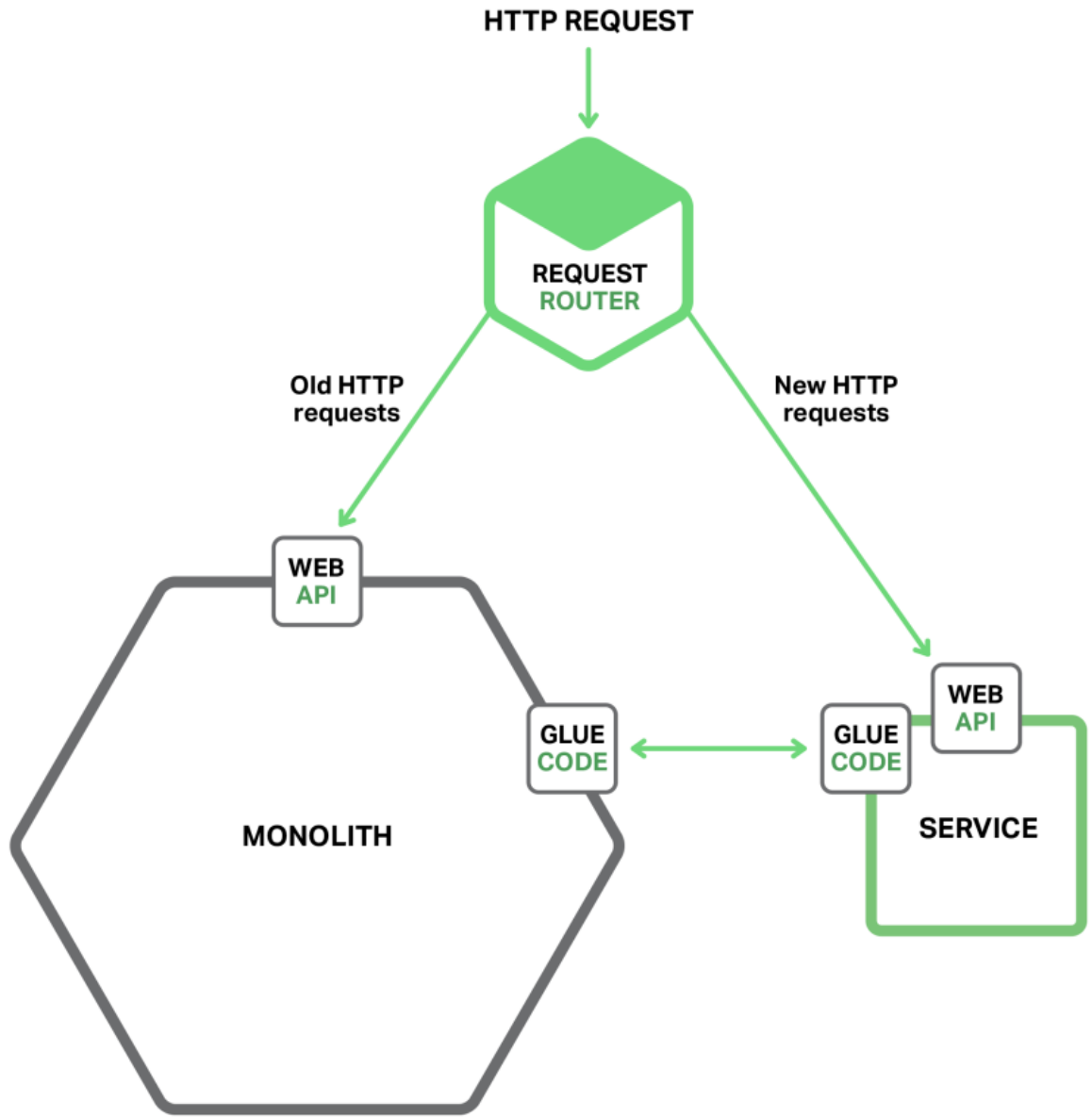
Strategy 1 – Stop Digging

The *Law of Holes* says that whenever you are in a hole we should stop digging.

When the monolithic application has become unmanageable, we should stop making the monolith bigger.

This means that when we are implementing new functionality , we should not add more code to the monolith. Instead, the big idea with this strategy is to put that new code in a standalone microservice


HTTP REQUEST
REQUEST
ROUTER
Old HTTP
requests
New HTTP
requests
WEB
API
GLUE
CODE
MONOLITH
GLUE
CODE
WEB
API
SERVICE

The “glue code” integrates the service with the monolith.

A service rarely exists in isolation and often needs to access data owned by the monolith.

The glue code, which resides in either the monolith, the service, or both, is responsible for the data integration.

The service uses the glue code to read and write data owned by the monolith.

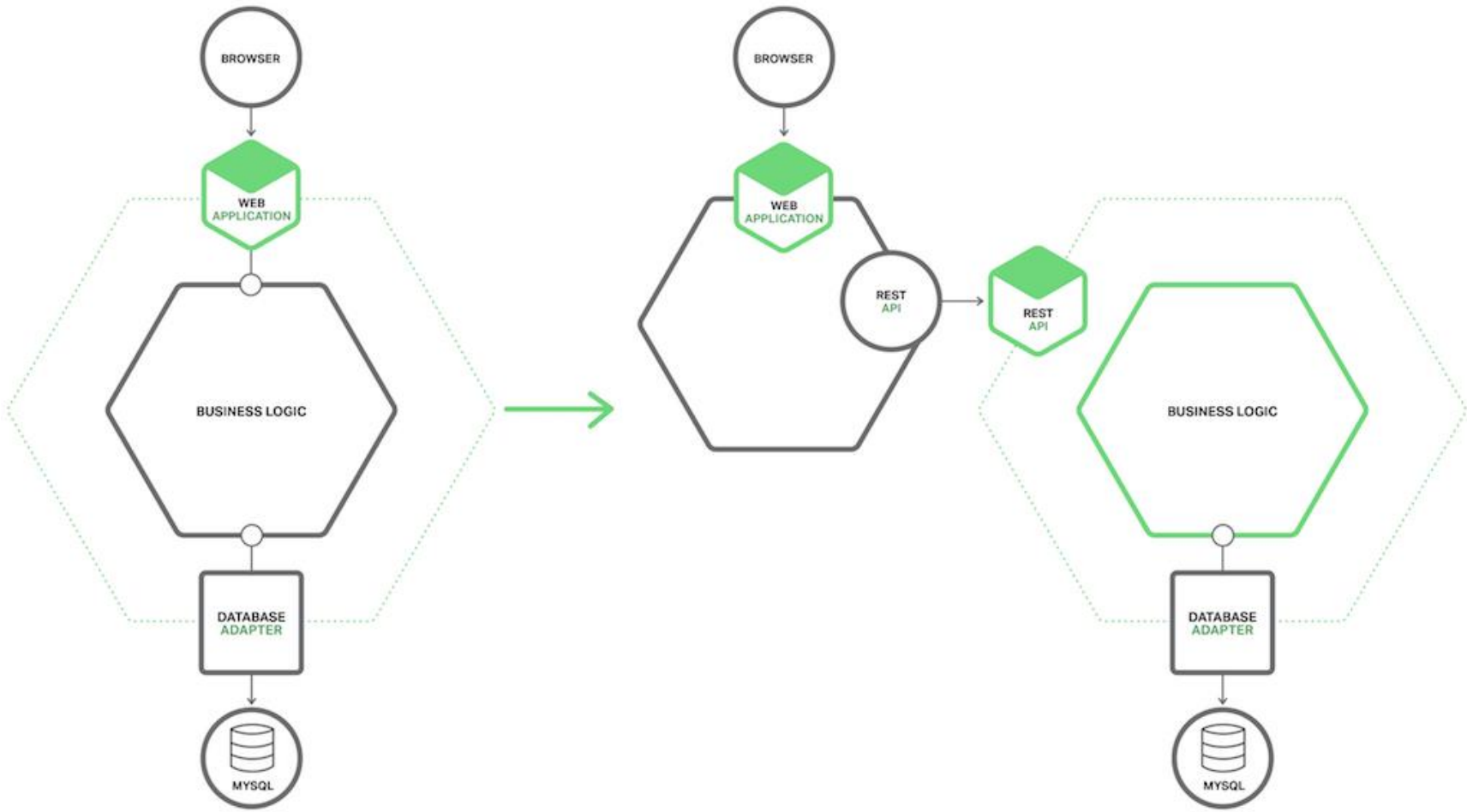
There are three strategies that a service can use to access the monolith’s data:

- ❑ Invoke a remote API provided by the monolith
- ❑ Access the monolith’s database directly
- ❑ Maintain its own copy of the data, which is synchronized with the monolith’s database

Strategy 2 – Split Frontend and Backend

A strategy that shrinks the monolithic application is to split the presentation layer from the business logic and data access layers.

A typical enterprise application consists of at least three different types of components: Presentation layer,Business logic layer & Data-access layer


BROWSER
WEB
APPLICATION
BUSINESS LOGIC
DATABASE
ADAPTER
MYSQL
BROWSER
WEB
APPLICATION
REST
API
REST
API
BUSINESS LOGIC
DATABASE
ADAPTER
MYSQL

Strategy 3 – Extract Services

The third refactoring strategy is to turn existing modules within the monolith into standalone microservices.

Each time you extract a module and turn it into a service, the monolith shrinks.

Once you have converted enough modules, the monolith will cease to be a problem. Either it disappears entirely or it becomes small enough that it is just another service.

Prioritizing Which Modules to Convert into Services

A good approach is to start with a few modules that are easy to extract. This will give you experience with microservices in general and the extraction process in particular.

After that we should extract those modules that will give you the greatest benefit.

How to Extract a Module

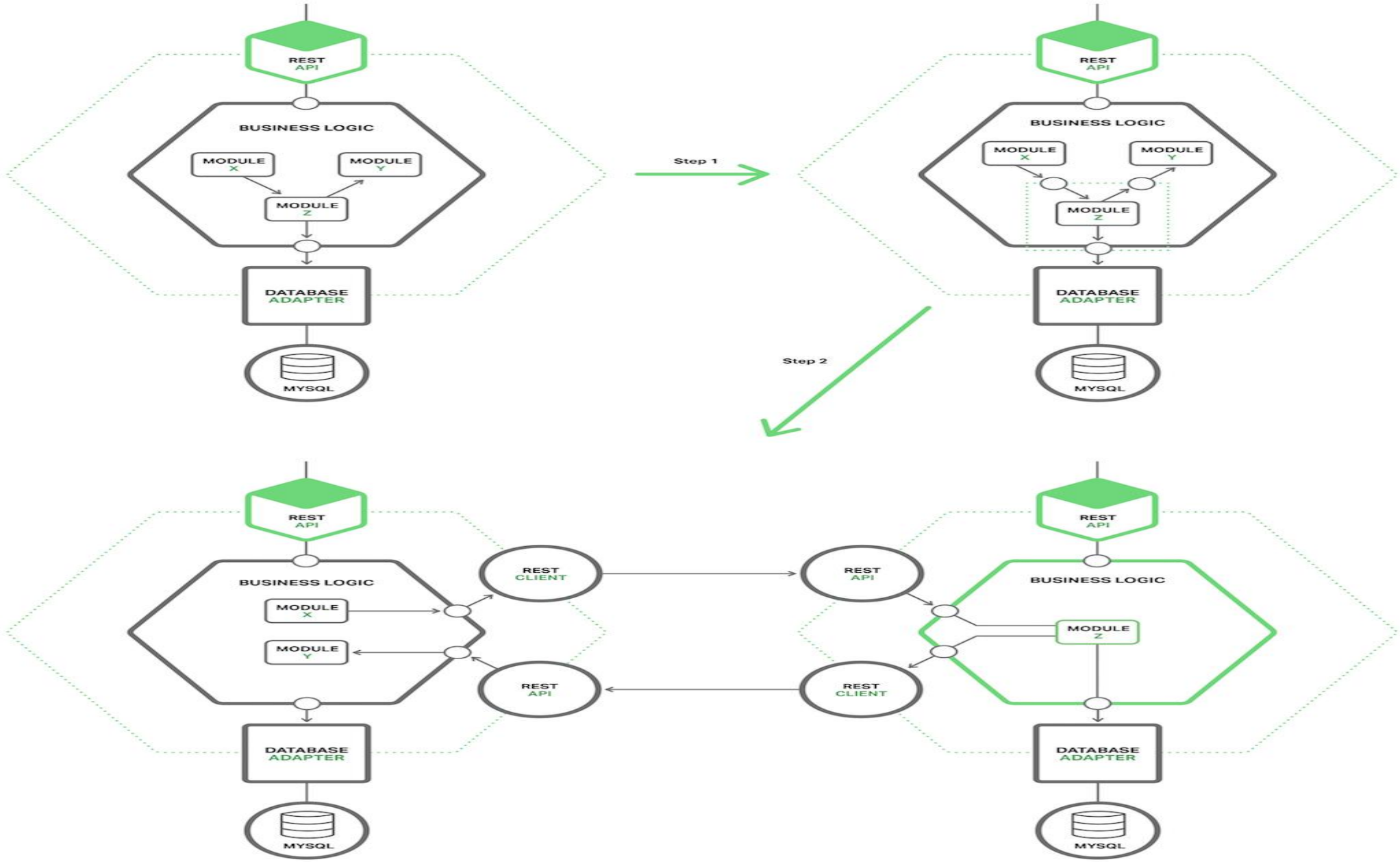
The first step of extracting a module is to define a coarse-grained interface between the module and the monolith

It is mostly likely a bidirectional API, since the monolith will need data owned by the service and vice versa.

It is often challenging to implement such an API because of the tangled dependencies and fine-grained interaction patterns between the module and the rest of the application.

Business logic implemented using the Domain Model pattern is especially challenging to refactor because of numerous associations between domain model classes

Once we implement the coarse-grained interface, we then turn the module into a free-standing service. To do that, we must write code to enable the monolith and the service to communicate through an API that uses an inter-process communication (IPC) mechanism.


REST
API
BUSINESS LOGIC
MODULE
X
MODULE
Y
MODULE
Z
DATABASE
ADAPTER
MYSQL
Step 1
Step 2
REST
CLIENT

In this example, Module Z is the candidate module to extract.
Its components are used by Module X and it uses Module Y.

The first refactoring step is to define a pair of coarse-grained APIs.

The first interface is an inbound interface that is used by Module X to invoke Module Z. The second is an outbound interface used by Module Z to invoke Module Y.

The second refactoring step turns the module into a standalone service. The inbound and outbound interfaces are implemented by code that uses an IPC mechanism.