



## **The Demo Java Microservices Application**



Welcome to the second module of the course “Building Scalable Java Microservices with Spring Boot and Spring Cloud.”

---

# Agenda

The Demo Application Architecture

Course Labs

In this module, I'll describe the structure and key components of the demo Java microservices application that you'll work with in the labs.

---

## Agenda

The Demo Application Architecture

[Course Labs](#)

You'll then start to work through the course labs.

---

## Learn how to ...

- Identify the components of the demo application and how they interact with each other

In this module, you'll learn what the components of the demo application are and how they interact with each other.

---

## Learn how to ...

- Identify the components of the demo application and how they interact with each other
- Integrate a range of Google Cloud services into Java applications with Spring Boot

In the labs, you'll learn how to integrate a range of Google Cloud services into Java applications with Spring Boot.

---

## Learn how to ...

- Identify the components of the demo application and how they interact with each other
- Integrate a range of Google Cloud services into Java applications with Spring Boot
- Deploy Spring Boot Java applications to App Engine and Google Kubernetes Engine

You'll also learn how to deploy Spring Boot Java applications to App Engine and Google Kubernetes Engine...

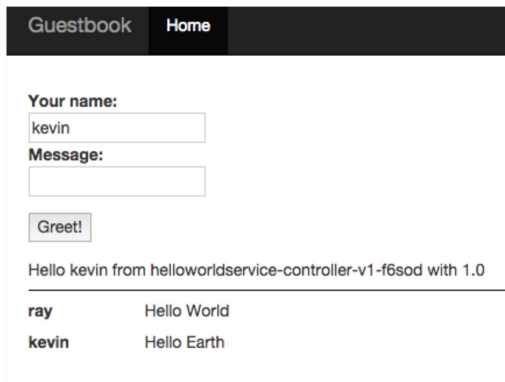
---

## Learn how to ...

- Identify the components of the demo application and how they interact with each other
- Integrate a range of Google Cloud services into Java applications with Spring Boot
- Deploy Spring Boot Java applications to App Engine and Google Kubernetes Engine
- Monitor deployed applications using Cloud Trace (previously Stackdriver Trace)

And monitor the deployed applications using Cloud Trace.

## The demo application



The screenshot shows a web application with a dark header bar containing two links: "Guestbook" and "Home". Below the header, there is a form with two input fields: "Your name:" with the value "kevin" and "Message:". A "Greet!" button is positioned below the message field. The output of the application is displayed below the button, showing a message: "Hello kevin from helloworldservice-controller-v1-f6sod with 1.0". At the bottom, there is a table with two rows of data:

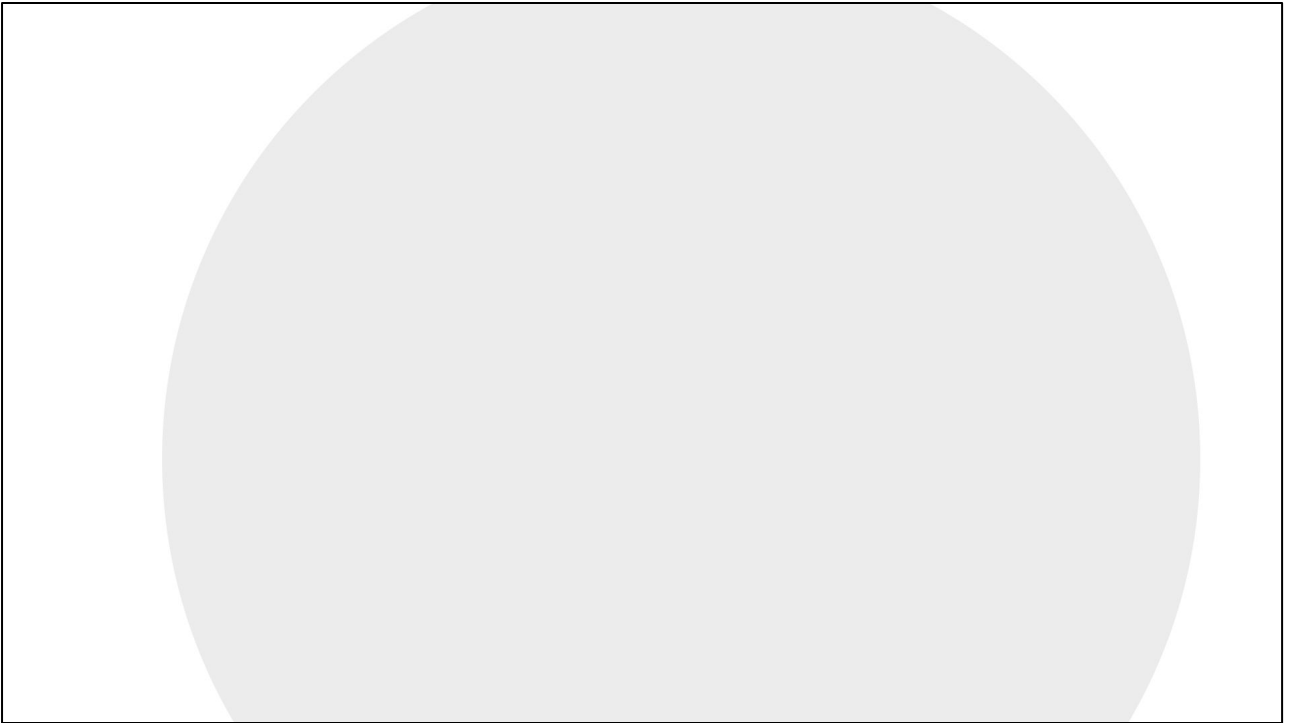
|       |             |
|-------|-------------|
| ray   | Hello World |
| kevin | Hello Earth |

The demo application used in this course is composed of two separate microservices-style component applications: the Guestbook frontend application and the Guestbook backend service.

With the initial default configuration, the two components communicate through the localhost network, assuming that they are both running on the same host.

Both components are written in Java with Spring Boot. The initial configuration of the components does not integrate with any Google or other cloud services.





The application can be run on any platform that has the appropriate Java edition and related libraries installed. The initial demo can be run on any such machine, and that includes Cloud Shell, by launching the frontend application and backend service in two separate console sessions.

## The demo application

Guestbook Home

Your name:

Message:

Greet!

Hello kevin from helloworldservice-controller-v1-f6sod with 1.0

|       |             |
|-------|-------------|
| ray   | Hello World |
| kevin | Hello Earth |

The Guestbook application delivers basic CRUD functionality, although only the Create and Retrieve functions are used. The Update and Delete functions aren't implemented because they're not needed for the purposes of the demo.

The frontend application provides a simple web form that allows a user to post a message composed of a user name and message body.

## The demo application

Guestbook Home

Your name:

Message:

Greet!

Hello kevin from helloworldservice-controller-v1-f6sod with 1.0

|       |             |
|-------|-------------|
| ray   | Hello World |
| kevin | Hello Earth |

When a message is submitted by the user, the frontend application sends it to the backend service. The backend service then creates a new record to store the message. The frontend also displays a list of previous messages retrieved from the backend service.

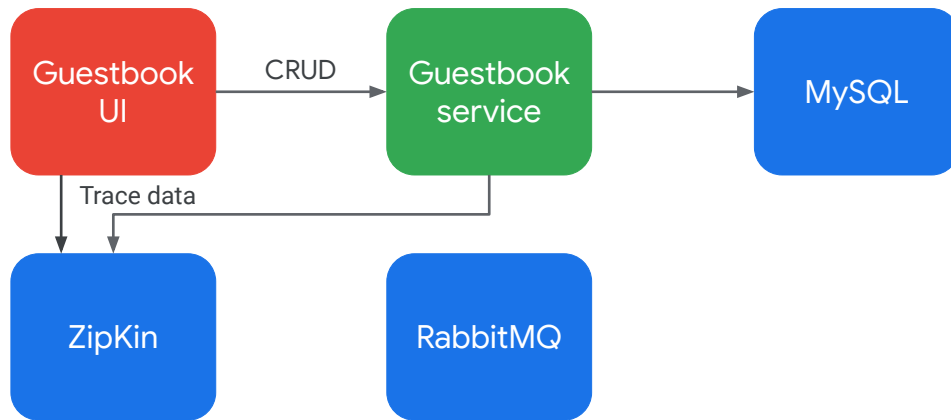
## Demo application architecture



The demo application follows a classic three-tier web architecture. The frontend application sends messages to the backend services to create, retrieve, update, and delete records. The single backend service in the demo sends responses back to the frontend and uses a database service for storage of the message content. The application is initially configured to use an embedded HSQL database.

Persistent database storage can be implemented using MySQL, or a similar database service, to replace the embedded HSQL database. In the first lab, you'll use Spring Boot to configure the application to use a Cloud SQL instance for the database functions that it needs to store and retrieve structured data.

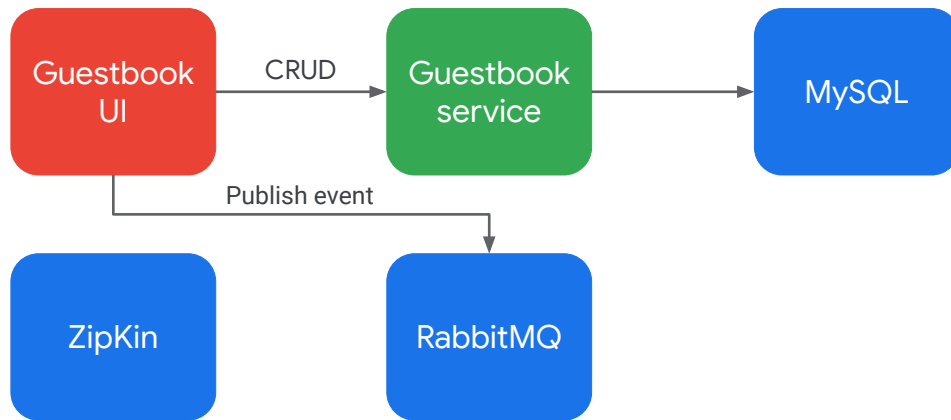
## Demo application architecture



Although the demo application is quite simple, it allows us to demonstrate how to implement distributed tracing, using a system like Zipkin. Zipkin is able to provide detailed timing data that can be used to troubleshoot latency issues with a microservices-style application like this. By implementing a Zipkin (or similar) solution yourself, you can provide the tracing services required to handle both the collection and analysis of this data.

In the labs, you'll use Spring Boot to configure the application to use Google's fully managed Cloud Trace solution. This enables you to implement tracing for your application without having to deploy and manage your own tracing services.

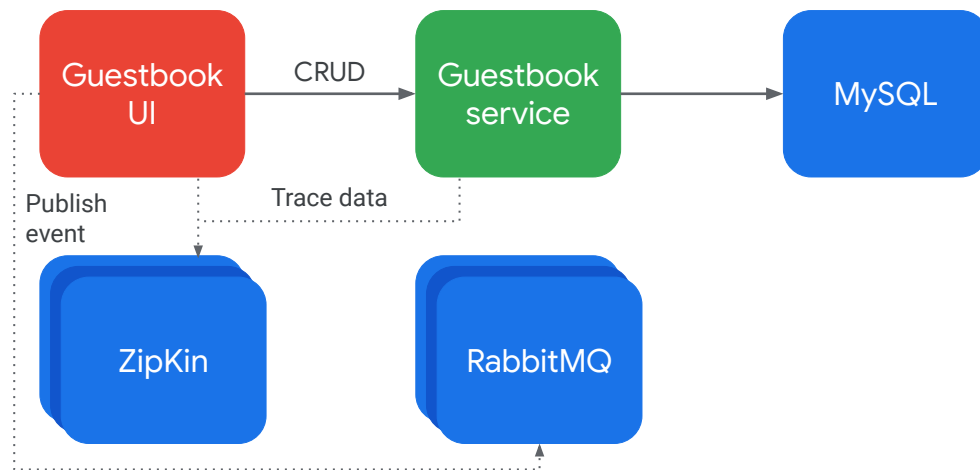
## Demo application architecture



The communication between services is a critical component of a microservices framework, and a message queuing system is critical to ensure reliable scaling. Message queuing solutions, like RabbitMQ, allow services to send or receive messages (or requests) asynchronously, instead of having to carry out resource-intensive processing at the same time. This helps to avoid component services getting stuck waiting to hand off a message. Message queueing makes it much easier to scale out component services independently because it allows you to distribute messages to multiple targets for processing or across multiple services for load-balancing. Message queuing also helps create connections between applications or services that are more independent, asynchronous, and therefore less tightly coupled.

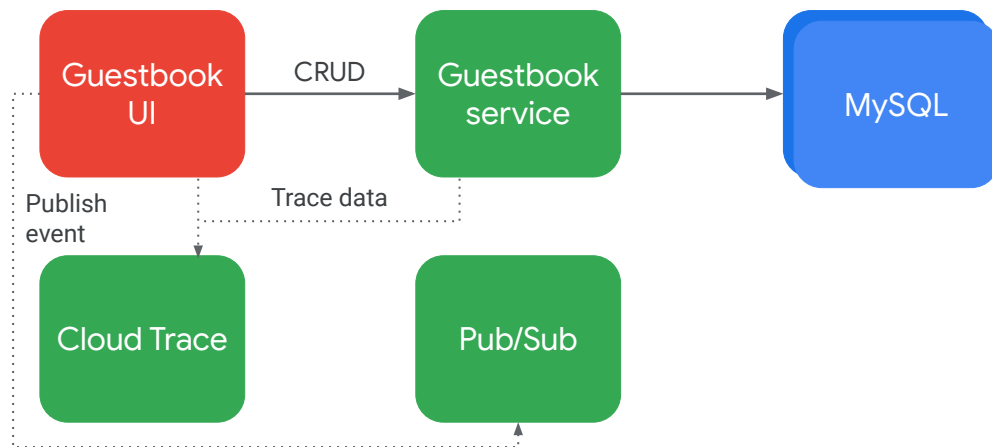
In the labs, you'll configure Pub/Sub as a message broker for the application. The frontend UI sends messages to a message queue that the backend service is subscribed to so that it can receive and process the messages asynchronously.

## Demo application architecture



These infrastructure services all require management and operational overhead in order to deliver scalable solutions. In a real world production environment, you'd need to consider the management of each of those systems, how they would need to be protected, how they need to be configured for scalability and resilience, how to manage software updates, and so on. That may or may not be practical in any given organization, but if you want to implement a solution that includes some or all of these components, that additional cost and operational overhead must be allowed for.

## Demo application architecture

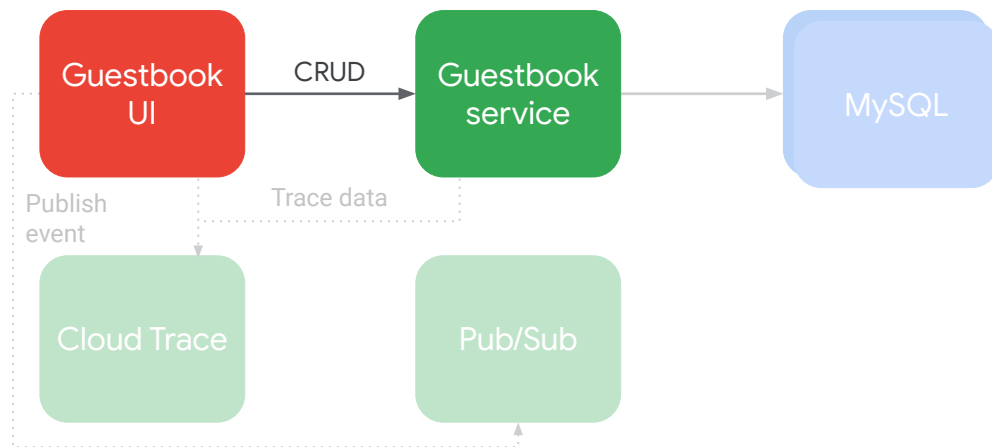


In contrast, adopting a cloud-native approach through the use of Google Cloud services for these capabilities, instead of building and managing your own, provides instant access to the capability without having to deploy any additional infrastructure yourself. This also avoids the overhead and complexity of managing the lifecycle of trace, message queueing, and database servers.

Google Cloud services provides the confidence that goes with knowing that the underlying infrastructure behind all of these services is not just fully managed, but resilient and implemented on a global scale.



## Demo application architecture



Ultimately, if you adopt a cloud-native approach, you can leave the management of those other services up to Google Cloud and focus on what you do best: developing the component parts of your application.

---

## Lab agenda

Lab 01 Bootstrapping the Application Frontend and Backend

Lab 02 Configuring and Connecting to Cloud SQL

Lab 03 Working with Cloud Trace

Lab 04 Messaging with Pub/Sub

Lab 05 Integrating Pub/Sub with Spring

Lab 06 Uploading and Storing Files

Lab 07 Using Cloud Platform APIs

You'll now start the labs that are the core of this course.

In this first group of labs, you'll modify and extend the application by configuring Spring Boot to use Google Cloud services. You'll also learn how to configure the services and authenticate to them from Java applications.

---

## Lab agenda, cont.

Lab 08 Deploying to App Engine

Lab 09 Working with Cloud Spanner

Lab 10 Deploying to Google Kubernetes Engine

Lab 11 Monitoring Google Kubernetes Engine with Prometheus

In this second group of labs, you'll learn how to deploy and monitor your application on App Engine or Google Kubernetes Engine.

You'll also learn how to add support for high-performance transactional database services using Cloud Spanner.

# Lab Intro

Bootstrapping the Application  
Frontend and Backend



In this lab, you'll start the two independent microservices components of the application by running them in the Cloud Shell using two separate Cloud Shell console tabs.

Cloud Shell is an interactive, Linux command-line shell environment for managing resources hosted on Google Cloud. When you start Cloud Shell, it provisions a small Compute Engine virtual machine running a Debian-based Linux operating system. Cloud Shell provides command-line access to the virtual machine instance in a terminal window that opens in the Cloud Console. You can open multiple shell connections to the same instance. You can use Cloud Shell code editor to browse file directories and view and edit files, with continued access to the Cloud Shell.

# Lab Intro

Configuring and Connecting to  
Cloud SQL



In this lab, you'll take the demo application and extend it to make use of Cloud SQL for database storage using the Java Persistence API, or JPA, and Hibernate with Spring Data Framework.

Cloud SQL is a fully managed database service that makes it easy to set up, maintain, manage, and administer your relational PostgreSQL and MySQL databases in the cloud. Cloud SQL offers high performance, scalability, and convenience. Hosted on Google Cloud, Cloud SQL provides a database infrastructure for applications running anywhere.

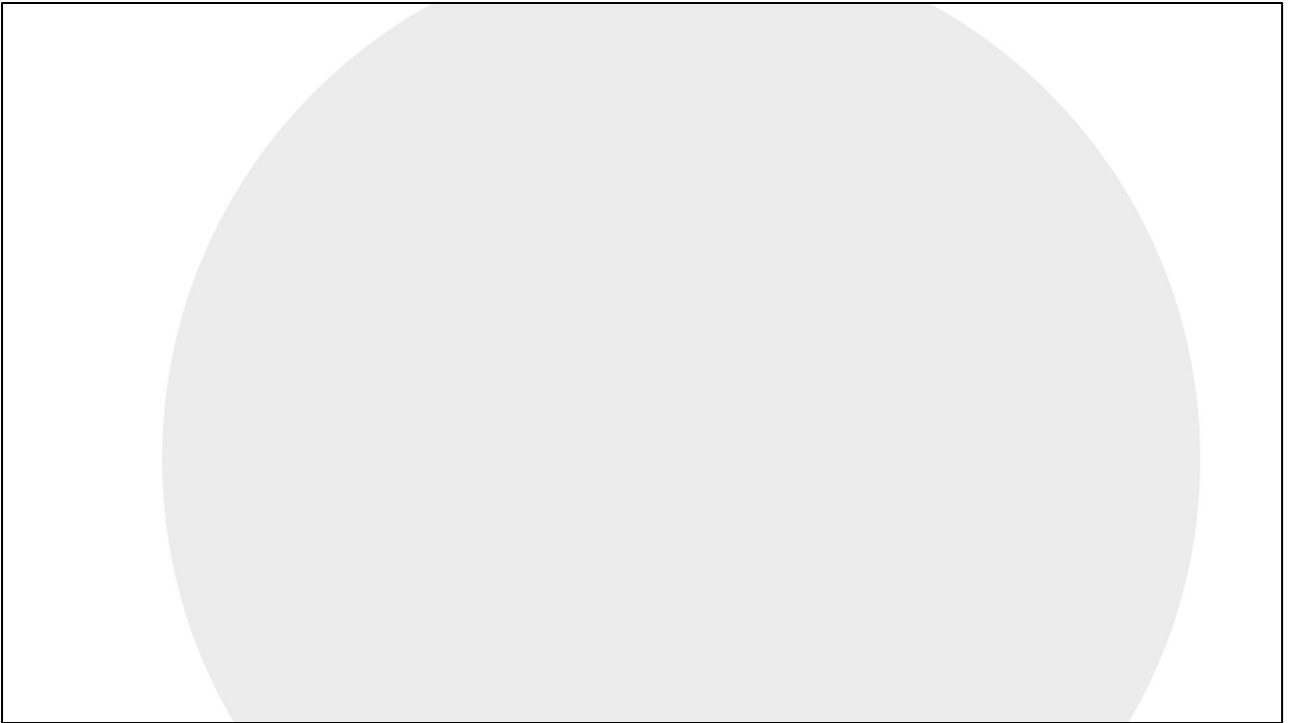
# Lab Intro

Working with Cloud Trace



In this lab, you'll add in distributed tracing across multiple services using Cloud Trace as the trace collection and analysis platform. You'll also learn how to handle authentication so that you can implement managed access to Google Cloud services from Java applications.

Trace is a distributed tracing system that collects latency data from your applications and displays it in the Cloud Console. You can track how requests propagate through your application and receive detailed near real-time performance insights. Trace automatically analyzes all of your application's traces to generate in-depth latency reports to surface performance degradations, and can capture traces from all of your VMs, containers, or App Engine projects.



The language-specific SDKs of Trace can analyze projects running on VMs, even those not managed by Google Cloud. The Trace SDK is currently available for Java, Node.js, Ruby, and Go, and the Cloud Trace API can be used to submit and retrieve trace data from any source. A Zipkin collector is also available that allows Zipkin tracers to submit data to Trace. Projects running on App Engine are automatically captured.

# Lab Intro

Messaging with Pub/Sub



In this lab, you'll learn how to configure Pub/Sub and create a simple application to demonstrate how to publish and receive messages.

Pub/Sub is a fully managed, real-time messaging service that allows you to send and receive messages between independent applications. Pub/Sub brings the scalability, flexibility, and reliability of enterprise message-oriented middleware to the cloud. By providing many-to-many, asynchronous messaging that decouples senders and receivers, it allows for secure and highly available communication between independently written applications. Pub/Sub delivers low-latency, durable messaging that helps developers quickly integrate systems hosted on Google Cloud and externally.



# Lab Intro

Integrating Pub/Sub with Spring



In this lab, you'll use Spring Integration to create a message gateway interface that abstracts from the underlying messaging system, instead of using direct integration with Pub/Sub.

Using this approach, you can swap messaging middleware that works with on-premises applications for messaging middleware that works with cloud-based applications. This approach also makes it easy to migrate between messaging middlewares. You will also use Spring Integration to add the message gateway interface and then refactor the code to use this interface, instead of implementing direct integration with Pub/Sub.

# Lab Intro

Uploading and Storing Files



In this lab, you extend the functionality of the application to allow users to upload images with their messages using a Spring Boot starter for Cloud Storage.

Cloud Storage provides fast, low-cost, highly durable, global object storage for developers and enterprises that need to manage unstructured file data. The consistent API of Cloud Storage, together with low latency and speed across storage classes, simplifies development integration and reduces code complexity.

# Lab Intro

Using Cloud Platform APIs



In this lab, you'll learn how to integrate other Google Cloud APIs that don't have a dedicated Spring Boot starter. You'll add support for the Vision API, which will then analyze the uploaded images automatically and classify the content it recognizes.

The Vision API enables developers to understand the content of an image by encapsulating powerful machine learning models in an easy-to-use REST API. It quickly classifies images into thousands of categories, detects individual objects and faces within images, and reads printed words contained within images. You can build metadata on your image catalog, moderate offensive content, or enable new marketing scenarios through image sentiment analysis. The Vision API allows your applications to easily detect broad sets of objects in your images: from flowers, animals, or transportation, to thousands of other object categories commonly found within images.

# Lab Intro

Deploying to App Engine



In this lab, you'll deploy the two component microservices to App Engine.

App Engine is Google's fully managed, serverless application platform. With App Engine, you can build and deploy applications on a fully managed platform without the worry of managing the underlying infrastructure. App Engine provides capabilities such as automatic scaling-up and scaling-down of your application, fully managed patching, and management of your servers.

# Lab Intro

Working with Cloud Spanner



In this lab, you'll practice how to migrate an application that uses Cloud SQL to Cloud Spanner.

Cloud Spanner is an enterprise-grade, globally distributed, and strongly consistent database service built for the cloud, specifically to combine the benefits of relational database structure with non-relational horizontal scale. This combination delivers high-performance transactions and strong consistency across rows, regions, and continents with an industry-leading 99.999% availability SLA, no planned downtime, and enterprise-grade security.

# Lab Intro

Deploying to GKE



In this lab, you'll create a Kubernetes cluster, package the application in containers, and deploy the containers to the Kubernetes cluster.

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services that facilitate both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Google Kubernetes Engine (GKE) is Google's managed, production-ready environment for deploying containerized applications. GKE enables rapid application development and iteration by making it easy to deploy, update, and manage your applications and services. GKE allows you to get up and running with Kubernetes in no time, by completely eliminating the need to install, manage, and operate your own Kubernetes clusters.

# Lab Intro

Working with Kubernetes Engine  
Monitoring



In this final lab, you'll enable Prometheus monitoring for GKE and modify the demo application to expose Prometheus metrics from within the application and its backend service.

Cloud Operations for GKE aggregates logs, events, and metrics from your GKE environment to help you understand your application's behavior in production. Prometheus is an optional monitoring tool often used with Kubernetes. If you configure Cloud Operations for GKE with Prometheus support, services that expose metrics in the Prometheus data model can be exported from the cluster and made visible as external metrics in Cloud Monitoring.