain thin and you have a dedicated class for your queries. After we defined the *InventoryBuilder* we need to tell Laravel that every time we do an *Inventory::something()* invocation, we want to use our custom query builder class.

This is done by overriding the *newEloquentBuilder* method in the *Inventory model*:

```

namespace App\Models;

use App\Builders\InventoryBuilder;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Inventory extends Model
{
    public function newEloquentBuilder($query)
    {
        return new InventoryBuilder($query);
    }
}

```

If you'd like to learn more about custom query builders, check out my blog post [here](#).

After the inventory has been created and we calculated the total quantity we need to instantiate an *InventoryData* DTO that looks like this:

```

use Ecommerce\Common\DataTransferObjects\warehouse;

class InventoryData
{
    public function __construct(
        public readonly int $productId,
        public readonly float $quantity,
    ) {}

    public static function fromArray(array $data): self
    {
        return new static(
            $data['productId'],
            $data['quantity'],
        );
    }
}

```

```
}
```

Ecommerce\Common\DataTransferObjects\Warehouse\InventoryData

This DTO can be found in the *common* package.

And finally, we can publish the event:

```
public function publishInventoryUpdated(  
    InventoryData $data  
): void {  
    $this->publish(new InventoryUpdatedEvent(  
        $data  
    ));  
}
```

App\Services\RedisService

The event looks like this in Redis:

```
event: {  
  "type":  
    "inventory:updated:v1",  
  "data": {  
    "productId":1,  
    "quantity":100  
  }  
},  
service: warehouse,  
createdAt: 2021-10-12 19:10:41
```

As I said earlier the *inventory:updated* event will be consumed by the Catalog service, which we don't have yet.

We need one more business logic to be implemented in the Warehouse service. When a user buys a product we need to decrease the available quantity. This will happen via a Redis event. The user clicks a button, the Frontend calls the Order service and it publishes an OrderCreated event with the product and quantity. The Warehouse service will listen to this event and decrease the quantity. Let's see how it's done:

```

namespace App\Actions;

class DecreaseInventoryAction
{
    public function __construct(
        private readonly RedisService $redis
    ) {}

    public function execute(OrderData $orderData): void
    {
        $product = Product::findOrFail($orderData->productId);

        $this->decrease($product, $orderData->quantity);

        $totalQuantity = Inventory::totalQuantity($product);

        $inventoryData = new InventoryData(
            $product->id,
            $totalQuantity
        );

        $this->redis->publishInventoryUpdated($inventoryData);
    }
}

```

App\Services\InventoryService

You haven't seen the *OrderData* class yet (I will introduce it in the Order service). It only contains a *productId* and a *quantity*. So we fetch the product and call the *decrease* function from the same class (we'll discuss it in a minute). After that, we need the new total quantity, and we publish an *inventory:updated* event because after the decrease the inventory has changed.

Now let's see the *decrease* method:

```
private function decrease(
    Product $product,
    float $quantity
): void {
    if (Inventory::totalQuantity($product) < $quantity) {
        throw new ProductInventoryExceededException(
            "There is not enough $product->name in inventory"
        );
    }

    $quantityLeft = $quantity;

    foreach ($product->inventories as $inventory) {
        if ($inventory->quantity >= $quantityLeft) {
            $inventory->quantity -= $quantityLeft;
            $inventory->save();

            $this->deleteInventoryIfEmpty($inventory);

            break;
        }

        $quantityLeft -= $inventory->quantity;

        $inventory->delete();
    }
}

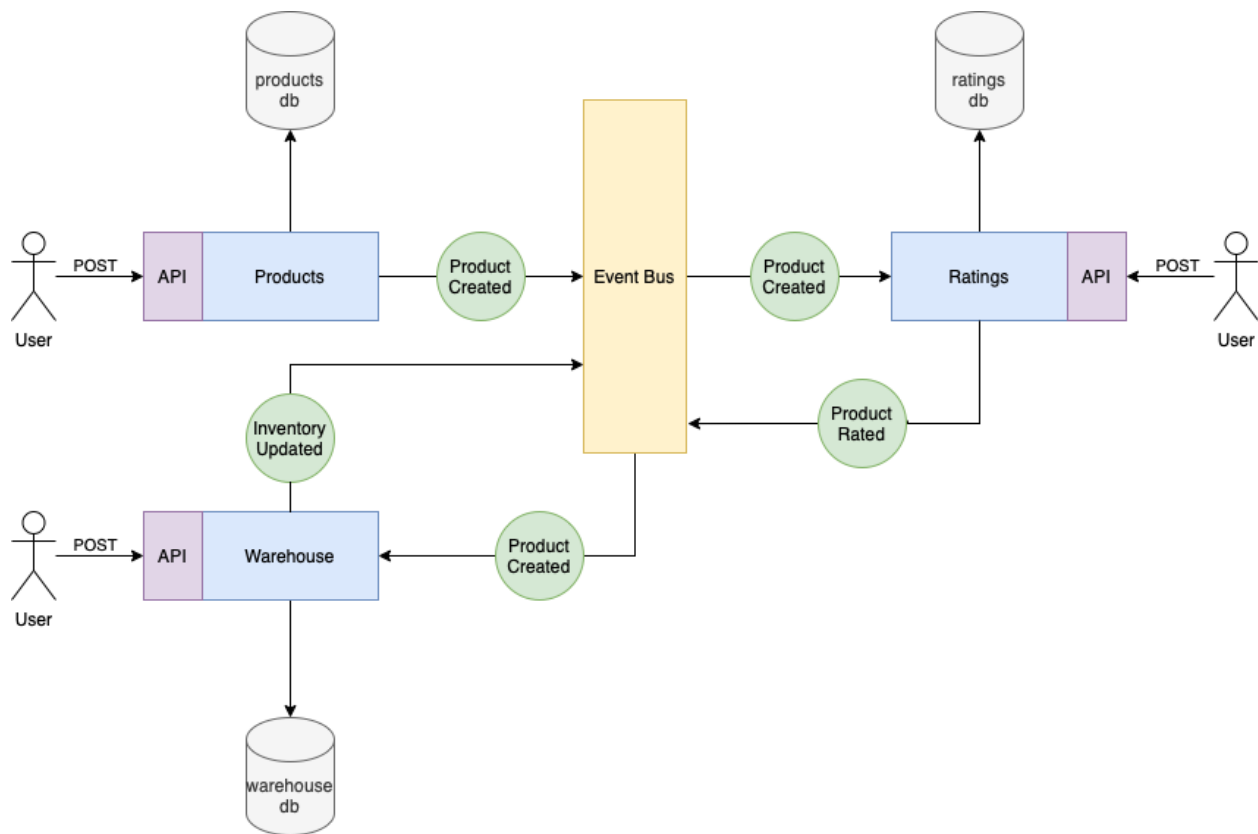
private function deleteInventoryIfEmpty(
    Inventory $inventory
): void {
    if ($inventory->quantity === 0.0) {
        $inventory->delete();
    }
}
```

App\Actions\DecreaseInventoryAction

First, it checks the total available quantity. If it's less than the *\$quantity*, it throws an exception.

After that, we can decrease the quantity. This is a naive algorithm, we don't specify from which warehouse we want to take out the product, and so on. The goal of this book is not to write advanced logistics, shipping, and warehouse business logic, but to teach you how microservices work. So we simply iterate through all the rows in the *inventories* table and simply subtract the appropriate quantity, so we don't create records with negative quantity values. Note that it deletes every row with 0 quantity. I created a helper called *deleteInventoryIfEmpty* to avoid nested if statements.

So far our e-commerce application looks like this:



The flow of events between 3 services

Catalog service

Finally, we get to our most “user-facing” service, the Catalog service. This is the API that's behind the homepage.

But first, we need to talk about data replication. On the catalog page users will see:

- Product name
 - This data stored in the Products service
- Product price
 - This data stored in the Products service
- Average ratings for the product
 - This data stored in the Ratings service
- Available quantity
 - This data stored in the Warehouse service
 - The quantity will not be shown on the catalog page, but we filter out products that are not available at the moment, so technically we need that information.

As you can see, this page needs data from at least three services. In this situation, we have at least three options:

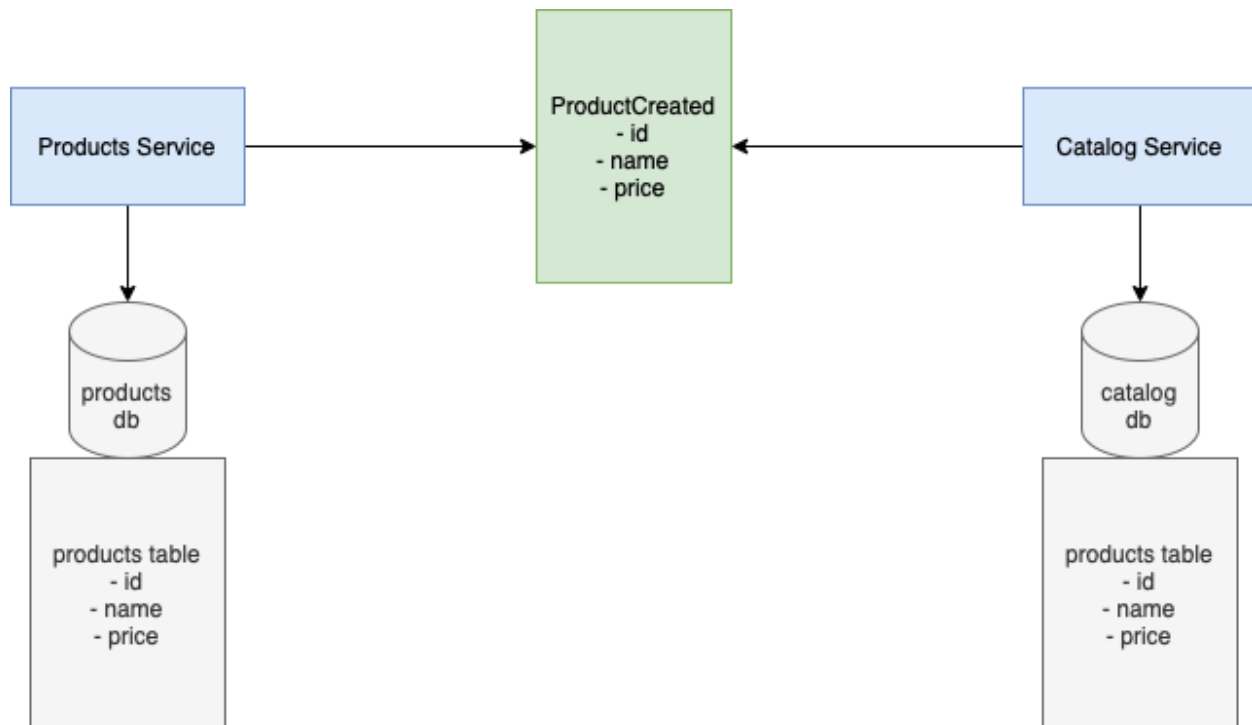
- No Catalog service at all. The frontend will simply call the 3 different services.
- “Aggregate” catalog service, or the event-driven approach. The catalog service has its own database and we use Redis to insert and update the data.
- “Proxy” catalog service. The catalog service does not have a database. Instead, it will call the other three services via HTTP, and act as a proxy to the frontend.

Let's see what are these options, and what are their benefits and drawbacks.

Aggregate Catalog service

This is the same story that we have done before. What's so special about the catalog service? Why don't we go with the default option? Here's what is different about the catalog service: it has no real functionality. All it has to do is to collect events, create a database from them, and serve it via an API. There's no real added value here. It just collects data from different sources and stores it in a database. And this can cause problems.

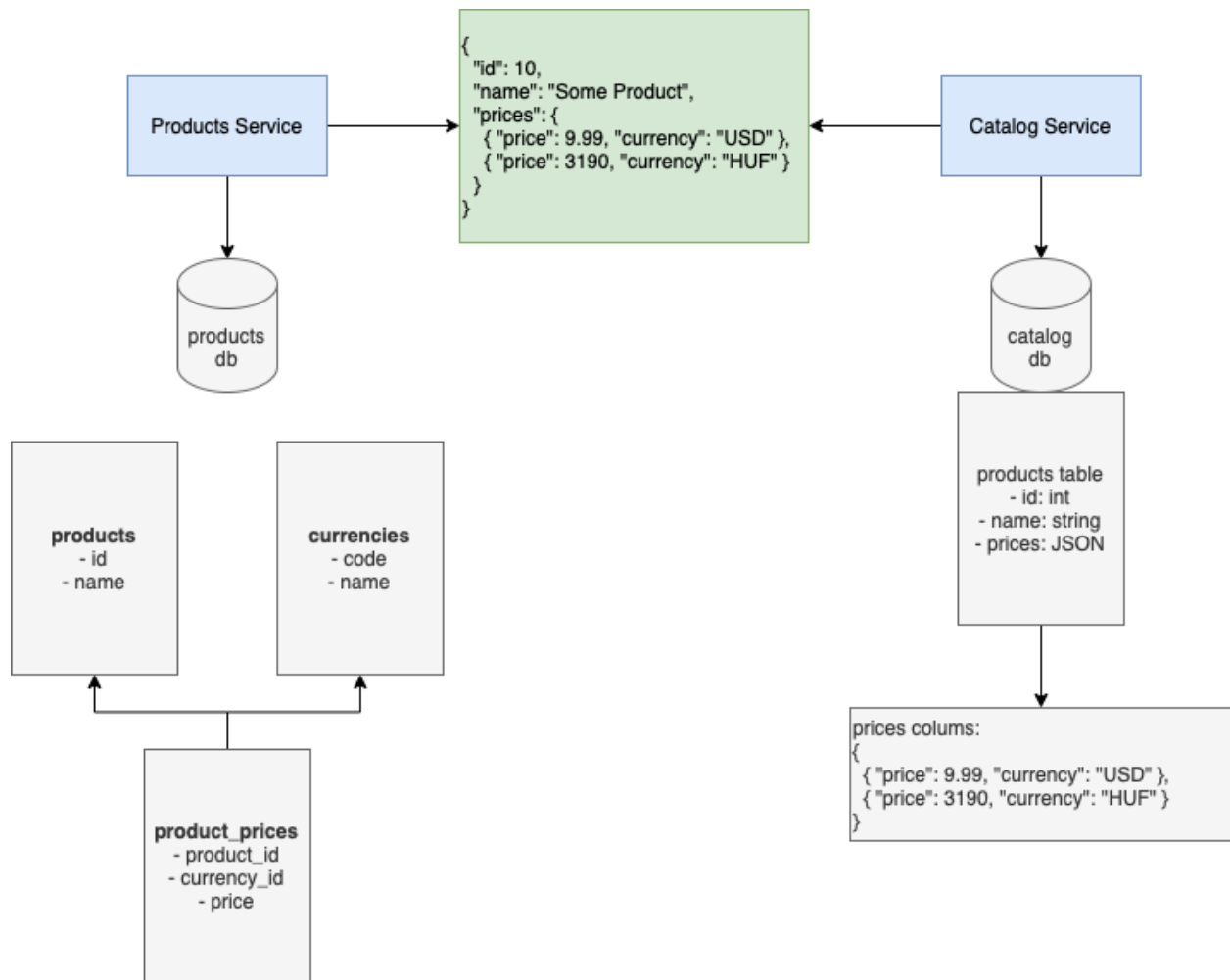
The Catalog service collects products and prices. What happens if we need to deal with currencies? We change the Product service and publish new events with currencies. Now, all of a sudden we need to reflect these changes in the Catalog service as well. Maybe we need to introduce the *currencies* and the *products_currencies* tables and write some logic to get the right currency for the catalog. **We are forced to change two services together.** Of course, there are always some situations when you have to change multiple services because of a single feature. But in this case, we have a tight coupling between the Product and the Catalog service. Let me illustrate the problem:



We are replicating the products table from the Products service. Let's imagine how the application works with currencies:

- Admins will manage the prices and currencies via an admin interface and we store them in the Products service
- The frontend will know the currency associated with the user. Maybe from the preferences, or the location.
- The frontend will send a request to the Catalog service: *GET /catalog?currency=USD*
- The Catalog service will query the price for that currency

Here's what it looks like:



As you can see the Product service has changed. It has two new tables and publishes a different Event. Please keep in mind that this is a very naive implementation of currencies, it can be *much* more complicated in the real world. Because in the Catalog service we don't want to manipulate prices or products we can store them in JSON format, which is quite convenient. But the important thing here is, that we need to change:

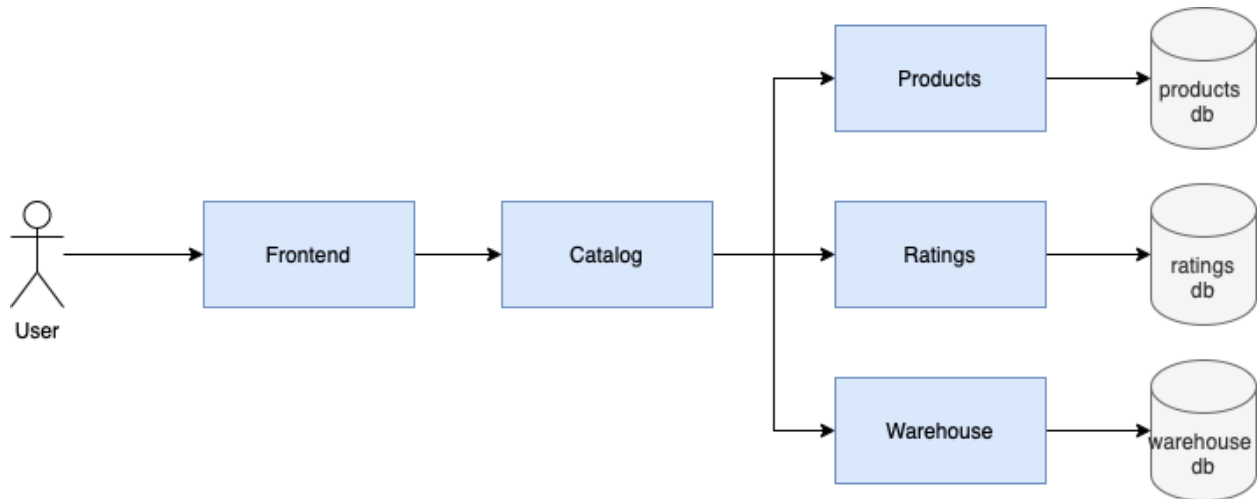
- The underlying database table
- The models associated with it
- Most of the queries
- The HTTP resources
- As a bonus, any class that used the old price column will be broken

What if the JSON column will not work? Now we need to introduce new tables and models in the Catalog service. This is tight coupling and low cohesion. The same old story as it was with monoliths. In fact, it can be much uglier.

I think we can say that this is not the best option, so let's continue with the other ones.

Proxy Catalog service

First of all, proxy in this context does mean the classic Nginx proxy or anything like that. What I mean is this:



So in this case the Catalog service doesn't have a database. It just sends requests to other services via HTTP and sends back the result to the Frontend. Earlier in this book, I told you not to do this. That's still true, but this is an exception. In this context, the Catalog service behaves like a facade, proxy, or entry point which makes it easier to get all the data that the Frontend needs. If there is no Catalog service then the Frontend needs to call three different APIs to get the catalog page. We will talk about it a bit later. So the "proxy" catalog service will call the other services, puts the data together, and immediately responds to the Frontend. Let's write some fake code to imitate the Catalog service:

```
public function getCatalog()
{
    $ids = $this->warehouseService->getAvailableProductIds();
    $products = $this->productService->getProducts($ids);
    $ratings = $this->ratingService->getRatings($ids);

    return $this->createResponse($products, $ratings);
}
```

This is just an imaginary method in an imaginary class. In this context, the *warehouseService*, *productService*, and the *ratingService* properties are some service classes, which send HTTP requests to our microservices. First, we ask the

Warehouse service to provide us with a list of the available product IDs. After that, we fetch the actual products, and finally, we request the ratings for those products.

The *createResponse* method is creating a response that may look like this:

```
{
  "products": [
    {
      "id": 1,
      "name": "First Product",
      "price": 9.99,
      "averageRating": 4.3,
      "numberOfRatings": 1273,
      "availableQuantity": 178
    }
  ]
}
```

So it merges all the data into one JSON. Now let's introduce currencies. How does it affect our imaginary class? Maybe now we have something like this:

```
public function getCatalog()
{
    $ids = $this->warehouseService->getAvailableProductIds();
    $products = $this->productService->getProducts($ids);
    $ratings = $this->ratingService->getRatings($ids);

    return $this->createResponse($products, $ratings);
}
```

Yes, nothing changed. The only thing that will change is the structure of the *\$products* collection. So in the worst case, we need to change the object which represents a product (the return value of *getProducts*) and the *createResponse* method, which will return a JSON like this:

```

{
  "products": [
    {
      "id": 1,
      "name": "First Product",
      "averageRating": 4.3,
      "numberOfRatings": 1273,
      "availableQuantity": 178,
      "prices": [
        { "price": 9.99, "currency": "USD" },
        { "price": 3190, "currency": "HUF" }
      ]
    }
  ]
}

```

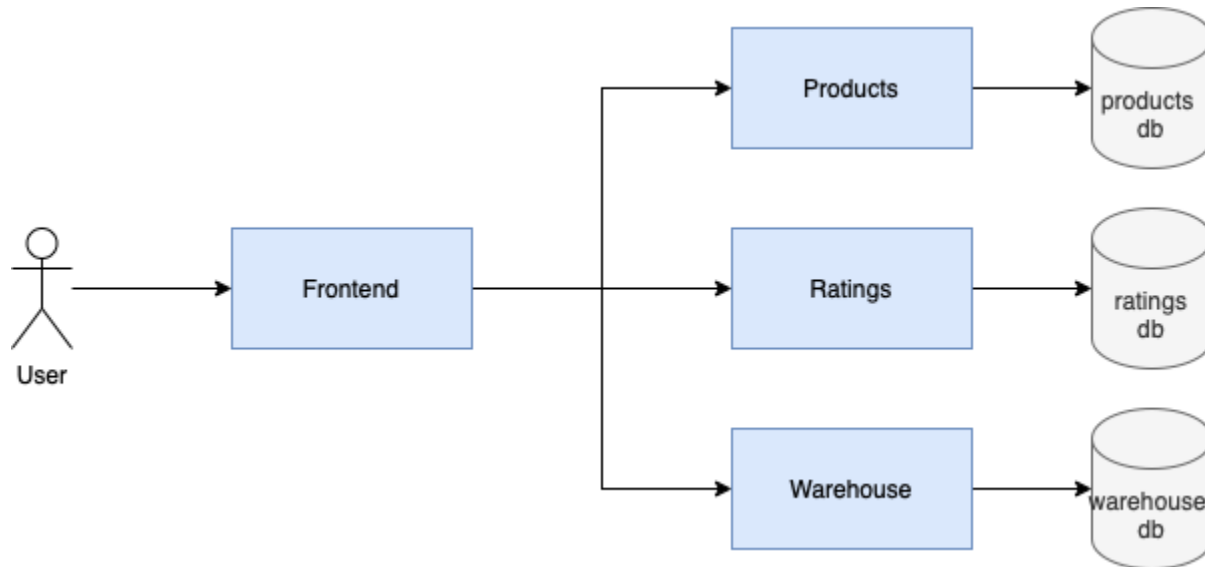
Because we don't have a database we only need to change some DTO or Resource objects to reflect the new structure. No models, actions, services, repositories, migrations, or event consumers to change.

I think that's a much less painful solution, but of course, it doesn't come without a cost:

- Now we have three times as many HTTP requests as before. The loading speed of the catalog page will be slower for sure. We can reduce this by using Redis as a cache. But it introduces more complexity.
- Now we can have circular dependencies or a "web of requests" in our application. Yes, it's unlikely that any other service will call the Catalog service (in fact it should not happen), but it's possible now.
- It's an extra cognitive load. Developers need to know that the Catalog service is different from the other services. They need to jump into multiple services if they want to debug something on the catalog page, etc.

No Catalog service at all

In this case, we just get rid of the Catalog service:



This can be a good solution in a lot of cases, but in this situation, I think that's a little bit difficult:

- Now all the work is done by the Frontend. By work, I mean, querying the warehouse for available products, then merging together products and ratings. I feel like it's just too messy to do it on the Frontend.
- What about mobile apps? If the project requires native mobile apps, this solution is not an option, because mobile developers need to implement the same logic in multiple applications.
- No single source of truth for the Frontend. It has to request data from all over the place.

The important thing here is one single page on the Frontend can call multiple services in the background. So it's not a one-to-one relationship between pages and services, but in this case, some filtering, processing, and merging need to be done, so it's a good idea to outsource this work to a separate service.

The main benefit of this solution is that you don't have to write a new service.

How to choose?

To be honest, I don't have a good answer. It depends on the situation. I think if you are not sure, you can start small. So first, you don't introduce a new service. The Frontend will call three services, and put the data together. If it works, good for

you. After that, you can evaluate the situation based on the current requirements, maybe you need a separate service, maybe you don't.

In this book, we will choose the second option. We will implement a proxy Catalog service. Why?

- Because it's a different technique, so it's a good thing if you see a service like that.
- I do think that it's a good solution for the current situation.
- As a bonus, it's gonna be relatively easy to refactor a service like that if you need to.

Implementation

We know that the Catalog service needs to access other services via HTTP API, so first let's make it happen. In the docker-compose file, we set up some new environment variables:

```
environment:
  - SERVICES_PRODUCTS_URL=http://products-api/api/v1/
  - SERVICES_WAREHOUSE_URL=http://warehouse-api/api/v1/
  - SERVICES_RATINGS_URL=http://ratings-api/api/v1/
```

docker-compose.yml

I prefixed all of them with SERVICES_ because Laravel has a services.php in the config directory which can be used to store the config of external services:

```
return [
    'products' => [
        'url' => env('SERVICES_PRODUCTS_URL')
    ],
    'ratings' => [
        'url' => env('SERVICES_RATINGS_URL')
    ],
    'warehouse' => [
        'url' => env('SERVICES_WAREHOUSE_URL')
    ],
];
```

config/services.php

I created an array for all of them in case we need some extra information later. We will make some HTTP request so let's create a small HttpClient class that wraps a Guzzle client, and implements some default behavior:

```

class HttpClient
{
    private Client $httpClient;

    public function __construct(private string $baseUri)
    {
        $this->httpClient = new Client([
            'base_uri' => $baseUri
        ]);
    }

    public function get(
        string $uri,
        array $queryParams = []
    ): Collection {
        $content = $this->httpClient
            ->get($uri, ['query' => $queryParams])
            ->getBody()
            ->getContents();

        $data = collect(json_decode(
            $content,
            associative: true
        ));

        return collect(Arr::get($data, 'data', $data));
    }
}

```

App\Services\HttpClient

In the *baseUri* parameter, we can inject the URLs from the config file. So when we need an *HttpClient* in the *ProductService* class to interact with the Products service, we can inject the *services.products.url* value. This way we will have an *HttpClient* instance that only interacts with the Products service in the *ProductService* class.

In order to interact with our services, we need to implement a class for each one. They are gonna be service classes in the *App\Services* namespace. I know it's a little bit confusing, so let's clarify. When I say *ProductService*, it's a class in the *App\Services* namespace. And when I say the Products service I'm referring to the microservice we implemented a few chapters earlier.

Warehouse

First, let's see how to interact with the Warehouse service since this is the most easier:

```
class WarehouseService
{
    public function __construct(private readonly HttpClient
$httpClient)
    {
    }

    public function getAvailableInventories(
        Collection $products
    ): Collection {
        return $this->httpClient
            ->get('inventory/products', [
                'productIds' => $products->pluck('id')->toArray()
            ])
            ->map(fn (array $inventory) =>
                new InventoryData(...$inventory)
            );
    }
}
```

App\Services\WarehouseService

It's quite simple, we are using the *HttpClient* to make a GET request to the */inventory/products* endpoint that takes a *productIds* array in the query params. After

that, we are mapping the results to a collection of *InventoryData*. Here we take advantage of two features of PHP 8: array destructuring, and named arguments.

You can see in the constructor we are accepting an instance of the *HttpClient* class. We don't want to instantiate it manually, but unfortunately, the *HttpClient* takes a string parameter called *\$baseUrl*. Laravel service container cannot resolve scalar values for us, but we can help it a little bit. We just have to configure when we want an instance of the *WarehouseService*, and it needs an *HttpClient* instance, just give it a predefined value, with the correct *\$baseUrl*. We can do such magic in the *AppServiceProvider*:

```
class AppServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->when(WarehouseService::class)
            ->needs(HttpClient::class)
            ->give(fn () => new HttpClient(config('services.warehouse.url')));
    }

    public function boot()
    {
    }
}
```

App\Providers\AppServiceProvider

So we tell Laravel: when the *WarehouseService* class needs an *HttpClient* just give it this instance with the URL of the Warehouse service. And now dependency injection will work. We can inject the *WarehouseService* into any class.

Right now we are using the *GET inventory/products* API which doesn't exist yet, so let's implement it in the **Warehouse service**.

Warehouse service API

We need a new API that takes product IDs and returns the inventory for these products. We will use the GET `/inventory/products?productIds[]=1&productsIds[]=2` endpoint that will call the *ProductInventoryController* class:

```
class ProductInventoryController extends Controller
{
    public function index(GetProductInventoryRequest $request)
    {
        $products = Product::find($request->getProductIds());

        $inventories = Inventory::totalQuantities($products);

        return [
            'data' => $inventories,
        ];
    }

    public function get(Product $product)
    {
        return [
            'data' => new ProductResource($product),
        ];
    }
}
```

App\Http\Controllers\ProductInventoryController

The *Inventory::totalQuantities()* is a simple method that loops through the given products and calls the *Inventory::totalQuantity()* for each one:

```
class InventoryBuilder extends Builder
{
    public function totalQuantities(
        Collection $products
    ): Collection {
        return $products
            ->mapWithKeys(fn (Product $product) => [
                $product->id => self::totalQuantity($product),
            ])
            ->map(fn (float $quantity, int $productId) =>
                new InventoryData(
                    $productId,
                    $quantity,
                )
            )
            ->values();
    }
}
```

App\Builders\InventoryBuilder

It also maps the data into *InventoryData* objects so in the controller we can just simply return the result of this method.

The API looks like this:

GET

127.0.0.1:8002/api/v1/inventory/products?productIds[]=1&productIds[]=2

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	productIds[]	1			
<input checked="" type="checkbox"/>	productIds[]	2			
	Key	Value	Description		

Body

Cookies

Headers (10)

Test Results

Status: 200 OK

Time: 991 ms

Size: 375 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1  {
2    "data": [
3      {
4        "productId": 1,
5        "quantity": 30
6      },
7      {
8        "productId": 2,
9        "quantity": 10
10     }
11   ]
12 }
```

Ratings

Next, we move on to communication with the Ratings service. First, we register the appropriate URL in the *AppServiceProvider*:

```
class AppServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->when(WarehouseService::class)
            ->needs(HttpClient::class)
            ->give(fn () => new HttpClient(
                config('services.warehouse.url'))
            );

        $this->app->when(RatingService::class)
            ->needs(HttpClient::class)
            ->give(fn () => new HttpClient(
                config('services.ratings.url'))
            );
    }
}
```

App\Providers\AppServiceProvider

After that, we can create the *RatingService* class:

```
class RatingService
{
    public function __construct(
        private HttpClient $httpClient
    ) {}

    public function getRatings(
        Collection $products
    ): Collection {
        return $this->httpClient
            ->get('products/ratings', [
                'productId' => $products->pluck('id')->toArray()
            ])
            ->map(fn (array $rating) =>
                new RatingData(...$rating)
            );
    }
}
```

App\Services\RatingService

It sends a request with the product IDs and maps the result into RatingData DTOs.

Ratings service API

The API endpoint is

GET /products/ratings?productIds[]=1&productIds[]=2, similar to what we have in the Warehouse service.

The Controller:

```
class RatingController extends Controller
{
    public function index(GetRatingsRequest $request)
    {
        $products = Product::whereHasRatings(
            $request->getProductIds()
        )->get();

        return RatingResource::collection($products);
    }
}
```

App\Http\Controllers\RatingController

There's a custom query builder in this service as well. It's called *ProductBuilder* and has the *whereHasRatings* method:

```
class ProductBuilder extends Builder
{
    public function whereHasRatings(Collection $ids): self
    {
        return $this->with('ratings')
            ->whereIn('id', $ids);
    }
}
```

App\Builders\ProductBuilder

This function returns a *ProductBuilder* instance. It works the same way as Eloquent scopes. In fact, this is the original form of of scopes. This is why we need to call the *get()* method in the controller after the *whereHasRatings* call.

Here's how the API looks like:

GET

127.0.0.1:8001/api/v1/products/ratings?productIds[]=1&productIds[]=2

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	productIds[]	1			
<input checked="" type="checkbox"/>	productIds[]	2			
	Key	Value	Description		

Body

Cookies

Headers (10)

Test Results

Status: 200 OK Time: 932 ms Size: 429 B Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1  {
2    "data": [
3      {
4        "productId": 1,
5        "averageRating": 4.33,
6        "numberOfRatings": 3
7      },
8      {
9        "productId": 2,
10       "averageRating": null,
11       "numberOfRatings": 0
12     }
13   ]
14 }
```


Products

First, we register the appropriate URL in the *AppServiceProvider*:

```
public function register()
{
    $this->app->when(ProductService::class)
        ->needs(HttpClient::class)
        ->give(fn () => new HttpClient(
            config('services.products.url')));

    $this->app->when(WarehouseService::class)
        ->needs(HttpClient::class)
        ->give(fn () => new HttpClient(
            config('services.warehouse.url')));

    $this->app->when(RatingService::class)
        ->needs(HttpClient::class)
        ->give(fn () => new HttpClient(
            config('services.ratings.url')));
}
```

App\Providers\AppServiceProvider

After that we create the *ProductService*:

```
class ProductService
{
    public function __construct(
        private HttpClient $httpClient
    ) {}

    public function getProducts(
        CatalogSearchData $data
    ): Collection {
        return $this->httpClient
            ->get('products', $data->toArray())
            ->map(fn (array $product) =>
                ProductData::fromArray($product)
            );
    }
}
```

App\Services\ProductService

As you can see, it's almost identical to the previous services. It calls the products API then maps the result into DTOs. It has only one difference. It accepts a *CatalogSearchData* DTO as its argument.

On the catalog page users can search products and sort the results based on some criteria. These data is represented with the *CatalogSearchData* DTO:

```
class CatalogSearchData
{
    public function __construct(
        public ?string $sortBy,
        public ?string $sortDirection,
        public ?string $searchTerm
    ) {
    }

    public function toArray(): array
    {
        return get_object_vars($this);
    }
}
```

App\DataTransferObjects\CatalogSearchData

This object holds everything that comes from the Catalog request and relates to searching products. The *GET /catalog* API will have query parameters like:

GET /catalog?sortBy=name&sortDirection=asc&searchTerm=notebook

The *CatalogSerachData* class holds these values and we create this object in the Controller from the Request. The reason we are using a separate object is that we don't want to pass the Request object around services. The Request should be used only in the Controller.

The Controller looks like this:

```
public function index(GetCatalogRequest $request)
{
    $data = new CatalogSearchData(
        $request->getSortBy(),
        $request->getSortDirection(),
        $request->getSearchTerm()
    );

    return [
        'data' => $this->catalogService->getCatalog($data)
    ];
}
```

App\Http\Controllers\CatalogController

We basically copy the Request into our DataTransferObject and pass it to the Service class. We will come back to see what's inside the *CatalogService*, but first, let's take a look at the Products service API.

Products service API

The controller is pretty straightforward:

```
class ProductController extends Controller
{
    public function index(GetProductsRequest $request)
    {
        $products = Product::search(
            $request->getSearchTerm(),
            $request->getSortBy(),
            $request->getSortDirection(),
        );

        return [
            'data' => $products->map->toData(),
        ];
    }
}
```

App\Http\Controllers\ProductController

As you can see, the *Product* model has a *toData()* method which creates a *ProductData* DTO from a model. We already discussed this approach earlier in the book, so let's move on to the *Product::search()* method.

This service also has a custom query builder just as before in other services. It's the *ProductBuilder* and it has a *search* method:

```
class ProductBuilder extends Builder
{
  public function search(
    ?string $searchTerm = null,
    ?string $sortBy = 'name',
    ?string $sortDirection = 'asc',
  ): Collection {
    return Product::select('products.*')
      ->when($searchTerm, fn (Builder $query) =>
        $query
          ->leftJoin(
            'categories', 'categories.id', 'products.category_id',
          )
          ->where('products.name', 'LIKE', "%$searchTerm%")
          ->orWhere('products.description', 'LIKE', "%$searchTerm%")
          ->orWhere('categories.name', 'LIKE', "%$searchTerm%")
        )
      ->when($sortBy, fn (Builder $query) =>
        $query->orderBy("products.$sortBy", $sortDirection)
      )
      ->get();
  }
}
```

App\Builders\ProductBuilder

The *Builder::when* executes the callback only if the first parameter is truthy, so in our case not null. In the first *when* clause we join the categories then search with the *LIKE* operator. I know it's not the best option from a performance perspective, but in this project it's okay. A better approach would be to store the product name, description, and category name in a *search_text* field in the *products* table. In this way, you only need to look inside this one column.

The API looks like:

GET

127.0.0.1:8000/api/v1/products?sortBy=name&sortDirection=asc&searchTerm=laptop

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	sortBy	name			
<input checked="" type="checkbox"/>	sortDirection	asc			
<input checked="" type="checkbox"/>	searchTerm	laptop			
	Key	Value	Description		

Body

Cookies

Headers (10)

Test Results

Status: 200 OK Time: 852 ms Size: 420 B Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1  {
2    "data": [
3      {
4        "id": 3,
5        "name": "Laptop",
6        "description": "My product",
7        "price": 12.99,
8        "category": {
9          "id": 2,
10         "name": "Notebooks"
11       }
12     ]
13   }
14 }
```

Put everything together

Our last task is to put those service classes together in the Catalog service and create a GET /catalog API. After all, this is the API that the frontend will consume. Here's the request:

```
class GetCatalogRequest extends ApiFormRequest
{
    public function getSortBy(): ?string
    {
        return $this->input('sortBy');
    }

    public function getSortDirection(): ?string
    {
        return $this->input('sortDirection');
    }

    public function getSearchTerm(): ?string
    {
        return $this->input('searchTerm');
    }

    public function rules()
    {
        return [
            'sortBy' => 'required_with:sortDirection|string',
            'sortDirection' => 'required_with:sortBy|string|in:asc,desc',
            'searchTerm' => 'sometimes|required|string',
        ];
    }
}
```

App\Http\Requests\GetCatalogRequest

The Controller is simple as always:

```
public function index(GetCatalogRequest $request)
{
    $data = new CatalogSearchData(
        $request->getSortBy(),
        $request->getSortDirection(),
        $request->getSearchTerm()
    );

    return [
        'data' => $this->catalogService->getCatalog($data)
    ];
}
```

App\Http\Controllers\CatalogController

It creates a *CatalogSearchData* from the request (shown earlier) and calls the *CatalogService* which will glue everything together:

```

/**
 * @return Collection<CatalogData>
 */
public function getCatalog(CatalogSearchData $data): Collection
{
    $products = $this->productService->getProducts($data);

    if ($products->isEmpty()) {
        return collect([]);
    }

    $inventories = $this->warehouseService
        ->getAvailableInventories($products);

    $ratings = $this->ratingService->getRatings($products);
    $catalog = [];

    foreach ($products as $product) {
        $inventory = $inventories->firstWhere('productId', $product->id);
        $rating = $ratings->firstWhere('productId', $product->id);

        if (!$inventory || $inventory->quantity === 0.0) {
            continue;
        }

        $catalog[] = $this->createData(
            $product,
            $rating,
            $inventory
        );
    }

    return collect($catalog);
}

```

App\Services\CatalogService

First, it calls the Product API through *ProductService*. This gives us product IDs we can use in the other requests. If it returns an empty collection we don't have to do anything else, we return right away.

After that, it gets the inventories and ratings for each product. Finally, it goes through all the products and skips the ones without available inventory. Remember, we don't want to show products that the user cannot buy. At the end of the loop it creates a *CatalogData*:

```
class CatalogData
{
    public function __construct(
        public readonly int $id,
        public readonly string $name,
        public readonly string $description,
        public readonly float $price,
        public readonly ?float $averageRating,
        public readonly ?int $numberOfRatings,
        public readonly float $availableQuantity
    ) {}
}
```

App\DataTransferObjects\CatalogData

This DTO is the response from the *CatalogController*.

The result in Postman:

GET 127.0.0.1:8003/api/v1/catalog?sortBy=name&sortDirection=asc

Params ● Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

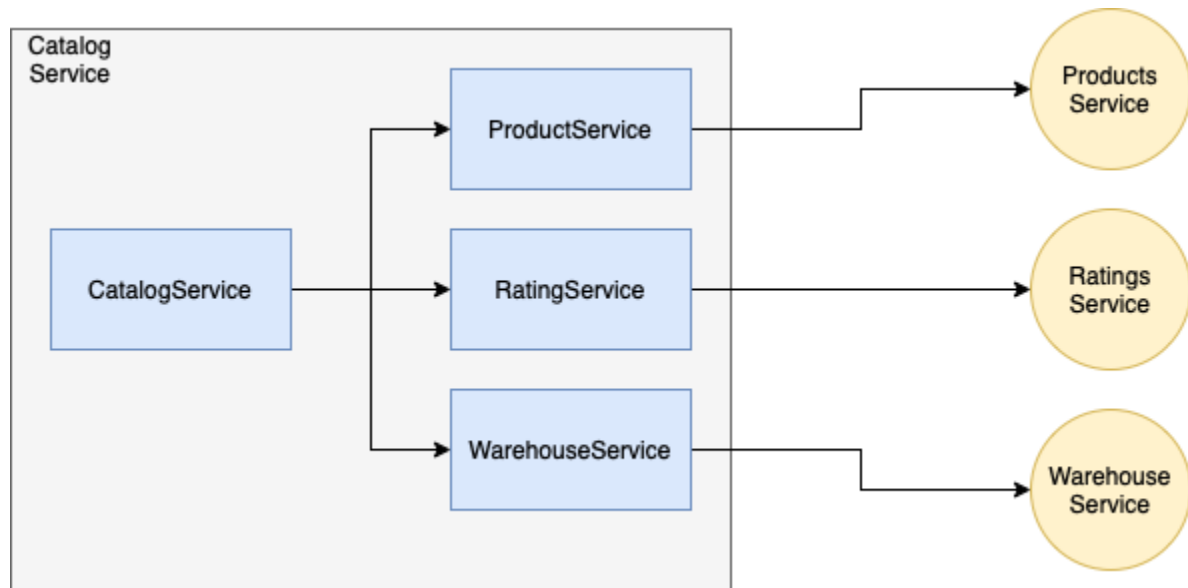
	KEY	VALUE
<input checked="" type="checkbox"/>	sortBy	name
<input checked="" type="checkbox"/>	sortDirection	asc

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1  {
2    "data": [
3      {
4        "numberOfRatings": 0,
5        "id": 4,
6        "name": "iPhone",
7        "price": 699,
8        "averageRating": null,
9        "availableQuantity": 10
10     },
11     {
12       "numberOfRatings": 2,
13       "id": 3,
14       "name": "Laptop",
15       "price": 12.99,
16       "averageRating": 2.5,
17       "availableQuantity": 20
18     },
```

In a nutshell, this is how the Catalog service works:



Orders service

Finally, we have arrived at our last service, the Orders service. This is a really simple one, with only one API endpoint, the *POST /orders*. We're not gonna implement some complex order handling logic, just the bare minimum. Users will be able to buy a product with a specified quantity. When they click the button the Frontend will call the *POST /orders* API, we create an order and decrease the inventory of the product. There is no cart functionality, so basically, one order contains only one product, similar to Amazon's one-click buy. This means we only need an *orders* table, something like this:

id	product_id	quantity	total_price	created_at	updated_at
1	1	2.00	58.00	2021-10-31 09:31:11 ↕	2021-10-31 09:31:11 ↕
2	1	3.00	87.00	2021-10-31 09:33:40 ↕	2021-10-31 09:33:40 ↕
3	1	6.00	174.00	2021-10-31 09:33:59 ↕	2021-10-31 09:33:59 ↕
4	1	1.00	29.00	2021-10-31 09:34:17 ↕	2021-10-31 09:34:17 ↕
5	1	5.00	145.00	2021-10-31 09:35:31 ↕	2021-10-31 09:35:31 ↕
6	1	1.00	29.00	2021-10-31 12:26:23 ↕	2021-10-31 12:26:23 ↕
7	1	1.00	29.00	2021-10-31 12:29:44 ↕	2021-10-31 12:29:44 ↕
8	1	2.00	58.00	2021-10-31 12:47:55 ↕	2021-10-31 12:47:55 ↕
9	1	3.00	87.00	2021-10-31 13:02:21 ↕	2021-10-31 13:02:21 ↕
10	1	3.00	87.00	2021-10-31 13:05:04 ↕	2021-10-31 13:05:04 ↕

orders table

We don't have an *order_items* table, because we don't implement a cart. Orders of course need products, so the service will listen to the ProductCreated event. That's the same logic as in previous services, so I won't bother you with it, you can see the implementation in *App\Console\Commands\RedisConsumeCommand*.

Let's see how the API is implemented:

```

class OrderController extends Controller
{
    public function store(
        StoreOrderRequest $request,
        CreateOrderAction $createOrder
    ) {
        try {
            $order = $createOrder->execute(
                Product::findOrFail($request->getProductId()),
                $request->getQuantity(),
            );

            return new OrderResource($order);
        } catch (ProductInventoryExceededException $ex) {
            return response([
                'errors' => ['quantity' => $ex->getMessage()]
            ], Response::HTTP_UNPROCESSABLE_ENTITY);
        }
    }
}

```

App\Http\Controllers\OrderController

It throws a *ProductInventoryExceededException* if the quantity of the order is greater than the available quantity. We check this in the *CreateOrderAction*:

```

class CreateOrderAction
{
    public function __construct(
        private readonly WarehouseService $warehouseService,
        private readonly RedisService $redis,
    ) {}

    public function execute(
        Product $product,
        float $quantity
    ): Order {
        $inventory = $this->warehouseService
            ->getTotalInventory($product);

        if ($quantity > $inventory) {
            throw new ProductInventoryExceededException(
                "There is not enough $product->name in
inventory"
            );
        }

        $order = Order::create([
            'product_id' => $product->id,
            'quantity' => $quantity,
            'total_price' => $product->price * $quantity,
        ]);

        $this->redis->publishOrderCreated($order->toData());

        return $order;
    }
}

```

App\Actions\CreateOrderAction

If the quantities are okay, we insert the order and publish an *order:created* event. We will talk about the *WarehouseService* in a minute, but first, let's take a look at the *OrderCreated* event and DTO (you can find them in the common package):

```
class OrderData
{
    public function __construct(
        public readonly int $productId,
        public readonly float $quantity,
        public readonly float $totalPrice,
    ) {}

    public static function fromArray(array $data): self
    {
        return new static(
            $data['productId'],
            $data['quantity'],
            $data['totalPrice'],
        );
    }
}
```

Ecommerce\Common\DataTransferObjects\Order\OrderData

It only stores the *productId*, the *quantity*, and the *totalPrice*, which is quantity * unit price. The event is even more simple:

```
class OrderCreatedEvent extends Event
{
    public string $type = Events::ORDER_CREATED;

    public function __construct(public OrderData $data)
    {
    }
}
```

Ecommerce\Common\Events\Order\OrderCreatedEvent

Now we can get back to the *WarehouseService* class which we call from the *CreateOrderAction* class:

```
class WarehouseService
{
    public function __construct(
        private Client $httpClient,
        private string $apiUrl
    ) {
    }

    public function getTotalInventory(Product $product): float
    {
        $content = $this->httpClient
            ->get($this->apiUrl . "inventory/products/$product->id")
            ->getBody()
            ->getContents();

        $data = json_decode($content, true);

        return (float) $data['data']['totalInventory'];
    }
}
```

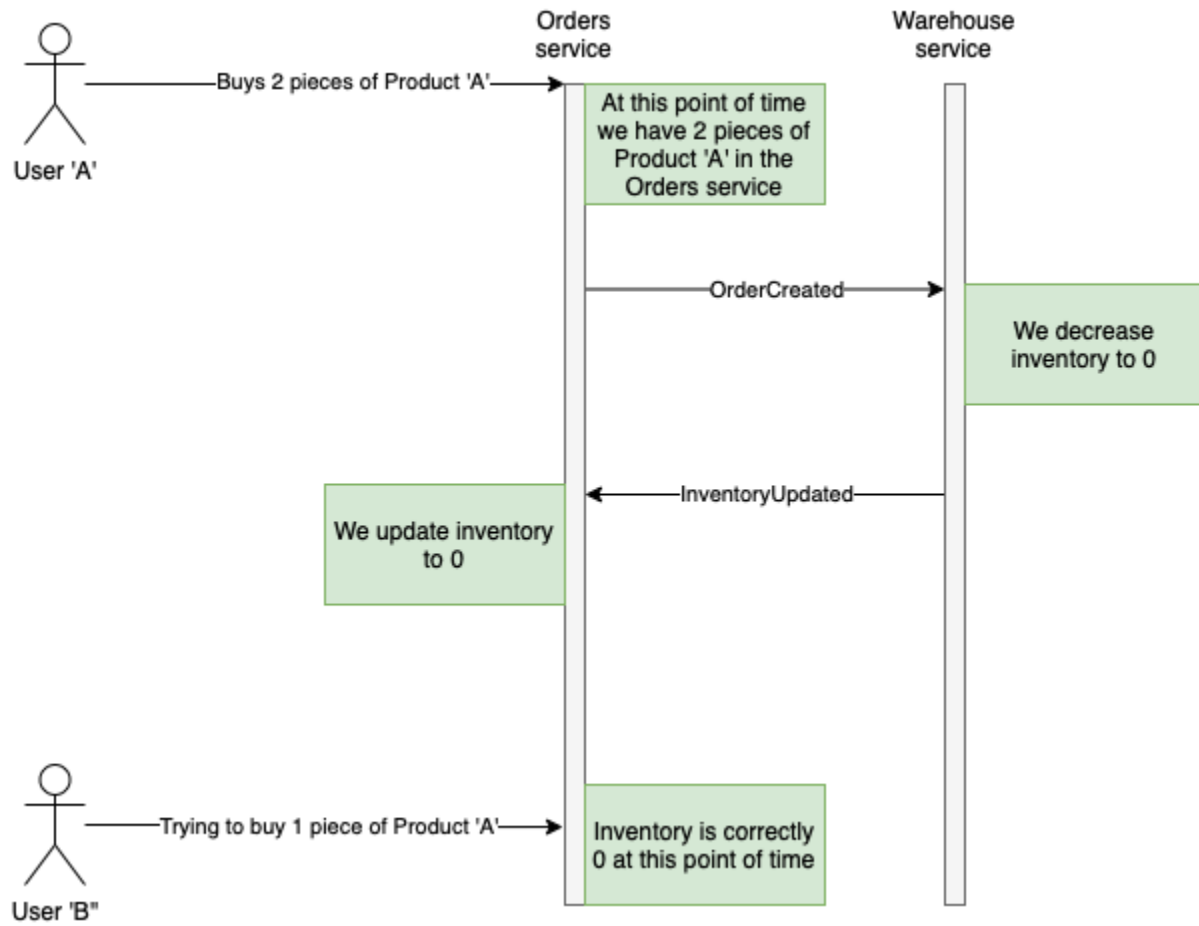
App\Services\WarehouseService

It calls the Warehouse service via HTTP. “But you told me it’s evil isn’t it?”. Yes and no. As I stated in the first part of this book, communicating only through HTTP APIs probably makes your life hard. But from time to time using this technique is perfectly fine. In this case, we call the Warehouse service from the Order service to get the total inventory for a product.

Why do we do this? Why doesn’t the Order service listen to the *InventoryUpdated* event, and store the inventory information in its database like before? The answer is too much knowledge. In real-world e-commerce or business applications, warehouse and inventory are one of the most important stuff to deal with. We cannot just spread this information across multiple services. Just think about the following situation:

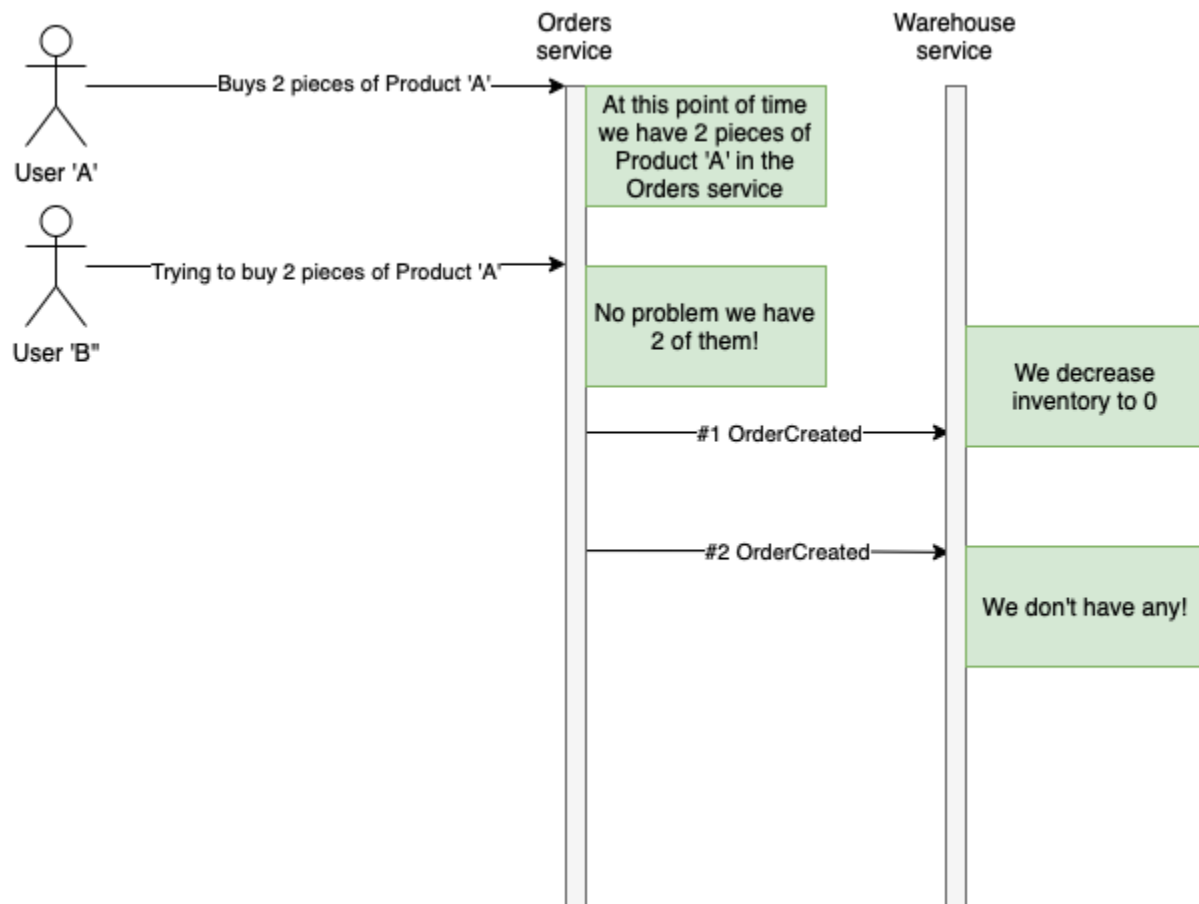
- We have two pieces of Product ‘A’ in Warehouse 1
- The Order service has this data
- User ‘A’ buys two pieces of Product ‘A’ through the Orders API
- 100 milliseconds later User ‘B’ buys 1 piece of Product ‘A’

What happens in this situation? We don't really know. We have two scenarios, let's visualize them:



This is the happy path

Now let's visualize the unhappy path:



This is the unhappy path

The basic problem is if User 'B' clicks the buy button **before** OrderCreated (and InventoryUpdated) events are processed we're in trouble because of the Order service thinks that Product 'A' is still available. But in fact, it's not anymore. And there are a lot of other edge-cases here. This is a common problem, async messaging has some concurrency issues.

And in this example, we only have two services that are interested in inventory. What if we have three other services? Like shipping, logistics, packaging, discounts? The situation gets worse with every new service. Of course, there are some workarounds for this problem, you can use some locking strategy, but it gets even more complicated. So the most simple and safe solution is to use sync communication.

Only some situations have this kind of concurrency issue, so you need to think about them carefully, and use sync communication if needed.

Resources vs DTOs

In this project, I'm using Resource classes and Container classes interchangeably. They serve a very similar purpose.

When you have an Event and a DTO for something, let's say a Product, then you can use the DTO in a GET /product/{product} API because the data will be the same (most of the time). On the other hand, if you don't have an Event or a DTO for something, but you need a GET API, you can write a simple Resource.

In other words:

- I use DTOs when the data needs to be transferred between services
- I use Resources when it's only used inside one service

I find it a good practice, but you can do things differently, for example:

- Always write resources for APIs. This is also a good practice, but you will have some "duplicated" classes. Not literally duplicated, because a Resource is different from a DTO, but very similar classes, with the same attributes. Probably you need to change both of them if a new requirement comes in.
- Never write Resources, always write DTOs. This is also a good practice because in this way you have an Event for everything. If you look inside the Warehouse service, there is a *POST /warehouses* but it doesn't publish any event. So right now we don't have the full history of our "data-flow" in the Redis stream. This is okay because there is no frontend component to create a warehouse. The API only exists because of testing purposes. You have only a few exceptions, like the *CatalogResource* in the Catalog service. You cannot write an Event because it's a GET operation, but still, you can easily have a *CatalogData* instead of a resource. So this is the reason I have DTOs and Resources as well in this project.

Docker Containers

If we want 5 services to communicate with each other and behave like one single application we obviously want to use Docker containers and Docker compose to run them. We will go through the main concepts in this chapter with the Product service as an example.

Dockerfile

First, we need a Dockerfile:

```
FROM php:8.1.8-fpm
WORKDIR /usr/local/src/products

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    libfreetype6-dev \
    libjpeg62-turbo-dev \
    libpng-dev \
    libpq-dev \
    libzip-dev && \
    apt-get install -y \
    git \
    unzip \
    zip \
    sqlite3 \
    openssh-client \
    default-mysql-client && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

RUN chown -R www-data:www-data .

RUN docker-php-ext-install pdo pdo_mysql sockets bcmath zip pcntl
posix
RUN pecl install redis
RUN pecl install xdebug
RUN docker-php-ext-enable xdebug
COPY . .
COPY ./docker/dev.ini /usr/local/etc/php/conf.d/dev.ini

COPY --from=composer:latest /usr/bin/composer /usr/bin/composer
RUN composer install
RUN chown -R www-data:www-data .
USER root
```

products/docker/Dockerfile

Let's take a look at each section.

```
FROM php:8.1.8-fpm
WORKDIR /usr/local/src/products
```

We need a PHP 8 base image, and we are using the FPM version. FPM is a FastCGI Process Manager tool. It's out of the scope of this book, but we will talk a bit more about it when we discuss Nginx.

After that, we set the working directory to */usr/local/src/products*. I like to use the */usr/local/src* directory, but you can use almost any folder, for example, */app*.

```
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    libfreetype6-dev \
    libjpeg62-turbo-dev \
    libpng-dev \
    libpq-dev \
    libzip-dev && \
    apt-get install -y \
    git \
    unzip \
    zip \
    sqlite3 \
    openssh-client \
    default-mysql-client && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

In the next section, we are installing Linux packages. This step installs mandatory dependencies to our system like git, zip, ssh, SQLite. The jpeg, png packages are needed for any image processing (like resizing an image after a user uploads it) logic, or PDF creation. In this service, we don't need anything like this, but I left it here because a lot of Laravel project needs these libs.


```
RUN chown -R www-data:www-data .
RUN docker-php-ext-install pdo pdo_mysql sockets bcmath zip
pcntl posix
RUN pecl install redis
RUN pecl install xdebug
RUN docker-php-ext-enable xdebug
COPY . .
COPY ./docker/dev.ini /usr/local/etc/php/conf.d/dev.ini
```

In the next few lines, we are installing PHP extensions. The *docker-php-ext-install* makes it possible to install the *extension.so* extensions and enables them in PHP ini. In the previous step, we only installed Linux packages, but we are not able to use them from PHP. In this step we are linking the two layers together, so now we are able to use PDO (used by Eloquent) from PHP.

After the extensions, we install Redis and XDebug. Redis is needed for the event streams. However, XDebug is not necessary so you can remove it if you want. It's obviously good for debugging PHP applications, and also makes code coverage generation for PHPUnit possible. So with this image, you are ready to write tests and generate code coverage reports out of the box.

After that, we are copying the source code from the host machine to the current working directory of the image. And we also copy the dev.ini configuration file, which is quite simple:

```
error_reporting = E_ALL
log_errors = On
error_log = /dev/stderr
xdebug.mode = develop,coverage

extension=redis.so
```

The important line is the Xdebug mode. This makes the code coverage possible. And we are also enabling Redis as a PHP extension. By the way, you can do this by running the *docker-php-ext-enable Redis* command as we do with Xdebug.

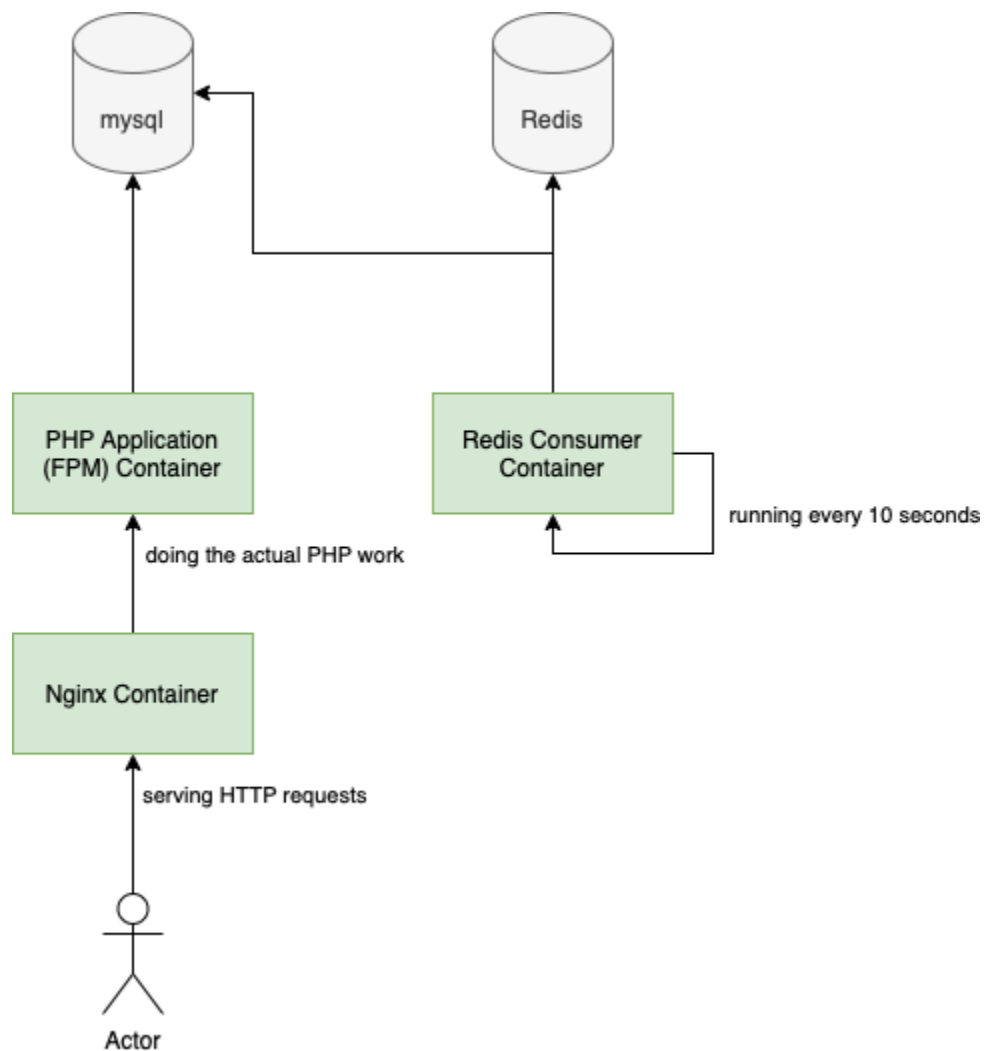
```
COPY --from=composer:latest /usr/bin/composer
/usr/bin/composer
RUN composer install
RUN chown -R www-data:www-data .
USER root
```

Finally, in the last few steps, we install Composer packages. The *COPY --from=composer:latest* command is copying the */usr/bin/composer* executable from the *composer:latest* image, and pasting it in the */usr/bin/composer* location in our image. By the way, using the latest tag is not a good practice.

The Dockerfiles of other services are the same. Now let's see what we need in the docker-compose.

Docker-compose

The SRP or single responsibility principle also applies to containers. This means that every container will run one main process. So if we have an API for the Ratings service, but also have a Redis consumer (which runs the *redis:consume* command every 10 seconds), then we need two containers. The first is only responsible for running an Nginx instance and listening to a port, the other is responsible for scheduling the *redis:consume* command. In fact, we need three containers. In the third one, we run our PHP application, the one with the Dockerfile from the previous section. Here's a picture:



The Nginx container will listen to a port and serve HTTP requests from the users. But it's just an Nginx process, it knows nothing about Laravel or PHP, so it will pass the request to our PHP container (the one with the Dockerfile from the previous section). This container contains our Laravel application and does the actual PHP work producing the response. The Redis consumer container will run the *redis:consume* command every X seconds and process the event log. The important thing here: the PHP and the Redis consumer container both contain the source code of the Products service. They both use our Dockerfile.

In the case of the Ratings service, for example, we will name these containers:

- ratings: PHP FPM container
- ratings-api: Nginx container
- ratings-consumer: PHP FPM container that schedules the *redis:consume* command

This is the docker-compose service for the ratings container:

```
ratings:
  build:
    context: ../ratings
    dockerfile: ../ratings/docker/Dockerfile
  volumes:
    - ../ratings:/usr/local/src/ratings
  environment:
    - APP_ENV=local
    - APP_KEY=base64:7t3iJGYKu1jqjeEQixGEL8mXCr6gVRmdRII3Jo=
    - APP_DEBUG=true
    - DB_CONNECTION=mysql
    - DB_HOST=mysql
    - DB_PORT=3306
    - DB_DATABASE=ratings
    - DB_USERNAME=root
    - DB_PASSWORD=root
    - REDIS_HOST=redis
    - REDIS_PASSWORD=null
    - REDIS_PORT=6379
    - REDIS_CLIENT=phpredis
    - REDIS_SCHEME=tcp
    - LOG_CHANNEL=stack
```

```
- LOG_LEVEL=debug
depends_on:
- mysql
```

ratings service in docker-compose.yml

This is a docker-compose file for development so we don't use images from some repository, we are building them from our Dockerfile. And we are also using volumes to make development easy.

In this project, we don't use .env files at all. I like to list all of the environment variables in my docker-compose files. In this way, **everything** is there. There's no need to copy files, ignore them from git, maintain a .env and a .env.example file, create a new one for CI jobs called .env.ci, create another one for testing called .env.testing, and so on. Everything is listed in the docker-compose file. If a new developer joins the team, he or she just needs to run docker-compose up, and that's it.

I won't list the complete docker-compose here, but you can see there's a Mysql container for the database. We have one instance of Mysql, and there are multiple databases in it for each service, like ratings. The same applies to Redis.

This service only depends on Mysql and not on Redis. The *redis:consume* command will run in another container, so the ratings container won't need it as a dependency.

The ratings-consumer is very similar:

```
ratings-consumer:
  build:
    context: ../ratings
    dockerfile: ../ratings/docker/Dockerfile
  entrypoint: sh -c "sleep 10 && php
/usr/local/src/ratings/artisan redis:consume"
  restart: unless-stopped
  volumes:
    - ../ratings:/usr/local/src/ratings
  environment:
    - APP_ENV=local
    - APP_KEY=base64:7t3JGKujqeEixGEL8mXwCr6XgVRmd6ryRII3Jo=
    - APP_DEBUG=true
    - DB_CONNECTION=mysql
    - DB_HOST=mysql
    - DB_PORT=3306
    - DB_DATABASE=ratings
    - DB_USERNAME=root
    - DB_PASSWORD=root
    - REDIS_HOST=redis
    - REDIS_PASSWORD=null
    - REDIS_PORT=6379
    - REDIS_CLIENT=phpredis
    - REDIS_SCHEME=tcp
    - LOG_CHANNEL=stack
    - LOG_LEVEL=debug
  depends_on:
    - mysql
    - redis
```

Ratings-consumer service in docker-compose.yml

We are building an image from the same Dockerfile, using the same environments. There are only three differences:

- It depends on Redis. You know, it's a Redis consumer.
- It has an entry point that does the scheduling. First, it sleeps for 10 seconds, then it runs the *redis:consume* command.
- It has a restart policy. This is necessary because the *redis:consume* command will run and then exit. It means the container will die unless we restart it.

And finally the Nginx container:

```
ratings-api:
  image: nginx:1.19-alpine
  volumes:
    - ../ratings:/usr/local/src/ratings
    - ../ratings/docker/nginx/conf.d:/etc/nginx/conf.d
  ports:
    - 8002:80
  depends_on:
    - ratings
```

ratings-api in docker-compose.yml

It uses an Nginx image and depends on the ratings container. The dependency is a must-have because it will pass requests to that container. You can see that in the Nginx config:

```
location ~ /\.php$ {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_pass products:9000;
    fastcgi_index index.php;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME
    $document_root$fastcgi_script_name;
    fastcgi_param PATH_INFO $fastcgi_path_info;
}
```

nginx config

I won't list the whole config here, you can find it in every service under the */docker/nginx* folder. The important line is the *fastcgi_pass*. The PHP FPM container

listens on the 9000 port by default. This is the PHP process, you can give it PHP scripts to execute. In the Nginx config, we say that every time a PHP file is requested, pass the job to the PHP FPM container. Every API request will translate to a PHP file:

```
location / {  
    try_files $uri $uri/ /index.php?$query_string;  
    gzip_static on;  
}
```

This location config tells Nginx that any request that comes to the / so the root location, transform it to *index.php?\$query_string*. Then the other location config will process it, and pass it to the PHP FPM container. In this container index.php is the entry point for Laravel, and it lives under the public folder. From that point, Laravel will take care of the rest, and run our application code.

If you make a request, you can see the process in the output of the docker-compose:

```
ratings-api: "GET /api/v1/products/ratings?productIds[]=1&productIds[]=2 HTTP/1.0"  
200 123 "-" "PostmanRuntime/7.28.4"  
ratings: "GET /index.php" 200
```

That's because we pass the request from Nginx to PHP FPM.

API gateway

Right now our application has 5 services:

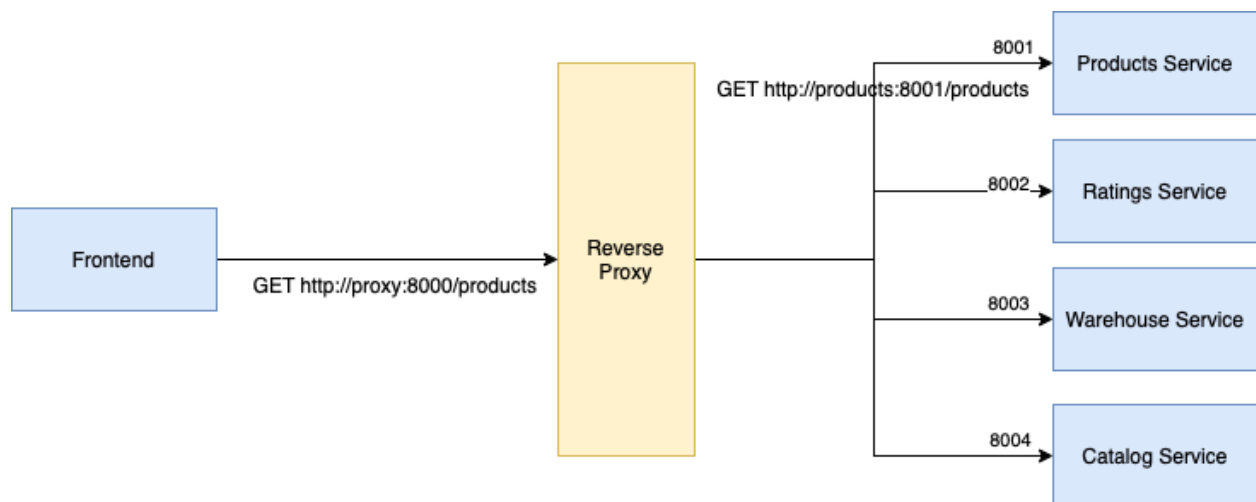
- Catalog
- Products
- Ratings
- Warehouse
- Orders

These services are using 5 different ports from 8001 through 8005. But as I mentioned earlier we want to publish all of the API via one port number for the frontend. We want the Frontend to be able to just use one domain name and one port number to communicate with all of the services. So we need some kind of gateway between the frontend and our backend services. There are two main kinds of solutions:

- Simple Nginx reverse proxy
- Some custom gateway implementation

Nginx reverse proxy

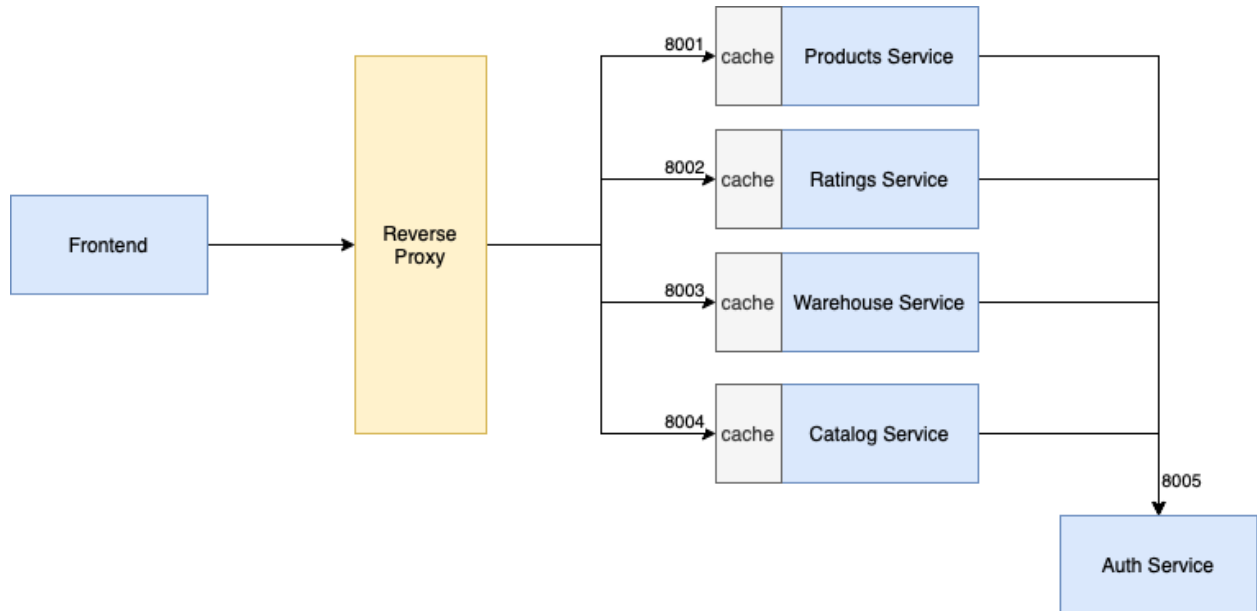
The first one is really just a couple lines of Nginx config and achieves the following:



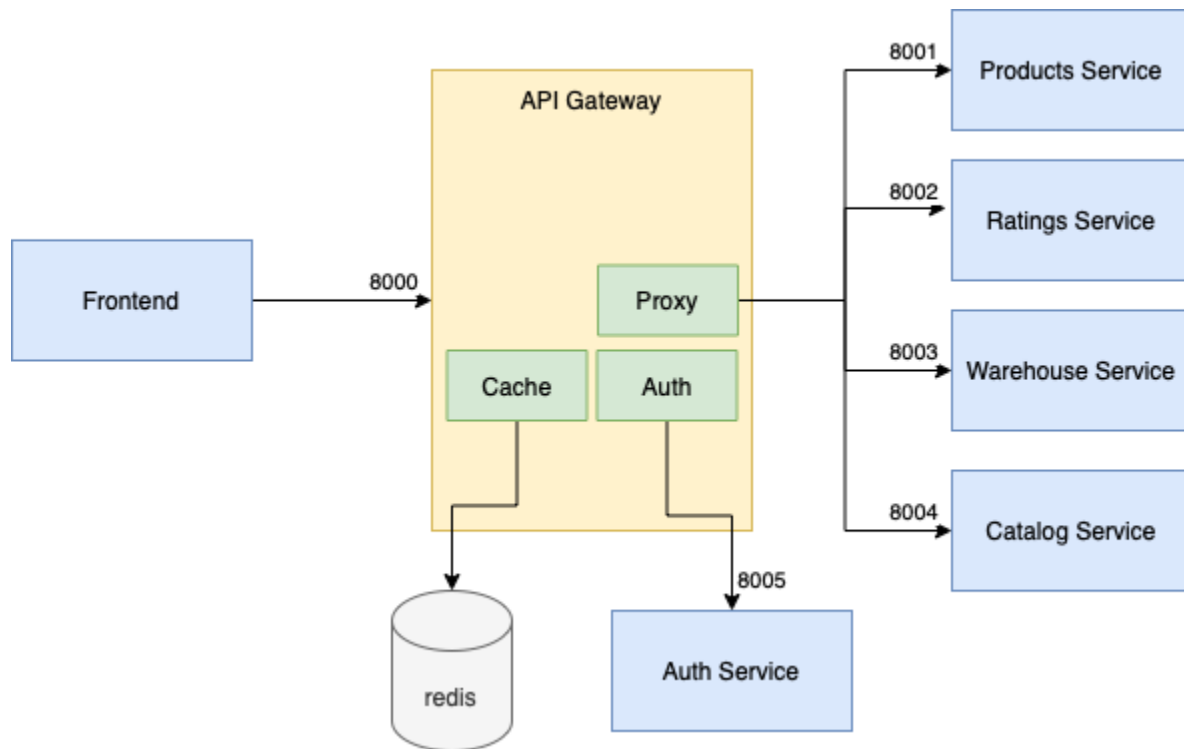
In the above figure the Frontend sends an HTTP request to the <http://proxy:8000/products>, the proxy knows that this request can be served by the Products service which lives under the product domain name and 8001 port number (these values are defined in the docker-compose.yml file), so it forwards the request to it. This is called a reverse proxy because multiple services or servers live behind a single proxy. This can be done really easily in Nginx.

Custom gateway

Maybe a simple Nginx config is not enough for our use case. For example, what if we want to cache some responses? With a simple reverse proxy, we need to implement it in every service. What if we want to authenticate the requests? We need to implement it in every service. Something like that:



You can see there's a lot of extra work to do within each service. Of course in the figure above I mean there's a Redis instance running as a separate service, the little "cache" boxes refer to the caching logic we need to implement in our services. But there's another way that looks like this:



In this case, there's no Nginx, but a more complex API Gateway. As you can see it has some logic that can handle cache, auth, and of course proxy logic. So it will:

- Query the cache if there's a response to the request
- Authenticate the request based on a token
- Proxy the request to one of our backend services

It's basically a separate service with no business, but infrastructure logic in it. One important thing: it has to be lightning-fast because **all** requests go through it. In my opinion, Laravel is out of the picture for an API Gateway. It's just too heavy. A simple Nodejs with Express or Python with Flask would be a much better choice. Nodejs is especially good when it comes to networking and routing. One more thing, earlier I wrote that there is no Nginx here. Technically that's not exactly true. You can implement your own gateway with cache and auth, but still, use Nginx to proxy requests.

Implementation

In this book we will implement a simple Nginx reverse proxy. This is how a simple proxy looks like:

```
server {  
    listen 80;  
    error_log /var/log/nginx/error.log;  
    access_log /var/log/nginx/access.log;  
  
    location /api/v1/catalog {  
        proxy_set_header Host $host;  
        proxy_set_header X-Real-IP $remote_addr;  
        proxy_pass http://catalog-api;  
    }  
}
```

/proxy/nginx.conf

Let's go through it line-by-line:

- **server**: we are creating a new server for Nginx
- **listen**: we want it to be on port 80. This is the internal port number, later we will map it to 8000 on our host machine in the docker-compose file
- **error_log**, **access_log**: location of log files. Nginx will redirect every log to stdout.
- **location**: we specify one endpoint inside our server. This is the catalog API endpoint. So, every time a request comes to the `proxy/api/v1/catalog` location we want to do the following:
- **proxy_set_header**: we set the `Host` header to the original sender (the browser of the user). Otherwise, every request would contain the Proxy server as the host which is not very helpful.
- **proxy_set_header**: we also set an `X-Real-IP` header that holds the user's IP address.
- **proxy_pass**: this is the most important line. Here we are telling Nginx that it should "pass" the request to the Catalog service. We're using the `catalog-api` domain name from the docker-compose. The Proxy server and the Catalog service are on the same internal network, so they know each other's hostname.

Ideally, we don't want to list all of the endpoints in this config, so we use some regex. I won't deep dive into Nginx regular expressions, but we can solve our problems with only one: *location ~ /catalog*

This location regex will match any request that contains the word “catalog”. With this in mind we can write the whole config:

```
server {
    listen 80;
    error_log /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;

    location ~ /catalog {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://catalog-api;
    }

    location ~ /ratings {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://ratings-api;
    }

    location ~ /inventory {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://warehouse-api;
    }

    location ~ /products {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_pass http://products-api;
    }
}
```

/proxy/nginx.conf

As you can see it's quite easy. The important thing is that order matters! Just as in Laravel. So with this config, if a request comes to */api/v1/products/inventory* and you want to forward it to the Product service you can't. It will always match the *~ /inventory* location and go to the Warehouse service. It's okay for now, but keep that in mind.

Now, we can add a new service to the docker-compose:

```
proxy:
  image: nginx:1.19-alpine
  volumes:
    - ./proxy/nginx.conf:/etc/nginx/conf.d/default.conf
  ports:
    - 8000:80
  depends_on:
    - catalog-api
    - products-api
    - warehouse-api
    - ratings-api
```

docker-compose.yml

Because in the *nginx.conf* we refer to our API services, we need to add them to the `depends_on` section.

As a result, we are able to access all of our API endpoints via the Proxy service on port 8000:

GET 127.0.0.1:8000/api/v1/catalog?sortBy=name&sortDirection=asc

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE
<input checked="" type="checkbox"/> sortBy	name
<input checked="" type="checkbox"/> sortDirection	asc
<input type="checkbox"/> searchTerm	laptop
Key	Value

Body Cookies Headers (12) Test Results

Pretty Raw Preview Visualize JSON

```
1  {
2    "data": [
3      {
4        "numberOfRatings": 0,
5        "id": 4,
6        "name": "iPhone",
7        "price": 699,
8        "averageRating": null,
9        "availableQuantity": 10
10     },
11     {
12       "numberOfRatings": 3,
13       "id": 3,
14       "name": "Laptop",
15       "price": 12.99,
16       "averageRating": 3.33,
17       "availableQuantity": 20
18     }
19   ]
20 }
```

In the output of docker-compose, you can see that every request goes through the Proxy:

```
e-commerce-catalog-api-1 | 172.22.0.14 - - [22/Oct/2021:20:40:05 +0000] "GET /api/v1/catalog?sortBy=name&sortDirection=asc HTTP/1.0" 200 425 "-" "PostmanRuntime/7.28.4"
e-commerce-catalog-1 | 172.22.0.12 - 22/Oct/2021:20:40:01 +0000 "GET /index.php" 200
e-commerce-proxy-1 | 172.22.0.1 - - [22/Oct/2021:20:40:05 +0000] "GET /api/v1/catalog?sortBy=name&sortDirection=asc HTTP/1.1" 200 437 "-" "PostmanRuntime/7.28.4"
```

Logging

Before moving on to the frontend, we need to discuss a topic, and that is logging. By default, a Laravel project will stream logs to the storage/laravel.log file. But now we have 5 services working together, and it can be really difficult to debug into 5 different log files. So we need one place, where we can see the logs. We need a single source of truth. And this place in development will be the output of docker-compose. That's the process that runs our services, and the one place we want to look at when debugging.

We need every service to log to the stdout which will be shown in the output of docker-compose. But we also want to keep the log files in the storage directory. To achieve this, we have to modify the *config/logging.php* file. First we need a stdout driver:

```
'stdout' => [
    'driver' => 'monolog',
    'level' => env('LOG_LEVEL', 'debug'),
    'handler' => StreamHandler::class,
    'formatter' => env('LOG_STDERR_FORMATTER'),
    'with' => [
        'stream' => 'php://stdout',
    ],
],
```

config/logging.php

Then we need to change the stack driver, to use the stdout driver:

```
'stack' => [
    'driver' => 'stack',
    'channels' => ['daily', 'stdout'],
    'ignore_exceptions' => false,
],
```

We are telling Laravel when we use the stack driver, please use the daily and the stdout drivers under the hood. The daily driver will create one log file for each day, and the stdout will log to the stdout. By using the daily driver we won't have a single huge laravel.log file, but instead, we have smaller, easier to read daily log files.

Then we need to tell our services, to use the stack driver. We achieve this in the docker-compose environment:


```
- LOG_CHANNEL=stack
- LOG_LEVEL=debug
```

Now if we want to log something by using the `info()` or the `logger()` helper, or the Log Facade, we will see the message in the terminal and in the daily log file:

```
public function index(GetCatalogRequest $request)
{

    info( message: 'INFO MESSAGE FROM CATALOG SERVICE');
```

App\Http\Controllers\CatalogController

You can see the result in the terminal window:

```
infra-catalog-1 | [2021-10-23 13:30:43] local.INFO: INFO MESSAGE FROM CATALOG SERVICE
infra-products-1 | 172.23.0.12 - 23/Oct/2021:13:30:43 +0000 "GET /index.php" 200
infra-products-api-1 | 172.23.0.4 - - [23/Oct/2021:13:30:44 +0000] "GET /api/v1/products?sort=price" 200 126 "-" "GuzzleHttp/7"
infra-warehouse-1 | 172.23.0.11 - 23/Oct/2021:13:30:44 +0000 "GET /index.php" 200
infra-warehouse-api-1 | 172.23.0.4 - - [23/Oct/2021:13:30:45 +0000] "GET /api/v1/inventory/products" 200
infra-ratings-api-1 | 172.23.0.4 - - [23/Oct/2021:13:30:45 +0000] "GET /api/v1/products/ratings" 200
```

Output of docker-compose

And also in the daily log file:

```
CatalogController.php x laravel-2021-10-23.log x
1 [2021-10-23 13:30:43] local.INFO: INFO MESSAGE FROM CATALOG SERVICE
2
```

Daily log file

In this way, you have an aggregated log stream by the docker-compose, but you also have the individual log files inside each service.

As you can see in the screenshots the access (or error) logs of Nginx always appear on the terminal window. This is because in the Nginx image the maintainers redirect the access and error logs to stdout by these two lines:

```
# forward request and error logs to docker log collector  
&& ln -sf /dev/stdout /var/log/nginx/access.log \  
&& ln -sf /dev/stderr /var/log/nginx/error.log \
```

This is a good pattern described in the 12factor. The important thing is you always want to log to the stdout and stderr in containerized environments. You don't want to run multiple services, where each can have its own "logging policy". You know, when App 1 writes logs into /var/logs but App B writes them in the storage directory, meanwhile for some reason App C writes everything in a database table. You want everything on the stdout or stderr.

The whole logging and monitoring are much more complicated than you think, especially when you have multiple servers, but the basic concept is that you need to log to stdout. From here you can use ELK or EFK stack to send the docker-compose logs to an Elasticsearch instance.

Frontend

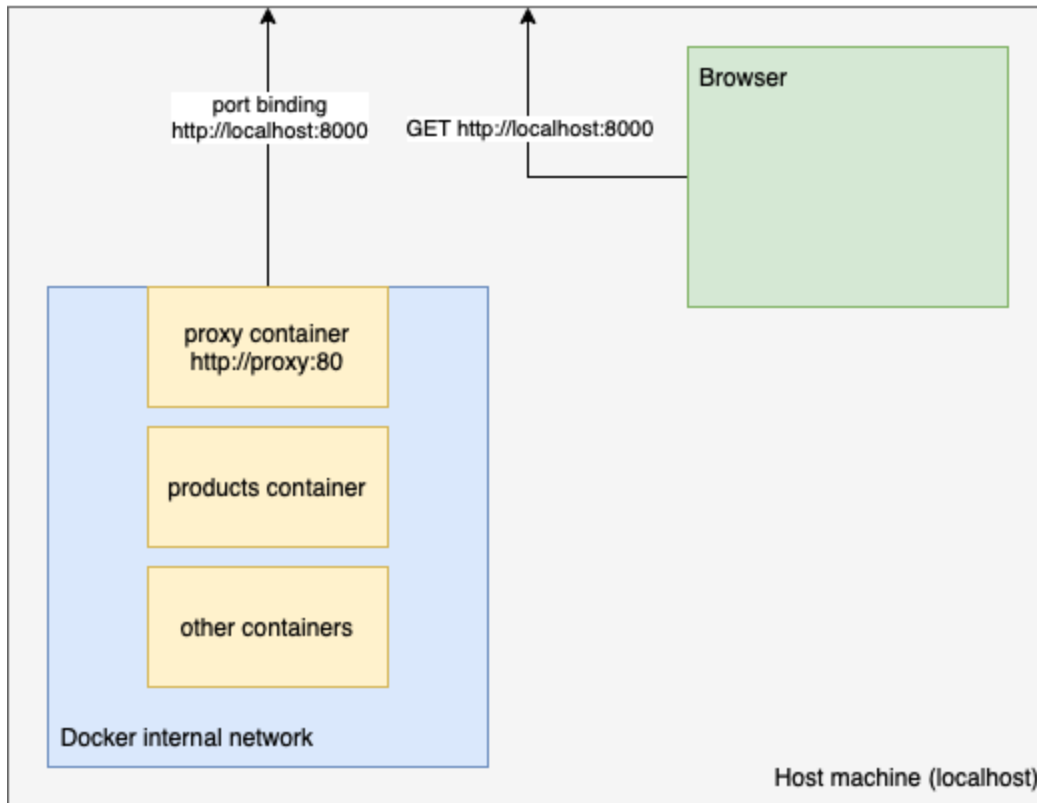
I will not discuss the frontend code in detail, because it's not strictly related to the topic of microservices, but there are some important things.

We've created a reverse proxy in order to make the connection easy between the frontend and the backend services. Let's see how we use it in the docker-compose:

```
frontend:
  build:
    context: ../frontend
    dockerfile: ../docker/Dockerfile
  ports:
    - 3000:8080
  volumes:
    - ../frontend:/usr/local/src
  environment:
    - NODE_ENV=local
    - VUE_APP_API_BASE_URL=http://localhost:8000/api/v1
  depends_on:
    - proxy
```

docker-compose.yml

The most important line is the `VUE_APP_API_BASE_URL=http://localhost:8000/api/v1`. But why localhost? That's because the browser is the one who sends the API requests to the backend, and the browser knows nothing about our docker-compose. It lives outside of the docker network. It lives on the host machine, let's visualize it:



So we need to inject a localhost URL into the frontend container:

- It will build the source code into static files. These files include the `http://localhost:8000/api/v1` value
- The browser will send the HTTP requests to `http://localhost:8000/api/v1`
- The localhost:8000 will map to the proxy container

In the frontend we can get this value in a config file:

```
export default {  
  baseUrl: process.env.VUE_APP_API_BASE_URL  
};
```

src/app.config.js

After that, we can use the *baseUrl* in our HTTP requests:

```
import axios from 'axios';
import config from '@app.config';

export default class Api {
  async get(uri, params = {}) {
    return (await axios.get(
      `${config.baseUrl}/${uri}`,
      { params }
    )).data.data;
  }

  async post(uri, data) {
    try {
      return (await axios.post(
        `${config.baseUrl}/${uri}`, data
      )).data.data;
    } catch (err) {
      this.displayError(err);
      return false;
    }
  }

  displayError(err) {
    let message = err;
    if (err.response) {
      message = JSON.stringify(err.response.data);
    }

    window.alert(message);
  }
}
```

src/services/ApiService

This *ApiService* is just a really simple wrapper around HTTP calls, in other service classes we can use it:

```
import Api from '@services/ApiService';

export default class ProductService {
  constructor() {
    this.api = new Api();
  }

  async getProduct(id) {
    return await this.api.get(`products/${id}`);
  }
}
```

src/services/ProductService

In this way, we don't have to deal with Axios, base URLs, or response parsing in our service classes.

Here are the routes:

```
const routes = [
  { path: '/', name: 'catalog', component: Catalog },
  { path: '/products/:id', name: 'product-show', component:
ProductShow },
  { path: '/products/create', name: 'product-create', component:
ProductCreate },
  { path: '/inventory/create', name: 'inventory-create',
component: InventoryCreate },
];
```

src/router/index

Here you can see the structure of our components:

- components
 - Catalog
 - Catalog.vue
 - Filter.vue
 - Inventory
 - Create.vue
 - Product
 - Card.vue
 - Create.vue
 - Show.vue
 - Rating
 - Create.vue
 - List.vue

If you open up the Catalog component, for example, you see the basic anatomy of a typical component:

```
setup() {
  const catalog = ref({});
  const catalogService = new CatalogService();

  const fetchCatalog = async () => {
    catalog.value = await catalogService.getCatalog(
      filters.value.searchTerm,
      filters.value.sortBy,
      filters.value.sortDirection,
    );
  };

  onMounted(async () => {
    await fetchCatalog();
  });

  return {
    catalog,
  };
}
```

src/components/Catalog/Catalog

We're initializing a *ref*, calling a service that will fetch our data via HTTP, and finally return the ref.

Every component follows the same logic, so I don't want to spend much time on it. The important part is to understand that the frontend needs a "localhost" API URL to communicate with the proxy.

Testing

Testing a service is exactly the same as testing in a monolith application. Literally, there is nothing special, nothing “microservicy” about it. In this book, I haven’t spent time on testing, because it’s out of the scope of this book. Now, we’re gonna write some very simple tests in the Product service. There is no real business logic or any calculation in this service, so we’re gonna write standard API tests for the `POST /products` endpoint.

Setup

First, we need a separate mysql and redis instance to use them in tests. Let’s add them to the `docker-compose.yml`:

```
mysql-test:
  image: mysql:8
  volumes:
    - ./bin/mysql:/docker-entrypoint-initdb.d
  environment:
    - MYSQL_ROOT_PASSWORD=root

redis-test:
  image: redis:6.2.5-alpine
```

docker-compose.yml

There are no port bindings or data volume for these containers, because we don’t want to browse the test databases. We just want them to be there and used by our tests.

Next, we configure phpunit to use these connections. We can do it in the `phpunit.xml`:

```
<server name="DB_HOST" value="mysql-test"/>
<server name="REDIS_HOST" value="redis-test"/>
```

phpunit.xml

Here we are setting environment variables via `phpunit.xml`. When we run `php artisan test` it will load this XML file and set the `DB_HOST` and `REDIS_HOST` environment variables with these values. Remember, we need Redis because the `POST /products` API will publish a `product:created:v1` event.

This is all the setup we need to do. Now let’s write some tests!

API tests

So we want to test the *POST /products* API. Let's create a class called *CreateProductApiTest*. We need to take care of two things first:

- MySQL
- Redis

We want them to be empty before every test. We can achieve this easily:

```
use Tests\TestCase;

class CreateProductApiTest extends TestCase
{
    use RefreshDatabase;

    protected function setUp(): void
    {
        parent::setUp();
        Redis::flushall();
    }
}
```

Tests\Feature\CreateProductApiTest

The *RefreshDatabase* will refresh the database before every test function. It will run our migrations once, and then it uses transactions in every test function. It will never commit the changes but roll back every single time. This means the test database will be empty before and after every test.

Laravel does not have a Trait like this for Redis, but it's quite simple, we only have to call the *flushall* function, in the *setUp* method. The *setUp* will run before *every* test function. So every test method will start with an empty database and an empty Redis. That's what we want.

After all this setup, we can write our actual test:

```

/** @test */
public function it_should_return_201()
{
    $category = Category::factory()->create();
    $response = $this->json('POST', '/api/v1/products', [
        'categoryId' => $category->id,
        'name' => 'Test Product',
        'description' => 'Description',
        'price' => 29,
    ]);

    $response->assertStatus(Response::HTTP_CREATED);
}

```

Tests\Feature\CreateProductApiTest

This is the most simple test. We create a random Category with CategoryFactory, after that we call the API endpoint. The *json* method is a Laravel helper, it makes an HTTP request and sends the data as JSON. By default, it returns a *TestResponse* object.

This is an *Illuminate\Testing\TestResponse* object that has some cool helper methods. For example the *assertStatus* function. But you can also check cookies, sessions, headers, redirects, and even file downloads. In this test, we only check that the status code is 201 created.

You can also test what happens if you call the API with invalid data:

```

/** @test */
public function it_should_return_422_if_name_missing()
{
    $category = Category::factory()->create();
    $response = $this->json('POST', '/api/v1/products', [
        'categoryId' => $category->id,
        'description' => 'Description',
        'price' => 39,
    ]);
}

```

```
$response->assertStatus(Response::HTTP_UNPROCESSABLE_ENTITY);  
}
```

For more complicated “data-intensive” use cases you can use PHPUnit data providers.

Now let’s make a successful API request and check the data in the response:

```
/** @test */  
public function it_should_return_the_new_product()  
{  
    $category = Category::factory()->create();  
    $product = $this->json('POST', '/api/v1/products', [  
        'categoryId' => $category->id,  
        'name' => 'Test Product',  
        'description' => 'Description',  
        'price' => 29,  
    ])->json('data');  
  
    $this->assertEquals('Test Product', $product['name']);  
    $this->assertEquals('Description', $product['description']);  
    $this->assertEquals(29, $product['price']);  
    $this->assertEquals($category->id,  
        $product['category']['id']);  
}
```

The setup is the same, but we are using other assert methods as before. The *assertEquals* simply compares the two parameters and fails if they are not equals. There is some other useful assert methods like:

- *assertNotEquals*: the opposite of *assertEquals*
- *assertSame*: while *assertEquals* uses `==` for comparison, *assertSame* also checks the types. So *assertSame*(29, 29.00) will fail, but *assertEqual*(29, 29.00) will pass.

So far we have tested the HTTP layer. Now we can check what happens in the database:

```

/** @test */
public function it_should_create_a_new_product()
{
    $category = Category::factory()->create();
    $product = $this->json('POST', '/api/v1/products', [
        'categoryId' => $category->id,
        'name' => 'Test Product',
        'description' => 'Description',
        'price' => 29,
    ]->json('data'));

    $this->assertDatabaseHas('products', [
        'category_id' => $category->id,
        'name' => 'Test Product',
        'description' => 'Description',
        'price' => 29,
    ]);
}

```

Here, the only difference is that we use the *assertDatabaseHas* method, which will perform a query with the given array as where expressions. *assertDatabaseHas* is a helper provided by Laravel. It also has some other helpers like:

- *assertDatabaseMissing*: the opposite of *assertDatabaseHas*
- *assertDatabaseCount*: it performs a count(*) query and checks against a given count

Lastly, we can test that our API will create a new entry in the Redis event stream:

```

/** @test */
public function it_should_create_a_product_created_event()
{
    $category = Category::factory()->create();
    $product = $this->json('POST', '/api/v1/products', [
        'categoryId' => $category->id,
        'name' => 'Test Product',
        'description' => 'Description',
        'price' => 29,
    ])->json('data');

    $events = Redis::xRange(
        RedisService::ALL_EVENTS_KEY,
        (int) now()->subHour()->valueOf(),
        (int) now()->valueOf()
    );

    $event = collect($events)->last();
    $data = json_decode($event['event'], true);

    $this->assertEquals(Events::PRODUCT_CREATED, $data['type']);
    $this->assertEquals($product['id'], $data['data']['id']);
}

```

After we call the API, we need to get the entries from the stream. This is what `xRange` does. We also need to parse the result into a PHP array, and we simply check the type of the event and the ID of the product. This test ensures that if we call the `POST /products` API, it will publish a new event into the Redis event stream.

You can test this behavior by mocking the `RedisService::publishProductCreated` method, but I'm personally not a big fan of mocking. I understand the reason behind isolated unit tests. But at the end of the day, the test above will behave **exactly** like our application in production. It calls the API via HTTP as our users will, and then it adds a new entry into Redis, just like our production application will. In my opinion, when you test it with a mock object you are saying: "my API can successfully call a mock object".

That's all for testing. I like to cover my service with API and other types of feature tests, but I don't always write integration tests. By integration test, I mean a test that invokes more than one service. The API test above tests the input of the Product service (the API call with the requested data), and the output of the Product service (the `product:created` event in Redis). If I want to make sure that the Rating service can react to the `product:created` event then I write a feature test in the Rating service that starts with the creation of the `product:created` event, after that I run the `redis:consume` command, and assert that the product is created in the database.

That's all for testing, now let's use it in CI.

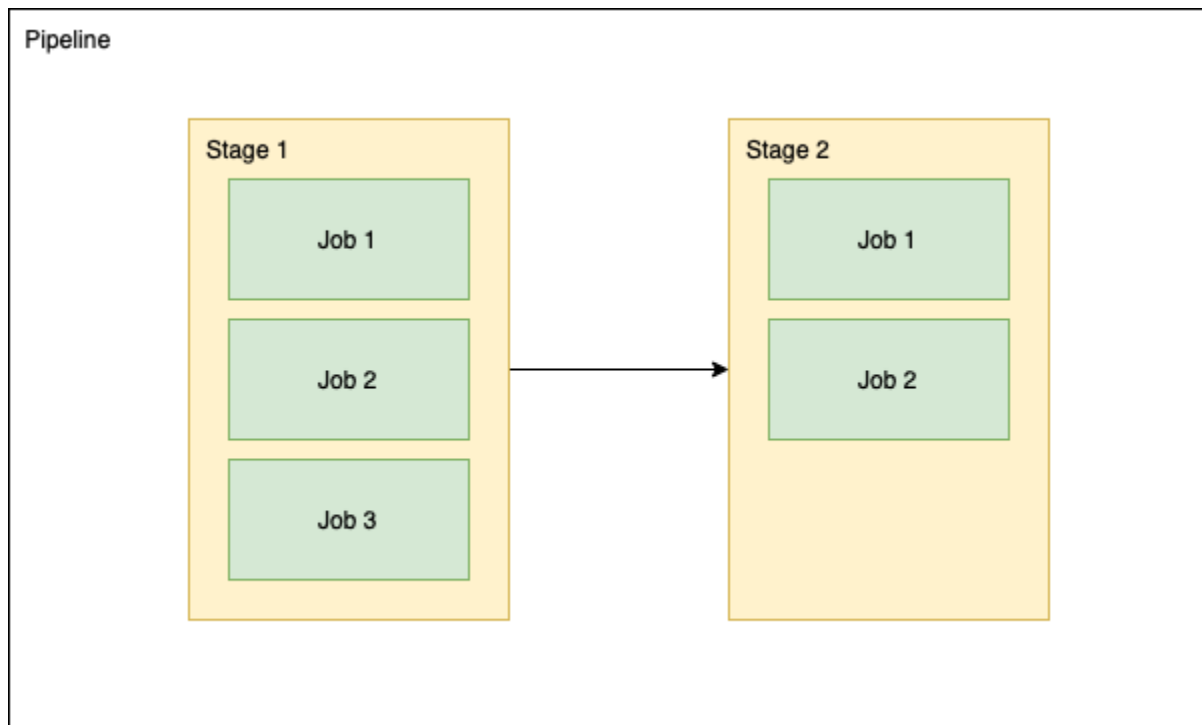
CI/CD Pipeline

One more important topic is continuous integration. This is a very complicated topic, so it's perfectly okay if you skip this part, but we're gonna create a simple, yet useful pipeline.

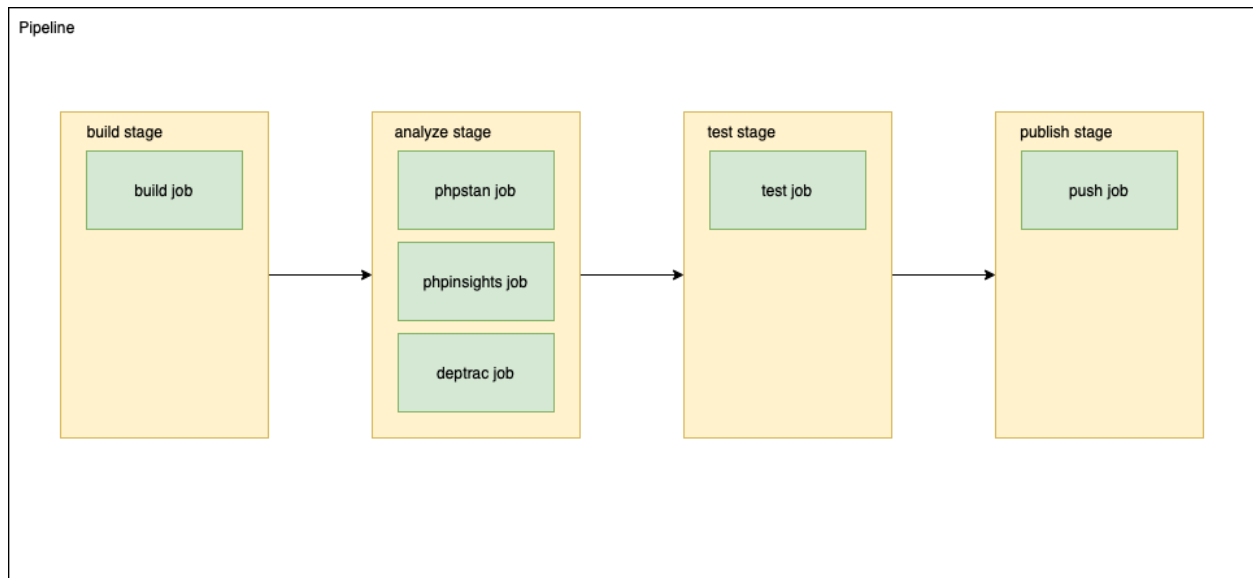
First of all, what is a pipeline? It's a number of steps that run after every push you make. These steps can be:

- Static code analysis
- Testing
- Performance tests
- Security analysis
- Building docker images
- Pushing docker image
- Deploy to servers

We will implement some of these steps using Gitlab CI. First, let's see the anatomy of a Gitlab pipeline:



A pipeline contains multiple stages. A stage contains multiple jobs. A stage is like a box for similar jobs. And jobs do the actual work. Stages are dependent on each other, meanwhile, jobs in a stage can run in parallel. Let's visualize the pipeline we will build:



The purpose of these jobs:

- Build: building a docker image
- phpstan: running phpstan code analysis
- phpinsight: running phpinsights code analysis
- deptrac: running deptrac dependency analysis

First, we declare our stages:

```
stages:  
- build  
- analyze  
- test  
- publish
```

After that we can declare some global variables that can be accessed in any job:

```
variables:  
  JOB_IMAGE_NAME_DEV: $CI_REGISTRY_IMAGE:dev
```

In this example, we only build an image with the *dev* tag. If you want to build a production image, you can tag it with the commit SHA. The `$CI_REGISTRY_IMAGE` is an environment variable provided by Gitlab. It refers to the docker image repository hosted on Gitlab (in the products project).

The first job is the build:

```
build:
  stage: build
  script:
    - docker build -t $JOB_IMAGE_NAME_DEV -f ./docker/Dockerfile .
```

We tag the image with the global variable. Next, we can run the static code analysis tools using the image:

```
phpstan:
  stage: analyze
  script:
    - docker run --rm -t $JOB_IMAGE_NAME_DEV ./vendor/bin/phpstan analyze
      --memory-limit=1G

phpinsights:
  stage: analyze
  script:
    - docker run --rm -t $JOB_IMAGE_NAME_DEV php artisan insights
      --no-interaction --min-quality=90 --min-complexity=90 --min-architecture=9
      --min-style=90

deptrac:
  stage: analyze
  script:
    - docker run --rm -t $JOB_IMAGE_NAME_DEV ./vendor/bin/deptrac
```

You can check the documentation of these tools, but in a nutshell:

- phpstan: it checks your code and finds bugs or violations based on type hints, PSR standards, and other rulesets. It is configured in the */phpstan.neon* config file.
- phpinsights: it does similar checks, but it has a summary where you can see a percentage of the quality of your code. In the example above we say: if any of this quality percentage is less than 90% break the pipeline. It is configured in the */app/config/insights.php* config file.
- deptrac: it checks the dependencies between your classes, and your “layers”. So you can define rules like a Repository could never use a Service class, and so on. It is configured in the */depfile.yaml* config file.

If any of these tools find something “bad” the pipeline will break.

Finally, we can run our tests:

```
test:
  stage: test
  before_script:
    - docker-compose -f docker-compose.ci.yml up -d --force-recreate
    - docker-compose -f docker-compose.ci.yml exec -T products chown -R
      www-data:www-data /usr/local/src
  script:
    - docker-compose -f docker-compose.ci.yml exec -T products php artisan
      test
  after_script:
    - docker cp products_products_1:/usr/local/src/products/tests/coverage
      tests/coverage
    - docker-compose -f docker-compose.ci.yml down
  artifacts:
    when: always
    paths:
      - tests/coverage
```

In the *before_script* we start our containers and set some permissions. In the *script*, we simply run the *php artisan test* command. Finally, we copy the test coverage report from the container to the host machine (Gitlab runner) and stop our containers.

Code coverage is made by XDebug that we have installed in the Dockerfile:

```
RUN pecl install xdebug
RUN docker-php-ext-enable xdebug
```

Configured in the */docker/docker/dev.ini* php ini file:

```
xdebug.mode = develop,coverage
```

And we also configured PHPUnit to put the coverage to the *tests/coverage* directory:

```
<coverage
  processUncoveredFiles="true"
  includeUncoveredFiles="true"
  cacheDirectory="/dev/shm"
  pathCoverage="false"
>
  <include>
    <directory suffix=".php">./app</directory>
  </include>
  <exclude>
    <directory suffix=".php">./app/Exceptions</directory>
    <directory suffix=".php">./app/Http/Middleware</directory>
    <file>./app/Providers/BroadcastServiceProvider.php</file>
  </exclude>
  <report>
    <html outputDirectory="./tests/coverage" />
  </report>
</coverage>
```

The last piece of the puzzle is the `docker-compose.ci.yml` file that we use in the pipeline. I don't list the whole file here, you can see it in the root directory, but the most important part is the:

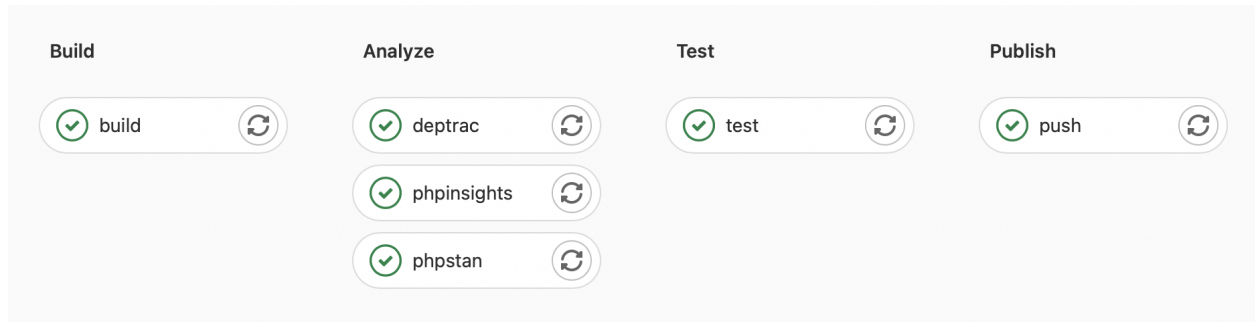
```
image: $JOB_IMAGE_NAME_DEV
```

In the pipeline above we declared the `$JOB_IMAGE_NAME_DEV` variable which we can use in the docker-compose file.

The artifact part makes it possible to download the coverage as a zip file from Gitlab (on the pipeline result page). Now, all we need to do is to push the docker image:

```
push:
  stage: publish
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
      $CI_REGISTRY
    - docker push $JOB_IMAGE_NAME_DEV
```

So that's our pipeline, you can see the whole config in the `.gitlab-ci.yml` file in the root directory. This is the result:



Final thoughts

Congratulation! You've just implemented an application in a microservices way. If you think about it, it wasn't that hard. Of course, there are some new things like the event stream, the consumers, and containers, but at the end of the day, every service remains very small and simple. And this is a huge gain for the developers, and for the company.

But life isn't perfect, so let's see some disadvantages of this architecture:

- Complicated. This e-commerce application would be a really really simple monolith application, that you can write in a single night in around 500 lines of code. But instead, you created 5 different services, 5 different databases, a Redis event stream, HTTP calls, Containers, Commands that read from Redis, a composer package, you have about 20 running docker containers, and you need to write 4 more pipelines. The project even contains a *seed.sh* because it needs an "orchestrated" way to seed fake data. You cannot just write some code in the *DataSeeder* class, because services need to "know" what is happening outside of them.
- It's even more complicated. For example, right now it's hard to sort the products in the catalog by ratings. It's because the sorting logic is in the Product service, but the ratings are in the Rating service. So right now if you want to sort by ratings, you need to do it with `Collection::sortBy()` in the Catalog service. This problem does not exist in the monolith world.
- Because of the same reason, you cannot join certain tables anymore. The ratings table in the Ratings service, the inventories table in the Warehouse service, and so on. What if you need to join some of them? There is no straightforward answer. This problem does not exist in the monolith world.
- It's redundant. You have the same data in multiple databases. If you want consistency you need to pay attention to having the right data at any time. Every create, update, or delete method needs an event, and you need to write a lot of consumer code. This problem does not exist in the monolith world.
- Race conditions. Let's say you have a bigger application, and you have 50 events. These events eventually will depend on each other. What happens if, for some reason, they are coming in the wrong order?
- Async. The async event processing can cause some interesting situations. For example, if you rate a product, and then refresh the page on a different browser window (like if you are a different user), it's impossible to say what you will get. Maybe you see your new rating, maybe you don't see it. The event processing is async, which means it takes time, so probably your users

will not see the most recent state of your application. This problem does not exist in a traditional request-response application.

- Sync. The Catalog service works in a synchronous way. And it's slooow. It sends 3 HTTP requests, but in a monolith, it would be 3 simple function calls. You can do some optimizations, like using the cache, or making async HTTP calls with Guzzle, but the point is you have to do some extra work.

Don't get me wrong, microservices are a really great way to build software, but it's not suitable for every project. It can be a real pain in the ass! Before you start to write everything in separate services at least ask yourself the following questions:

- Is this project big enough to use microservices?
- Is our team big enough for a microservices project? In theory, every team is responsible for one service. Just think about it. Let's say you have standard 4-5 member agile teams. It means to write this e-commerce application you need about 20-25 developers. Of course, in practice, one team can handle 2 or 3 services, but still, this requires a lot of people.
- Is our organization big enough and well organized for this stuff? You need to coordinate between teams. I guarantee you that you have situations when you need to change 2, 3, or even 4 services to implement a feature. This means a lot of communication and coordination between developers, product owners, project managers, testers, and so on.
- Do we have the right infrastructure and ops for this?

Sometimes all you need is a high-quality, hand-crafted, well-written monolith.