Interservice communication in a microservices setup

22-28 minutes

This document is the third in a four-part series about designing, building, and deploying microservices. This series describes the various elements of a microservices architecture. The series includes information about the benefits and drawbacks of the microservices architecture pattern, and how to apply it.

1. Introduction to microservices
2. Refactoring a monolith into microservices
3. Interservice communication in a microservices setup (this document)
4. Distributed tracing in a microservices application

This series is intended for application developers and architects who design and implement the migration to refactor a monolith application to a microservices application.

This document describes the tradeoffs between asynchronous messaging compared to synchronous APIs in microservices. The document walks you through deconstruction of a monolithic application and shows you how to convert a synchronous request in the original application to asynchronous flow in the new microservices-based setup. This conversion includes implementing distributed transactions between services.

Example application

In this document, you use a pre-built ecommerce application called Online Boutique. The application implements basic ecommerce flows such as browsing items, adding products to a cart, and checkout. The application also features recommendations and ads based on user selection.

Logical separation of service

In this document, you isolate the payment service from the rest of the application. All flows in the original Online Boutique application are synchronous. In the refactored application, the payment process is converted to an asynchronous flow. Therefore, when you receive a purchase request, instead of processing it immediately, you provide a "request received" confirmation to the user. In the background, an asynchronous request is triggered to the payment service to process the payment.

Before you move payment data and logic into a new service, you isolate the payment data and logic from the monolith. When you isolate the payment data and logic in the monolith, it's easier to refactor your code in the same codebase if you get payment service boundaries wrong (business logic or data).

The components of the monolith application in this document are already modularized, so they're isolated from each other. If your application has tighter interdependencies, you need to isolate the business logic and create separate classes and modules. You also need to decouple any database dependencies into their own tables and create separate repository classes. When you decouple database dependencies, there can be foreign key relationships between the split tables. However, after you completely decouple the service from the monolith, these dependencies cease to exist, and the service interacts exclusively through predefined API or RPC contracts.

Distributed transactions and partial failures

After you isolate the service and split it from the monolith, a local transaction in the original monolithic system is distributed among multiple services. In the monolith implementation, the checkout process followed the sequence shown in the following diagram.

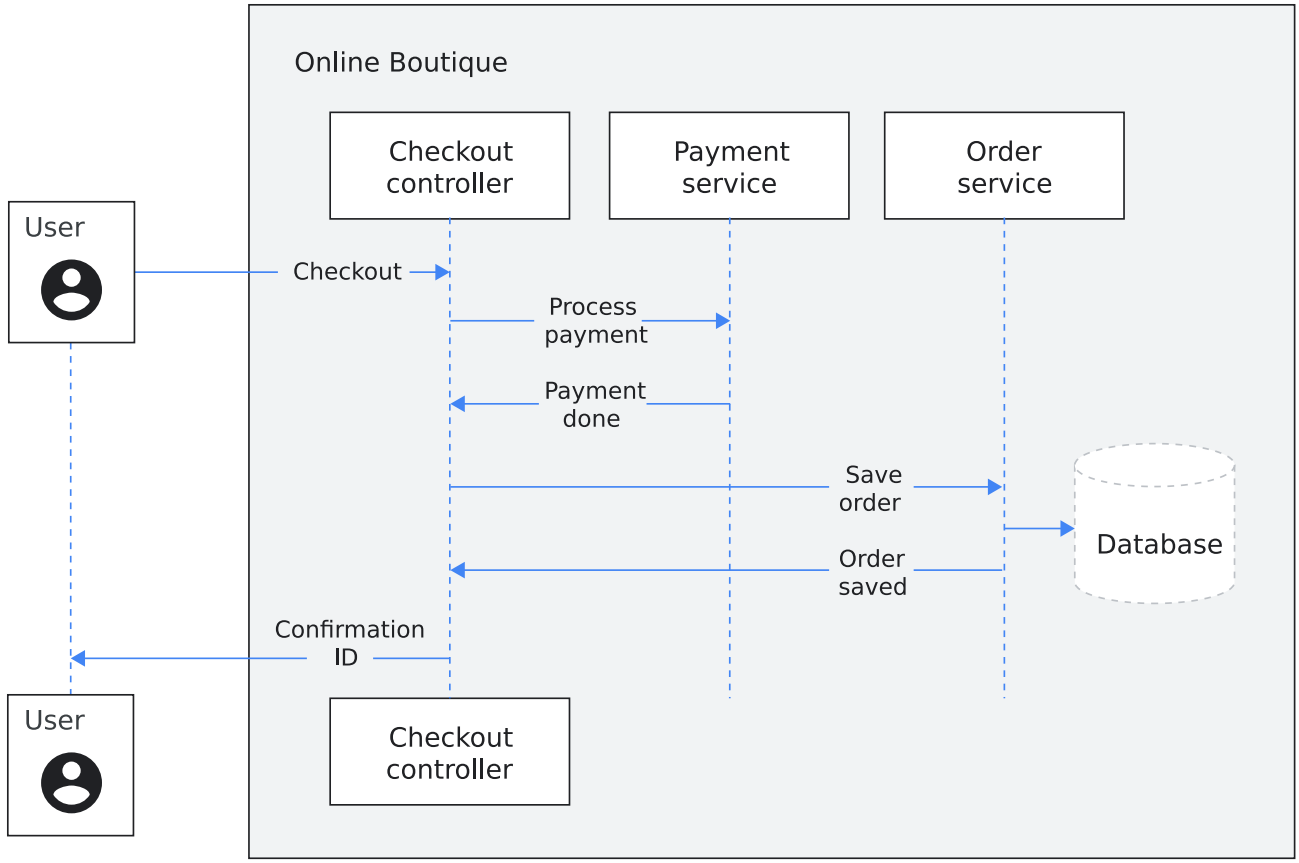

Figure 1. A checkout process sequence in a monolith implementation.

In figure 1, when the application receives a purchase order, the checkout controller calls the payment service and order service to process payment and save the order respectively. If any step fails, the database transaction can be rolled back. Consider an example scenario in which the order request is successfully stored in the order table, but the payment fails. In this scenario, the entire transaction is rolled back and the entry is removed from the order table.

After you decouple payment into its own service, the modified checkout flow is similar to the following diagram:

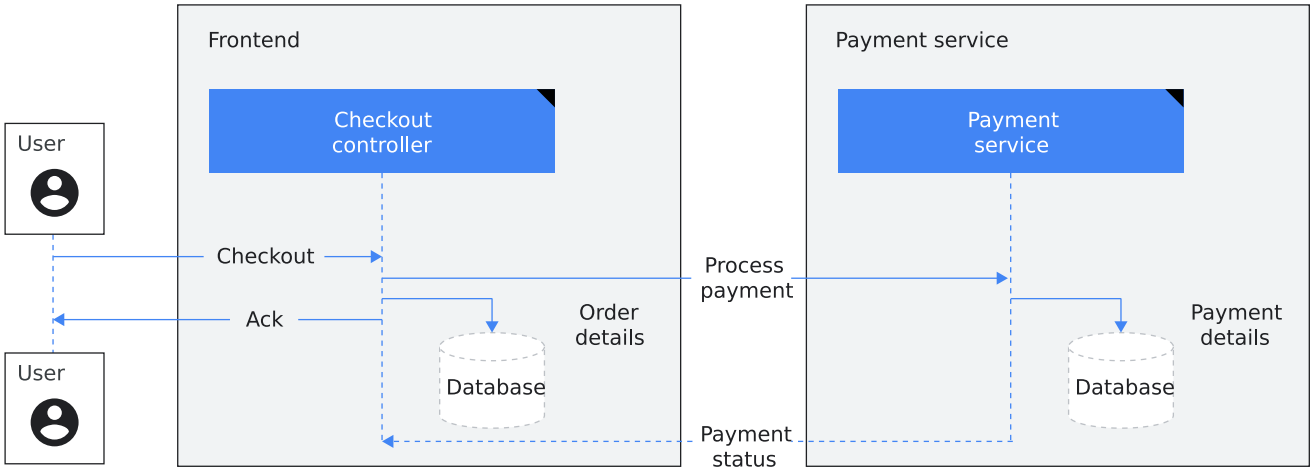

Figure 2. A checkout process sequence after payment is decoupled into its own service.

In figure 2, the transaction now spans multiple services and their corresponding databases, so it's a *distributed transaction*. On receiving an order request, the checkout controller saves the order details in its local database and calls other services to complete the order. These services, such as the payment service, can use their own local database to store details about the order.

In the monolithic application, the database system ensures that the local transactions are atomic. However, by default, the microservice-based system that has a separate database for each service doesn't have a global transaction coordinator that spans the different databases. Because transactions aren't centrally coordinated, a failure in processing a payment doesn't roll back changes that were committed in the order service. Therefore, the system is in an inconsistent state.

The following patterns are commonly used to handle distributed transactions:

- **Two-phase commit protocol (2PC)**: Part of a family of consensus protocols, 2PC coordinates the commit of a distributed transaction and it maintains atomicity, consistency, isolation, durability (ACID) guarantees. The protocol is divided into the *prepare* and *commit* phases. A transaction is committed only if all the participants voted for it. If the participants don't reach a consensus, then the entire transaction is rolled back.

- **Saga**: The Saga pattern consists of running local transactions within each microservice that make up the distributed transaction. An event is triggered at the end of every successful or failed operation. All microservices involved in the distributed transaction subscribe to these events. If the following microservices receive a success event, they execute their operation. If there is a failure, the preceding microservices complete compensating actions to undo changes. Saga provides a consistent view of the system by guaranteeing that when all steps are complete, either all operations succeed or compensating actions undo all the work.

We recommend Saga for long-lived transactions. In a microservices-based application, you expect interservice calls and communication with third-party systems. Therefore, it's best to design for eventual consistency: retry for recoverable errors and expose compensating events that eventually amend non-recoverable errors.

There are various ways to implement a Saga—for example, you can use task and workflow engines such as Apache Airflow,, Apache Camel, or Conductor. You can also write your own event handlers using systems based on Kafka, RabbitMQ, or ActiveMQ.

The Online Boutique application uses the checkout service to orchestrate the payment, shipping and email notification services. The checkout service also handles the business and order workflow. As an alternative to building your own workflow engine, you can use third-party component such as Zeebe. Zeebe provides a UI-based modeler. We recommend that you carefully evaluate the choices for microservices orchestrator based on your application's requirements. This choice is a critical part of running and scaling your microservices.

Refactored application

To enable distributed transactions in the refactored application, the checkout service handles the communication between the payment, shipping, and email service. The generic Business Process Model and Notation (BPMN) workflow uses the following flow:

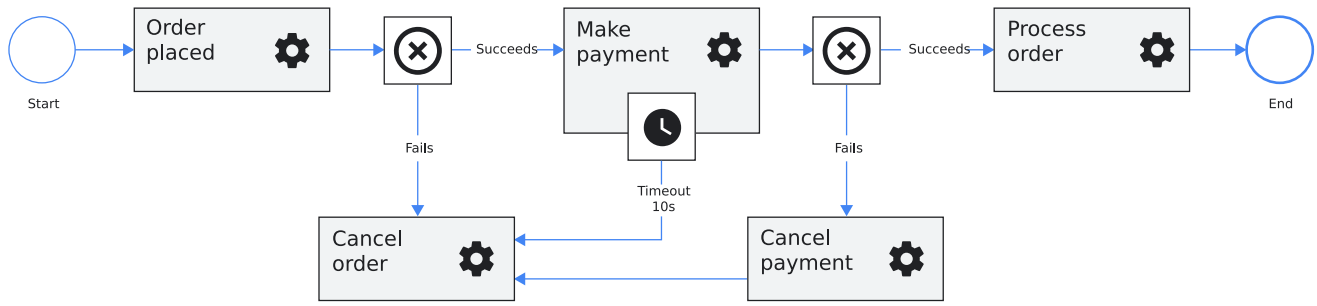

Figure 3. An order workflow that helps ensure distributed transactions in typical microservices.

The preceding diagram shows the following workflow:

- The frontend service receives an order request and then does the following:
  - Sends the order items to cart service. The cart service then saves the order details (Redis).
  - Redirects to checkout page. The checkout service pulls orders from the cart service, sets the order status as `Pending`, and asks the customer for payment.
  - Confirms that the user paid. Once confirmed, the checkout service tells the email service to generate a confirmation email and send it to the customer.
- The payment service subsequently processes the request.
  - If the payment request succeeds, the payment service updates the order status to `Complete`.
  - If the payment request fails, then payment service initiates a compensating transaction.
    - The payment request is canceled.
    - The checkout service changes the order status to `Failed`.
  - If the payment service is unavailable, the request times out after *N* seconds and the checkout service initiates a compensating transaction.
  - The checkout service changes the order status to `Failed`.

Objectives

- Deploy the monolithic Online Boutique application on Google Kubernetes Engine (GKE).
- Validate the monolithic checkout flow.
- Deploy the microservices version of the refactored monolithic application
- Verify that the new checkout flow works.
- Verify that the distributed transaction and compensation actions work if there is a failure.

Costs

In this document, you use the following billable components of Google Cloud:

- GKE
- Cloud SQL
- Container Registry

To generate a cost estimate based on your projected usage, use the pricing calculator. New Google Cloud users might be eligible for a free trial.

When you finish this document, you can avoid continued billing by deleting the resources you created. For more information, see Cleaning up.

Before you begin

1. In the Google Cloud console, on the project selector page, select or create a Google Cloud project.

   Go to project selector

2. Make sure that billing is enabled for your Google Cloud project.

3. In the Google Cloud console, activate Cloud Shell.

   Activate Cloud Shell

   At the bottom of the Google Cloud console, a Cloud Shell session starts and displays a command-line prompt. Cloud Shell is a shell environment with the Google Cloud CLI already installed and with values already set for your current project. It can take a few seconds for the session to initialize.

4. Enable the APIs for Compute Engine, Google Kubernetes Engine, Cloud SQL, Artifact Analysis, and Container Registry:

```
gcloud services enable \
    compute.googleapis.com \
    sql-component.googleapis.com \
    servicenetworking.googleapis.com\
    container.googleapis.com \
    containeranalysis.googleapis.com \
    containerregistry.googleapis.com \
    sqladmin.googleapis.com
```

5. Export the following environment variables:

```
export PROJECT=$(gcloud config get-value project)
export CLUSTER=$PROJECT-gke
export REGION="us-central1"
```

Deploying the ecommerce monolith

In this section, you deploy the monolithic Online Boutique application in a GKE cluster. The application uses Cloud SQL as its relational database. The following diagram illustrates the monolithic application architecture:

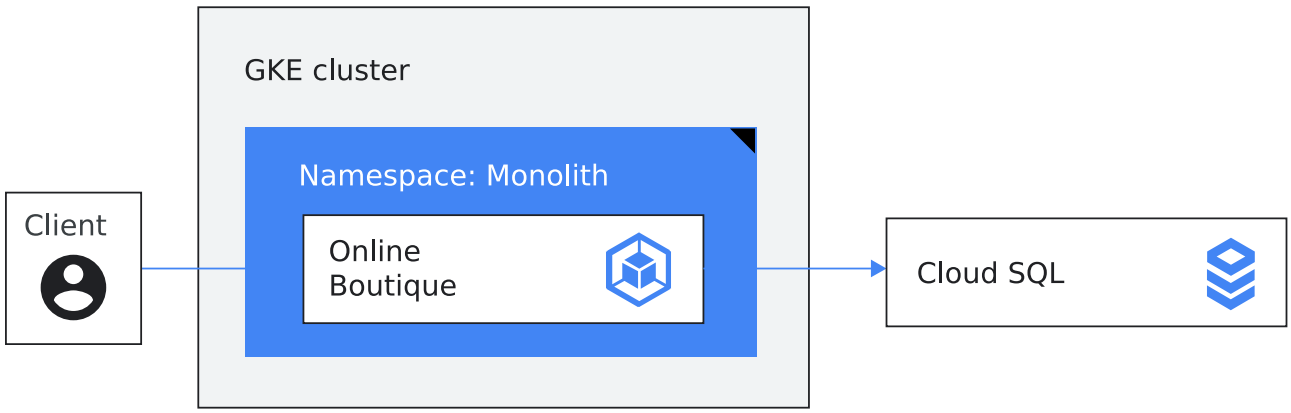

Figure 4. A client connects to the application in a GKE cluster, and the application connects to a Cloud SQL database.

To deploy the application, complete the following steps:

1. Clone the GitHub repository:

```
git clone https://github.com/GoogleCloudPlatform/monolith-to-microservices-example
```

2. Replace the `PROJECT_ID` placeholder in the Terraform variables manifest file:

```
cd monolith-to-microservices-example/setup && \
sed -i -e "s/\[PROJECT_ID\]/$PROJECT/g" terraform.tfvars
```

3. Run the Terraform scripts to complete the infrastructure setup and deploy the infrastructure. To learn more about Terraform, see Getting started with Terraform on Google Cloud:

```
terraform init && terraform apply -auto-approve
```

   The Terraform script creates the following:

   - A VPC network named *PROJECT_ID*-vpc
   - GKE cluster named *PROJECT_ID*-gke
   - A Cloud SQL instance named *PROJECT_ID*-mysql

- A database named `ecommerce` that the application uses
- A user `root` with the password set to `password`

You can modify the Terraform script to auto-generate a password. This setup uses a simplified example that you shouldn't use in production.

Infrastructure provisioning can take up to 10 minutes. When the script is successful, the output looks like the following:

```
...

Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

kubernetes_cluster_name = PROJECT_ID-gke
sql_database_name = PROJECT_ID-mysql
vpc_name = PROJECT_ID-vpc
```

4. Connect to the cluster and create a namespace named `monolith`. You deploy the application in its own namespace in the GKE cluster:

```
gcloud container clusters get-credentials $CLUSTER \
   --region $REGION \
   --project $PROJECT && \
   kubectl create ns monolith
```

5. The application running on GKE uses Kubernetes Secrets to access the Cloud SQL database. Create a secret that uses the user credentials for the database:

```
kubectl create secret generic dbsecret \
  --from-literal=username=root \
  --from-literal=password=password -n monolith
```

6. Build the monolith image and upload it to Container Registry:

```
cd ~/monolith
gcloud builds submit --tag gcr.io/$PROJECT_ID/ecomm
```

7. Update the reference in the `deploy.yaml` file to the newly created Docker image:

```
cd ~/monolith
sed -i -e "s/\[PROJECT_ID\]/$PROJECT/g" deploy.yaml
```

8. Replace placeholders in deployment manifest files and then deploy the application:

```
cd .. && \
DB_IP=$(gcloud sql instances describe $PROJECT-mysql | grep "ipAddress:" | tail -1 | awk -F ":" '{print
$NF}')

sed

-i -e "s/\[DB_IP\]/$DB_IP/g" monolith/deploy.yaml
kubectl apply -f monolith/deploy.yaml
```

9. Check the status of the deployment:

```
kubectl rollout status deployment/ecomm -n monolith
```

The output looks like the following.

```
Waiting for deployment "ecomm" rollout to finish: 0 of 1 updated replicas are available...
deployment "ecomm" successfully rolled out
```

10. Get the IP address of the deployed application:

```
kubectl get svc ecomm -n monolith \
        -o jsonpath="{.status.loadBalancer.ingress[*].ip}" -w
```

Wait for the load balancer IP address to be published. To exit the command, press `Ctrl+C`. Note the load balancer IP address and then access the application at the URL `http://IP_ADDRESS`. It might take some time for the load balancer to become healthy and start passing traffic.

Validate the monolith checkout flow

In this section, you create a test order to validate the checkout flow.

1. Go to the URL that you noted in the previous section, `http://IP_ADDRESS`.
2. On the application home page that appears, select any product and then click **Add to Cart**.
3. To create a test purchase, click **Place your order**:
4. When checkout is successful, the order confirmation window appears and displays an Order Confirmation ID.
5. To view order details, connect to the database:

```
gcloud sql connect $PROJECT-mysql --user=root
```

   You can also use any other supported methods to connect to the database. When prompted, enter the password as `password`.

6. To view saved order details, run the following command:

```
select cart_id from ecommerce.cart;
```

7. The output looks like the following:

```
+--------------------------------------+
| cart_id                              |
+--------------------------------------+
| 7cb9ab11-d268-477f-bf4d-4913d64c5b27 |
+--------------------------------------+
```

Deploying the microservices-based ecommerce application

In this section, you deploy the refactored application. This document only focuses on decoupling frontend and payment services. The next document in this series, Distributed tracing in a microservices application, describes other services, such as recommendation and ads services, that you can decouple from the monolith. The checkout service handles the distributed transactions between the frontend and the payment services and is deployed as a Kubernetes service in the GKE cluster, as shown in the following diagram:

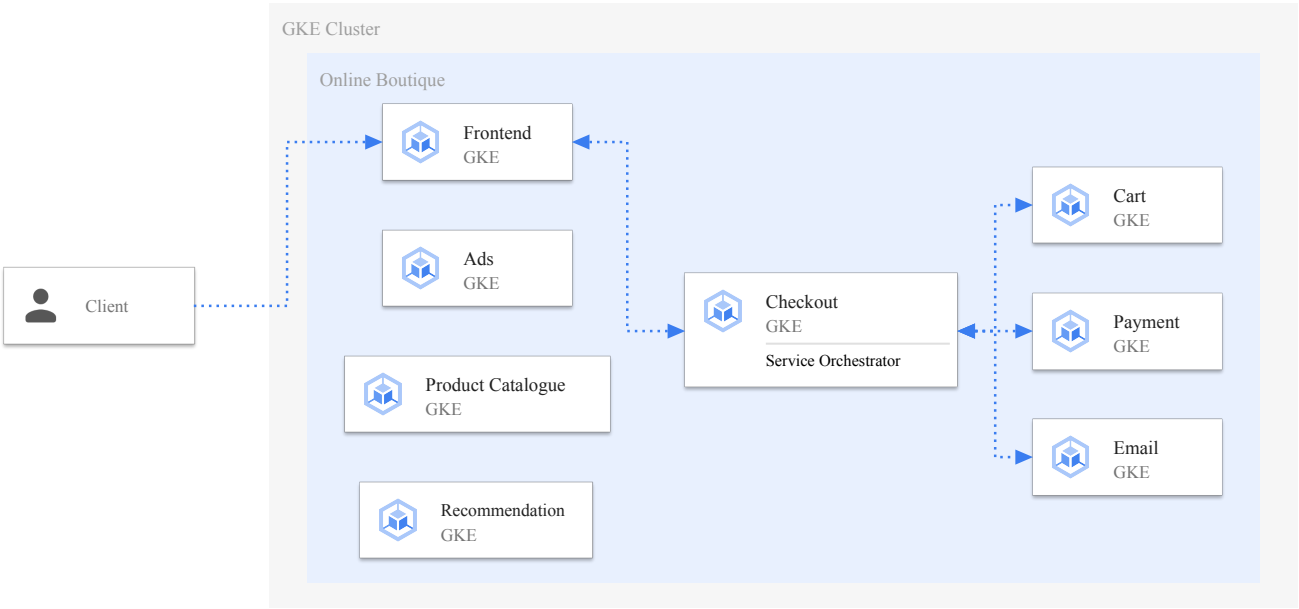

Figure 5. The checkout service orchestrates transactions between the cart, payment, and email services.

Deploy the microservices

In this section, you use the infrastructure that you provisioned earlier to deploy microservices in their own namespace `microservice`:

1. Ensure that you have the following requirements:
   - Google Cloud project

- Shell environment with `gcloud`, `git`, and `kubectl`

2. In Cloud Shell, clone the microservices repository:

```
git clone https://github.com/GoogleCloudPlatform/microservices-demo
cd microservices-demo/
```

3. Set the Google Cloud project and region and ensure the GKE API is enabled:

```
export PROJECT_ID=PROJECT_ID
export REGION=us-central1
gcloud services enable container.googleapis.com \
--project=${PROJECT_ID}
```

Substitute with the ID of your Google Cloud project.

4. Create a GKE cluster and get the credentials for it:

```
gcloud container clusters create-auto online-boutique \
    --project=${PROJECT_ID} --region=${REGION}
```

Creating the cluster may take a few minutes.

5. Deploy microservices to the cluster:

```
kubectl apply -f ./release/kubernetes-manifests.yaml
```

6. Wait for the pods to be ready:

```
kubectl get pods
```

After a few minutes, you see the Pods in a `Running` state.

7. Access the web frontend in a browser using the frontend's external IP address:

```
kubectl get service frontend-external | awk '{print $4}'
```

Visit `http://`*`EXTERNAL_IP`* in a web browser to access your instance of Online Boutique.

Validate the new checkout flow

1. To verify the checkout process flow, select a product and place an order, as described in the earlier section Validate the monolith checkout flow.
2. When you complete order checkout, the confirmation window doesn't display a confirmation ID. Instead, the confirmation window directs you to check your email for confirmation details.

3. To verify that the order was received, that the payment service processed the payment, and that the order details were updated, run the following command:

```
kubectl logs -f deploy/checkoutservice --tail=100
```

The output looks like the following:

```
[...]
{"message":"[PlaceOrder] user_id=\"98828e7a-b2b3-47ce-a663-c2b1019774a3\"
user_currency=\"CAD\"","severity":"info","timestamp":"2023-08-10T04:19:20.498893921Z"}
{"message":"payment went through (transaction_id: f0b4a592-026f-4b4a-9892-
ce86d2711aed)","severity":"info","timestamp":"2023-08-10T04:19:20.528338189Z"}
{"message":"order confirmation email sent to
\"someone@example.com\"","severity":"info","timestamp":"2023-08-10T04:19:20.540275988Z"}
```

To exit the logs, press `Ctrl+C`.

4. Verify that the payment was successful:

```
kubectl logs -f deploy/paymentservice -n --tail=100
```

The output looks like the following:

```
[...]
{"severity":"info","time":1691641282208,"pid":1,"hostname":"paymentservice-65cc7795f6-
r5m8r","name":"paymentservice-charge","message":"Transaction processed: visa ending 0454     Amount:
CAD119.30128260"}
{"severity":"info","time":1691641300051,"pid":1,"hostname":"paymentservice-65cc7795f6-
r5m8r","name":"paymentservice-server","message":"PaymentService#Charge invoked with request
{\"amount\":{\"currency_code\":\"USD\",\"units\":\"137\",\"nanos\":850000000},\"credit_card\":
{\"credit_card_number\":\"4432-8015-6152-
0454\",\"credit_card_cvv\":672,\"credit_card_expiration_year\":2039,\"credit_card_expiration_month\":1}}"}
```

To exit the logs, press `Ctrl+C`.

5. Verify that order confirmation email is sent:

```
kubectl logs -f deploy/emailservice -n --tail=100
```

The output looks like the following:

```
[...]
{"timestamp": 1691642217.5026057, "severity": "INFO", "name": "emailservice-server", "message": "A
request to send order confirmation email to kalani@examplepetstore.com has been received."}
```

The log messages for each microservices indicate that the distributed transaction across the checkout, payment, and email services have completed successfully.

Validate compensation action in a distributed transaction

This section simulates a scenario in which a customer is placing an order and the payment service goes down.

1. To simulate the service's unavailability, delete the payment deployment and service:

```
kubectl delete deploy paymentservice && \
kubectl delete svc paymentservice
```

2. Access the application again and complete the checkout flow. In this example, if the payment service doesn't respond, the request times out and a compensation action is triggered.

3. In the UI frontend, click the **Place Order** button. The output resembles the following:

```
HTTP Status: 500 Internal Server Error
rpc error: code = Internal desc = failed to charge card: could not charge the card: rpc error: code =
Unavailable desc = connection error: desc = "transport: Error while dialing: dial tcp: lookup
paymentservice on 34.118.224.10:53: no such host"
failed to complete the order
main.(*frontendServer).placeOrderHandler
    /src/handlers.go:360
```

4. Review the frontend service logs:

```
kubectl logs -f deploy/frontend --tail=100
```

The output resembles the following:

```
[...]
{"error":"failed to complete the order: rpc error: code = Internal desc = failed to charge card: could
not charge the card: rpc error: code = Unavailable desc = connection error: desc = \"transport: Error
while dialing: dial tcp: lookup paymentservice on 34.118.224.10:53: no such
host\"","http.req.id":"0a4cb058-ee9b-470a-9bb1-
3a965636022e","http.req.method":"POST","http.req.path":"/cart/checkout","message":"request
error","session":"96c94881-a435-4490-9801-c788dc400cc1","severity":"error","timestamp":"2023-08-
11T18:25:47.127294259Z"}
```

5. Review the Checkout Service logs:

```
kubectl logs -f deploy/frontend --tail=100
```

The output resembles the following:

```
[...]
{"message":"[PlaceOrder] user_id=\"96c94881-a435-4490-9801-c788dc400cc1\"
user_currency=\"USD\"","severity":"info","timestamp":"2023-08-11T18:25:46.947901041Z"}
{"message":"[PlaceOrder] user_id=\"96c94881-a435-4490-9801-c788dc400cc1\"
user_currency=\"USD\"","severity":"info","timestamp":"2023-08-11T19:54:21.796343643Z"}
```

Notice that there's no subsequent call to email service to send notification. There is no transaction log, like `payment went through (transaction_id: 06f0083f-fa47-4d91-8258-6d61edfab1ca)`

6. Review the email service logs:

```
kubectl logs -f deploy/emailservice --tail=100
```

Notice that there are no log entries created for the fail transaction on email service.

As an orchestrator, if a service call fails, the checkout service returns an error status and exits the checkout process.

Clean up

To avoid incurring charges to your Google Cloud account for the resources used in this tutorial, either delete the project that contains the resources, or keep the project and delete the individual resources.

If you plan to complete the steps in the next document of this series, Distributed tracing in a microservices application, you can reuse the project and resources instead of deleting them.

Delete the project

1. In the Google Cloud console, go to the **Manage resources** page.

   Go to Manage resources

2. In the project list, select the project that you want to delete, and then click **Delete**.
3. In the dialog, type the project ID, and then click **Shut down** to delete the project.

Delete the resources

If you want to keep the Google Cloud project that you used in this document, delete the individual resources.

1. In Cloud Shell, run the following command:

```
cd setup && terraform destroy -auto-approve
```

2. To delete the microservices cluster using the Google Cloud CLI, run the following command:

```
gcloud container clusters delete online-boutique \
    --location $REGION
```

What's next

- Learn more about microservices architecture.
- Read the first document in this series to learn about microservices, their benefits, challenges, and use cases.
- Read the second document in this series to learn about application refactoring strategies to decompose microservices.
- Read the final document in this series to learn about distributed tracing of requests between microservices.