Table of Contents

Migrating to Serverless (15 pages)

In this chapter, we will migrate the components developed thus far to adopt a serverless style of architecture.

Serverless Primer

Services as Functions

Serverless Persistence

Next Steps

Legacy Application Migration Patterns

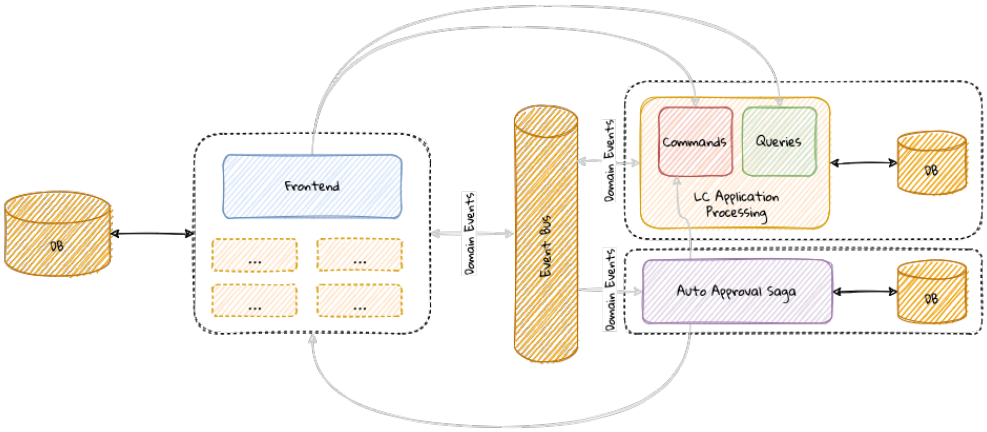
Decomposing and distributing big balls of mud

Code-first

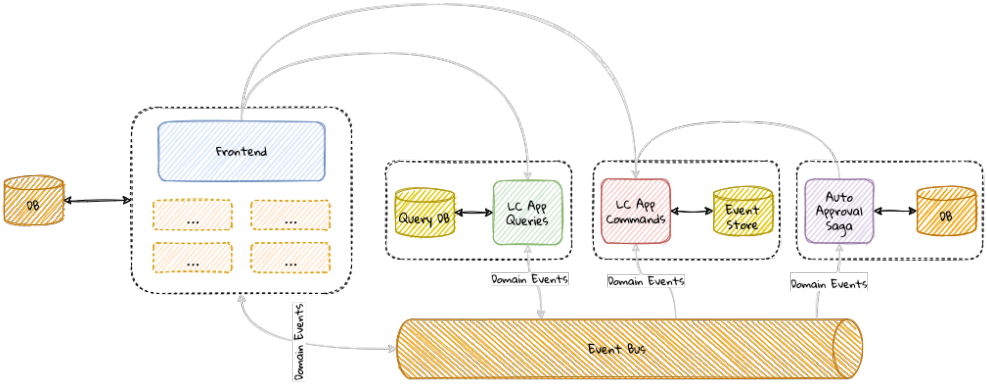
Data-first

Potential next steps

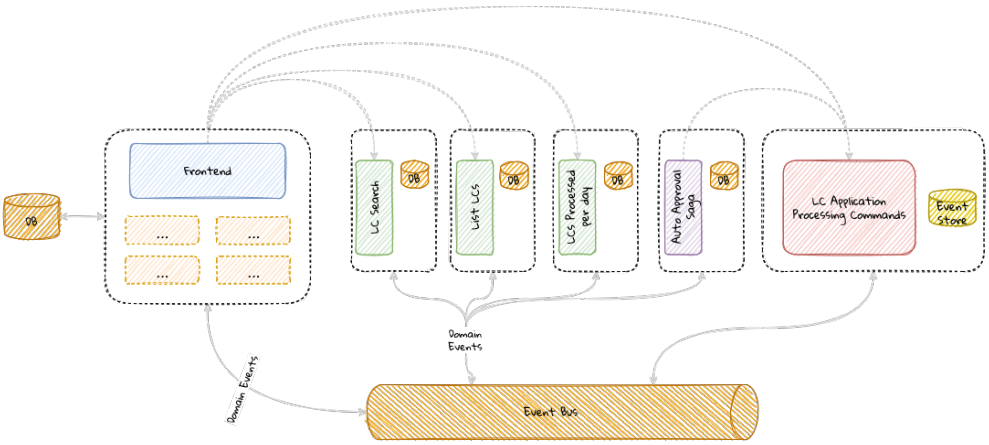
Sagas as standalone components



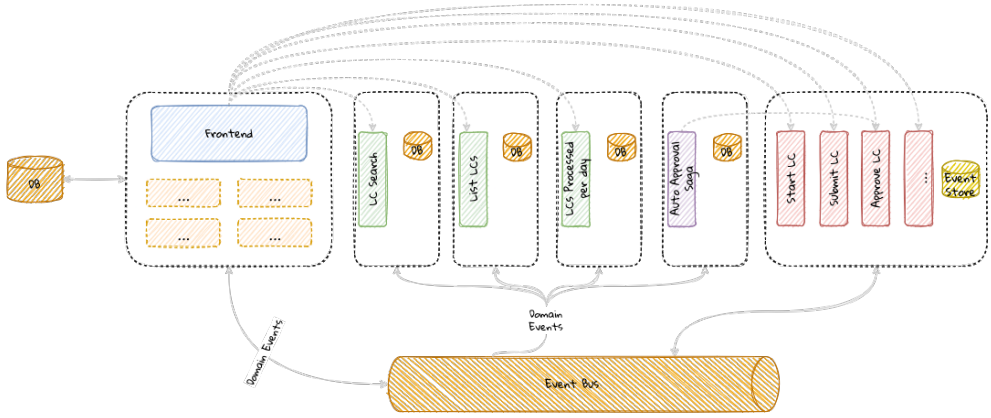
Commands and queries as standalone components



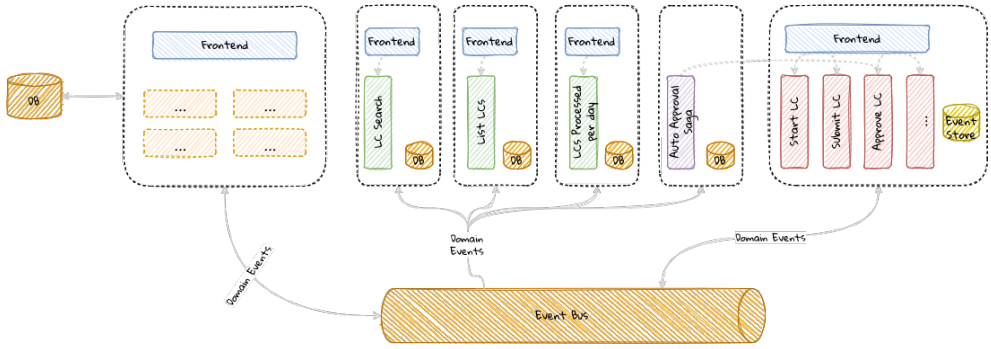
Distributing individual query components



Even more fine-grained distribution



Customer experiences and frontends



Technical implications of distribution

Performance

Resilience

Exception handling

> Everything fails all the time.
>
> — Werner Vogels, CTO -- Amazon Web Services

Unexpected failures in software systems are bound to happen. Instead of expending all our energies trying to prevent them from occurring, it is prudent to embrace and design for failure as well. Let's look at the scenario in the `AutoApprovalSaga` and identify things that can fail:

```
class AutoApprovalSaga {
    //...
    @SagaEventHandler(associationProperty = "lcApplicationId")
    public void on(ProductLegalityValidatedEvent event) {
        //..
        productLegalityValidated = true;
        if (productValueValidated && applicantValidated) {
               gateway.send(new ApproveLCApplicationCommand(lcApplicationId)); ①
        }
    }
}
```

① When dispatching commands, we have a few styles of interaction with the target system (in this case, the LC application bounded context):

- **Fire and forget**: This is the style we have used currently. This style works best when system scalability is a prime concern. On the flip side, this approach may not be the most reliable because we do not have definitive knowledge of the outcome.
- **Wait infinitely**: We wait infinitely for the dispatch and handling of the `ApproveLCApplicationCommand`.
- **Wait with timeout**: We wait for a certain amount of time before concluding that the handling of the command has likely failed.

Which interaction style should we use? While this decision appears to be a simple one, it has quite a few, far-reaching consequences:

- If command dispatching itself fails, the `CommandGateway#send` method will fail with an exception. Given that the CommandGateway is an infrastructure component, this will happen because of technical reasons (like network blips, etc.)
- If the command handling for the `ApproveLCApplicationCommand` fails, and we will not know about it because the `#send` method does not wait for handling to complete. One way to mitigate that problem is to wait for the command to be handled using the `CommandGateway#sendAndWait` method. However, this variation waits infinitely for the handler to complete — which can be a scalability concern.
- We can choose to only wait

Recovery

Automated recovery

Manual recovery

Compensating actions

Retries

Integration strategies

Testing strategies

Not yet finalized

Transitional architecture

Understanding the costs of distribution

Handling exceptions

Testing the Overall System

Compatibility

Non-technical implications of distribution

Team topologies

Tech stack