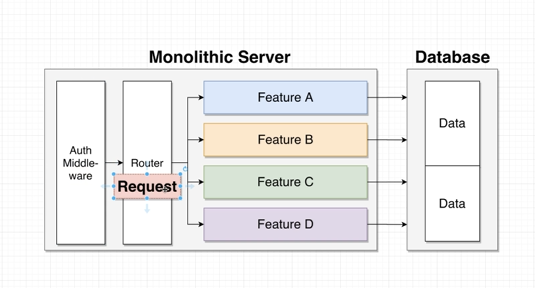
**Monolithic Architecture:**



This is probably how you are building servers right now in a monolithic server.

We have all of our code needed to implement our application inside of one single code base, and we deploy that code base as one discrete unit. We have some request coming in from a user’s browser or mobile device that will flow into our application and go through maybe some pre-processing middleware, then maybe it goes off to some router.

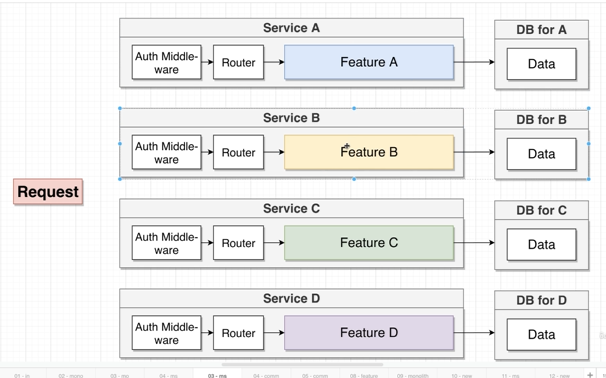
That router might then inspect the request and decide to send it off to some very specific feature to be further processed.

So maybe in this case it goes off to feature a feature a might decide to read or write some data out of a database, eventually formulate a response and then send a response back to whoever made the request.

**Monoliths**: It contains all the routing, all the middleware, all the business logic and all the database access code required to implement all features of our application. So that would characterize a monolith.

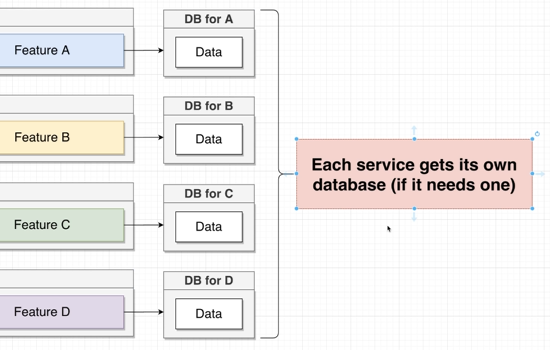
**Micro-service:**

A single micro-service contains all of the routing, all of the middleware, all the business logic and database access required to implement one feature of our application. That is the big difference. So a monolith has all the code needed to implement every feature of our application. A micro service has all the code needed to implement just **one** feature.



SO, the actual logic behind a micro-service is: Our working definition for right now is going to be that a micro service contains all of the code required to make one feature work correctly.

**Challenges Faced by Micro-services:**

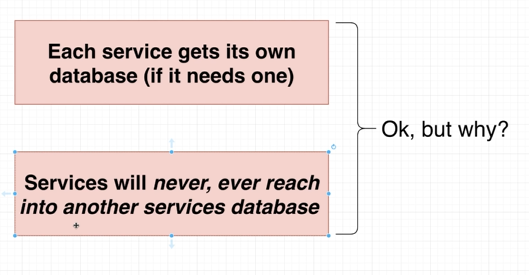


**Data Management between the services:** When I say data management, I'm talking about the way in which we store data inside of a service and how we communicate that data between different services.

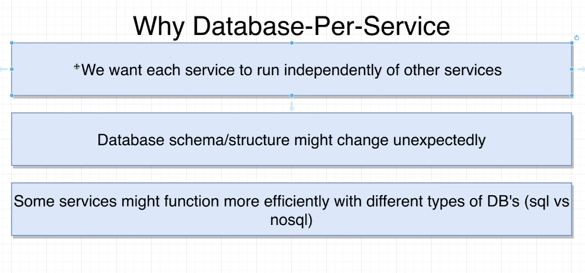
Why this is a problem? First let us see how we store and access data in a micro-service approach->

“So, whenever we are making use of micro-services, we are going to create a separate database for each service. If the service actually needs a database, if a service doesn't needs a database, that's fine. We're not going to give it one. But the point is, every service gets its own database.

Also regarding the data access, we are never going to access data by reaching from one service into another services database. So under no circumstance will we ever have service-A right here, try to reach into the database for service-B.”

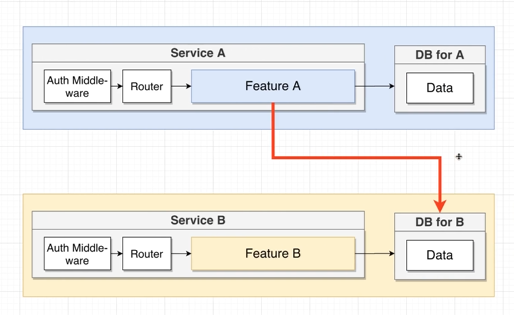


**Reason for this type of storing and access methods:**



**Problems with single data base and attempt to access database of different service:**

The problem with “this (monolith)” approach is that if anything bad ever happens to this database, all of our services are going to crash immediately. The other big problem is that scaling this database right here would be really challenging. If every service inside of application uses the same database. We're going have to scale up that one database instance or that one database to service all the different services of our app. It would be a lot easier to scale just the databases that need additional capacity or throughput.



The other scenario that we could run in, let's imagine that we've got service reaching out to database B service right here.

If anything ever goes wrong with the database B right there then all of a sudden, service is going to start to crash because we have introduced a dependency between service A and service B. So if a request comes into service A and feature A here tries to reach over to this database B and something is wrong with that database (B), boom service A is going to crash as well. And so rather than just losing service B, we would now also be losing service A, So by giving each service its own database and making sure that each service only uses its own database we dramatically increase the uptime of our entire system.

Also in case of this dependency for exchanging data between services, there would be tight coupling between services and so there would be difficulties in maintaining databases and schema update. It could also happen that one service might use different type of query language or database than the other service.

**Example:**

Let us say we have a web application with 3 basic features:

* Sign in
* List available product
* Purchase a product

These are the basic features we have in our web application. In case of Monolithic server, we have all these service’s code in one place with one database with collections like user, products and orders. Let's say that we want to add in a new feature where we can show all the products ordered by a very particular user. First, we might want to take a look at the user collection and make sure that the particular user that we're trying to look up information for exists. We might try to then look in the orders collection and find the order that a user has created. And maybe on that order, we would see the IDs of each product that was actually ordered. And then if we want you to actually show information about those products, well, we probably have to do a query over to the products collection as well. And inside of a monolith this will all be fine. Now let us consider the idea of same application with under Micro-services architecture.

In here, all services would have their separate databases, isolated. Now, let's imagine once again that we want to add in a new feature, so we want to add in a fourth service your service D and the goal of Service D is to give us the ability to show not products, but orders made by a very particular user.

So if we wanted to follow the same pattern that we used back in that monolith style approach, we could say that we want to have this service right here, reach out to that database. Reach out to database B and see. And it would have to do that if it wanted to get information about the user by the order they had created and then find details about those given products. But as we just discussed a moment ago in this Microsoft Services style world, we do not allow a service to reach out to databases owned by other services. And so the question here very quickly becomes how would we create service D?

How in the world would that happen? There are 2 methods to do so:

* **Sync communication**:

In this Synchronous world, if we are going to communicate between different services using this pattern, we're going to have one service communicate with another directly using requests.

We can apply the idea here in our earlier discussed web-app. So, let us say we need to implement Service D, a request might come into service D and it might say show me all the products that have been ordered by the user with ID number say-one. The first thing that service might do is make a direct request over to service A again, that might be a plain HTTP request, might exchange JSON, whatever its form is.

It is a direct request from over to a social service. D would then check to make sure that maybe that user exists, would then get a response back. And if that user exists, it might then make a follow up request over to its service, see and find all the orders that have been created for this user.

It would get back a response and then finally make a request over to service B and say give me details about these particular products. So after making those three different requests, service would then have all the information it needs to respond to the overall request.

**Pros**: Easy to understand and service D would not require a database.

**Cons**: Introduces a dependencies between services and risk of working failure. Also if any request within the workflow fails, overall request fails. Also the entire request is only as fast as the slowest request.

* **Async Communication(2)**:

1. **Service communicate with each other using an event bus**:

OK, so we're going to imagine that service receives a request asking to see all the products that are been ordered by a particular user. Service D needs information from its services, A, B and C to complete this request. So the first thing that service will do is emit an event on event bus connected to all services. This event might consist of request type like UserQuery or UserQueryResult and a data section consisting of information needed within the query. So this event would flow into this event bus and this event bus can handle automatically the incoming events and root them off to different services that might be able to handle that event. So in this case, we might be able to configure our event bus to send a copy of this event over to service A and so we can imagine that this event is going to flow over like so. Then inside of service A, we would have some code to be executed any time this event comes in to service, they might see this event come in. It might say, oh, someone's trying to look up some information about a user.

It's a service. They could then look up information about the user with I.D. number one, and then to respond or get some information back over to service D Service A could emit a new event so it could emit an event over to the event bus. And maybe it would be an event with a type of something like user query result, and then the data could be the actual information about this user. So maybe their ID and a name of Jill and whatever else. So this event would flow into our event bus and the event. Plus we could configure it so that it will send an event of type user query results automatically back over to service to us. So we'll go back over to Service D. And we'll have some code inside of service to receive an event of type user equal results and do some processing on it, you can then kind of imagine that we would repeat the same exact process to take a look at all the different orders that this user has created and then retrieve details about all the different products associated with those orders as well from Service B.

**Cons**: Single point of failure because of central event bus. Also same cons as Sync approach.

We're going to once again focus on this idea of introducing Service D into our e commerce application that just has three existing services.