Introduction to Microservices

Vikash Verma

Agenda

Building Monolithic Applications

Marching Toward Monolithic Hell

Microservices – Tackling the Complexity

The Benefits of Microservices

The Drawbacks of Microservices

Get Started

What are Microservices ?

Rebranded SOA?

Benefits ?

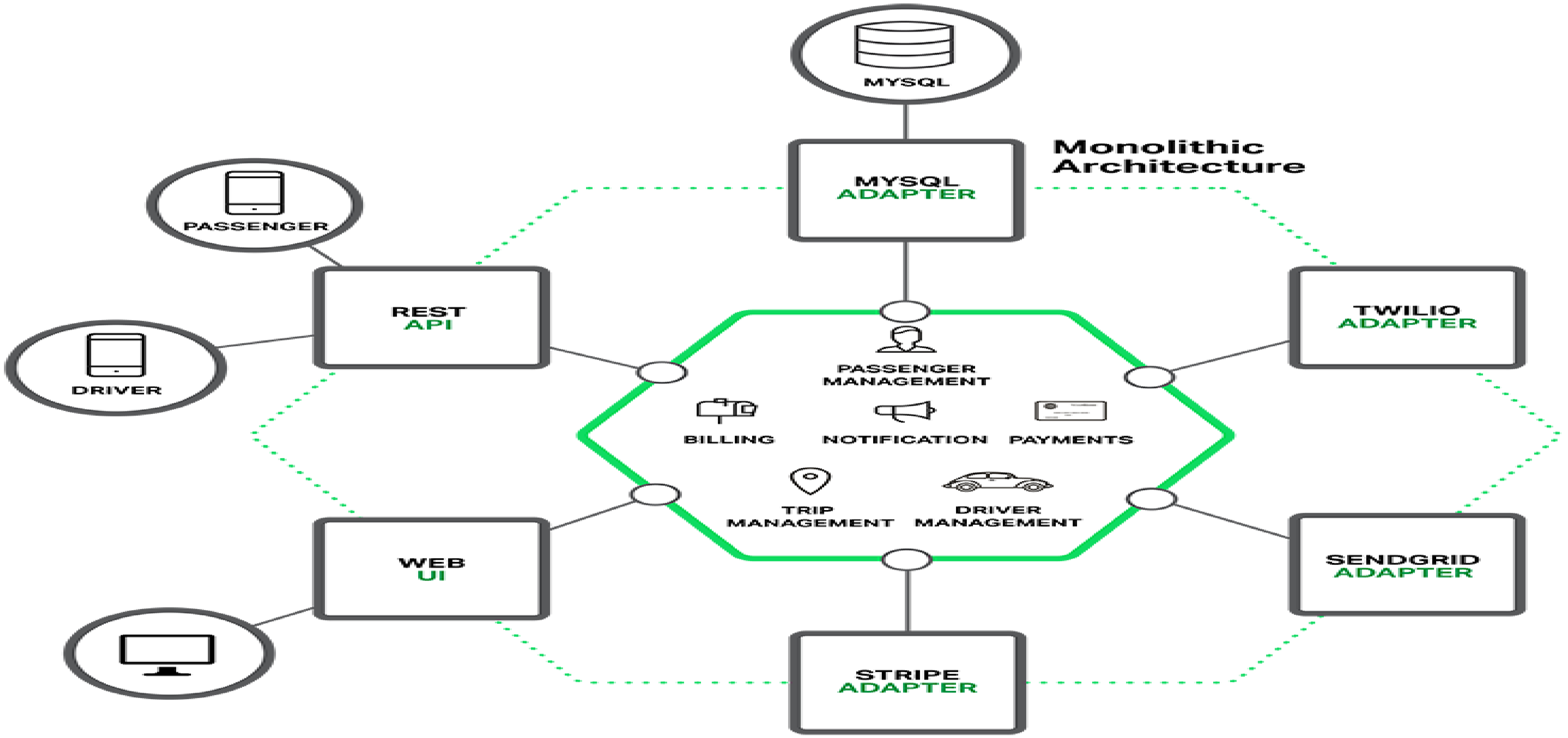
Building Monolithic Applications

Case Study: Taxi-hailing application intended to compete with Uber

After some preliminary meetings and requirements gathering, you would create a new project either manually or by using a generator that comes with a platform such as Rails, Spring Boot, Play, or Maven.

This new application would have a modular hexagonal architecture

Building Monolithic Applications



Building Monolithic Applications

Advantages

Adapters around business logic

Database adapters

Simple to develop since IDEs and other tools are focused on building a single application.

Simple to test.

Can implement end-to-end testing by simply launching the application and testing the UI with a testing package such as Selenium.

Monolithic applications are also simple to deploy. You just have to copy the packaged application to a server.

Can be scaled by running multiple copies behind a load balancer. In the early stages of the project it works well.

Marching Toward Monolithic Hell

The large monolithic code base intimidates developers, especially ones who are new to the team.

- The application can be difficult to understand and modify. As a result, development typically slows down.
- Since there are not hard module boundaries, modularity breaks down over time.
- It can be difficult to understand how to correctly implement a change the quality of the code declines over time. It’s a downwards spiral.

Overloaded IDE

- the larger the code base the slower the IDE and the less productive developers are.

Marching Toward Monolithic Hell

Continuous deployment is difficult

In order to update one component you have to redeploy the entire application.

This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems.

There is also the chance that components that haven't been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates.

This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.

Marching Toward Monolithic Hell

Scaling the application can be difficult –

A monolithic architecture is that it can only scale in one dimension.

It can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load.

Can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic.

Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently

Marching Toward Monolithic Hell

Obstacle to scaling development -

Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc.

The trouble with a monolithic application is that it prevents the teams from working independently.

The teams must coordinate their development efforts and redeployments.

It is much more difficult for a team to make a change and update production.

Marching Toward Monolithic Hell

Overloaded web container

The larger the application the longer it takes to start up.

Huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.

Requires a long-term commitment to a technology stack

A monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development .

Difficult to incrementally adopt a newer technology.

If your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It's possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.

Microservices – Tackling the Complexity

Used by Amazon, eBay, and Netflix

Split your application into set of smaller, interconnected services.

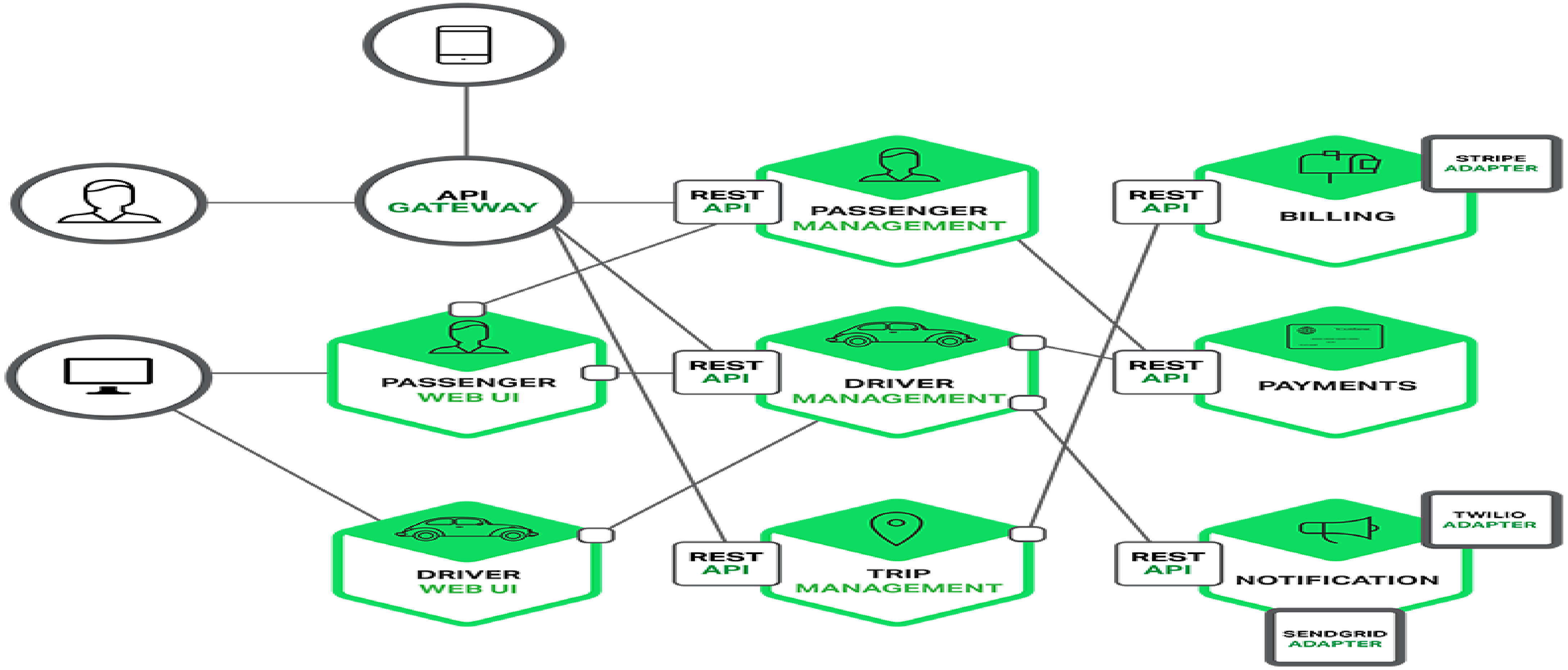
A service typically implements a set of distinct features or functionality, such as order management, customer management, etc.

Each microservice is a mini-application that has its own hexagonal architecture consisting of business logic along with various adapters.

Some microservices would expose an API that’s consumed by other microservices or by the application’s clients.

Other microservices might implement a web UI. At runtime, each instance is often a cloud virtual machine (VM) or a Docker container.

Microservices – Tackling the Complexity



Microservices – Tackling the Complexity

Each functional area of the application is now implemented by its own microservice.

Moreover, the web application is split into a set of simpler web applications – such as one for passengers and one for drivers, in our taxi-hailing example.

This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases.

Each backend service exposes a REST API and most services consume APIs provided by other services. For example, Driver Management uses the Notification server to tell an available driver about a potential trip.

The UI services invoke the other services in order to render web pages.

Services might also use asynchronous, message-based communication.

Microservices – Tackling the Complexity

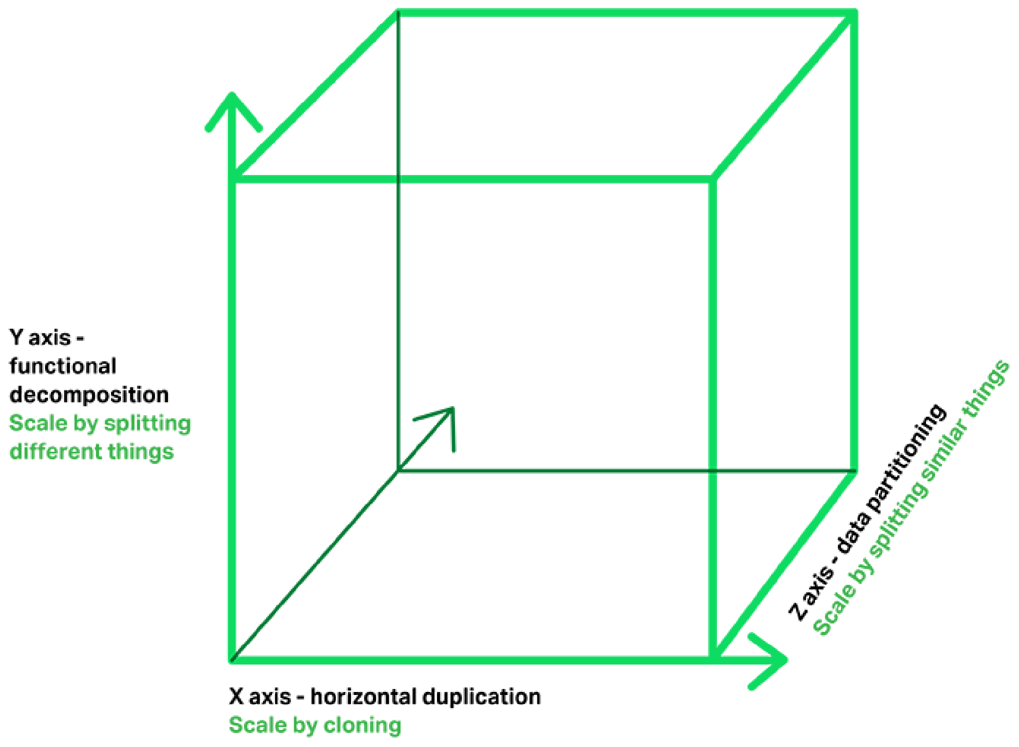
Some REST APIs are also exposed to the mobile apps used by the drivers and passengers. The apps don’t, however, have direct access to the backend services.

Instead, communication is mediated by an intermediary known as an API Gateway.

The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring.

Microservices – Tackling the Complexity

Where Microservices pattern fits in Scale Cube 3D model


Y axis - functional decomposition
Scale by splitting different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things

Microservices – Tackling the Complexity

X-axis scaling

X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application.

One drawback of this approach is that because each copy potentially accesses all of the data, caches require more memory to be effective. Another problem with this approach is that it does not tackle the problems of increasing development and application complexity.

Microservices – Tackling the Complexity

Y-axis scaling

- Y-axis axis scaling splits the application into multiple, different services.
- Each service is responsible for one or more closely related functions. There are a couple of different ways of decomposing the application into services.
  - *Verb-based decomposition and define services that implement a single use case such as checkout.*
  - *Noun based decomposition and create services responsible for all operations related to a particular entity such as customer management.*
  - *An application might use a combination of verb-based and noun-based decomposition.*

Microservices – Tackling the Complexity

Z-axis scaling

Each server runs an identical copy of the code.

Each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server.

For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity.

Z-axis splits are commonly used to scale databases. Data is partitioned (a.k.a. sharded) across a set of servers based on an attribute of each record

Z-axis scaling can also be applied to applications. In this example, the search service consists of a number of partitions. A router sends each content item to the appropriate partition, where it is indexed and stored. A query aggregator sends each query to all of the partitions, and combines the results from each of them.

Microservices – Tackling the Complexity

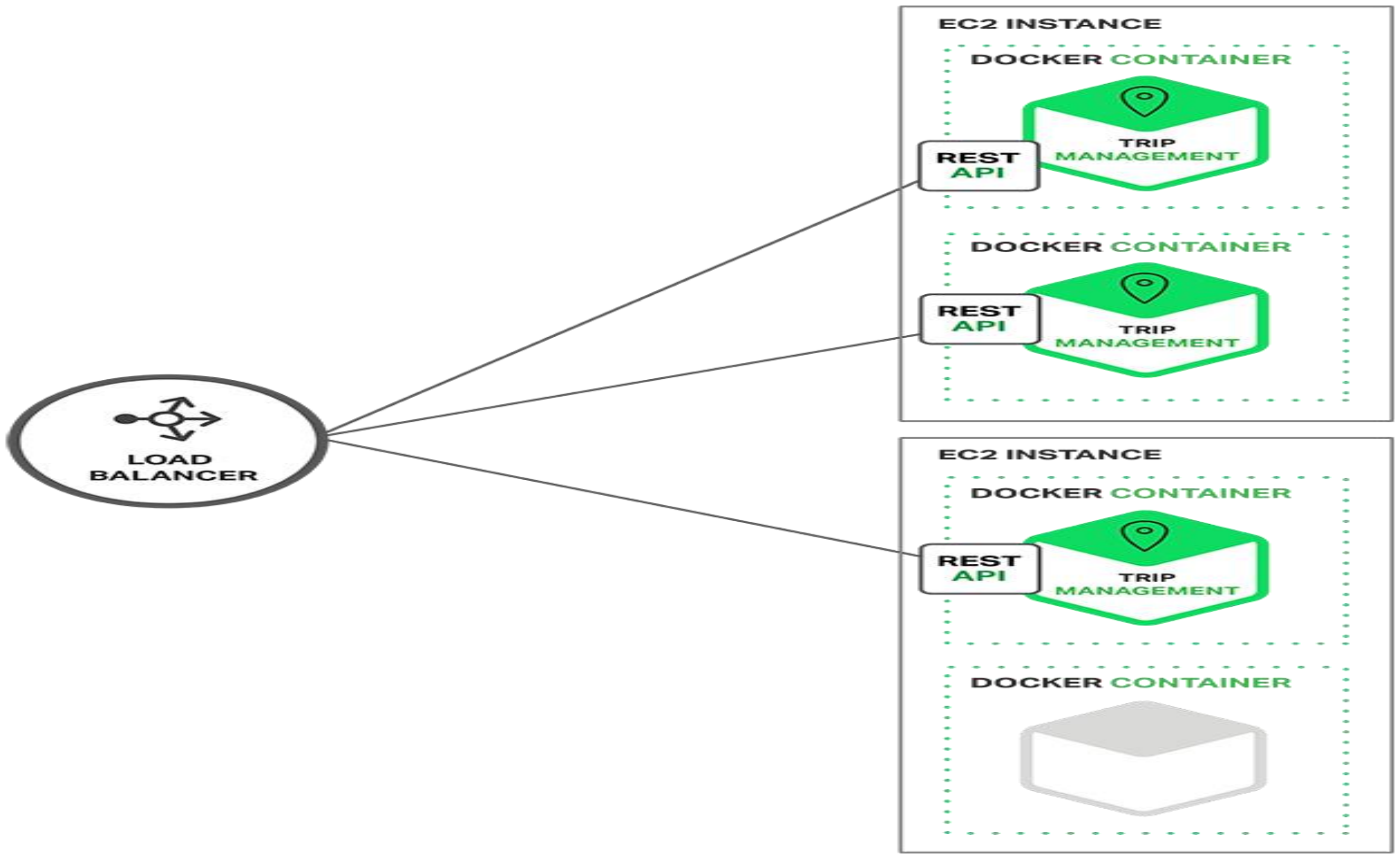
Z-axis scaling has a number of benefits.

- Each server only deals with a subset of the data.
- This improves cache utilization and reduces memory usage and I/O traffic.
- It also improves transaction scalability since requests are typically distributed across multiple servers.
- Also, Z-axis scaling improves fault isolation since a failure only makes part of the data in accessible.

Z-axis scaling has some drawbacks.

- One drawback is increased application complexity.
- We need to implement a partitioning scheme, which can be tricky especially if we ever need to repartition the data.
- Another drawback of Z-axis scaling is that doesn’t solve the problems of increasing development and application complexity. To solve those problems we need to apply Y-axis scaling.

Docker Solution



Docker Solution

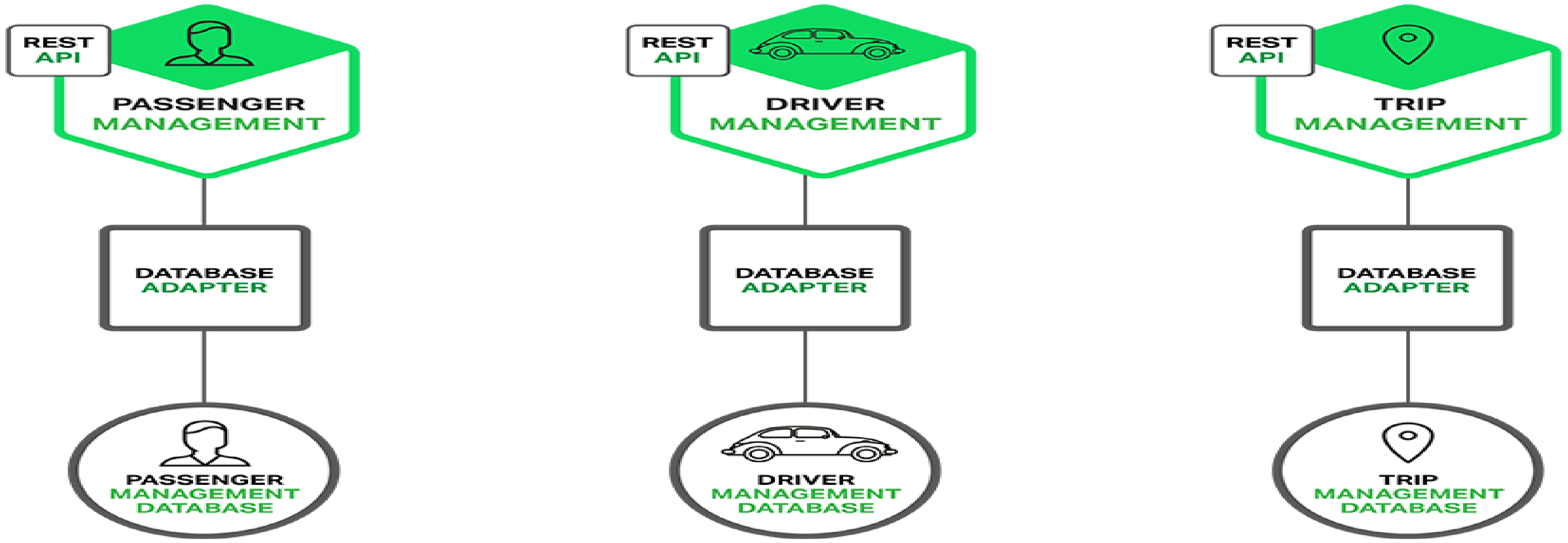
The Microservices Architecture pattern significantly impacts the relationship between the application and the database.

Rather than sharing a single database schema with other services, each service has its own database schema.

On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data.

However, having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling.

Docker Solution



Microservices Vs SOA

On the surface, the Microservices Architecture pattern is similar to SOA.

With both approaches, the architecture consists of a set of services.

However, one way to think about the Microservices Architecture pattern is that it’s SOA without the commercialization and perceived baggage of web service specifications (WS-*) and an Enterprise Service Bus (ESB).

Microservice-based applications favor simpler, lightweight protocols such as REST, rather than WS-*.

They also very much avoid using ESBs and instead implement ESB-like functionality in the microservices themselves.

The Microservices Architecture pattern also rejects other parts of SOA, such as the concept of a canonical schema for data access.

The Benefits of Microservices

It tackles the problem of complexity. By decomposing monstrous monolithic application into a set of services. While the total amount of functionality is unchanged, the application has been broken up into manageable chunks or services.

Each service has a well-defined boundary in the form of a remote procedure call (RPC)-driven or message-driven API.

The Microservices Architecture pattern enforces a level of modularity that in practice is extremely difficult to achieve with a monolithic code base.

Consequently, individual services are much faster to develop, and much easier to understand and maintain.

The Benefits of Microservices

This architecture enables each service to be developed independently by a team that is focused on that service.

The developers are free to choose whatever technologies make sense, provided that the service honors the API contract.

Of course, most organizations would want to avoid complete anarchy by limiting technology options.

However, this freedom means that developers are no longer obligated to use the possibly obsolete technologies that existed at the start of a new project.

When writing a new service, they have the option of using current technology.

Moreover, since services are relatively small, it becomes more feasible to rewrite an old service using current technology.

The Benefits of Microservices

Microservices Architecture pattern enables each microservice to be deployed independently.

Developers never need to coordinate the deployment of changes that are local to their service. These kinds of changes can be deployed as soon as they have been tested.

The UI team can, for example, perform A|B testing and rapidly iterate on UI changes.

The Microservices Architecture pattern makes continuous deployment possible.

The Benefits of Microservices

Microservices Architecture pattern enables each service to be scaled independently.

You can deploy just the number of instances of each service that satisfy its capacity and availability constraints.

Moreover, you can use the hardware that best matches a service's resource requirements. For example, you can deploy a CPU-intensive image processing service on EC2 Compute Optimized instances and deploy an in-memory database service on EC2 Memory-optimized instances.

The Drawbacks of Microservices

Complexity that arises from the fact that a microservices application is a distributed system.

Developers need to choose and implement an inter-process communication mechanism based on either messaging or RPC. Moreover, they must also write code to handle partial failure, since the destination of a request might be slow or unavailable.

Another challenge with microservices is the partitioned database architecture.

Business transactions that update multiple business entities are fairly common. These kinds of transactions are trivial to implement in a monolithic application because there is a single database. In a microservices-based application, however, you need to update multiple databases owned by different services.

The Drawbacks of Microservices

Using distributed transactions is usually not an option, and not only because of the CAP theorem.

They simply are not supported by many of today's highly scalable NoSQL databases and messaging brokers.

You end up having to use an eventual consistency-based approach, which is more challenging for developers.

Testing a microservices application is also much more complex

- Will require launching many services

The Drawbacks of Microservices

Another major challenge with the Microservices Architecture pattern is implementing changes that span multiple services.

- Say A -> B -> C

Deploying a microservices-based application is also much more complex. A monolithic application is simply deployed on a set of identical servers behind a traditional load balancer.

Each application instance is configured with the locations (host and ports) of infrastructure services such as the database and a message broker.

In contrast, a microservice application typically consists of a large number of services.

- E.g. Hailo has 160 different services and Netflix has more than 600

The Drawbacks of Microservices

Each service will have multiple runtime instances.

That’s many more moving parts that need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism that enables a service to discover the locations (hosts and ports) of any other services it needs to communicate with.

Consequently, successfully deploying a microservices application requires greater control of deployment methods by developers and a high level of automation.

Thank You