Scalable Web Systems

Splitting the Monolith Part 1

Overview

1. Seams - where to break apart the monolith?
2. Reasons - why might you want to break apart the monolith?
3. Tangled Dependencies and the Database - how do you untangle a database?
    a. Breaking Foreign Keys
    b. Dealing with Shared Static Data
    c. Dealing with Shared Mutable Data
4. Staging - how to migrate from monolith to services

Seams

Two important criteria we want our services to have and maintain:

- Highly Cohesive
- Loose Coupling

The monolith is typically the opposite of both!

Seams

Two important criteria we want our services to have and maintain:

- **Highly Cohesive**
- Loose Coupling

The monolith is typically the opposite of both!

Rather than tend toward cohesion, and keep things together that tend to change together, we acquire and stick together all sorts of unrelated code.

Seams

Two important criteria we want our services to have and maintain:

- Highly Cohesive
- **Loose Coupling**

The monolith is typically the opposite of both!

Loose coupling doesn't really exist: if I want to make a change to a line of code, I may be able to do that easily enough, but I cannot deploy that change without potentially impacting much of the rest of the monolith, and I'll certainly have to redeploy the entire system.

Seams

A **seam** is a portion of code that can be treated in isolation and worked on without impacting the rest of the system.

Identifying seams in a monolith will help us determine service boundaries.

So, what makes a good seam?

The first step is to the boundaries in the system. Once we do that we can begin to clearly delineate seams.

Often we can use the packaging facilities of a programming language to guide us.

Example: MusicCorp

Imagine we have a large backend monolithic service that represents a substantial amount of the behavior of MusicCorp's online systems.

1. identify the high-level bounded contexts that we think exist in our organization.
2. Understand what bounded contexts the monolith maps to.

Imagine that initially we identify four contexts we think our monolithic backend covers.

Example: MusicCorp

Imagine we have a large backend monolithic service that represents a substantial amount of the behavior of MusicCorp's online systems.

1. identify the high-level bounded contexts that we think exist in our organization.
2. Understand what bounded contexts the monolith maps to.

Imagine that initially we identify four contexts we think our monolithic backend covers.

Catalog
Everything to do with metadata about the items we offer for sale

Example: MusicCorp

Imagine we have a large backend monolithic service that represents a substantial amount of the behavior of MusicCorp’s online systems.

1. identify the high-level bounded contexts that we think exist in our organization.
2. Understand what bounded contexts the monolith maps to.

Imagine that initially we identify four contexts we think our monolithic backend covers.

Catalog
Everything to do with metadata about the items we offer for sale

Finance
Reporting for accounts, payments, refunds, etc.

Example: MusicCorp

Imagine we have a large backend monolithic service that represents a substantial amount of the behavior of MusicCorp's online systems.

1. identify the high-level bounded contexts that we think exist in our organization.
2. Understand what bounded contexts the monolith maps to.

Imagine that initially we identify four contexts we think our monolithic backend covers.

Catalog
Everything to do with metadata about the items we offer for sale

Finance
Reporting for accounts, payments, refunds, etc.

Warehouse
Dispatching and returning of customer orders, managing inventory levels, etc.

Example: MusicCorp

Imagine we have a large backend monolithic service that represents a substantial amount of the behavior of MusicCorp's online systems.

1. identify the high-level bounded contexts that we think exist in our organization.
2. Understand what bounded contexts the monolith maps to.

Imagine that initially we identify four contexts we think our monolithic backend covers.

Catalog
Everything to do with metadata about the items we offer for sale

Finance
Reporting for accounts, payments, refunds, etc.

Warehouse
Dispatching and returning of customer orders, managing inventory levels, etc.

Recommendation
Our patent-pending, revolutionary recommendation system, which is highly complex code written by a team with more PhDs than the average science lab

Example: MusicCorp

Imagine we have a large backend monolithic service that represents the behavior of MusicC

1. identify the high-l that we think exis
2. Understand what monolith maps to

Imagine that initially we think our monolithic backend covers:

Catalog
Everything to do with metadata about the items we offer for sale

ayments, refunds, etc.

of customer orders, , etc.

utionary recommendation system, which is highly complex code written by a team with more PhDs than the average science lab

Create Packages Representing Contexts

We have this big tangled mess of code.

What do we do to begin untangling?

Organize and Refactor Code
Create packages representing these contexts, and then move the existing code into them.

Modern IDEs allow us to move parts of code around and will automatically perform refactorings for us.

It is not always a walk in the park!

Different programming languages provide different support for this.

Statically Typed Languages
These languages (Java, C++, Go, Rust) provide a robust type system. This gives the IDE an understanding of how things are put together.

Dynamically Typed Languages
Other languages (Python, JS, Ruby) lack explicit typing making it more difficult for an IDE to perform refactorings.

Did it work?

It is easy to move code around, but did it work?

To verify that the organizational changes to the system didn't break anything it is **critical** to continually run tests!

This is a good reason to always pay attention to **writing tests** to ensure that any changes made will validate that the code is still doing what it should be doing (at least according to the tests).

It is even more important for dynamically typed languages or those (e.g., JS) with package management that is not built-in.

Static Code Analysis

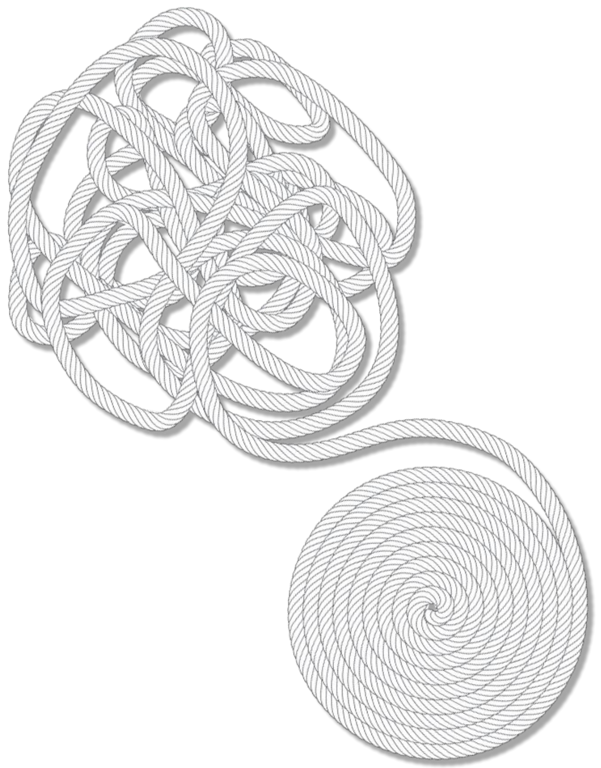
There are many code analysis tools available that can help with this migration process.

There are many code analysis tools for all sorts of languages (see this on wikipedia).

The book mentions Structure 101 which will analyze the dependencies between packages and provide a graphical depiction.

This is not a new thing - indeed it has been around since the beginning of computing.

Of course, static analysis is hard as Alan Turing proved with the halting problem in 1936.



A Piece of Cake?

Is it really that easy?

No. In fact, the problem of disentangling code can be so difficult it may warrant an entire rewrite.

What if the code uses threads? Static analysis of multithreaded code is extremely difficult.

What if the code uses events and callbacks? Static analysis is hard in the context of mutable state.

A well written software system is easier to split apart then one that is poorly designed.

Reasons To Split The Monolith

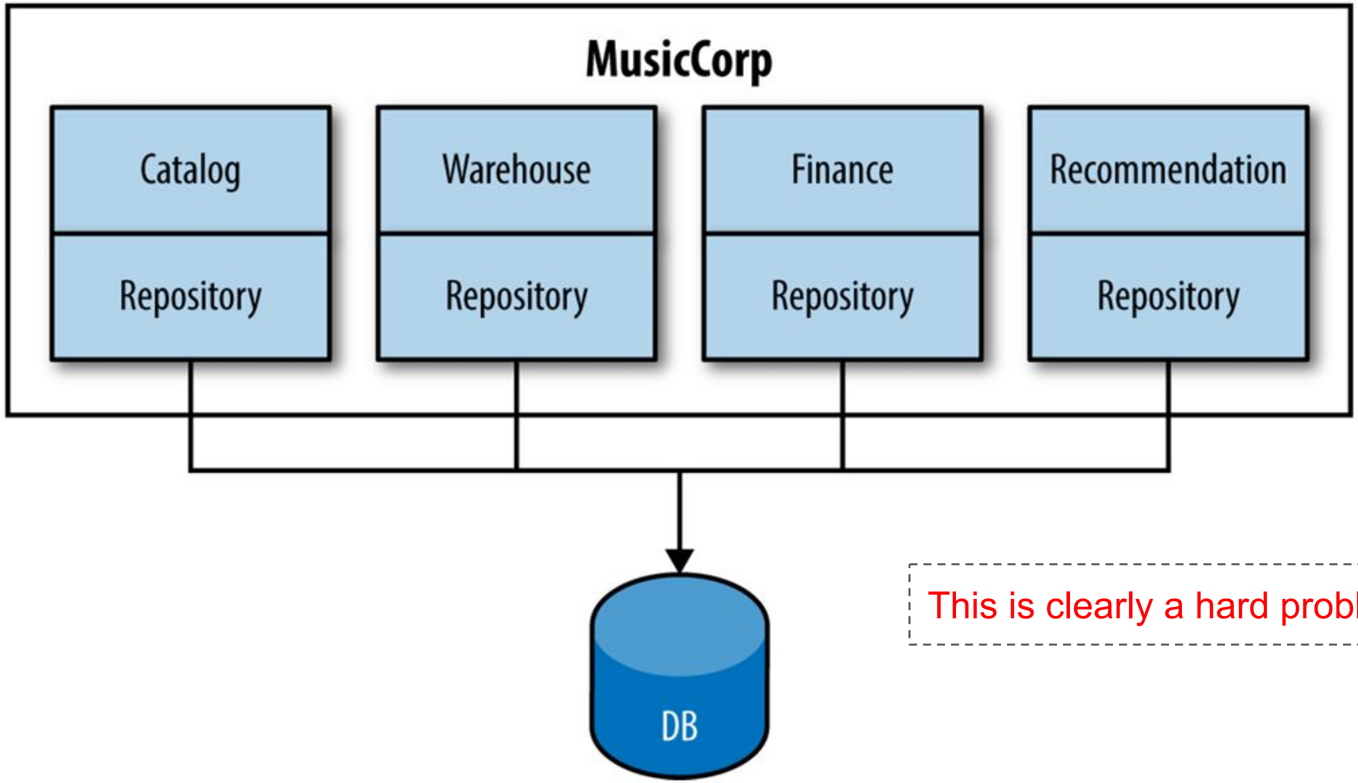
Pace Of Change
If there are a number of changes coming soon to the system it might be an opportunity to split the system into microservices.

Team Structure
If a team is split across several geographic regions it might make sense to move towards several services giving each part of the team ownership over a specific service or set of services.

Security
If the application has sensitive data splitting that sensitive data out into a specific service will help protect and monitor the security of that data.

Technology
There is an opportunity to use a different programming language, algorithm, or 3rd party service to improve the application.

Database Detanglement



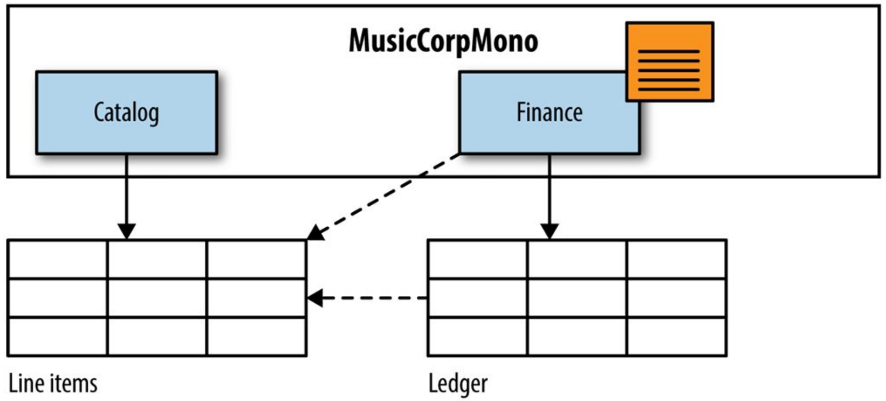
This is clearly a hard problem!

Breaking Foreign Keys

In this example, the Finance part of the monolith accesses the “Line Items” table using a *foreign key* from the Ledger table.

How do you pull this apart?

1. Need to stop the Finance module from accessing the “Line Items” table. **Solution:** make this an API call.
2. Because of the foreign key relationship we will need to implement our own consistency checks to ensure the data is accurate.



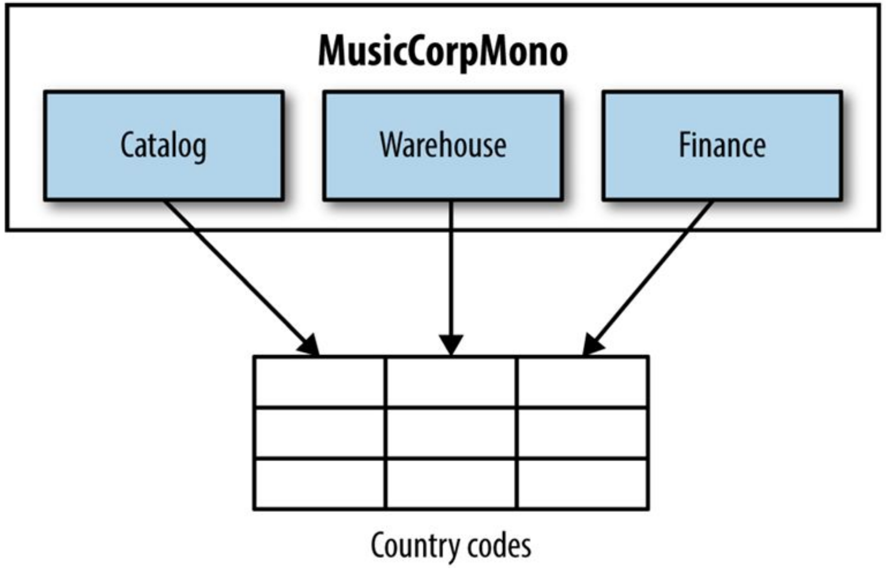
Static Data Detanglement

Static data are data that does not change.

It is very common to store that data in a database for uniformity.

This creates a dependency problem. Here are a couple of solutions:

1. Each service will keep its own tables in a database with this information. **Problem:** what happens if there is change?
2. Keep static data in files that are distributed to each service when there is a change.



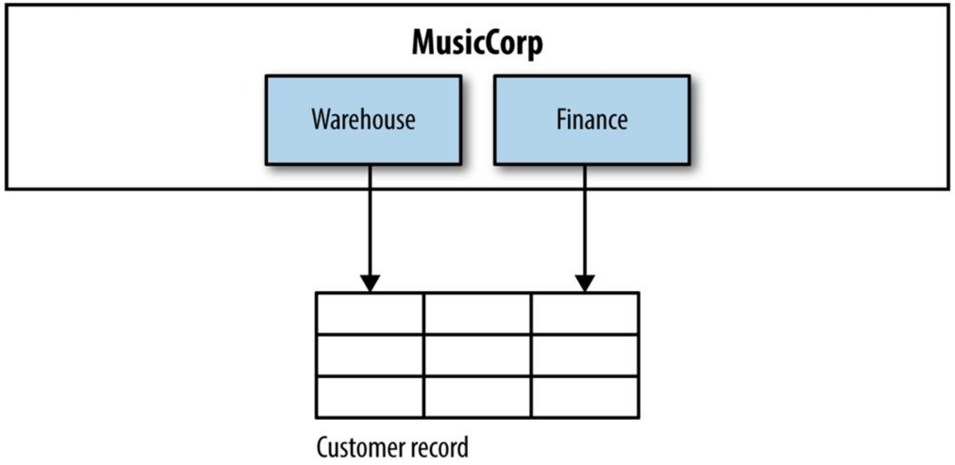
Shared Mutable Data Detanglement

What if we have 2 or more parts of the monolith accessing the same shared mutable data?

In this case, we have two components that are accessing and modifying the customer records.

This is clearly a problem. They are strongly coupled in that they share the same data.

How do we make these two components into services that are loosely coupled?



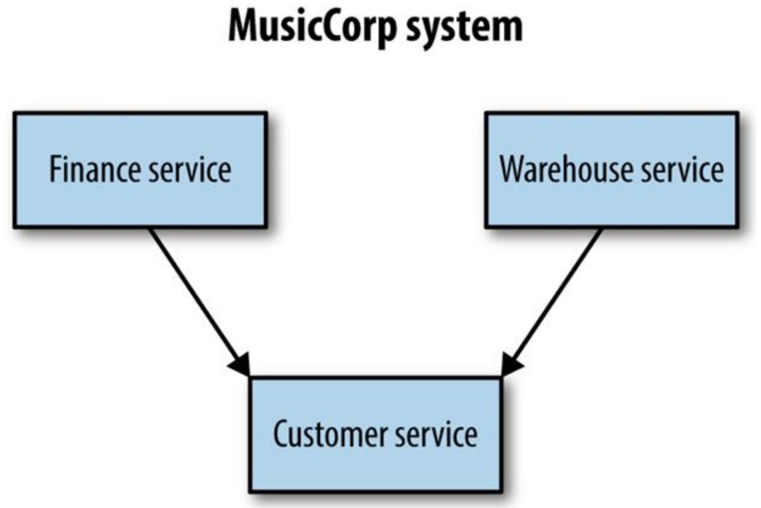
Shared Mutable Data Detanglement

Doing this properly might require the creation of another service that will accept requests to access the data.

In this way, we are able to pull apart 2 components of a monolith into 2 services with addition of a third that will be responsible for the mutable data.

This is certainly a large amount of change for a codebase and may take months to implement.

It makes sense that this should be done slowly and incrementally.

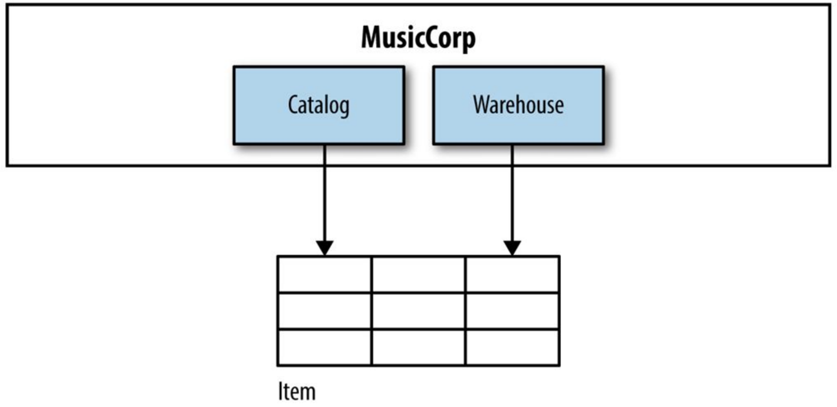


Shared Table Detanglement

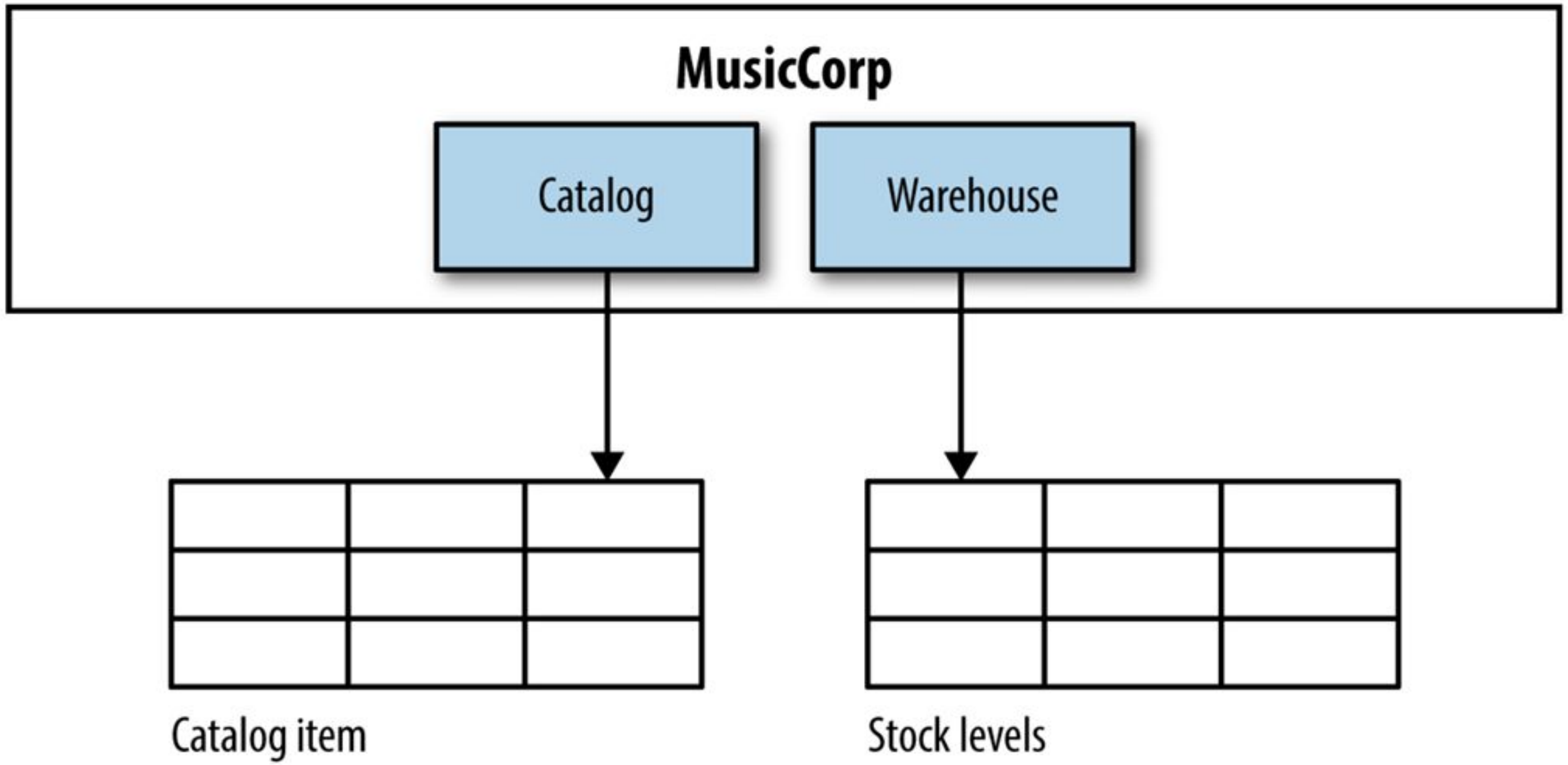
In some cases we share individual records in a database table to make things more “efficient” or “convenient”.

Another approach to this problem is to split the table (rather than create a 3rd service) if there is a clear delineation between the 2 components.

This can allow us to create only the necessary services we need without the overhead of a 3rd service to provide access.



Shared Table Detanglement



Staging The Transition

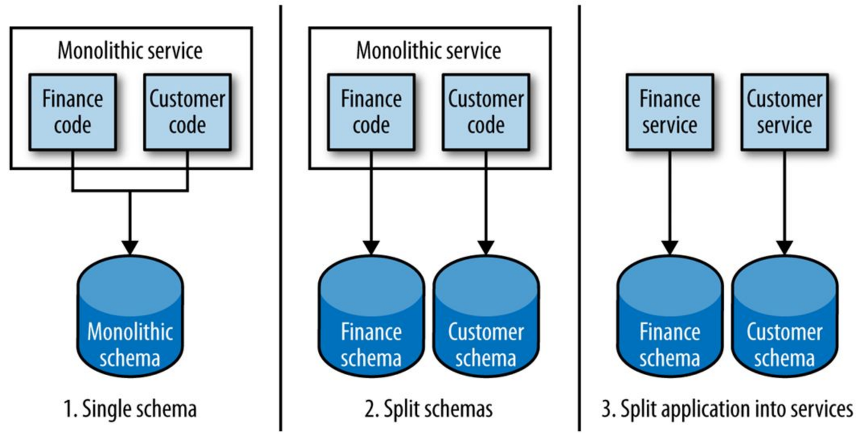
We have talked about the different ways in which we can migrate a monolith that uses a single shared database.

In this case, we are splitting the database.

It is not always the case that we can do this all at once.

So, how do we stage the transition to ensure that the system is stable and doesn't fall apart?

Again, this might take time, but it can be done if there is firm testing in place and is accomplished in an incremental fashion.


Monolithic service
Finance code
Customer code
Monolithic schema
1. Single schema
Monolithic service
Finance code
Customer code
Finance schema
Customer schema
2. Split schemas
Finance service
Customer service
Finance schema
Customer schema
3. Split application into services