Appendix C

Catalogue of refactorings: A guide for the migration towards a microservices architecture

The following catalogue corresponds to the full catalogue addressed in Chapter 5.

This catalogue guides the refactoring process toward a microservices architecture. It contains large-scale refactorings, the smaller-scale refactorings that compose it, and the steps to implement the smaller-scale ones, like a recipe to follow, touching very different topics of the migration process across eight different chapters:

- Chapter 1 (Appendix C.1) describes refactoring sequences for breaking existing dependencies between microservices.
- Chapter 2 (Appendix C.2) outlines infrastructure changes needed for a microservices design.
- Chapter 3 (Appendix C.3) explores microservice deployment and orchestration strategies.
- Chapter 4 (Appendix C.4) describes the properties of microservices and how the refactorings in this catalogue help to achieve them.
- Chapter 5 (Appendix C.5) outlines different refactoring techniques that lead to intermediate states of refactoring when we have constraints and limitations during our refactoring process.
- Chapter 6 (Appendix C.6) contains sequences of common refactorings.
- Chapter 7 (Appendix C.7) tackles additional challenges when transferring a system to a microservices design.
- Chapter 8 (Appendix C.8) discusses how to prepare for the migration process.

The refactorings are presented using a structure similar to the one proposed by Martin Fowler and Kent Beck in their 2nd edition of the book *"Refactoring: Improving the Design of Existing Code"* [33]. They are composed of four sections:

- **Title** corresponds to the name of the refactoring. This name is designed to be as self-explanatory as in any code refactoring naming.
- **Motivation** describes a scenario, the set of observed conditions that may lead to the need to apply the refactoring. We can see it as the driving cause.
- **Mechanics**, presents a step-by-step guide of the refactoring implementation.
- **Example** exemplifies the refactoring application, either by using some code snippets, descriptions or schemas.

C.1 Breaking Dependencies

To separate the system into microservices, we need to identify the dependencies between the clusters of classes that will make our intended microservices and "break" (cf. Section 5.1) the dependencies between those clusters so that the microservices can function independently.

We can consider multiple types of dependencies. This catalogue chapter (Appendix C.1) describes how to break the different dependencies and how the code should be refactored to achieve that. It is common to find these dependencies types together, so some refactorings described below may link to others and form a sequence of refactorings to solve the dependencies between microservices.

C.1.1 Change Local Method Call Dependency to a Service Call

Motivation

When splitting the monolith into microservices, it is prevalent to have code dependencies in the form of a method invocation between components. As the classes in question will become part of different services, method invocations often have to be refactored to service calls in the microservices architecture. This service call should be made with a protocol that can be synchronous or asynchronous, depending on the requirements and goals.

When we do not want a service waiting for the other to respond and an instant response is not required because we do not need that information immediately, asynchronous calls are usually a good option. This is the case when we accept having eventual consistency when it comes to data operations. It is especially common when a service call triggers other service calls, and we do not want the first service to be busy waiting for these calls to complete. It helps in scalability and availability. It is common to implement it through an asynchronous remote procedure call or messaging with a publisher/subscriber protocol, where services publish messages to a message broker, and the subscriber consumes the messages when available to process them.

Asynchronous calls are not the best solution to ensure we are reading accurate data and need consistency in the moment of the action, as consistency in asynchronous calls is eventual; in this case, a synchronous request/response protocol is usually preferred.

Making this solution asynchronous allows for better scalability and fault tolerance as the services become decoupled.

Mechanics

1. Decide the communication strategy and make the initial configurations to use it.
   (a) Synchronous (using strategies like REST or RPC, for example)
      i. Store the necessary information (e.g. URL) to make the remote calls to the microservice.
   (b) Asynchronous (with technologies like Mosquitto, RabbitMQ, Apache Kafka, etc.): using strategies like Event Sourcing or some form of asynchronous RPC.
      i. Set up a message broker/event bus.
      ii. Create a topic.
2. Configure the microservice that makes the invocation to use the communication strategy.
   (a) Synchronous - Change the method calls to and from local components to be remote calls to a different service:
      i. Create an interface with the declaration of the identified methods.
      ii. Create a class that implements that interface and makes the service calls, a Request Class.
   (b) Asynchronous
      i. This microservice will act like a subscriber, so make it subscribe to the topic you have created.
3. Configure the microservice that has the method to use the communication strategy.
   (a) Synchronous - arrange the microservice owning the method to respond to this communication protocol, creating an API to respond to the service calls. Example:
      i. Create a class that defines the resource paths for the requests and processes them producing a response.
      ii. Add methods to the class to perform the actions required by the service calls.
   (b) Asynchronous
      i. This microservice will act like a publisher, so you make it push the messages it wants to communicate in the topic created.

Note: Make sure to guarantee fault tolerance (check **Chapter 7** (**Appendix C.7**)).

Example

Let us imagine we have in the monolith a set of classes for managing orders (candidate to become a new *OrderManagement* microservice) and another set of classes for managing inventory (candidate to become a new *Inventory* microservice). The former includes a class *OrderProcessor* that makes a local method call to the *updateInventory* method defined by the class *InventoryService* of the latter, as shown in Listing C.1.

```
1  public class OrderProcessor {
2      private final InventoryService inventoryService;
3
4      public OrderProcessor(InventoryService inventoryService) {
5          this.inventoryService = inventoryService;
6      }
7
8      public void processOrder(Order order) {
9          inventoryService.updateInventory(order);
10         // Other processing logic
11     }
12 }
13
14 public class InventoryService {
15     private final InventoryManager inventoryManager;
16
17     public InventoryService(InventoryManager inventoryManager) {
18         this.inventoryManager = inventoryManager;
19     }
20
21     public void updateInventory(Order order) {
22         inventoryManager.updateInventory(order);
23     }
24 }
```

Listing C.1: The *OrderProcessor* class from proposed *OrderManagement* microservice makes a local method call to the method *updateInventory* of the *InventoryManager* call of the proposed *Inventory* microservice.

In Listing C.1, we can see that the *OrderProcessor* class performs a local call to the method of *InventoryManager*, and as we want to create two separate microservice, we need to break this dependency. For that, the method calls should become remote service calls. An example for a synchronous solution using REST is shown in Listing C.2.

```
1 //OrderManagement Microservice
2 public interface InventoryService {
3     void updateInventory(Order order);
4 }
```

```
5
6  @Service
7  public class RemoteInventoryService implements InventoryService {
8      private final RestTemplate restTemplate;
9      private final String inventoryServiceUrl;
10
11     public RemoteInventoryService(RestTemplate restTemplate, @Value("${inventory.
           service.url}") String inventoryServiceUrl) {
12         this.restTemplate = restTemplate;
13         this.inventoryServiceUrl = inventoryServiceUrl;
14     }
15
16     @Override
17     public void updateInventory(Order order) {
18         String url = inventoryServiceUrl + "/update";
19         restTemplate.postForObject(url, order, Void.class);
20     }
21 }
22
23 public class OrderProcessor {
24     private final InventoryService inventoryService;
25
26     public OrderProcessor(InventoryService inventoryService) {
27         this.inventoryService = inventoryService;
28     }
29
30     public void processOrder(Order order) {
31         inventoryService.updateInventory(order);
32         // Other processing logic
33     }
34 }
35
36 // Inventory Microservice
37 @RestController
38 @RequestMapping("/api/inventory")
39 public class InventoryController {
40     @PostMapping("/update")
41     public ResponseEntity<Void> updateInventory(@RequestBody Order order) {
42         // Update inventory logic
43         // ...
44         return ResponseEntity.ok().build();
45     }
46 }
```

Listing C.2: *OrderProcessor* uses the *InventoryService* interface to make a synchronous service call to the *Inventory* service using REST.

In the *OrderManagement* microservice, we created the *InventoryService* interface that defined the *updateInventory* method and created the implementation of this interface, the *RemoteInven-*

toryService class. This class uses *RestTemplate* to send a POST request to the inventory service URL.

The *OrderProcessor* class remains the same, although now it depends on the *InventoryService* interface for updating the inventory.

In the *Inventory* microservice, we created the *InventoryController* class that handles the HTTP POST request for updating the inventory. The *updateInventory* method contains the logic to update the inventory based on the received order.

This refactoring changes the direct local method call dependency to a synchronous RESTful API call, allowing the *OrderManagement* microservice to communicate with the *Inventory* microservice via remote synchronous service calls using HTTP requests.

The asynchronous solution using *Apache Kafka* is shown in Listing C.3.

```
// OrderManagement microservice
    public class OrderEvent {
    private Order order;

    // Constructors, getters, and setters
}

public class OrderUpdatedEvent {
    private Order order;

    // Constructors, getters, and setters
}


@Component
public class KafkaEventProducer {
    private final KafkaTemplate<String, OrderEvent> kafkaTemplate;
    private final String topic;

    public KafkaEventProducer(KafkaTemplate<String, OrderEvent> kafkaTemplate,
        @Value("${kafka.topic}") String topic) {
        this.kafkaTemplate = kafkaTemplate;
        this.topic = topic;
    }

    public void publishOrderEvent(OrderEvent orderEvent) {
        kafkaTemplate.send(topic, orderEvent);
    }
}

@Component
public class OrderEventListener {

    private final InventoryService inventoryService;
```

```
34
35      public OrderEventListener(InventoryService inventoryService) {
36          this.inventoryService = inventoryService;
37      }
38
39      @KafkaListener(topics = "${kafka.topic}")
40      public void handleOrderEvent(OrderEvent orderEvent) {
41          inventoryService.updateInventory(orderEvent.getOrder());
42      }
43  }
44
45  public class OrderProcessor {
46      private final KafkaEventProducer kafkaEventProducer;
47
48      public OrderProcessor(KafkaEventProducer kafkaEventProducer) {
49          this.kafkaEventProducer = kafkaEventProducer;
50      }
51
52      public void processOrder(Order order) {
53          OrderEvent orderEvent = new OrderEvent(order);
54          kafkaEventProducer.publishOrderEvent(orderEvent);
55          // Other processing logic
56      }
57  }
58
59  // Inventory microservice
60  @Service
61  public class InventoryService {
62      public void updateInventory(Order order) {
63          // Update inventory logic
64          // ...
65      }
66  }
```

Listing C.3: *OrderProcessor* uses the *InventoryService* interface to make an asynchronous service call to the *Inventory* service using *Apache Kafka* and Events.

We created the event class, *OrderEvent*. We then created a *Kafka* event producer, implemented an event listener to process the order events and updated the *OrderProcessor* to use the *Kafka* event producer.

The *Kafka Event Producer* publishes the *OrderEvent* to the *Kafka* option, while the *OrderEventListener* listens to the *Kafka* topic and invokes the *updateInventory* method of the *Inventory* service, to perform the actual update. This service contains the logic to update the inventory.

The *OrderProcessor* class starts using the *Kafka Event Producer* to publish the *OrderEvent* so the events can be handled asynchronously instead of directly calling the *Inventory* service.

By making this solution asynchronous, it allows for better scalability and fault tolerance as the services become decoupled.

C.1.2 Move Foreign-key relationship to code

Motivation

When two entities are related and dependent on one another, their relationship is frequently characterised by a foreign-key relationship between the database tables representing each entity. One-to-one, Many-to-One, and One-to-Many relationships are possible.

When we extract a service and realise that these entities should be in different microservices, the database tables that describe them should belong to different database schemas, as each microservice should have its database.

Because one service will own the table containing the foreign key and another will own the table from which the foreign key comes, we must break the database and eliminate the foreign-key relationship. Therefore, as the constraint is no longer in the database, we must move this relationship to the code itself.

We have to keep in mind that sometimes the migration needs to carry more information than a single database table. When it happens, it is not always clear that those dependencies need to be migrated. Besides, integration layers can sometimes be required for functional decoupling.

Mechanics

The following steps can either be performed after breaking the code or with the code breakage in mind. Whenever services are mentioned, they are mentioned as what will be the end service division, but they do not have to be already implemented.

1. Remove the foreign-key constraint from the table that stores it.

2. In the class of entity (database table) that used to have the foreign-key constraint, create an instance variable that represents the other entity involved in the said relationship and create a column for that variable in this entity table. This variable will no longer be a foreign key but a filter of the select query to retrieve data.

3. Separate the tables into the databases of the different owners (at this moment, this might be more conceptual, but in the future, this will represent the databases of the different microservices).

4. Create an interface for each of these databases that implements the methods of data manipulation.

5. Identify the methods that use/manipulate data from different databases and change them to use the newly created interfaces.

6. When you separate the services, don't forget to use the previous refactoring to "Change local method call dependency to a service call"(C.1.1) to change these local methods to service calls using the primary key as a parameter.

Note: We may need to remove code annotations when using specific programming languages, frameworks or ORMs that use it. The way we join the information of these two entities is no longer through a join query, so data consistency has to be a concern. Do not forget to implement mechanisms to guarantee data integrity and consistency (check **Chapter 7 (Appendix C.7)**). Additionally, we should be aware that the latency of requests increases as we transform the database calls into service calls.

Example

Suppose we have two entities, *Order* and *Customer*, with a foreign-key relationship where an *Order* belongs to a *Customer*. The *Order* entity has a *ManyToOne* relationship with the entity *Customer*, using the *customer_id* foreign-key column for the association, as can be seen in Listing C.4.

```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;

    // Other properties, getters, and setters
}

@Entity
@Table(name = "customers")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other properties, getters, and setters
}

@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

```

```
33      public void processOrder(Order order) {
34          // Perform business logic
35
36          Customer customer = order.getCustomer();
37          // Use customer data for processing
38
39          orderRepository.save(order);
40      }
41  }
42
43  @Repository
44  public interface OrderRepository extends JpaRepository<Order, Long> {
45      // Order-related methods for data manipulation
46  }
```

Listing C.4: *Order* has a foreign-key constraint with *Customer*

The *OrderService* class depends on the *OrderRepository* class for data access and manipulation. In the *processOrder* method, the *Customer* entity is accessed directly through the *getCustomer* method.

Listing C.5 shows the code after applying the refactoring.

```
1  @Entity
2  @Table(name = "orders")
3  public class Order {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private Long id;
7
8      private Long customerId;
9
10     // Other properties, getters, and setters
11 }
```

Listing C.5: The foreign key constraint between *Order* and *Customer* is now in the code.

We removed the foreign-key constraint from the *Order* table referencing the *Customer* table. In the *Order* class, we created an instance variable *customerId* to represent the association with the *Customer* entity. In the *Order* table, we added a column for the *customerId*. Each table goes to a separate database; the *Order* table goes to *orders_db*, and the *Customer* table goes to *customers_db*.

We created the interfaces for data manipulation. *OrderRepository* defines the methods for manipulating the *Order* entity in the *orders_db* database. *CustomerRepository* defines the methods for manipulating the *Customer* entity in the *customers_db* database. We identified that the method *processOrder* interacted with both entities, so we modified it to use the respective interfaces.

C.1.3 Replicate Data Across Microservices

Motivation

Sometimes, different microservices need the same data. Still, if they access the same data source, they won't be totally independent, as one microservice will also be managing the data of another.

To solve this, each service can have its own dedicated database with replication to the shared data source. Ideally, in a microservices architecture, this should occur with no distributed transaction, assuming eventual consistency; however, synchronous replication is also possible. One of the services is the data owner and source of truth. The other simply holds a copy of the data it needs to access and manipulate.

Mechanics

1. Split the database by deciding which service will be the owner of the shared data.
2. There are a few strategies to replicate the data:
   (a) Using mechanisms of the database engine, you can create one or more replication channels between it and the shared data source.
   (b) Using event sourcing, a method of storing (or communicating) data which facilitates data replication because events may be easily repeated. It is a way to keep eventual consistency. It holds events that are frequently objects, and because event sourcing does not need to know its consumers, other technologies can be utilised concurrently (for more on event sourcing, check Martin Fowler's article here).
   (c) Using "Change Data Capture" refactoring (check refactoring C.5.12)

Example

In this example, we will show how to use Event Sourcing to replicate data across microservices.

We will use the example of the entity *Order* containing a relationship with the entity *User* of the type ManyToOne and these two entities will belong to different microservices. Since the entity *User* needs to know its order, one solution is to replicate the data of the entity *Order* for the *User* using Event Sourcing.

To apply event sourcing, we have to identify the entity for data replication, which is the *Order* in this case; implement event sourcing; publish events; subscribe to events and update the local data. Listing C.6 shows the *OrderManagement* microservice code to implement event sourcing.

```
1 @Service
2 public class OrderService {
3     private final EventPublisher eventPublisher;
4     private final OrderRepository orderRepository;
5
```

```
6       public OrderService(EventPublisher eventPublisher, OrderRepository
            orderRepository) {
7           this.eventPublisher = eventPublisher;
8           this.orderRepository = orderRepository;
9       }
10
11      public Order createOrder(OrderDTO orderDTO) {
12          // Create the order entity
13          Order order = new Order(orderDTO);
14
15          // Save the order in the repository
16          Order savedOrder = orderRepository.save(order);
17
18          // Publish the order created event
19          OrderCreatedEvent event = new OrderCreatedEvent(savedOrder.getId());
20          eventPublisher.publish(event);
21
22          return savedOrder;
23      }
24
25      // Other methods for updating and deleting orders
26  }
```

Listing C.6: The *OrderManagement* microservice publishes *OrderCreatedEvent* events every time a new order is created

The *OrderManagement* microservice is responsible for creating and managing orders, so every time a new order is created, it publishes an *OrderCreatedEvent* that contains the newly created order using the *EventPublisher*. The microservices interested in replicating the *Order* data can subscribe to the *OrderCreatedEvent* and update their local order records accordingly. Listing C.7 shows the code in the *User* side, the microservice that wants to replicate the data.

```
1   @Service
2   public class UserService {
3       private final EventSubscriber eventSubscriber;
4       private final UserRepository userRepository;
5
6       public UserService(EventSubscriber eventSubscriber, UserRepository
            userRepository) {
7           this.eventSubscriber = eventSubscriber;
8           this.userRepository = userRepository;
9           eventSubscriber.subscribe(OrderCreatedEvent.class, this::
                handleOrderCreatedEvent);
10      }
11
12      private void handleOrderCreatedEvent(OrderCreatedEvent event) {
13          // Retrieve the order details based on the event
```

```
14          Order order = getOrderDetails(event.getOrderId());
15
16          // Update the user associated with the order
17          User user = userRepository.findById(order.getUserId()).orElse(null);
18          if (user != null) {
19              user.addOrder(order);
20              userRepository.save(user);
21          }
22      }
23
24      private Order getOrderDetails(String orderId) {
25          // Retrieve the order details from the appropriate source (e.g., call the
                Order service)
26          // ...
27          return new Order(orderId, "User123", /* other order details */);
28      }
29
30      // Other methods for user management
31  }
```

Listing C.7: The *User* microservice listens to *OrderCreatedEvent* events to update its own data storage

As the *User* microservice is interested in replicating *Order* data, it subscribes to the *OrderCreatedEvent* using *EventSubscriber* and the *handleOrderCreatedEvent* method is invoked whenever a new order is created.

Inside this method, the *Order* details are retrieved based on the event. The associated *User* is fetched, and if it exists, the *Order* is added to its list of orders, and this way, it keeps the user's order data updated.

This way, each microservice can independently handle the received events and update its own data storage, ensuring data consistency.

C.1.4 Split Database Across Microservices

Motivation

When extracting a service, we commonly find that some monoliths aggregate in the same database table data that will further need to be split across different microservices, as different microservices access the same database table. Therefore, splitting a monolithic database is not so trivial.

When deciding to choose which service is going to be the owner of the data, we should keep in mind the business requirement and what makes sense for the system in our hands. Also, we should be able to clearly see which data fields remain master/slave on each service.

Mechanics

1. Separate into each microservice database only the tables that are only accessed/manipulated by that microservice.
2. Find the columns in the table used by the different newly defined microservices.
3. Analyse which is the case of each table and what solution better fits your system's requirements:
   (a) Two microservices access the same database table but do not update the same columns
      i. You can replicate the data for both microservices using **"Replicate Data Across Microservices"** (C.1.3) and use a data replication mechanism to keep it consistent or
      ii. Decide to which of these microservices each column belongs.
      iii. Split this table inside the monolith schema between the two tables belonging to the different components that will soon be microservices.
      iv. In each component, include the corresponding table and adapt the code to use that table.
      v. If the different microservices interact with what used to be foreign keys on the monolith schema, use the **"Move Foreign-key relationship to code"** refactoring (C.1.2).
   (b) Two microservices use the same database table and update the same columns
      i. You can replicate the data for both microservices using **"Replicate Data Across Microservices"** (C.1.3) and use a data replication mechanism to keep it consistent or
      ii. Decide which microservice should own this data.
      iii. Make the other microservice make a service call to update this column.
         A. To perform this incrementally, we can first make the change in the monolith to the soon microservice that does not own the data, make the update through a method call, and then, when creating the microservices, use the refactoring **"Change local method call dependency to a service call"** (C.1.1).

Note: Guarantee data consistency (check **Chapter 7 (Appendix C.7)**)

Example

We will present an example for option (b) of the Mechanics without data replication.

Suppose we have two microservices, the *Inventory* microservice and the *OrderManagement* microservice and that both use the same database table called *Product*. If first need to identify the microservice that should own the data, and in this case, we believe the *Inventory* microservice is more suitable. Then we need to define a service API for the *Inventory* microservices with methods

to update the specific columns related to inventory management. Then we must remove the direct database updates from *OrderManagement* microservice to the shared columns in the *Product* table and change them to make service calls to the API provided by *Inventory* microservice, whenever updates to the shared columns related to inventory management are required. Listing C.8 shows the code on the *Inventory* microservice side.

```
@RestController
@RequestMapping("/api/inventory")
public class InventoryController {
    private final InventoryService inventoryService;

    public InventoryController(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }

    @PutMapping("/product/{productId}/stock")
    public ResponseEntity<String> updateProductStock(@PathVariable("productId")
        Long productId, @RequestParam("stock") int stock) {
        // Call the service method to update the stock of the product
        inventoryService.updateProductStock(productId, stock);
        return ResponseEntity.ok("Product stock updated successfully");
    }
}
```

Listing C.8: *Inventory* microservice code - the owner of the data)

Inventory microservice owns the shared data related to product inventory and exposes and API endpoint *'api/inventory/product/productId/stock'* to update the stock of a product.

Listing C.9 shows the code on the *OrderManagement* microservice side.

```
@Service
public class OrderService {
    private final RestTemplate restTemplate;
    private final String inventoryServiceUrl = "http://localhost:8082/api/inventory
        /product";

    public OrderService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public void placeOrder(Order order) {
        // Perform order placement logic

        // Make a service call to the Inventory Service to update the product stock
        String updateUrl = String.format("%s/%s/stock?stock=%s",
            inventoryServiceUrl, order.getProductId(), order.getQuantity());
```

```
15          restTemplate.put(updateUrl, null);
16
17          // Continue with the remaining order placement logic
18      }
19  }
```

Listing C.9: *OrderManagement* microservice code - the service that uses data owned by Inventory microservice

The *OrderManagement* microservice is responsible for placing orders, and whenever it places an order, it makes a service call to the *Inventory* microservice API endpoint *'api/inventory/product/productId/stock'* to update the stock of the ordered product.

With this refactoring, the *Inventory* microservice has control over the inventory-related data, while the *OrderManagement* microservice interacts with the *Inventory* microservice through service calls to update the shared inventory columns. This way, each microservice focus on its specific responsibilities.

C.1.5 Create Data Transfer Object

Motivation

This refactoring is commonly necessary when we extract a service and there is a relationship between entities that will belong to different microservices. It suggests transferring more data in each call through a data transfer object that will hold all the call's data. Hence, this object will contain all the data that must be shared between the microservices to reduce the number of service calls performed, as service calls are expensive regarding latency.

It decouples presentation from the service layer and the domain model.

Note: This data transfer object must be serialisable to be sent through the connection.

Mechanics

1. Create an entity (Data Transfer Object) to hold the data necessary in a call between those services.

Example

In this example, we must create a Data Transfer Object to hold the information about an *Order* to be transferred between microservices during order-related communication. An example of the *Order* object being sent in the communication can be seen in Listing C.10.

```
1  @Entity
2  public class Order {
3      @Id
4      @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
5       private Long id;
6
7       private String customerName;
8
9       // Other fields and relationships
10
11      // Constructors, getters, and setters
12  }
13
14  @Service
15  public class OrderService {
16      private final OrderRepository orderRepository;
17
18      public OrderService(OrderRepository orderRepository) {
19          this.orderRepository = orderRepository;
20      }
21
22      public Order getOrderDetails(Long orderId) {
23          return orderRepository.findById(orderId);
24      }
25  }
```

Listing C.10: *Order* object is being sent through the communication channel.

In the *getOrderDetails* method from *Order* microservice class, an object of type Order is being sent through the communication. However, we want to create a Data Transfer Object that can hold the necessary data in a call to this method that contains more than information only present in the *Order* class. This way, the services will not have to share the same entity because we are encapsulating the specific data for communication, creating an abstraction. Listing C.11 shows the creation of a DTO for *Order*.

```
1   public class OrderDTO {
2       private Long orderId;
3       private String customerName;
4       private List<String> products;
5       // Other fields as needed
6
7       // Constructors, getters, and setters
8   }
9
10  @Service
11  public class OrderService {
12      private final OrderRepository orderRepository;
13
14      public OrderService(OrderRepository orderRepository) {
15          this.orderRepository = orderRepository;
16      }
```

```
17
18      public OrderDTO getOrderDetails(Long orderId) {
19          Order order = orderRepository.findById(orderId);
20          // Transform Order entity into OrderDTO
21          OrderDTO orderDTO = new OrderDTO();
22          orderDTO.setOrderId(order.getId());
23          orderDTO.setCustomerName(order.getCustomer().getName());
24          orderDTO.setProducts(order.getProducts().stream().map(Product::getName)
25              .collect(Collectors.toList()));
26          // Set other fields as needed
27
28          return orderDTO;
29      }
30  }
```

Listing C.11: Creation of a DTO to be sent through the communication

We defined a new class representing the DTO and declared the necessary fields to hold the data. In the future, more fields can be added to this DTO as they correspond to the transferred data. Then, we transform the data being transferred into the DTO. The *getOrderDetails* method in the *Order* microservice will no longer retrieve an *Order* object but a *OrderDTO* object.

The DTO provides a standard format for transferring the data of orders between services.

C.1.6 Break Data Type Dependency

Motivation

A data type dependency is common between different microservices. When we separate the microservices by business capabilities, some microservices might still need information about some entities in specific parts of their operations that are part of a different business capability. This dependency can appear on instance variables types, parameter types, return types and even method variables types.

We must identify where this data type is used and break this dependency to separate the microservices smoothly.

Mechanics

1. Identify where the data type is used (for example, the method invocations from the data type class, variable, parameters, or return types of the data type).
2. There are three ways of doing this:
   (a) Assuming it belongs only to the microservice where it was first defined:
      i. if there are method invocations:

A. Create an interface with the same name as the data type that defines the methods invocations identified for use through the data transfer object to make service calls to the data owner.

B. The method invocations will change from local calls to calls to the service that owns the data types and its methods, using the refactoring **"Change local method call dependency to a service call"** (C.1.1).

ii. The return types, variables and parameters shall use a Data Transfer Object to create the data type in the microservice because it will be sent through the service calls. Use the refactoring **“Create Data Transfer Object”** (C.1.5).

iii. Make the necessary changes in the code to use the new data type and the right interface for the method calls.

(b) Keep it in both microservices

i. Replicate the data type in both microservices and use event sourcing to ensure eventual consistency. Check the refactorings **"Change local method call dependency to a service call: asynchronous"** (C.1.1) and **"Replicate Data Across Microservices"** (C.1.3).

(c) Keep it in both microservices, but one of them is a proxy.

Example

We will focus this example on the first way of solving this, assuming it belongs only to the microservice where it was first defined.

Imagine we have two microservices *OrderManagement* microservice and *Inventory* microservice, where the *OrderManagement* microservice depends on a data type called *Product*. We want to break the dependency of the *Product* data type in the *OrderManagement* microservice and let the data type be owned by *Inventory* microservice. The *OrderManagement* microservice before the refactoring using the *Product* data type as a variable type, as can be seen in Listing C.12.

```
1  public class OrderService {
2      private ProductService productService;
3
4      public OrderService(ProductService productService) {
5          this.productService = productService;
6      }
7
8      public void createOrder(Order order) {
9          // Perform order creation logic
10
11         // Directly access the ProductService to get product information
12         Product product = productService.getProductById(order.getProductId());
13
14         // Use the product to complete the order creation process
```

```
15      }
16  }
```

Listing C.12: *OrderManagement* microservice before the refactoring

To resolve it, we create a *ProductInterface* that defines the necessary methods invocations to interact with *Product* data in the *Inventory* microservice. The *ProductService* implements this interface and makes the requests to the Inventory microservice that owns the data type *Product*. This way, We then replace the local method invocations in the *Order* service that involves the *Product* data type with calls to the *ProductService* interface, which will make service calls to the *Inventory* microservice to retrieve or manipulate the *Product* data.

We create a *ProductDTO* to use for transferring the *Product* data between the microservices through service calls, and we modify the return types, variables and parameters in the service's communications to use the DTO. Lastly, we update the *Order* service to use the new data type and the *ProductService* interface for method invocations. All changes performed to the *Inventory* microservice can be found in Listing C.13 and all changes performed to the *OrderManagement* microservice can be found in Listing C.14.

```
1  @Service
2  public class InventoryService {
3      public ProductDto getProductById(Long productId) {
4          // Logic to fetch product information from the inventory database or any
               other source
5          // ...
6
7          // Assume product information is retrieved and stored in the 'product'
               variable
8          ProductDto product = new ProductDto();
9          product.setId(productId);
10         product.setName("Example Product");
11         product.setPrice(BigDecimal.valueOf(9.99));
12
13         return product;
14     }
15 }
16
17 public class ProductDto {
18     private Long id;
19     private String name;
20     private BigDecimal price;
21
22     // Getters and setters
23 }
```

Listing C.13: *Inventory* microservice after the refactoring

```java
public class ProductDto {
    private Long id;
    private String name;
    private BigDecimal price;

    // Getters and setters
}

public interface ProductInterface {
    ProductDto getProductById(Long productId);
}
@Service
public class ProductService implements ProductInterface {
    private final RestTemplate restTemplate; // or any HTTP client

    public ProductService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public ProductDto getProductById(Long productId) {
        // Make an HTTP request to the InventoryService to fetch the product
        String inventoryServiceUrl = "http://inventory-service/api/products/" +
            productId;
        ResponseEntity<ProductDto> responseEntity = restTemplate.getForEntity(
            inventoryServiceUrl, ProductDto.class);
        return responseEntity.getBody();
    }
}

@Service
public class OrderService {
    private final ProductService productService;

    public OrderService(ProductService productService) {
        this.productService = productService;
    }

    public void createOrder(OrderDto orderDto) {
        // Process the order details
        // ...

        // Retrieve product information from the ProductService
        Long productId = orderDto.getProductId();
        ProductDto product = productService.getProductById(productId);

        // Perform other order-related operations
        // ...
    }
}
```

Listing C.14: *OrderManagement* microservice after the refactoring

C.1.7 Duplicate file Across Microservices

Motivation

When extracting a microservice from the monolith, it is common to find the need to have an interface or an abstract class in different microservices.

Mechanics

In this case, duplicating the file for each microservice is okay, as these classes do not handle business logic.

Example

Imagine you have a file called *Utils.java* that defines multiple functions useful for this domain but does not handle any business logic. If the microservice *Order* and the microservice *Inventory* both use multiple functions from that file, we can just simply add the *Utils.java* file to both microservices repositories. Suppose the microservice *Order* and the microservice *Inventory* both use multiple functions from that file. In that case, we can just simply add the *Utils.java* file to both microservices repositories.

C.2 Infrastructure Improvement

When we migrate a monolithic application into a microservices architecture, we must make one significant change to get the full advantage of this new architecture: improving the infrastructure. This chapter introduces the changes to implement to the system's infrastructure to accommodate the microservices paradigm.

C.2.1 Introduce the circuit breaker

Motivation

Services make requests to each other to collaborate. Our system should be able to gracefully handle faults with an unknown recovery time, bringing resilience and stability to the application. For example, when a service makes a synchronous call to another service and is unavailable, it may block waiting for a response if nothing is prepared. Therefore, the failure of one service can potentially cascade to other services.

Adding a circuit breaker will provide a wrapper over remote invocations and a barrier that will shut off additional attempts to invoke a remote service based on a set threshold, allowing the system to recover from problems rather than propagate them throughout the system.

Mechanics

1. Add a Circuit Breaker to the system (either implement it or use a third-party library).
2. Configure and parametrise the Circuit Breaker's thresholds so that when the number of consecutive failures crosses them, the circuit breaker trips and during the timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires, the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes regular operation. Otherwise, if there is a failure, the timeout period begins again.
3. Configure the service client to invoke a remote service via the circuit breaker.

Note: There are various ways to implement this refactoring. These steps explain that it can be done in a separate service, within an API Gateway, on the client or server side. On the client's side, it stops making the calls after a threshold of failing calls. The services implement the circuit breaker logic on the server's side, so the calls still hit the service.

Example

The schema in Figure C.1 represents an example of the implementation of a circuit breaker.

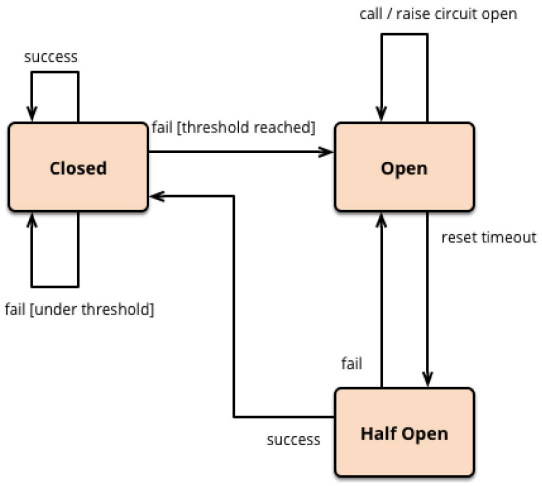


Figure C.1: Circuit Breaker implementation schema
From Fowler [66]

It starts in the close state. If the service is up and running, the circuit breaker passes the requests to services. However, when it fails to reach the service for a threshold of times, it enters the open state. In this state, the circuit breaker returns an error without processing the request. Additionally, it has a timeout setting that, once it is over, will make it enter the half-open state. In the half-open state, the circuit breaker tries once again to pass the calls to the service that was previously failing. If it fails again, it goes back to the open state. If it succeeds, it will return to the closed (default) state, where the calls are passed to the services.

C.2.2 Introduce service registry

Motivation

Each service can have one or more instances in production that can be modified dynamically. The introduction of a Service Registry enables services to locate one another dynamically.

Mechanics

1. Set up a service registry, which stores service instances' addresses.
2. Make each service register itself during the initiation.
3. You can unregister a service when they terminate or fail their health check.

Note: Implement this correctly, as failure to do so can create a single point of failure. Remember that the service registry directly impacts the system's availability; therefore, replication strategies should be implemented.

Example

A service requests the load balancer to communicate with a service. The load balancer queries the service registry for the existing instances of the requested service and load balance, choosing the endpoint of one of the less busy instances.

Figure C.2 demonstrates the communication between services using the load balancer and the service registry.

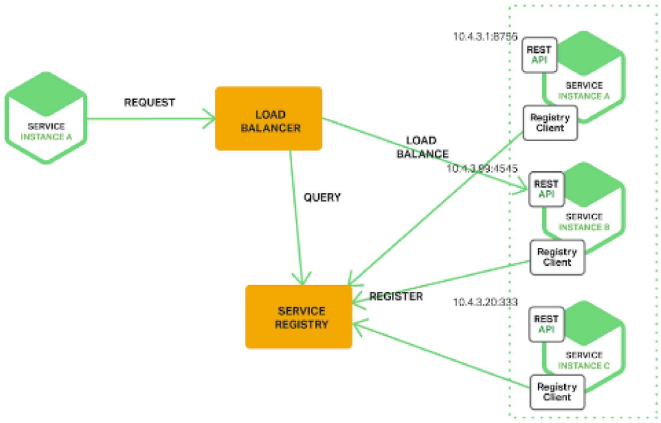


Figure C.2: Service Registry Implementation schema
Source: "Service Discovery in a Microservices Architecture", [17]

C.2.3 Introduce internal/external load balancer

Motivation

Assume you've already established a service registry. This service registry contains information about each operating microservice's many instances. It is feasible to balance the load on a service amongst its instances using this information. Its focus is to balance the load based on the client's requirements without needing an external load balancer.

Internal

Its focus is to balance the load based on the client's requirements without needing an external load balancer, allowing different load-balancing mechanisms to be used in different services.

Mechanics

1. Implement on each service an internal load balancer. This load balancer queries the service registry for the existing available instances for that service.
2. Make it balance the load between the available instances using the appropriate metrics for that service.

External

The goal is to balance the load with as few changes to the service code as possible and to create a centralised load-balancing approach for all services.

Mechanics

1. Set up an external load balancer that queries the service registry for the available instances of a given service and uses an algorithm to balance the load between the different services depending on the requests.
2. This load balancer can act as a proxy or an instance address (recommended).

Note: The load balancer can be built-in the service proxy, being the requests made to the service registry that return the address if the less busy instance of the requested service.

Example

The example provided in the previous refactoring (Section C.2.2) works perfectly for this case. It shows how a load balancer works in the systems infrastructure, whether internal or external. In the case of being internal, the requesting service would query the service registry directly for the existing instances and make the load balancer itself.

C.2.4 Introduce configuration server

Motivation

When breaking an architecture in microservices, each microservice might have numerous instances in production. Although, as mentioned in Section C.2.2, the list of available instances is accessible through the service registry, it may be necessary to update some configurations while they are operating without redeploying them. By using a configuration server, we can propagate the changes in the configuration to all running instances.

Mechanics

1. Create a separate repository for the software configurations.
2. Separate the source code and the software configurations into different repositories.
3. Every time a change happens to configuration keys, the configuration repository has to be synchronised.
4. When changes occur to the configuration repository, these should be propagated to the running instances that should adapt to the new configuration.

Note: If many programming languages are utilised, each programming language must implement the configuration propagation and adapting strategy.

Example

When a new configuration is committed to git, it updates the configuration server that propagates to the apps. The user then refreshes the apps. Figure C.3 shows an example of the application of this technique.

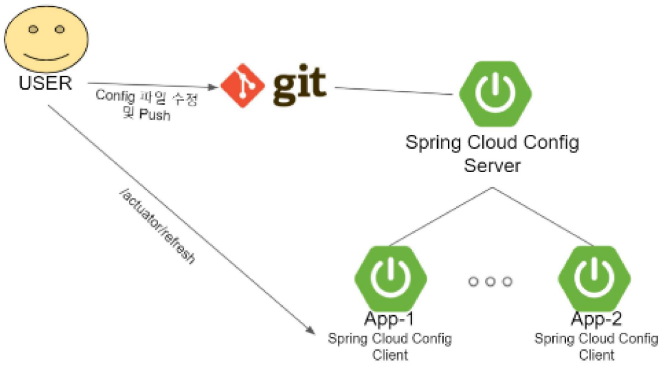


Figure C.3: Configuration Server implementation schema
Source: https://joomn11.tistory.com/79?category=936835

C.2.5 Introduce edge server or API Gateway

Motivation

With this new architecture, new services can be simply introduced, and existing ones can be rearchitected based on new requirements. On the other hand, the user does not need to be aware of this, and these actions should be tracked. Monitoring the status and usage of a service in production allows for the dynamic rerouting of external requests to internal services.

The API Gateway enables services to leverage several communication methods and form a single API with calls to many services.

Mechanics

Implementing an edge server or an API Gateway is quite similar. We create a new layer between the outside world and the set of microservices. This layer can be an API gateway or an edge server and perform dynamic routing based on its configuration. This layer can interact with the service registry to discover the addresses of the instances of each service. This layer becomes an excellent place to monitor the usage of services.

Note: As the communication is all performed through this layer, this layer should also be replicated through load balancing mechanisms to avoid becoming a single point of failure.

Example

Figure C.4 represents the layer of communication. The client requests to the API Gateway (or edge server), and this layer performs dynamic rerouting to the requested service to the most favourable instance.

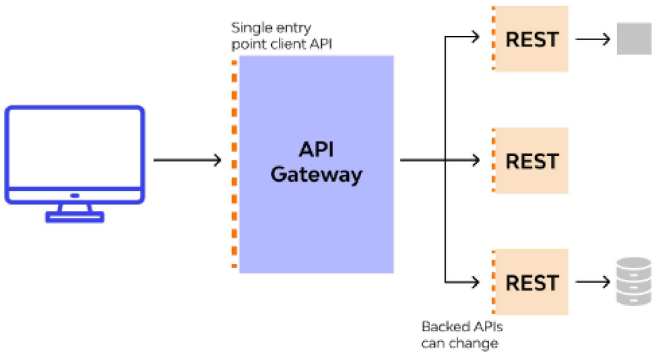


Figure C.4: API Gateway implementation schema
Source: "API Gateway", https://www.wallarm.com/what/the-concept-of-an-api-gateway

C.2.6 Configure service discovery

Motivation

It is common in a microservices architecture to use cloud solutions to host the systems; in these cases, the addresses where the instances are hosted are frequently dynamically allocated. Because the number of instances running changes dynamically, it is difficult to predict each service's address. It is usual in these scenarios to set up a service discovery with a service registry to keep track of the locations of existing systems.

Mechanics

1. Configure and run a service registry.
2. Whenever a service starts running, it registers itself on the service registry; whenever it shuts down, it de-registers itself from the service registry.
3. Add a service discovery client to the services to be used. Whenever they need to communicate with another service, they call service discovery to obtain the service location.
4. Replace previous configurations and remote calls to use the discovery client instead of a static address.

Note: It is common to have a replica of the service registry registered within itself so it does not become a single point of failure.

Example

Figure C.5 shows how the Service Discovery Pattern is implemented.

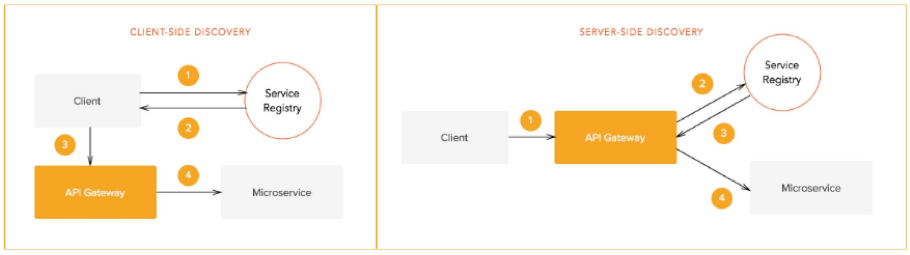


Figure C.5: Service Discovery implementation schema
Source: [86]

C.2.7 Configure health-check

Motivation

Microservices must be notified when a service fails to avoid routing requests to that microservice. This is often accomplished by sending a health check request to the microservice to determine

whether or not it is operational. It should either respond positively or return the specific error that is causing it to malfunction.

Mechanics

1. Decide what type of health check verification you want to perform, which can be a simple 200 status code response.
2. Implement the health check verification in the microservice. This can be verifying all the resources it needs, like the database, or simply returning a positive response.
3. Add the endpoint to the microservice.

Example

In Figure C.6, we can see the mechanics of implementing the health check. The load balancer periodically makes a health-check request to the services' instances it knows and waits for a positive response. If it does receive a positive response, it will take note of that and not forward the requests to that instance.

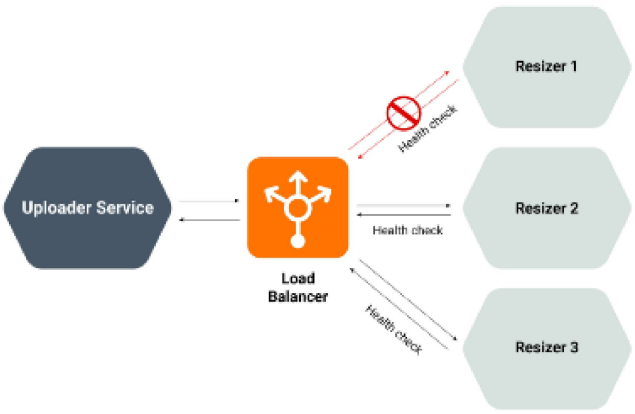


Figure C.6: Health Check implementation schema
Source: "In Brief: The Circuit Breaker Pattern", https://wso2.com/blogs/thesource/in-brief-the-circuit-breaker-pattern/

C.3 Deployment and Orchestration

Another major difference between monoliths and microservices is their deployment. As seen in the previous chapter, besides each microservice having to be deployed, each service usually has multiple running instances. Each of these instances has to be deployed, configured and monitored.

In addition, having multiple instances running simultaneously, orchestration becomes more demanding. This chapter approaches the strategies of deployment and orchestration to adapt to a microservices architecture.

C.3.1 Enable continuous integration

Motivation

The number of services expands due to the migration to microservices, yet we want the system to be "production-ready". We want to build and test the microservices automatically and verify their availability. Because we execute minor and frequent changes, continuous integration shortens the time it takes to validate and release software. It is a step to achieve continuous delivery/deployment.

Mechanics

1. Place each service in a separate repository, having its history and separating its build life cycle.
2. Create an artifact repository.
3. Create a continuous integration server.
4. Create a continuous integration job for each service, where the new code is fetched, all the tests are run in the new code, the corresponding artifacts are built, and these artifacts are pushed into the artifact repository.
5. Create a trigger so the job runs each time the code changes in that repository. The development team should be informed of the errors if the job fails.
6. When the job fails, resolving it becomes the priority, as new changes should pass all the tests so it does not break the existing system.

Example

Figure C.7 represents how continuous integration should work. Every time a developer commits to a git repository, this action should trigger the CI pipeline to run the job that builds the code, runs the unit tests and produces the result: either pass or fail and this result should be notified to the engineering team.

Figure C.8 shows how we connect continuous integration with continuous deployment. After merging the new code to the system and successfully passing all tests, we build and create an image, push it to the registry and after manual approval, it is deployed to production.

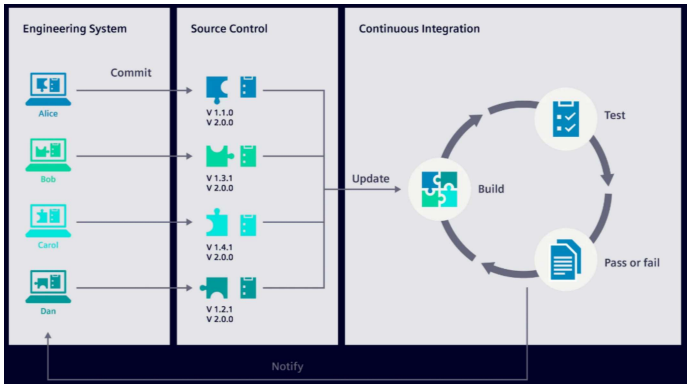


Figure C.7: Continuous Integration implementation schema
Source: "Continuous Integration with TIA Portal", https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/highlights/continuous-integration.html

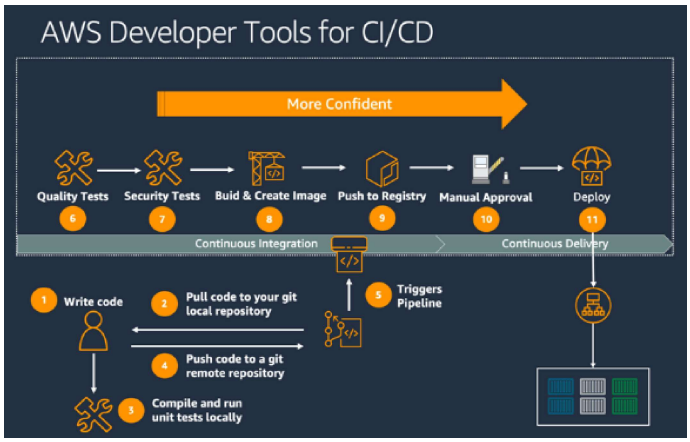


Figure C.8: Aws Developer Tools for CI/CD
Source: "AWS Prescriptive Guidance Patterns", https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/welcome.html

C.3.2 Containerize services

Motivation

One of the goals of breaking down a system into microservices is to make them totally independent of one another. This means that each microservice can be designed with a different technology, will have different configurations and different versions of the same packages, etc. and, as a result, each microservice will require a distinct environment for deployment.

Containerisation is a type of virtualisation in which a service is deployed into a single container image and runs isolated from the other services with its own desired environment. Besides this advantage, containers are lightweight and portable and facilitate automation.

Mechanics

1. Choose a containerisation technology. Docker is a widespread technology to use in these cases.
2. Analyse the system's dependencies.
3. In each service's code repository, create a script with the container image configuration and the script to make it run.
4. Have a list of the required environment variables for running the service in its code repository and the values it takes in a different environment (development, production, etc.). It is common to see this in a .env file.
5. Deploy the container.

Note: Containerising the services adds two more tasks for the continuous integration pipeline: building the container images and storing them in a private repository. Also, the development environment should adapt to embrace containers.

C.3.3 Orchestrate service

Motivation

The more microservices that are created, the more difficult it is to manage all of them and their dependencies since they all have different characteristics. Each microservice may require different initialisation strategies or specific configurations to run correctly. When availability, provisioning, scaling, and other configurations are added, it becomes overwhelming to accomplish this manually. Fortunately, orchestration tools/frameworks allow you to describe all of these behavioural preferences in a configuration file that establishes a distributed cluster with the appropriate nodes, builds the system, and operates it automatically.

Mechanics

1. Choose an orchestration tool/framework (e.g. docker-compose, Kubernetes).
2. Create a configuration file that describes how you want the system to work. This file is expected to be similar to the service's configuration file.
3. Start your system using the configuration files.

Example

Figure C.9 represents an example of container orchestration.

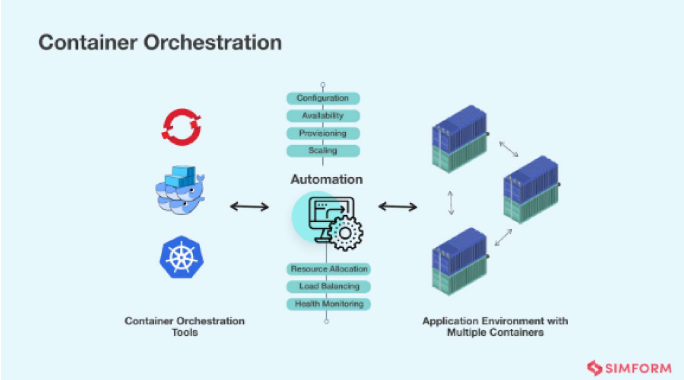


Figure C.9: Container Orchestration schema
Source: [17]

C.3.4 Deploy into a cluster and orchestrate containers

Motivation

We want to deploy all service instances into a cluster and orchestrate them with the least effort possible.

Mechanics

1. Choose a cluster management tool. Ideally, this tool should allow the definition of the deployment architecture of the services declaratively. This way, any additional effort for deployment, for instance, auto-scaling, monitoring, name resolution, and failure management of the deployed services, should be delegated to the cluster management tool.

2. Integrate this tool into your system.

Note: Pay close attention to single points of failure. Choose the management tool wisely, for example, one that provides means for auto-scaling the services.

Example

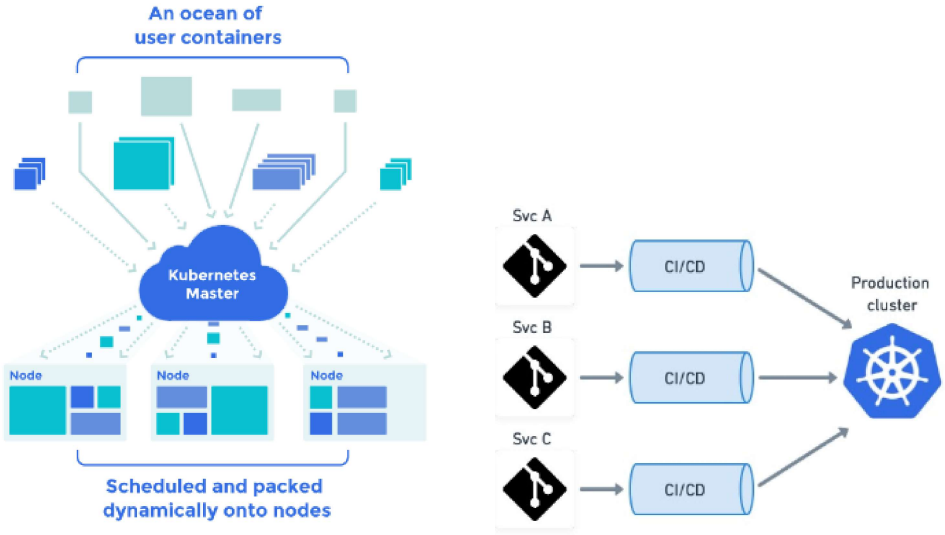
Figures C.10 show how it works to deploy services into clusters and its orchestration.

C.3.5 Centralize logging

Motivation

Each microservice can or not have its logging infrastructure. Logs aid in the diagnosis, troubleshooting, and tracking of issues. They usually get stored in the source directory in a log file.

If each microservice had its log file, the analysis of the execution of the entire system and debugging would be more challenging, as we couldn't easily track interaction between microservices or a global view of what is happening in the system.



(a) Kubernetes Master , Source: [28] (b) Production cluster, Source: [101]

Figure C.10: Deployment into clusters

This is simplified by having a centralised logging infrastructure to view the logs produced by each microservices. This centralised logging allows the addition of tags to the logs or even preferentially arrange them.

Mechanics

1. Configure the centralised logging platform.
2. Configure logging for your applications if it doesn't already exist.
3. Redirect the logs from the application to the centralised logging platform.

C.3.6 Centralize Configuration

Motivation

Similarly to the logs, each microservice may have its own set of configuration files. Because there are system-wide configurations, some of these configurations can be duplicated for all microservices. When one of these global configurations changes, all microservices must be updated, which is inefficient.

Extracting these configurations into a configuration server centralises the configurations, making modification easier.

Mechanics

1. Extract the information in the configuration files from each microservice into a source code repository to be versioned.

2. Configure and run a configuration server that reads the configuration from the repository and serves it to existing microservices.

3. Configure existing microservices to read the configuration from the configuration server instead of using the local configuration file.

C.4 Building Microservices

Following James Lewis and Martin Fowler's description of the common characteristics of microservices [62], we will dive into their definition of each characteristic and identify the refactorings offered in this catalogue that help to achieve them. As they said, "[...] not all microservice architectures have all the characteristics, but we expect that most microservice architectures exhibit most characteristics.".

C.4.1 Componentization via Services

The authors argue that we should treat services as components since they can be deployed independently. Thus, if a service changes, only that service needs to be redeployed, not the entire application.

"A service may consist of multiple processes that will always be developed and deployed together, such as an application process and a database that's only used by that service." [62]

Useful refactorings: All refactorings of chapters 1, 2, 3 and 6 contribute to having services as components.

C.4.2 Organized around Business Capabilities

There are numerous ways to break a system down into different services. However, the authors state that *"The microservice approach to division is different, splitting up into services organized around business capability. Such services take a broad-stack implementation of software for that business area, including user interface, persistent storage, and any external collaborations. Consequently, the teams are cross-functional, including the full range of skills required for the development: user experience, database, and project management"*. [62]

Useful Refactorings: This is not a refactoring issue but rather a concern when determining the boundaries of each service, which is not covered in this catalogue. With the company and project characteristics in mind, this decision should always be made to enhance project productivity and efficiency.

C.4.3 Products, not Projects

Following up the line of thought of organising the microservices around business capabilities, the authors believe that we should shift to the products over projects mentality. This way, you don't see the software development as completing a set of functionalities but rather as a way to make the

most of the business capability. With the typical small granularity of the services and each team owning a product over its lifetime, a closer connection is built with the user.

Useful refactorings: 3.1, 3.2., 3.3, and 3.4 are useful for letting a team own a product, and 6.1 to create these products teams will be working on.

C.4.4 Smart endpoints and dumb pipes

This trait is related to the shift in communication paradigms that occurs when we migrate to microservices. This structure must be carefully considered since calls between microservices can cause increased latency.

The basis of communication in microservices is to make a request, which is then processed and applied some logic by the service receiving the request, which produces a response.

According to the authors, communication is most typically implemented via HTTP request-response protocols with resource APIs or lightweight messaging. The first makes use of web principles and protocols. The second, a lightweight message bus, often only works as a message router ("dumb") and leaves the "smart" part to the endpoints. These services will design a strategy to operate without introducing additional latency, which is common in service calls.

Useful refactorings: 1.1., 1.5, 6.4, 6.5., and 6.7 are useful refactorings to create smart endpoints as well as the concerns mentioned in Chapter 7.

C.4.5 Decentralized Governance

As we split the monolith into microservices, each microservice can have its own stack of technologies, standards, and so on. It also opens the door to employing tools to handle the demands of each microservice, as well as using and developing open-source code that other developers can use to solve similar challenges. Decentralised governance allows each service to have its own governance strategy, contributing to the reality that each team should own their product.

Useful refactorings: 2.1 to 2.7, 3.1, 3.2, 3.3 and 3.4.

C.4.6 Decentralized Data Management

Microservices decentralise both conceptual models and data storage decisions.

"Microservices prefer letting each service manage its own database, either different instances of the same database technology or entirely different database systems". [62]

This creates implications for managing updates, it can be solved by using transactions (even distributing transactions, when needed), but usually, it is mitigated by assuming eventual consistency to answer the demand quickly and through compensating operations to deal with mistakes.

Useful Refactorings: 1.2, 1.3, 1.4, 1.6, 1.7, 6.4, 6.5, 6.6, 6.7 and 6.8.

C.4.7 Infrastructure Automation

Infrastructure automation solutions have grown substantially with the emergence of the cloud. Systems designed with microservices use Continuous Integration and Continuous Delivery strategies, which benefit greatly from infrastructure automation techniques.

The most popular applications of infrastructure automation are automated testing, deployments, and microservices management in production.

Useful refactorings: 2.1 to 2.7 and 3.1 to 3.6.

C.4.8 Design for failure

"A consequence of using services as components is that applications need to be designed to tolerate the failure of services. Any service call could fail due to the unavailability of the supplier, and the client has to respond to this as gracefully as possible". [62]

Services fail frequently, and we must be able to detect and automatically restore them. This can be accomplished through real-time monitoring of the application and logging configurations for each service. How we achieve this is irrelevant, but maintaining the system available is critical.

Useful refactorings: 2.1, 2.2, 2.3, 2.6, 2.7, 3.3, 3.4, 3.5, 6.5, 6.7, 6.8, and all concerns mentioned in chapter 7.

C.4.9 Evolutionary Design

When decomposing a monolith into microservices, we must keep the concepts of independent replacement and upgradeability in mind.

Code that changes together should be in the same microservice, and code that changes rarely should be in a distinct microservice than code that changes regularly. If two services tend to change simultaneously, they should generally be merged.

It is also useful to use microservices when we know that something will be temporary; in this manner, we build and deploy it fast, and when it is no longer necessary, we remove it, speeding up the release process. However, this raises the concern of one service breaking its consumers, so the services should be designed to be as tolerant as possible of the changes that their suppliers may suffer.

Useful refactorings: All refactorings of chapters 2 and 3 contribute to an evolutionary design and refactorings 6.1, 6.2, 6.4, 6.5, 6.7, 6.8, 6.9 and the concerns mentioned in chapter 7.

C.5 Intermediate refactoring states (common strategies)

The migration should be a step-by-step process as it is complex. Therefore, multiple ways order these steps, leading to a microservices architecture. In this chapter, we present some common refactorings that lead to intermediate states of the refactoring that contribute to the end goal of migrating to a microservices architecture but do not directly apply refactorings that go along with microservices characteristics. For instance, not always database splitting is the first step, and there

are numerous states the database can be refactored as intermediate steps to allow the evolution of the migration.

C.5.1 Shared Database

Motivation

When we start breaking the monolith, we may not want to split the database straight away to the different microservices. We may want to avoid performing all these changes in the first moment and focus on the functionality migration. This state is usually an initial state, usually a state of the system's evolution, but it does not represent a refactoring. Situations where using a shared database might be a good solution are:

1. In a microservices architecture is on read-only static reference data because there aren't multiple services trying to change the same data, only reading it;
2. When a service exposes its database as a defined endpoint that was thought to handle multiple consumers;
3. Or when it is an intermediate step before refactoring it.

Mechanics

The database is shared among multiple microservices. Each service can access data owned by other services using local ACID transactions. If, at some point, two microservices share a purely read-only database part, it may not be necessary to keep copies of these in multiple places, and it may be kept that way.

Note: This is not ideal in the microservices paradigm as it brings inconsistencies, no possibility of keeping things hidden, etc., so splitting the database is preferred.

Example

In Figure C.11, we can see that all three services, Dispatch, Finance and Order Processing, use the same database and can change its information directly.

C.5.2 Database Wrapping Service

Motivation

When multiple services depend on the same datastore, it is hard to consider pulling apart the underlying schema. The database wrapping service uses a service as a wrapper of the database schema, hiding the schema complexity behind a service. This refactoring is seen as a step toward more fundamental changes. It is common in a many-to-many relationship between entities in different microservices, where there usually is a join table. Instead of using the move foreign-key

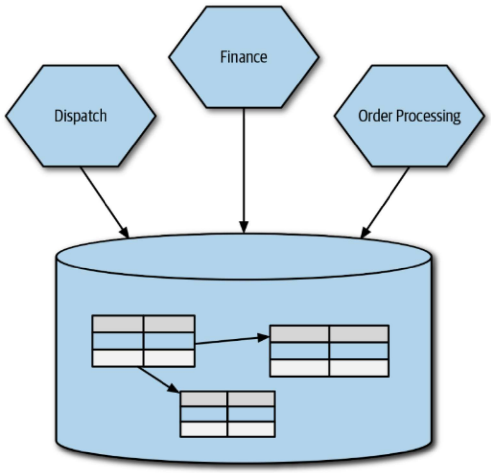


Figure C.11: Shared Database Example
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

connection to code refactor and lose the table, this refactor is applied, and the relationship is kept the way it is.

Mechanics

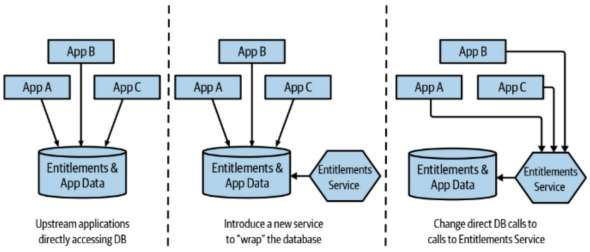
1. Create a new service to hide the database that acts as a thin wrapper.
2. Extract the classes mapped to the involved entities to the new service.
3. Replace clients' requests to the database with requests for this new service.

Note: Encourage developers writing the different clients to think of this new service as someone else's and start storing their data locally. Ensure that the database schema is unchanged and that there is low coupling.

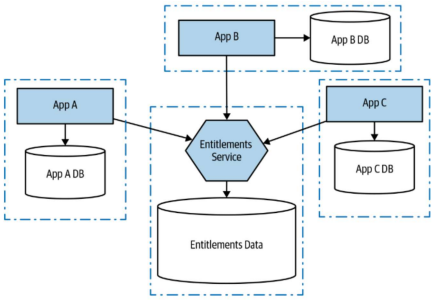
Example

The new service will act as an interface to consumers while changes are being made in the database to improve the schema. This way, it can later adapt to our end goal of breaking the tables, so each service owns its data, and there is loose coupling.

In this example, Figure C.12, all apps must access the database. This database stores each app's data and entitlements. The entitlements are the accounts each individual can access and the operations they can do. As the entitlements are very complex and have a lot of logic behind them, we create the Entitlements Service that owns the entitlements and app data and hides this complexity. Then we create a database for each service, so each service stores its app data, and the entitlement service only stores the entitlements data in its database. This way, other services think of the entitlements as owned by another service.



(a) Using a service to wrap a database



(b) Using the database wrapping service pattern to reduce dependence on a central database

Figure C.12: Database Wrapping Service Example
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

C.5.3 Database-as-a-Service

Motivation

When clients need a database to query, this could be because they need to query large amounts of data or because external parties already use toolchains that require a SQL endpoint to work against.

Mechanics

1. Create a dedicated database to be exposed as a read-only endpoint and have this database populated when the data in the underlying database changes.
2. Separate the database we expose from the one we use inside our service boundary.
3. In the external database, make a mapping engine that takes changes in the internal database and determines what changes.
4. The mapping engine must change when the internal database changes structure to ensure the public-facing database remains consistent.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.4 Database view

Motivation

We want a single data source for multiple services, but changing all clients to point to the new service(s) is impractical. For clients considering the monolith as their data source, we want to

mitigate concerns regarding coupling to specific data schema parts. There are more clients reading data than writing it. We can use database views to show only the relevant data for each service.

Mechanics

1. Create a dedicated schema that hosts views like the old schema assembled from data now owned by the new service(s).
2. Have clients that only read data points at those views instead of the original tables.
3. Change the clients that need to write data to use the new services directly instead of the monolith.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.5 Synchronize data in the application

Motivation

When we want to split the schema of the monolith into two separate data stores in preparation for extracting a new service.

Mechanics

1. Create the new data store and bulk-import data from the monolith—only the part of the schema we are interested in splitting needs to be imported.
2. Make the monolith write the same data to both data stores, ensuring they are kept in sync, and that writes to the new data store work well.
3. Make the new data store the source of truth and ensure that the reads also work. As we still write to both data stores, we can easily fall back to reading from the old data store if any issue is found.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.6 Tracer writer

Motivation

When we need to move the ownership of some data from the monolith to a new service in an incremental fashion, tolerating there being two sources of truth during the migration.

Mechanics

1. Identify the part of the monolith data schema whose ownership we want to change and the service that will host it.
2. Ensure this data syncs between the monolith and the new service. This can be achieved in several ways:
   (a) Allowing writing only on one of the sources of truth and making it in charge of replicating the change in the other one;
   (b) Having clients send writes to both sources; or
   (c) Allow clients to send writes to either source and synchronise data behind the scenes.
3. Once all clients use the new source of truth, the old source of truth can be retired.

Note: *Synchronize Data in Application* focuses on splitting the schema, and this technique focuses on moving a part of the schema to the ownership of a previously created service.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.7 Separating libraries from their dependents

Motivation

Martin Fowler defines libraries as "components that are linked into a program and called using in-memory function calls", distinguishing them from services that are "out-of-process components who communicate with a mechanism such as a web service request or remote procedure call". In this situation, there is code used by multiple services using a method call, but it is not business related.

Scalability needs between a shared library and the dependent services can create the need for the library to become a service that can scale independently.

Mechanics

1. Extract the code used by multiple services into a library.
2. Extract the library as a service creating an API or other types of interface to access it.

Note: Sometimes, separating libraries from their dependents and using a service call instead of a method call may introduce performance issues. Although performance issues can be handled using careful caching mechanisms, it adds another layer of complexity to the dependent service.

C.5.8 UI Composition

Motivation

As we incrementally migrate a monolith to microservices, the user interface's functionality must be served partly by an existing monolith and partly by the new microservice architecture.

Mechanics

Different variants of UI Composition may be used:

1. Page-based Composition consists of having different services serve different pages, migrating each one at a time.
2. Widget Composition consists of embedding a piece of the user interface provided by a new service into the monolith-provided user interface.
3. Micro Frontends consist of taking a microservice-based approach to frontend development. The idea is to break down a user interface into different components that can work independently.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.9 Branch by Abstraction

Motivation

Changes to the existing codebase will likely take time, and we want to avoid any disruption. It assumes we can change the code of the current system.

Mechanics

1. Develop, in the monolith, an alternative implementation for a module so that the new and old implementations comply with the same interface/abstraction.
2. The new implementation typically calls a new external service, whereas the old implements the functionality internally.
3. At some point, switch from the old to the new one.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.10 Parallel Run

Motivation

When the failure of the functionality being worked implies a high risk for the business.

Mechanics

1. With a parallel run, rather than calling either the old or the new implementation, we call both, allowing us to compare the results to ensure they are equivalent.
2. Despite calling both implementations, only one is considered the source of truth at any given time. Typically, the old implementation is considered the source of truth until the ongoing verification reveals that we can trust the new implementation.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.11 Decorating Collaborator

Motivation

We use it to trigger some behaviour based on something happening inside the monolith, but we want to avoid changing it. This technique assumes we can intercept responses returned by the monolith before they reach the clients.

Use the decorator pattern to make it appear from the outside that we have added new logic to the monolith without actually changing it.

Mechanics

1. Route client requests to a proxy, allowing the call to reach the monolith as usual.
2. Based on the result of this call, call out to a new microservice and possibly modify the result before returning it to the client.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.12 Change Data Capture

Motivation

When we need to react to a change in data in the monolith but cannot intercept this change at the perimeter of the monolith (e.g., using a decorator) or change its implementation.

Mechanics

Rather than trying to intercept and act on calls made into the monolith, we react to changes made in a data store. The underlying capture system will be coupled to the monolith's datastore. It can rely on database triggers, transaction log pollers (usually a file into which is written a record of all the changes that have been made) or batch delta copier (a program that regularly scans the database in question for what data has changed).

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.13 Aggregate Exposing the Monolith

Motivation

When the monolith database still owns the data we want to access, we need a way for our new service to access it. Calling back to the monolith to access the data is harder than accessing the database directly; however, it is better in the long run. So we can expose a service endpoint (an API or a stream of events) in the monolith to access this data.

Mechanics

1. Create an endpoint that exposes the data the microservice needs to access.
2. This endpoint can limit the level of access the microservice can have to that data.

3. Make the microservice use this endpoint to access the data.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.6 Common Sequences of Refactorings

This chapter introduces common sequences of refactorings found when migrating from monoliths to microservices.

C.6.1 Extract Service

Motivation

We may need to extract a service from the rest for multiple reasons, like scalability, ease of deployment, etc. We have multiple functionalities when we have a monolith system, but not all are used as much as the others. As one functionality is used more frequently than the other, scalability can become an issue. This refactoring will transform one or more regular class(es) into a remote service.

Mechanics

1. Analyse all the dependencies of the functionality we will extract to a service. Both the dependencies to the rest of the monolith and that the monolith has with the service to be extracted.

2. Resolve these dependencies. This usually involves refactoring foreign keys and changing method calls to other files/classes/etc. to remote calls (synchronous or asynchronous). Appendix C.1 describes interesting refactorings to resolve these dependencies.

   (a) If external libraries are dependencies, they will later need to be added to the new service.

   (b) All its methods used by other components inside the monolith had to be ready to be called over service calls, like services communication, etc.

   (c) Its original clients have to access it through remote service calls.

3. Create a new project folder with the files identified to be owned by this functionality with the same characteristics and the changes made during the previous steps.

4. Make it capable of running independently, installing the necessary dependencies and preparing its production environment.

Note: Make sure to guarantee fault tolerance and data consistency and to solve performance issues (check **Chapter 7 (Appendix C.7)**).

Example

We have seen in the previous section (cf. Section C.1) many refactorings that lead to the extraction of the services *Inventory* and *Order*. To apply this refactoring, the same way we identified those dependencies, we identify all dependencies these microservices have between themselves and to the rest of the monolith and use similar refactorings to the ones identified in that section to extract them. When all dependencies are resolved, we create a new project for each of these microservices with the designated files that belong to them; we build them and deploy them after correctly preparing them with the necessary dependencies for the production environment.

C.6.2 Strangler Fig

Motivation

When we have decided to evolve a system to a microservices architecture, we want to take incremental steps toward the new architecture and ensure that each step is easily reversible, reducing risks.

Mechanics

1. You can start by deciding if you want to wrap the monolith with an API that allows us to access the new system in the old way. If so, perform the necessary implementations (Branch by Abstraction refactoring, **Chapter 5 (Appendix C.5)**).
2. Start small, with the macro, then micro mindset and identify the functionalities you want to extract and their boundaries.
3. Identify the order by which you want to extract the functionalities. And program how you want to do the extraction.
4. Gradually move functionality over to the new microservices architecture. This technique usually uses as the main refactoring of the extract service refactoring (C.6.1).
5. Reroute calls from the monolith over to the new microservice using the change local method call dependency to a service call (C.1.1).
6. If the new extracted functionality uses functionalities that remain inside the monolith, then the monolith can expose this functionality (C.5.13). Iteratively extract all functionalities.
7. Write new code as microservices.

   Note: Do this by replacing or rewriting existing features parallel to the old architecture, one at a time, until the old architecture has been entirely replaced. Usually, we must create a proxy or façade that provides a stable API for old clients throughout the migration.

Example

As an example, we are going to use the same example given by Sam Newman on *"Monolith to Microservices: Evolutionary Patterns to Transform your Monolith"* [75].

In Figure C.13, we can see that the *InventoryManagement* functionality is self-contained and, therefore, has no dependencies. So, we can simply extract this service, rerouting existing calls regarding this functionality to this new service instead of to the monolith.

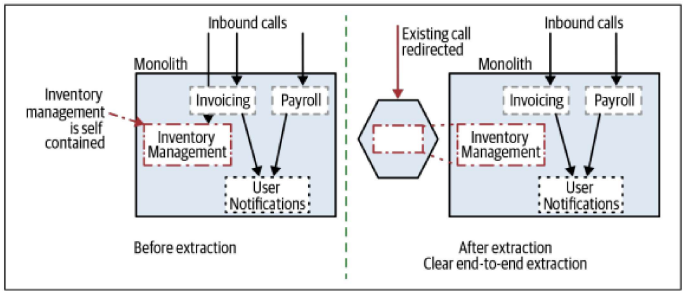


Figure C.13: Example of application of the Strangler Fig refactoring to the InventoryManagement refactoring
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

We gradually plan the extraction of the functionalities to services according to an order we define and at our own pace, and all the new code is written as microservices. These changes can be rolled back whenever we find it is not suitable. If the functionality were to be used by the functionalities inside the monolith, we would have to redirect those too.

C.6.3 Typical functionality library pattern

Motivation

When many services have functionalities in common that do not have memory or state.

Mechanics

1. Move the common functionalities into a repository with common code libraries.
2. Import this library into the services that used the functionalities.
3. Change the service calls for these functionalities to code dependencies.

C.6.4 API Composition

Motivation

Each microservice has its database, and it is no longer straightforward to implement queries that join data from multiple services. Creating an aggregator service allows one to gather data from the different services, merge it and show it to the end user.

It is very common to use this refactoring when there are a lot of reads.

Mechanics

1. Define an API Composer.
2. Make the composer identify the services it needs to invoke to answer the user query.
3. After identifying, it queries the microservices.
4. Perform an in-memory join of the results.
5. Return the response to the user.

C.6.5 SAGA

Motivation

Each service has its own database, but some business transactions involve multiple services. This refactoring corresponds to a distributed transaction and increases data consistency across services. This represents another case where we expect eventual consistency because each transaction may take some time. It is common when multiple writes to the database are triggered by one action.

Mechanics

1. Implement each business transaction as a SAGA - sequence of local transactions.
2. Each transaction updates the database and publishes a message or event to trigger the next local transaction. Or an orchestrator tells the participants what local transactions to execute.
3. If a local transaction fails because some business rule was violated, then a series of transactions are executed to undo the changes made by the previous transactions.

Example

Figure C.14 shows an example of the SAGA works.

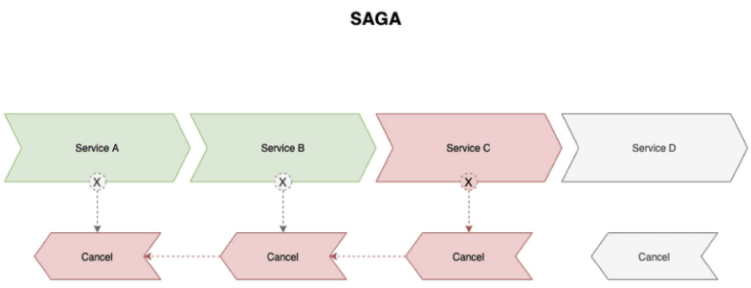


Figure C.14: SAGA example
Source: [44]

C.6.6 Change Data Ownership

Motivation

When extracting a new service that encapsulates the business logic of some data, that data should belong to that service and, therefore, should be moved into the new service. The new service has to be the new source of truth of that data.

Mechanics

To do this, we have to break the dependencies the monolith may have with this data through refactorings like: C.1.2, C.1.3, C.1.4, C.5.2, and C.6.5.

Example

As an example, we are going to use the same example given by Sam Newman on *"Monolith to Microservices: Evolutionary Patterns to Transform your Monolith"* [75].

In Figure C.15, we can see that we want to move invoice-related data into the new *Invoice* service. Therefore, we need to change the monolith to take the *Invoice* service as the source of truth for invoice-related data and change all calls to read or change the invoice-related data to be made directly to the *Invoice Service*. This can initiate other refactorings like "Move Foreign-key relationship to code" to make this work.

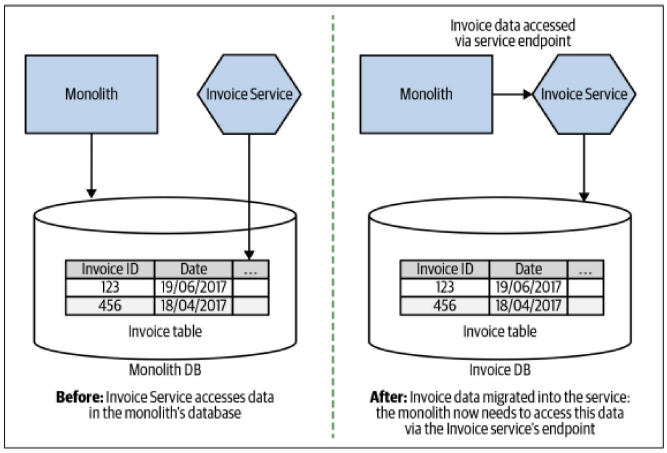


Figure C.15: Example of application of the Change Data Ownership refactoring of the Invoice table to the Invoice Service
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

C.6.7 Tolerant Reader

Motivation

Being tolerant when reading data from a service is an excellent advantage because it prevents issues from happening whenever the format changes. The idea is to read only the parts of the message it needs to read and tolerate changes as long as the required fields are present and readable. Even if some fields are added or removed, the ones the reader needs are expected to remain.

Mechanics

1. Take the minimum assumptions about the structure of the message.
2. Make sure there's only one bit of code that reads data payloads like this.
3. For example, if you use a DTO, use generic collections to make it serialisable, allowing the technologies to differ from service to service.

C.6.8 Anti-corruption Layer

Motivation

We want to avoid data corruption during any communication at all costs. Therefore, creating an anti-corruption layer is often a good solution when the services use different communication technologies.

Mechanics

1. Create a layer between the two services as a proxy to allow them to communicate.
2. This layer should verify the data flowing in the communication and transform it to fit the receiver, acting like a translator if needed.

Example

Figure C.16 shows how typically an Anti-corruption layer is added to the system.

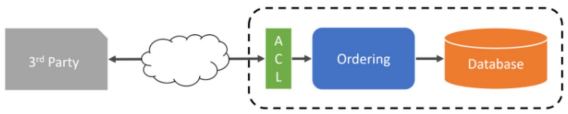


Figure C.16: Anti-corruption Layer schema
Source: [23]

C.6.9 Adapt Service Interface

Motivation

The availability of business functionality across different providers is quite typical in cloud-based services. Multiple service implementations may be required for various business and technical reasons, including fault tolerance and a better division of responsibilities (as less similar code is repeated).

The issue is that it is also common for this multiple services implementation to have different service interfaces. When this happens, there are two ways of solving it: you either treat the invocation of the similar services functionalities independently, and each service has a replica of that code, or you can systematically adapt the services' interface to each other, so we can use the same interface no matter which service we invoke.

Mechanics

These mechanics explain how to implement the second methodology to solve this issue.

1. Create an adapter service that translates the requested method to the method on the adaptee interface.
2. Incrementally start with two services interface, identify the differences and map the names to create one interface.
3. Solve the differences by reordering the parameters, providing missing parameters or omitting extra parameters. In some cases where different types for the same parameter must be solved manually, adjusting to each one seems better.
4. It ends when you can create one single interface for all services.

C.7 Extra Concerns

In most circumstances, we should have additional considerations in addition to the procedures indicated in each refactoring to design a resilient system. Numerous inputs exist on these topics, so we leave links to helpful resources to address these concerns.

More concerns can be added in this chapter, as the change in architectural paradigm raises many new concerns to keep in mind. Two sources we want to add to this catalogue because it comprises a lot of information on microservices architecture are the following Source 1, Source 2.

C.7.1 Data consistency

Ensures the data format or the data concerning other data. It basically refers to the usability of the data.

This is ensured in a monolith by the ACID transactions of the relational database. ACID means:

- Atomicity;
- Consistency;
- Isolation;
- Durability.

Even though, in a microservices architecture, the data is distributed across the microservices, it still needs the ACID properties.

To achieve this, three common strategies are distributed transaction (check SAGA refactoring, C.6.5), Two-Phase commit and Eventual Consistency.

For instance, in cases of moving the foreign key relationship to code, we can check before deletion, handle deletion gracefully or don't allow deletion when the data the foreign key refers to is being deleted so we can ensure the data consistency on the service that uses that data through the foreign-key.

Some useful sources to look for information about this are: Source 1, Source 2, Source 3, Source 4, Source 5, Source 6 and Source 7.

C.7.2 Resilience and Fault Tolerance

Refers to the system's ability to continue operating without interruption when one or more components fail.

Some good strategies to create a resilient and fault-tolerant system are timeouts, retries, circuit breaker, chaos testing, etc.

Some valuable sources to look for information about this are: Source 1, Source 2, Source 3, Source 4, Source 5 and Source 6.

C.7.3 Performance

Performance is a common bottleneck for microservices. In this section, we want to provide some sources to identify and solve the performance challenges.

Some recommended sources are: Source 1, Source 2, Source 3 and Source 4.

It is also common to apply the CQRS pattern as it allows to maximize performance, scalability and security. Check this Source for more about it.

C.7.4 Security and Network

In distributed systems, network and security problems are very likely to rise. The following sources provide good insights on this theme: Source 1, Source 2, Source 3, Source 4, Source 5, Source 6 and Source 7.

C.8 Migration Notes

Preparing well is the best way to successfully journey from a monolithic architecture into a microservices architecture. This means:

- Having a good understanding of the monolith and of the reasons why you are performing the migration;
- Give training to the teams that will handle the microservices;
- Creating a good plan for the migration and how you will handle the synchronisation of the monolith and the microservices;
- Using feature flags is a good technique to turn on and off portions of the monolith when they are migrated;
- Have a good set of automated tests implemented, from unit to integration and end-to-end;
- Many times, putting everything in a mono repo and creating a shared CI pipeline is recommended;
- Sometimes, modularising the monolith before the migration is a good strategy;
- Creating a monitoring server that monitors, parses the information and aggregates it into structured information that can be pooled timely to update some visualisation tool to help each service team improve performance and detect anomalies.
- Going Macro and then Micro. Start with larger services around a logical domain concept, then break them into multiple services.

A piece of advice is to start small and analyse well if the partitions being performed are productive. Picking something with the fewer dependencies possible, for instance, can be a good starting point. The strangler fig refactoring is one of the most used approaches.

C.9 References

Table C.1 shows the refactorings in the catalogue and the main references we used.

Table C.1: Refactorings references of the complete catalogue

Refactoring Name	**Chapter**	**References**
Change local method call dependency to a service call		[9], [112], [79]
Move Foreign-key relationship to code		[37], [75], [79]
Replicate Data Across Microservices		[35]

Continued on next page

Table C.1 – continued from previous page

Refactoring Name	Chapter	References
Split Database Across Microservices	1	[112], [75]
Create Data Transfer Object		[37]
Break data type dependency		[37]
Duplicate file Across Microservices		-
Introduce the circuit breaker		[112], [79], [82]
Introduce service registry		[9]
Introduce internalexternal load balancer		[9]
Introduce configuration server	2	[9]
Introduce edge server or API Gateway		[9]
Configure service discovery		[79]
Configure health-check		[79]
Enable continuous integration		[9]
Containerize services		[79]
Orchestrate service		[79]
Deploy into a cluster and orchestrate containers		[9]
Centralize logging	3	[79]
Centralize Configuration		[79]
Shared Database		[75]
Database Wrapping Service		[75]
Database-as-a-Service		[75]
Database view		[75]
Synchronize data in the application		[75]
Tracer writer		[75]
Separating libraries from their dependents	5	[75, 9]
UI Composition		[75]
Branch by Abstraction		[75]
Parallel Run		[75]
Decorating Collaborator		[75]
Change Data Capture		[75]
Aggregate Exposing the Monolith		[75]
Extract Service		[108], [79]
Strangler Fig		[75], [108]
Typical functionality library pattern		[112]
API Composition		[68]
SAGA		[68]
Change Data Ownership		[75]
Tolerant Reader	6	[112], [32]

Continued on next page

Table C.1 – continued from previous page

Refactoring Name	**Chapter**	**References**
Anti-corruption Layer		[112], [58], [23]
Adapt Service Interface		[61], [59], [100]