## Reference from Cloud Architect Learning Path

* A Cloud Developer designs, builds, analyzes, and maintains cloud-native applications.
* 1.Tour\_GC – Ref Cloud Architect
* 2. GC\_Fundamentals\_Core\_Infrastructure – Ref Cloud Architect
* 3. Getting Started With Application Development
* 4. Securing and Integrating Components of your Application
* 5. App Deployment, Debugging, and Performance
* 6. Application Development with Cloud Run
* 7. Getting Started with Google Kubernetes Engine – Ref Cloud Architect
* 8. Serverless Cloud Run Development
* 9. Serverless Firebase Development
* 10. Deploy to Kubernetes in Google Cloud3.Create and Manage Cloud Resources

## 3. Getting Started With Application Development

* application developers learn how to design and develop cloud-native applications that seamlessly integrate managed services from Google Cloud. Participants learn how to apply best practices for application development and use the appropriate Google Cloud storage services for object storage, relational data, caching, and analytics.

## 3.2 Best Practices for Application Development

* This module introduces best practices for application development.
* Loosely Couple Microservice and AI Gateways - https://youtu.be/Sj0RuNAzn5U
* Security, Reliability, and Migration 12 minutes - https://youtu.be/WWRElbV1HRw
* Practice Quiz: Best Practices for Application Development
  + You need to design a social application to reach a much broader audience than before. You want to achieve scalability, reliability, and security. Select two best practices that you can implement to build scalable, more secure, and highly available applications?
    - Manage your application’s code and environment by using a code repository and a dependency management system.
    - Design application components so that they are loosely coupled at runtime. Tightly coupled components can make an application less resilient to failures, spikes in traffic, and changes to services.
  + Which of the following statements is true?
    - For transient network errors, applications should implement retry logic with exponential backoff and fail gracefully if the errors persist.
    - When accessing services and resources in a distributed system, applications need to be resilient to temporary and long-lasting errors.
  + You have a mission-critical application that is accessed globally. You must make sure that your application is able to serve traffic reliably. What is the best way to check if your application is ready to serve traffic?
    - Implement a health-check endpoint for each service.
    - The endpoint handler should check the health of all dependencies and infrastructure components required for the service to function properly.
* Quiz: Final Quiz: Best Practices for Application Development
  + Which of the following is considered a best practice when developing cloud-native applications?
    - Implement API gateways to make backend functionality available to consumer applications.
  + Identify two key aspects of this application's architecture.
  + (Select all 2 correct answers).
    - The application will require longer development and QAcycles as the system grows in features and complexity.
    - The application is monolithic because all layers need to be deployed as a single unit.
  + Identify three key aspects of a sound architecture for a continuous integration and delivery (CI / CD) system. (Select all 3 correct answers)
    - Canary deployments can help catch unexpected issues before they affect a large number of users in production.
    - When a developer commits code into the code repository, a continuous integration tool such as Jenkins builds a container image for the application.
    - Builds are tested in the development environment and canary deployment before promoting to the production deployment.

## 3.3 Cloud Client Libraries, the Cloud SDK, and Firebase SDKs

* Cloud SDK, Cloud Client Libraries, & Firebase SDK 8 minutes - https://youtu.be/Y0jndUeQ5XE
* Practice Quiz: Google Cloud SDK, Google Cloud Client Libraries, & Firebase SDK
  + What are some uses for the API Explorer? Choose all that are correct (3 correct answers).
    - View details about the API request and response.
    - Execute an API method with some test parameter values.
    - Search for services and methods.
* Demo API Explorer walk-through 3 minutes - https://youtu.be/7r6erw5EHtY
* App Dev - Setting up a Development Environment: Node.js 2 hours
  + Overview
    - In this lab, you provision a Google Compute Engine virtual machine and install software libraries for Node.js software development on Google Cloud Platform (GCP).
  + Objectives
    - Provision a Google Compute Engine instance.
    - Connect to the instance using SSH.
    - Install software on the instance.
    - Verify the software installation.
  + Task 1: Create a Compute Engine Virtual Machine instance
    - Create Instance.
      * Name the instance dev-instance.
      * Set the Region to us-central1.
      * Set the Zone to us-central1-a.
      * Identity and API access > Access Scopes section, select Allow full access to all Cloud APIs.
      * In the Firewall section, enable Allow HTTP traffic.
      * Leave the remaining settings as their defaults, and click Create.
  + Task 2: Use the SSH to Install software on the VM instance
    - update the Debian package list, execute the following command:
      * sudo apt-get update
    - To install Git, execute the following command:
      * sudo apt-get install git
    - To download the Node.js setup script, execute the following command:
      * curl -sL https://deb.nodesource.com/setup\_12.x | sudo -E bash -
    - To install Node Package Manager (npm) and Node.js, execute the following command:
      * sudo apt install nodejs
  + Task 3: Configure the VM to Run application software
    - In this section, you verify the software installation and run some sample codes.
    - To check the version of Node.js, execute the following command:
      * node -v
    - To clone the class repository, execute the following command:
      * git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - Create a soft link as a shortcut to the working directory.
      * ln -s ~/training-data-analyst/courses/developingapps/v1.3/nodejs/devenv ~/devenv
    - Navigate to the directory that contains the sample files for this lab:
      * cd ~/devenv
    - To run a simple web server, execute the following command:
      * sudo node server/app.js
    - From VM , click on the External IP address, browser show a Hello GCP dev! msg from Node.js.
    - Return to the SSH window and stop the application by pressing Ctrl+C.
    - To install the Node.js library for Compute Engine, execute the following command:
      * npm install
    - To run a simple Node.js application that lists Compute Engine instances:
      * node list-gce-instances.js
    - Many details about your machine should appear in the terminal window.
    - Warning: If you try to do this on your own machine, it will not work if credentials have not been set up to access GCP on your machine.
* App Dev - Setting up a Development Environment: Java 2 hours (same above)
  + Overview
    - In this lab, you set up a Java development environment on GC Platform. You use Google Compute Engine to create a virtual machine (VM) and install software libraries for software development.
  + Objectives
    - Install a Java library on the instance.
  + Google Cloud Platform
    - Google Cloud Platform (GCP) consists of a set of physical assets, such as computers and hard disk drives, and virtual resources, such as virtual machines (VMs), that are contained in Google's data centers around the globe. Each data center location is in a global region. Regions include Central US, Western Europe, and East Asia. Each region is a collection of zones, which are isolated from each other within the region. Each zone is identified by a name that combines a letter identifier with the name of the region. For example, zone a in the East Asia region is named asia-east1-a.
    - This distribution of resources provides several benefits, including redundancy in case of failure and reduced latency by locating resources closer to clients. This distribution also introduces some rules about how resources can be used together.
    - Projects
      * Any GCP resources that you allocate and use must belong to a project. You can think of a project as the organizing entity for what you're building. A project is made up of the settings, permissions, and other metadata that describe your applications. Resources within a single project can work together easily, for example by communicating through an internal network, subject to the regions-and-zones rules. The resources that each project contains remain separate across project boundaries; you can only interconnect them through an external network connection.
      * Each GCP project has:
      * A project name, which you provide. A project ID, which you can provide or GCP can provide for you. A project number, which GCP provides. As you work with GCP, you'll use these identifiers in certain command lines and API calls.
      * The Google Cloud Platform Console displays project name, ID and number under Home > Dashboard.
      * Each project ID is unique across GCP. Once you have created a project, you can delete the project but its ID can never be used again.
      * When billing is enabled, each project is associated with one billing account. Multiple projects can have their resource usage billed to the same account.
      * A project serves as a namespace. This means every resource within each project must have a unique name, but you can usually reuse resource names if they are in separate projects. Some resource names must be globally unique. Refer to the documentation for the resource for details.
      * In this lab, you provision a Google Compute Engine virtual machine (VM) and install software libraries for Java software development on Google Cloud Platform (GCP).
    - Ways to interact with the services
      * GCP gives you three basic ways to interact with the services and resources.
      * Google Cloud Platform Console: a web-based, graphical user interface that you can use to manage your GCP projects and resources.
      * Command-line interface
        + Google Cloud SDK: provides the gcloud command-line tool, which gives you access to the commands you need.
        + Cloud Shell: a browser-based, interactive shell environment for GCP. You can access Cloud Shell from the GCP console. If you prefer to work in a terminal window, the Google Cloud SDK provides the gcloud command-line tool, which gives you access to the commands you need. The gcloud tool can be used to manage both your development workflow and your GCP resources. See the gcloud reference for the complete list of available commands.
      * Client libraries: The Cloud SDK includes client libraries that enable you to easily create and manage resources. GCP client libraries expose APIs to provide access to services and resource management functions. You also can use the Google API client libraries to access APIs for products such as Google Maps, Google Drive, and YouTube.
  + Task 1: Create a Compute Engine Virtual Machine instance
  + Install java 11 software and configure the VM instance
    - sudo apt-get update
    - Install Java 11:
      * sudo apt-get install -yq openjdk-11-jdk
    - Apply workaround for certificate issue in OpenJDK 11:
      * sudo sed -i 's/^\(keystore\.type\s\*=\s\*\).\*$/\1jks/' /etc/java-11-openjdk/security/java.security; sudo rm /etc/ssl/certs/java/cacerts; sudo /usr/sbin/update-ca-certificates -f
    - Install Git:
      * sudo apt-get install git -y
    - Install Maven:
      * sudo apt-get install -yq maven
    - Configure IP tables:
      * sudo iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 8080
    - This command (above) to configure the IP tables redirects requests on Port 80 to Port 8080 - the Java Web application listens on Port 8080.
    - Export the Project ID as an environment variable:
      * export GCLOUD\_PROJECT="$(curl -H Metadata-Flavor:Google http://metadata/computeMetadata/v1/project/project-id)"
  + Configure the VM to run application software
    - Clone the class repository:
      * git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - To keep the navigation simple, create a soft link as a shortcut to the working directory:
      * ln -s ~/training-data-analyst/courses/developingapps/v1.3/java/devenv ~/devenv
    - Change to the directory that contains the sample files:
      * cd ~/devenv
    - Run a simple web application:
      * mvn clean install
    - Run the application.
      * mvn spring-boot:run
    - From VM, click External IP address, browser opens to display a Hello GCP dev! msg from Java.
    - Return to the SSH window, and stop the application by pressing Ctrl+C.
    - To run a simple Java application that lists Compute Engine instances:
      * mvn exec:java@list-gce
    - Which one of the following protocol-port combination is responsible for allowing HTTP traffic?
      * tcp:80
* App Dev - Setting up a Development Environment: Python 2 hours (same as node & java)
  + In the SSH session, update the Debian package list:
    - sudo apt-get update
    - sudo apt-get install git
  + Install Python:
    - sudo apt-get install python3-setuptools python3-dev build-essential
  + Install pip:
    - curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
    - sudo python3 get-pip.py
  + Configure the VM to run application software
    - Clone the class repository:
      * git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
      * ln -s ~/training-data-analyst/courses/developingapps/v1.3/java/devenv ~/devenv
      * cd ~/devenv
    - Run a simple web server:
      * sudo python3 server.py
    - From VM, click External IP address, browser opens to display a Hello GCP dev! msg from Java.
    - Return to the SSH window, and stop the application by pressing Ctrl+C.
    - Install the Python packages needed to enumerate Compute Engine VM instances:
      * sudo pip3 install -r requirements.txt
    - Now list your instance in Cloud Shell. Enter the following command to run a simple Python application that lists Compute Engine instances. Replace <PROJECT\_ID> with your Google Cloud Project ID and <YOUR\_VM\_ZONE> is the region you specified when you created your VM. Find these values on the VM instances dialog of the console:
      * python3 list-gce-instances.py <PROJECT\_ID> --zone=<YOUR\_VM\_ZONE>
    - Your instance name shows in the SSH terminal window.
    - pip is a package management system used to install and manage software packages written in Python.
      * True
* Final Quiz: Cloud Client Libraries, Cloud SDK, and Firebase SDK
  + What is the gcloud command to list compute instances?
    - gcloud compute instances list
  + Which of the following statements about Google Cloud Client Libraries are accurate? (Select all 3 correct answers)
    - Cloud Client Libraries are helpful because they support a language's natural conventions and styles.
    - Cloud Client Libraries handle low-level communication, retry logic, and authentication.
    - Cloud Client Libraries are the latest and recommended approach to making requests to the server.
  + Your code throws errors because the actual response from a Google Cloud API is not what you expect. You want to quickly find the response values for a set of test parameter values. How can you execute the API with minimal effort to determine the response data?
    - Use the API Explorer to invoke the API with test parameter values.
  + What tools does the Google Cloud SDK include?
    - bq, gsutil, gcloud

## 3.4 Overview of Data Storage Options

* This module introduces the various data storage options available to your applications in Google Cloud.
* Cloud Storage, Datastore, Cloud Bigtable, Cloud SQL, and Cloud Spanner - https://youtu.be/VaBvsVm\_CT8
* BigQuery, Microsoft SQL Server images on GC, and Firebase Storage Options - https://youtu.be/bvKcWuVdY3g
* Demo: Connecting Securely to a Cloud SQL Database 7 minutes - https://youtu.be/VsfQu4a3voE
* Final Quiz: Data Storage Options
  + You have a very large database that you are primarily using for queries in a business intelligence application. You want to move the data to a fully-managed solution. Which data storage option is ideal for such use cases?
    - BigQuery
  + You are building a banking application that is expected to have a very large number of users across the world. When users make a deposit, they want to see the result of this deposit reflected immediately when they view their balance. What data storage option is ideal for storing account balance information for users?
    - Cloud Spanner is ideal because it supports strongly consistency reads in addition to horizontal scalability, low latency, and high throughput.
  + A restaurant in your neighborhood wants to put up a website that displays a menu, restaurant hours, and location on a map. You want to help set up the website. What is the best way to host the website on Google Cloud Platform?
    - Serve the website's content from a Cloud Storage bucket.

## 3.5 Best Practices for Using Datastore

* Datastore Concepts and Indexes 6 minutes - https://youtu.be/ywD1CRZx0F4
* Demo Explore Cloud Datastore 14 minutes - https://youtu.be/GSG-OenwkLU
* Design Considerations & Sharding 2 minutes - https://youtu.be/Monkzlq4c78
* Replication, Query Types, Transactions, and Handling Errors 5 minutes - https://youtu.be/Dbduhpg7xEA
* Demo Use Cloud Dataflow to bulk-load data into Cloud Datastore 8 minutes - <https://youtu.be/u-nLPQ85jFk>
* Lab overview - https://youtu.be/ePmtGpk1qwQ
* App Dev - Storing Application Data in Cloud Datastore: Node.js 2 hours
  + Overview
    - In this lab, you review the case study application, an online Quiz. You store application data for the Quiz application in Cloud Datastore.
    - The Quiz application skeleton has already been written for you. You clone a repository that contains the skeleton using Google Cloud Shell, review the code using the Cloud Shell code editor, and view it using the Cloud Shell web preview feature.
    - Then you modify the code that stores data to use Cloud Datastore.
  + Previewing the Case Study Application
    - To clone the repository for the class, execute the following command in Cloud Shell:
      * git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - Configure and run the case study application
      * ln -s ~/training-data-analyst/courses/developingapps/v1.3/nodejs/datastore ~/datastore
      * cd ~/datastore/start
      * export GCLOUD\_PROJECT=$DEVSHELL\_PROJECT\_ID
    - Install the application dependencies:
      * npm install
    - To run the application, execute the following command:
      * npm start
    - Review the case study application
      * In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
    - You should see the user interface for the web application. The three main parts to the application are:
      * Create Question
      * Take Test
      * Leaderboard
    - In the navigation bar, click Create Question.
      * You should see a simple form that contains textboxes for the question and answers with radio buttons to select the correct answer.
      * Quiz authors can add questions in this part of the application.
      * This part of the application is written as a server-side web application using the popular Node.js web application framework Express.
    - In the navigation bar, click Take Test.
      * The client application opens.
    - In the navigation bar, click GCP.
      * You should see a sample (Dummy) question.
      * Quiz takers can answer questions in this part of the application.
      * This part of the application is written as a client-side web application using the popular JavaScript framework AngularJS.
      * It receives JSON data from the server and uses JavaScript in the browser to display questions and collect answers.
    - To return to the server-side application, click Quite Interesting Quiz in the navigation bar.
  + Examining the Case Study Application Code
    - From Cloud Shell, click Open Editor.
    - Navigate to the datastore/start/server folder.
      * The application is a standard Node.js application written using the popular Express application framework.
    - Review the Express Web application
      * Select the datastore/start/server/app.js file.
        + This file contains the entrypoint for the Express application and registers the main application routes for creating questions and delivering quizzes to users.
      * Select the datastore/start/server/web-app folder.
        + This folder contains the Node.js modules for the server-side web application.
      * Select the datastore/start/server/web-app/questions.js file.
        + This file contains the handlers that display the form and collect form data posted by quiz authors in the web application.
      * In the questions.js file, find the handler that responds to HTTP POST requests for the /questions/add route.
        + Review the source code comments for a detailed description of this handler's functionality.
      * Select the datastore/start/server/web-app/views folder.
        + This folder contains templates for the web application user interface using the pug (formerly jade) layout engine.
      * View the datastore/start/server/web-app/views/questions/add.pug file.
        + This file contains the pug template for the Create Question form.
        + Notice how there is a select list to pick a quiz, textboxes where an author can enter the question and answers, and radio buttons to select the correct answer.
      * Select the datastore/start/server/api/index.js file.
        + This file contains the handler that sends JSON data to students taking a test.
        + Notice that the handlers also make use of the model object.
      * Select the datastore/start/server/gcp/datastore.js file.
        + This is the file where you write Datastore code to save and load quiz questions to and from Cloud Datastore.
        + This module will be imported into the web application and API as the model object for the web application and API
  + Adding Entities to Cloud Datastore
    - In this section, you write code to save form data in Cloud Datastore.
    - Important: Update code within the sections marked as follows:
      * // TODO
      * // END TODO
    - To maximize your learning, try to write the code without reference to the completed code block at the end of the section. In addition, review the code, inline comments, and related API documentation.
    - For more information, view the API documentation. - https://googleapis.dev/nodejs/datastore/latest/index.html
    - Create an App Engine application to provision Cloud Datastore
      * Return to Cloud Shell and press Ctrl+C to stop the application.
      * To create an App Engine application in your project, execute the following command:
        + gcloud app create --region "us-central"
    - Import and use the NodeJS Datastore module
      * Open the ...gcp/datastore.js file in the Cloud Shell code editor.
      * Load the config module from the parent folder.
      * Load the @google-cloud/datastore module.
      * Declare a Datastore client object named ds.
      * datastore.js
        + 'use strict';
        + // TODO: Load the ../config module
        + const config = require('../config');
        + // END TODO
        + // TODO: Load the @google-cloud/datastore module
        + const {Datastore} = require('@google-cloud/datastore');
        + // END TODO
        + // TODO: Create a Datastore client object, ds
        + // The Datastore(...) factory function accepts an options
        + // object which is used to specify which project's
        + // Datastore should be used via the projectId property.
        + // The projectId is retrieved from the config module. This
        + // module retrieves the project ID from the GCLOUD\_PROJECT
        + // environment variable.
        + const ds = new Datastore({
        + projectId: config.get('GCLOUD\_PROJECT')
        + });
        + // END TODO
    - Write code to create a Cloud Datastore entity
      * Declare a constant named kind, initialized with the value 'Question'.
      * datastore.js
        + // TODO: Declare a constant named kind
        + // The Datastore key is the equivalent of a primary key in a
        + // relational database.
        + // There are two main ways of writing a key:
        + // 1. Specify the kind, and let Datastore generate a unique
        + // numeric id
        + // 2. Specify the kind and a unique string id
        + const kind = 'Question';
        + // END TODO
      * In the create(...) function, remove the existing Promise.resolve({}) placeholder statement from the create(...) function.
      * Declare a constant called key to store the key for this entity.
      * Declare a constant named entity and initialize it with the key and the quiz question properties extracted from the form data.
      * Use the Datastore client object (ds) to save the entity by calling the save(entity) method.
      * datastore.js
        + // The create({quiz, author, title, answer1, answer2,
        + // answer3, answer4, correctAnswer}) function uses an
        + // ECMAScript 2015 destructuring assignment to extract
        + // properties from the form data passed to the function
        + function create({ quiz, author, title, answer1, answer2, answer3, answer4, correctAnswer }) {
        + // TODO: Remove Placeholder statement
        + // return Promise.resolve({});
        + // END TODO
        + // TODO: Declare the entity key,
        + // with a Datastore generated id
        + const key = ds.key(kind);
        + // END TODO
        + // TODO: Declare the entity object, with the key and data
        + const entity = {
        + key,
        + // The entity's members are represented in a data property.
        + // This is an array where each element represents one
        + // member in the entity. Each element is an object with a
        + // name and a value
        + data: [
        + { name: 'quiz', value: quiz },
        + { name: 'author', value: author },
        + { name: 'title', value: title },
        + { name: 'answer1', value: answer1 },
        + { name: 'answer2', value: answer2 },
        + { name: 'answer3', value: answer3 },
        + { name: 'answer4', value: answer4 },
        + { name: 'correctAnswer', value: correctAnswer },
        + ]
        + };
        + // END TODO
        + // TODO: Save the entity, return a promise
        + // The ds.save(...) method returns a Promise to the
        + // caller, as it runs asynchronously.
        + return ds.save(entity);
        + // END TODO
        + }
    - Run the application and create a Cloud Datastore entity
      * Save the ...gcp/datastore.js file, and then return to the Cloud Shell command prompt.
      * To start the application, execute the following command:
        + npm start
    - In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
    - Click Create Question.
    - Complete the form with the following values, and then click Save.

|  |  |
| --- | --- |
| **Form Field** | **Value** |
| Author | Your Name |
| Quiz | Google Cloud Platform |
| Title | Which company owns GCP? |
| Answer 1 | Amazon |
| Answer 2 | **Google** (select the Answer 2 radio button!) |
| Answer 3 | IBM |
| Answer 4 | Microsoft |

* + - You should be returned to the home page for the application.
    - Go to the Cloud Platform Console and, on the Navigation menu, click Datastore.
    - Your new question is in the Entities list!
  + Choose the two correct options to create a key in Cloud Datastore.
    - You can specify the kind and let Cloud Datastore generate a unique numeric ID.
    - You can specify the kind and a unique string ID.
    - You can ask Cloud Datastore to leave the key as a null value.
    - You can create an entity without a key.
  + Which method enables you to store entities in Cloud Datastore?
    - persist
    - save
  + Bonus: Querying Cloud Datastore
    - In this section, you write code to retrieve entity data from Cloud Datastore.
    - Write code to retrieve Cloud Datastore entities
  + Return to the code editor.
    - In the ...gcp/datastore.js file, remove the code from the list(quiz)method.
    - In the list(...) method, a query that retrieves Question entities for a specific quiz from Cloud Datastore.
    - Use the Datastore client to run the query, and assign the returned promise object to a constant.
    - Write a statement to return the promise.
    - Chain a then(...) method to the promise.
    - Write an arrow function in the then(...) method to retrieve the response from Cloud Datastore.
    - In the arrow function extract the results from the response.
    - Reshape the data by adding each entity ID and removing the correct answer from the data returned from Cloud Datastore.
    - Complete the code in the arrow function body to return a page of entities or an object that indicates that this is the last page of results.
  + Run the application and test the Cloud Datastore query
    - Save the ...gcp/datastore.js file, and then return to the Cloud Shell command.
    - Stop the application by pressing Ctrl+C.
    - Start the application.
    - In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
    - Replace the querystring at the end of the application's URL with /api/quizzes/gcp.
      * The URL will be in the form: https://8080-####-####.ql-####-####.cloudshell.dev/api/quizzes/gcp
      * You should see that JSON data has been returned to the client corresponding to the question you added in the web application!
    - Return to the application home page, and click Take Test.
    - Click GCP.
      * You should see that the quiz question has been formatted inside the client-side web application!
    - You can find the solution to the bonus in the lab's bonus folder.
* App Dev - Storing Application Data in Cloud Datastore: Java 2 hours (same as above)
  + Overview
    - Google Cloud Datastore is a NoSQL document database built for automatic scaling, high performance, and ease of application development. In this lab, you use Datastore to store application data for an online Quiz application. You also configure the application to retrieve from Datastore and display the data in the quiz.
    - The Quiz application skeleton has already been written. You clone the repository that contains the skeleton using Google Cloud Shell, review the code using the Cloud Shell editor, and view it using the Cloud Shell web preview feature. You then modify the code that stores data to use Cloud Datastore.
  + Objectives
    - Harness Cloud Shell as your development environment
    - Preview the application
    - Update the application code to integrate Cloud Datastore
  + Prepare the Quiz Application
    - git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - ln -s ~/training-data-analyst/courses/developingapps/v1.3/java/datastore ~/datastore
    - cd ~/datastore/start
    - export GCLOUD\_PROJECT=$DEVSHELL\_PROJECT\_ID
    - Install the application dependencies:
      * mvn clean install
    - Run the application:
      * mvn spring-boot:run
  + Review the code
    - The application is a standard Java application written using the popular Spring Boot application framework.
    - Navigate to the /training-data-analyst/courses/developingapps/v1.3/java/datastore/start folder using the file browser panel on the left side of the editor.
      * This is the root folder for the application.
      * In the datastore folder, notice the end folder. The end folder contains the same files as the start folder, but each file in the end folder contains the complete code required to perform this lab.
    - From the start folder, navigate to the src/main/java/com/google/training/appdev folder using the file browser panel on the left side of the editor.
      * The paths for Java source code files are relative to the appdev folder.
  + Review the Spring Boot Web application
    - Select the .../QuizApplication.java file.
      * In this file, the class contains the entrypoint for the Spring Boot application.
    - Select the .../services/gcp/domain/Question.java file.
      * In this file, the domain class represents question data submitted in the question form and questions displayed when taking a quiz.
    - Select the .../web/QuestionsController.java file.
      * This file contains the handlers that display the form and collect form data posted by quiz authors in the web application.
      * In the QuestionsController.java file, find the handler that responds to HTTP POST requests for the /questions/add route.
      * Notice that the controller delegates the implementation of the handler to a service, questionService.
    - Navigate to the /training-data-analyst/courses/developingapps/v1.3/java/datastore/start/src/main/resources folder using the file browser panel on the left side of the editor.
      * This folder contains templates for the web application user interface and static content displayed in the client-side web application.
    - Select the templates folder.
      * This folder contains the template for the web application user interface using the Thyme templating engine.
    - Select the .../templates/new\_question.html file.
      * This file contains the template for the Create Question form. Notice how there is a select list to pick a quiz, textboxes where an author can enter the question and answers, and radio buttons to select the correct answer.
      * Return to the folder containing Java source code using the file browser panel on the left side of the editor. (Do you remember? It's start/src/main/java/com/google/training/appdev.)
    - Select the .../api/QuizEndpoint.java file.
      * This file contains the handler that sends JSON data to students taking a test. Notice that the handlers also make use of the questionService object.
    - Select the .../services/gcp/datastore/QuestionService.java file.
      * This is the file where you write Datastore code to save and load quiz questions to and from Cloud Datastore. The web application and API use this class.
  + Add Entities to Cloud Datastore
    - In this section, you write code to save form data in Cloud Datastore.
    - Important: Update code within the `// TODO` and `// END TODO` comment lines. To maximize your learning, review the code, inline comments, and related API documentation.
  + Create an App Engine application to provision Cloud Datastore
    - From Cloud Shell, click on the Open Terminal icon and stop the application by pressing Ctrl+C.
    - To create an App Engine application in your project, execute the following command in Cloud Shell:
      * gcloud app create --region "us-central"
  + Import and use the Java Datastore package
    - Open the .../services/gcp/datastore/QuestionService.java file in the Cloud Shell editor.
    - Write a star import for the com.google.cloud.datastore.\* package.
      * // TODO: Import the com.google.cloud.datastore.\* package
      * import com.google.cloud.datastore.\*;
      * // END TODO
    - Declare a Datastore client object named datastore and initialize it.
      * // TODO: Create a Datastore client object, datastore
      * // The DatastoreOptions class has a getDefaultInstance()
      * // static method.
      * // Use the getService() method of the DatastoreOptions
      * // object to get the Datastore client
      * private Datastore datastore =
      * DatastoreOptions.getDefaultInstance().getService();
      * // END TODO
  + After the updates, the first part of QuestionService.java is as follows:
    - package com.google.training.appdev.services.gcp.datastore;
    - // TODO: Import the com.google.cloud.datastore.\* package
    - import com.google.cloud.datastore.\*;
    - // END TODO
    - import com.google.training.appdev.services.gcp.domain.Question;
    - import java.util.ArrayList;
    - import java.util.Iterator;
    - import java.util.List;
    - import org.springframework.stereotype.Service;
    - @Service
    - public class QuestionService {
    - // TODO: Create a Datastore client object, datastore
    - // The DatastoreOptions class has a getDefaultInstance()
    - // static method.
    - // Use the getService() method of the DatastoreOptions
    - // object to get the Datastore client
    - private Datastore datastore =
    - DatastoreOptions.getDefaultInstance().getService();
    - // END TODO
  + Write code to create a Cloud Datastore entity
    - Declare a static final string named ENTITY\_KIND, initialized with the value "Question".
      * // TODO: Declare a static final String named kind
      * //The Datastore key is the equivalent of a primary key in a // relational database.
      * // There are two main ways of writing a key:
      * // 1. Specify the kind, and let Datastore generate a unique // numeric id
      * // 2. Specify the kind and a unique string id
      * private static final String ENTITY\_KIND = "Question";
      * // END TODO
    - Create a KeyFactory for Question entities.
      * // TODO: Create a KeyFactory for Question entities
      * private final KeyFactory keyFactory =
      * datastore.newKeyFactory().setKind(ENTITY\_KIND);
      * // END TODO
    - In the createQuestion(Question question) method, modify the method's return type to Key.
      * // TODO: Modify return type to Key
      * public Key createQuestion(Question question) {
      * // END TODO
    - Declare a key with an allocated ID for the Question entity using the datastore client and Key Factory.
      * // TODO: Declare the entity key,
      * // with a Datastore allocated id
      * Key key = datastore.allocateId(keyFactory.newKey());
      * // END TODO
    - Declare an entity named questionEntity, and initialize it using an entity builder.
      * // TODO: Declare the entity object, with the key and data
      * // The entity's members are set using the Entity.Builder.
      * // This has a set method for property names and values
      * // Values are retrieved from the Domain object
      * Entity questionEntity = Entity.newBuilder(key)
      * .set(Question.QUIZ, question.getQuiz())
      * .set(Question.AUTHOR, question.getAuthor())
      * .set(Question.TITLE, question.getTitle())
      * .set(Question.ANSWER\_ONE,question.getAnswerOne())
      * .set(Question.ANSWER\_TWO, question.getAnswerTwo())
      * .set(Question.ANSWER\_THREE,question.getAnswerThree())
      * .set(Question.ANSWER\_FOUR, question.getAnswerFour())
      * .set(Question.CORRECT\_ANSWER,
      * question.getCorrectAnswer())
      * .build();
      * // END TODO
    - Use the Datastore client object (datastore) to save the entity by calling its put(questionEntity) method.
      * // TODO: Save the entity
      * datastore.put(questionEntity);
      * // END TODO
    - Modify the return statement to return the key for the entity.
      * // TODO: Return the key
      * return key;
      * // END TODO
    - Save the file.
    - The following is the QuestionService.java content with all updates to this point.
      * // The createQuestion(Question question) method
      * // is passed a Question object using data from the form
      * // Extract the form data and add it to Datastore
      * // TODO: Modify return type to Key
      * public Key createQuestion(Question question) {
      * // END TODO
      * // TODO: Declare the entity key,
      * // with a Datastore allocated id
      * Key key = datastore.allocateId(keyFactory.newKey());
      * // END TODO
      * // TODO: Declare the entity object, with the key and data
      * // The entity's members are set using the Entity.Builder.
      * // This has a set method for property names and values
      * // Values are retrieved from the Domain object
      * Entity questionEntity = Entity.newBuilder(key)
      * .set(Question.QUIZ, question.getQuiz())
      * .set(Question.AUTHOR, question.getAuthor())
      * .set(Question.TITLE, question.getTitle())
      * .set(Question.ANSWER\_ONE,question.getAnswerOne())
      * .set(Question.ANSWER\_TWO, question.getAnswerTwo())
      * .set(Question.ANSWER\_THREE,question.getAnswerThree())
      * .set(Question.ANSWER\_FOUR, question.getAnswerFour())
      * .set(Question.CORRECT\_ANSWER,
      * question.getCorrectAnswer())
      * .build();
      * // END TODO
      * // TODO: Save the entity
      * datastore.put(questionEntity);
      * // END TODO
      * // TODO: Return the key
      * return key;
      * // END TODO
    - Run the application and create a Cloud Datastore entity
      * mvn spring-boot:run
    - In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
    - Click Create Question, complete the form with the following values, and then click Save.
    - In the Console, click Navigation menu > Datastore > Entities to see your new question!
  + Query Cloud Datastore
    - In this section, you write code to retrieve entity data from Cloud Datastore.
    - Write code to retrieve Cloud Datastore entities
    - From Cloud Shell, click on the Open Editor icon. Move to the getAllQuestions(String quiz) method in the .../services/gcp/datastore/QuestionService.java file, and remove the code for the existing Dummy questions.
      * // TODO: Remove this code
      * List<Question> questions = new ArrayList<>();
      * Question dummy = new Question.Builder()
      * .withQuiz("gcp")
      * .withAuthor("Dummy Author")
      * .withTitle("Dummy Title")
      * .withAnswerOne("Dummy Answer One")
      * .withAnswerTwo("Dummy Answer Two")
      * .withAnswerThree("Dummy Answer Three")
      * .withAnswerFour("Dummy Answer Four")
      * .withCorrectAnswer(1)
      * .withId(-1)
      * .build();
      * questions.add(dummy);
      * return questions;
      * // END TODO
    - Still in the getAllQuestions(String quiz)method, create a query object and initialize it with a query that retrieves Question entities for a specific quiz from Cloud Datastore.
      * // TODO: Create the query
      * // The Query class has a static newEntityQueryBuilder()
      * // method that allows you to specify the kind(s) of
      * // entities to be retrieved.
      * // The query can be customized to filter the Question
      * // entities for one quiz.
      * Query<Entity> query = Query.newEntityQueryBuilder()
      * .setKind(ENTITY\_KIND)
      * .setFilter(StructuredQuery.PropertyFilter.eq(
      * Question.QUIZ, quiz))
      * .build();
      * // END TODO
    - Call the Datastore client object's datastore.run(query) method, and assign the result to entity iterator named entities.
      * // TODO: Execute the query
      * // The datastore.run(query) method returns an iterator
      * // for entities
      * Iterator<Entity> entities = datastore.run(query);
      * // END TODO
    - Return the transformed results, using buildQuestions(entities) method to convert Datastore entities to domain objects.
      * // TODO: Return the transformed results
      * // Use the buildQuestions(entities) method to convert
      * // from Datastore entities to domain objects
      * return buildQuestions(entities);
      * // END TODO
    - Uncomment the buildQuestions(...) and entityToQuestion(...) helper methods provided in the QuestionService class and use them to map the iterator to a list of question domain objects.
    - The following is the updated QuestionService.java:
      * public List<Question> getAllQuestions(String quiz){
      * // TODO: Create the query
      * // The Query class has a static newEntityQueryBuilder()
      * // method that allows you to specify the kind(s) of
      * // entities to be retrieved.
      * // The query can be customized to filter the Question
      * // entities for one quiz.
      * Query<Entity> query = Query.newEntityQueryBuilder()
      * .setKind(ENTITY\_KIND)
      * .setFilter(StructuredQuery.PropertyFilter.eq(
      * Question.QUIZ, quiz))
      * .build();
      * // END TODO
      * // TODO: Execute the query
      * // The datastore.run(query) method returns an iterator
      * // for entities
      * Iterator<Entity> entities = datastore.run(query);
      * // END TODO
      * // TODO: Return the transformed results
      * // Use the buildQuestions(entities) method to convert
      * // from Datastore entities to domain objects
      * return buildQuestions(entities);
      * // END TODO
      * }
      * private List<Question> buildQuestions(Iterator<Entity> entities){
      * List<Question> questions = new ArrayList<>();
      * entities.forEachRemaining(entity-> questions.add(entityToQuestion(entity)));
      * return questions;
      * }
      * private Question entityToQuestion(Entity entity){
      * return new Question.Builder()
      * .withQuiz(entity.getString(Question.QUIZ))
      * .withAuthor(entity.getString(Question.AUTHOR))
      * .withTitle(entity.getString(Question.TITLE))
      * .withAnswerOne(entity.getString(Question.ANSWER\_ONE))
      * .withAnswerTwo(entity.getString(Question.ANSWER\_TWO))
      * .withAnswerThree(entity.getString(Question.ANSWER\_THREE))
      * .withAnswerFour(entity.getString(Question.ANSWER\_FOUR))
      * .withCorrectAnswer(entity.getLong(Question.CORRECT\_ANSWER))
      * .withId(entity.getKey().getId())
      * .build();
      * }
      * }
    - Run the application and test the Cloud Datastore query
    - Save the .../services/gcp/datastore/QuestionService.java file, and then return to the Cloud Shell command.
    - From Cloud Shell, click on the Open Terminal Stop the application by pressing Ctrl+C.
    - Start the application.
    - In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
    - Replace the query string at the end of the application's URL with /api/quizzes/gcp.
    - The URL is in the form: https://8080-####-####.cloudshell.dev/api/quizzes/gcp You should see that JSON data has been returned to the client corresponding to the question you added in the web application!
    - Return to the application home page, and click Take Test and then click GCP.
    - You should see that the quiz question has been formatted inside the client-side web application!
* App Dev - Storing Application Data in Cloud Datastore: Python 2 hours
  + Overview
    - Google Cloud Datastore is a NoSQL document database built for automatic scaling, high performance, and ease of application development. In this lab, you use Datastore to store application data for an online Quiz application. You also configure the application to retrieve from Datastore and display the data in the quiz.
    - The Quiz application skeleton has already been written. You clone the repository that contains the skeleton using Google Cloud Shell, review the code using the Cloud Shell editor, and view it using the Cloud Shell web preview feature. You then modify the code that stores data to use Cloud Datastore.
  + Create a virtual environment
    - virtualenv is used to create user space virtual environments that allow you to install different sets of Python packages for different projects. Using virtualenv also means you don't have to install Python packages globally which can cause system tools or other Python projects to break. In this lab virtualenv is also used to make sure that Python3 is used for all Python commands.
      * virtualenv -p python3 vrenv
      * source vrenv/bin/activate
  + Prepare the Quiz application
    - The repository that contains the Quiz application is located on GitHub.com. In this section, you use Cloud Shell to enter commands that clone repository and run the application.
      * git clone https://github.com/GoogleCloudPlatform/training-data-analyst
      * ln -s ~/training-data-analyst/courses/developingapps/v1.3/python/datastore ~/datastore
      * cd ~/datastore/start
      * export GCLOUD\_PROJECT=$DEVSHELL\_PROJECT\_ID
    - Install the application dependencies:
      * pip install -r requirements.txt
    - Run the application:
      * python run\_server.py
    - In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
      * Graphical user interface, text, application

        Description automatically generated
  + Examine the Quiz application code
  + Review the Flask Web application
    - Navigate to the /datastore/start folder using the file browser panel on the left side of the editor.
      * Note: Paths will be relative to this folder. This application is a standard Python application written using the popular Flask application framework.
    - Select the ...run-server.py file.
      * This file contains the entrypoint for the application, and runs it on port 8080.
    - Select the ...quiz/\_\_init\_\_.py file.
      * This file imports routes for the web application and REST API.
    - Select the ...quiz/webapp/questions.py and ...quiz/webapp/routes.py file.
      * These files contain the routes that map URIs to handlers that display the form and collect form data posted by quiz authors in the web application.
    - Select the ...quiz/webapp/templates folder.
      * This folder contains templates for the web application user interface using Jinja2 templates.
    - View the ...quiz/webapp/templates/add.html file.
      * This file contains the Jinja2 template for the Create Question form.
      * Notice how there is a select list to pick a quiz, textboxes where an author can enter the question and answers, and radio buttons to select the correct answer.
    - Select the ...quiz/api/api.py file.
      * This file contains the handler that sends JSON data to students taking a test.
    - Select the ...quiz/gcp/datastore.py file.
      * This is the file where you write Datastore code to save and load quiz questions to and from Cloud Datastore.
    - This module will be imported into the web application and API.
  + Adding Entities to Cloud Datastore
    - In this section, you write code to save form data in Cloud Datastore.
    - Important: Update or add code between the following comments:
      * // TODO
      * // END TODO
  + Create an App Engine application to provision Cloud Datastore
    - Return to Cloud Shell and stop the application by pressing Ctrl+c.
    - To create an App Engine application in your project, use the following command:
      * gcloud app create --region "us-central"
  + Import and use the Python Datastore module
    - Open the ...quiz/gcp/datastore.py file in the Cloud Shell editor and add the following code to perform the following:
      * Import the os module.
      * Use the os module to get the GCLOUD\_PROJECT environment variable.
      * Import the datastore module from the google.cloud package.
      * Declare a datastore.Client client object named datastore\_client.
    - Updated datastore.py
      * # TODO: Import the os module
      * import os
      * # END TODO
      * # TODO: Get the GCLOUD\_PROJECT environment variable
      * project\_id = os.getenv('GCLOUD\_PROJECT')
      * # END TODO
      * from flask import current\_app
      * # TODO: Import the datastore module from the google.cloud package
      * from google.cloud import datastore
      * # END TODO
      * # TODO: Create a Cloud Datastore client object
      * # The datastore client object requires the Project ID.
      * # Pass through the Project ID you looked up from the
      * # environment variable earlier
      * datastore\_client = datastore.Client(project\_id)
      * # END TODO
  + Write code to create a Cloud Datastore entity
    - Still in ...quiz/gcp/datastore.py,
    - Move to the save\_question() function and remove the existing pass placeholder statement. Add the following code to perform the following:
      * Use the Datastore client object to create a key for a Datastore entity whose kind is 'Question'.
      * Use Datastore to create a Datastore question entity with the key.
      * Iterate over the items in the dictionary of values supplied from the Web application form.
      * In the body of the loop, assign each key and value to the Datastore entity object.
      * Use the Datastore client to save the data.
    - datastore.py - save\_question() function
      * """
      * Create and persist and entity for each question
      * The Datastore key is the equivalent of a primary key in a relational database.
      * There are two main ways of writing a key:
      * 1. Specify the kind, and let Datastore generate a unique numeric id
      * 2. Specify the kind and a unique string id
      * """
      * def save\_question(question):
      * # TODO: Create a key for a Datastore entity
      * # whose kind is Question
      * key = datastore\_client.key('Question')
      * # END TODO
      * # TODO: Create a Datastore entity object using the key
      * q\_entity = datastore.Entity(key=key)
      * # END TODO
      * # TODO: Iterate over the form values supplied to the function
      * for q\_prop, q\_val in question.items():
      * # END TODO
      * # TODO: Assign each key and value to the Datastore entity
      * q\_entity[q\_prop] = q\_val
      * # END TODO
      * # TODO: Save the entity
      * datastore\_client.put(q\_entity)
      * # END TODO
    - Save datastore.py.
  + Run the application and create a Cloud Datastore entity
    - Save the ...quiz/gcp/datastore.py file and then return to the Cloud Shell command prompt.
    - To run the application, execute the following command:
      * python run\_server.py
    - In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
    - Click Create Question.
  + Retrieve Cloud Datastore entities
    - In this section, you write code to retrieve entity data from Cloud Datastore to view your question in the application.
    - Write code to retrieve Cloud Datastore entities
    - In the code editor, in the ...quiz/gcp/datastore.py file, remove the code for the list\_entities(quiz, redact) function and replace it with a query that:
      * Retrieves Question entities for a specific quiz from Cloud Datastore.
      * Uses the Datastore client to fetch the query, and uses the returned data to create a list.
      * Enumerate the list of items, and promote each entity's Key identifier to a top level property.
      * Return the results.
    - Replace this code:
      * """
      * Returns a list of question entities for a given quiz
      * - filter by quiz name, defaulting to gcp
      * - no paging
      * - add in the entity key as the id property
      * - if redact is true, remove the correctAnswer property from each entity
      * """
      * def list\_entities(quiz='gcp', redact=True):
      * return [{'quiz':'gcp', 'title':'Sample question', 'answer1': 'A', 'answer2': 'B', 'answer3': 'C', 'answer4': 'D', 'correctAnswer': 1, 'author': 'Nigel'}]
      * """
    - With this code:
      * """
      * Returns a list of question entities for a given quiz
      * - filter by quiz name, defaulting to gcp
      * - no paging
      * - add in the entity key as the id property
      * - if redact is true, remove the correctAnswer property from each entity
      * """
      * def list\_entities(quiz='gcp', redact=True):
      * query = datastore\_client.query(kind='Question')
      * query.add\_filter('quiz', '=', quiz)
      * results =list(query.fetch())
      * for result in results:
      * result['id'] = result.key.id
      * if redact:
      * for result in results:
      * del result['correctAnswer']
      * return results
      * """
    - Save datastore.py.
  + Run the application and test the Cloud Datastore query
    - Now to test if your question is retrieved from Datastore and loaded into your Quiz application.
    - In Cloud Shell, press Ctrl+c to stop the application, then restart the application:
      * python run\_server.py
    - Preview the quiz: If the browser running the quiz is still open, reload the browser. Otherwise, click Web preview > Preview on port 8080.
    - Click Take Test > GCP.
* Practice Quiz: Best Practices for Using Datastore
  + Which one of the following statements about Cloud Datastore is accurate?
    - Cloud Datastore supports atomic transactions.
  + What best practices can you apply when creating a Cloud Datastore entity with a numeric key? Choose all that are correct (3 correct answers) .
    - When creating keys manually, get a block of IDs using the allocateIds() method.
    - Avoid sequential numbering of keys.
    - Let Cloud Datastore automatically assign the numeric ID for the key.
  + You receive the following error code from a Cloud Datastore request: INTERNAL. What action should you take?
    - Retry only once.
    - Requests that return an INTERNAL error should not be retried more than once.
* Quiz: Final Quiz - Best Practices for Using Cloud Datastore
  + Which of the following statements about Cloud Datastore entities are accurate? (Select all 2 correct answers)
    - Entities of the same kind can have different properties.
    - Entity keys can have manually generated numeric ids.
  + Your expense report application allows users to submit multiple expenses in a single report. You want to add each expense as a separate entity in Cloud Datastore. How can you reduce latency when adding expenses to Cloud Datastore?
    - Use a batch operation to add multiple entities in one request.
  + An employee can have multiple expense exports and each expense report can have multiple expenses. You need to store expense report information in Cloud Datastore. What is the best way to structure the data?
    - Create entities with the following ancestor relationship :
    - Root entity is Employee: Key = John Doe
    - Child entity of Employee is ExpenseReport: Key = 2018\_06
    - Child entity of ExpenseReport is Expense: Key = 11111

## 3.6 Best Practices for Using Storage

* Performing operations on buckets and objects 5 minutes - https://youtu.be/6yc\_jReHT1g
* Demo Explore Cloud Storage 11 minutes - https://youtu.be/pRUclGy8m-k
* Bucket / Object Operations and Truncated Exponential Backoff 5 minutes - https://youtu.be/93AcYpUoVAs
* Demo Enable CORS configuration in Cloud Storage 5 minutes - https://youtu.be/E7bxktebw7c
* Best Practices for Using Cloud Storage (Part 1) 9 minutes - https://youtu.be/HT7W6Bn-pAI
* Best Practices for Using Cloud Storage (Part 2) 7 minutes - https://youtu.be/2z0BqV5WIYM
* App Dev - Storing Image and Video Files in Cloud Storage: Node.js 2 hours
  + Overview
    - In this lab, you enhance the online Quiz application to work with images and videos. You store files as objects in a Cloud Storage bucket.
    - The Quiz application user interface now includes a file input button in the add question form, and the handler can receive image data in the server-side application.
    - You add the code that stores the uploaded file into Cloud Storage, make the object publicly available, and then store the object URL and other application data in Google Cloud Datastore.
  + Objectives
    - Create a Cloud Storage bucket.
    - Review file upload UI and code changes.
    - Write code to store uploaded file data into Cloud Storage.
  + Reviewing the Case Study Application
    - git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - ln -s ~/training-data-analyst/courses/developingapps/v1.3/nodejs/cloudstorage cloudstorage
    - cd ~/cloudstorage/start
    - To configure the application, execute the following command:
      * . prepare\_environment.sh
      * This script file
        + Creates an App Engine application.
        + Exports an environment variable, GCLOUD\_PROJECT.
        + Runs npm install.
        + Creates entities in Cloud Datastore.
        + Prints out the Google Cloud Platform Project ID.
    - To run the application, execute the following command:
      * npm start
    - To view the application, click Web preview > Preview on port 8080.
  + Examining the Case Study Application Code
    - Navigate to the /cloudstorage/start folder using the file browser panel on the left side of the editor.
    - Review the Express Web application
    - Select the add.pug file in the .../server/web-app/views/questions folder.
      * This file contains the pug template for the Create Question form.
      * Notice how the form has been modified to use multipart/form-data as the enc-type, and there are two new form controls:
        + A file upload control called image
        + A hidden field called imageUrl
    - Select the questions.js file in the .../server/web-app folder.
    - This file has been modified so that the POST handler chains together three distinct middleware calls:
      * Configures Multer to accept a single image file from a form field called image. Multer is a popular Express middleware package for handling file uploads. The Multer handler consumes the file multipart upload from the browser and holds the file data in memory.
      * Consumes the file processed by Multer. This handler is where you will write the Cloud Storage code to upload a file, make it publicly available, and store the public URL for the object.
      * Consumes the public URL to store the data into Cloud Datastore. You already wrote the code to store an entity in Cloud Datastore in the previous lab!
    - Select the .../server/gcp/cloudstorage.js file.
      * This is the file where you will write code to save image file data into Cloud Storage.
  + Creating a Cloud Storage Bucket
    - To create a Cloud Storage bucket named <Project ID>-media, execute the following command:
      * gsutil mb gs://$DEVSHELL\_PROJECT\_ID-media
    - To export the Cloud Storage bucket name as an environment variable named GCLOUD\_BUCKET
      * export GCLOUD\_BUCKET=$DEVSHELL\_PROJECT\_ID-media
  + Import and use the NodeJS Cloud Storage module
    - In the editor, move to the top of the ...server/gcp/cloudstorage.js file.
    - Load the '@google-cloud/storage module, and assign it to a constant named Storage.
      * // TODO: Load the module for Cloud Storage
      * const {Storage} = require('@google-cloud/storage');
      * // END TODO
    - Use the Storage(...) factory to construct a Cloud Storage client named storage.
      * // TODO: Create the storage client
      * // The Storage(...) factory function accepts an options
      * // object which is used to specify which project's Cloud
      * // Storage buckets should be used via the projectId
      * // property.
      * // The projectId is retrieved from the config module.
      * // This module retrieves the project ID from the
      * // GCLOUD\_PROJECT environment variable.
      * const storage = new Storage({
      * projectId: config.get('GCLOUD\_PROJECT')
      * });
      * // END TODO
    - Declare a string constant named GCLOUD\_BUCKET, and initialize it with the name of the bucket you previously exported as an environment variable.
      * // TODO: Get the GCLOUD\_BUCKET environment variable
      * // Recall that earlier you exported the bucket name into an
      * // environment variable.
      * // The config module provides access to this environment
      * // variable so you can use it in code
      * const GCLOUD\_BUCKET = config.get('GCLOUD\_BUCKET');
      * // END TODO
    - Declare a constant named bucket to reference the Cloud Storage bucket.
      * // TODO: Get a reference to the Cloud Storage bucket
      * const bucket = storage.bucket(GCLOUD\_BUCKET);
      * // END TODO
  + Write code to send a file to Cloud Storage
    - In the sendUploadToGCS(req, res, next) handler, use the Cloud Storage client to upload a file to your Cloud Storage bucket and make it publicly available.
    - Get a reference to a Cloud Storage file object in the bucket.
      * // TODO: Get a reference to the new object
      * const file = bucket.file(oname);
      * // END TODO
    - Create a writeable stream to send data to the Cloud Storage object. Pass through the MIME type for the object.
      * // TODO: Create a stream to write the file into
      * // Cloud Storage
      * // The uploaded file's MIME type can be retrieved using
      * // req.file.mimetype.
      * // Cloud Storage metadata can be used for many purposes,
      * // including establishing the type of an object.
      * const stream = file.createWriteStream({
      * metadata: {
      * contentType: req.file.mimetype
      * }
      * });
      * // END TODO
    - Attach an event handler to the stream to handle errors. In the error event handler, invoke the next Express middleware.
      * // TODO: Attach two event handlers (1) error
      * // Event handler if there's an error when uploading
      * stream.on('error', err => {
      * // TODO: If there's an error move to the next handler
      * next(err);
      * // END TODO
      * });
      * // END TODO
    - Attach a second event handler to the stream. This handler will be invoked when the data has finished uploading. In the finish event handler, make the file public, and set a new property on the Datastore entity that references the new public URL, and invoke the next middleware handler.
      * // TODO: Attach two event handlers (2) finish
      * // The upload completed successfully
      * stream.on('finish', () => {
      * // TODO: Make the object publicly accessible
      * file.makePublic().then(() => {
      * // TODO: Set a new property on the file for the
      * // public URL for the object
      * // Cloud Storage public URLs are in the form:
      * // https://storage.googleapis.com/[BUCKET]/[OBJECT]
      * // Use an ECMAScript template literal (`https://...`)to
      * // populate the URL with appropriate values for the bucket
      * // ${GCLOUD\_BUCKET} and object name ${oname}
      * req.file.cloudStoragePublicUrl = `https://storage.googleapis.com/${GCLOUD\_BUCKET}/${oname}`;
      * // END TODO
      * // TODO: Invoke the next middleware handler
      * next();
      * // END TODO
      * });
      * // END TODO
      * });
      * // END TODO
    - At the end of the sendUploadToGCS(req, res, next) handler, to upload the file's data into Cloud Storage, call the end() method to write the file to the stream.
      * // TODO: End the stream to upload the file's data
      * stream.end(req.file.buffer);
      * // END TODO
    - cloudstorage.js
    - After making the above changes, your file should look like this:
  + 'use strict';
  + const config = require('../config');
  + // TODO: Load the module for Cloud Storage
  + const {Storage} = require('@google-cloud/storage');
  + // END TODO
  + // TODO: Create the storage client
  + // The Storage(...) factory function accepts an options
  + // object which is used to specify which project's Cloud
  + // Storage buckets should be used via the projectId
  + // property.
  + // The projectId is retrieved from the config module.
  + // This module retrieves the project ID from the
  + // GCLOUD\_PROJECT environment variable.
  + const storage = new Storage({
  + projectId: config.get('GCLOUD\_PROJECT')
  + });
  + // END TODO
  + // TODO: Get the GCLOUD\_BUCKET environment variable
  + // Recall that earlier you exported the bucket name into an
  + // environment variable.
  + // The config module provides access to this environment
  + // variable so you can use it in code
  + const GCLOUD\_BUCKET = config.get('GCLOUD\_BUCKET');
  + // END TODO
  + // TODO: Get a reference to the Cloud Storage bucket
  + const bucket = storage.bucket(GCLOUD\_BUCKET);
  + // END TODO
  + // Express middleware that will automatically pass uploads to Cloud Storage.
  + // req.file is processed and will have a new property:
  + // \* ``cloudStoragePublicUrl`` the public url to the object.
  + // [START sendUploadToGCS]
  + function sendUploadToGCS(req, res, next) {
  + // The existing code in the handler checks to see if there
  + // is a file property on the HTTP request - if a file has
  + // been uploaded, then Multer will have created this
  + // property in the preceding middleware call.
  + if (!req.file) {
  + return next();
  + }
  + // In addition, a unique object name, oname, has been
  + // created based on the file's original name. It has a
  + // prefix generated using the current date and time.
  + // This should ensure that a new file upload won't
  + // overwrite an existing object in the bucket
  + const oname = Date.now() + req.file.originalname;
  + // TODO: Get a reference to the new object
  + const file = bucket.file(oname);
  + // END TODO
  + // TODO: Create a stream to write the file into
  + // Cloud Storage
  + // The uploaded file's MIME type can be retrieved using
  + // req.file.mimetype.
  + // Cloud Storage metadata can be used for many purposes,
  + // including establishing the type of an object.
  + const stream = file.createWriteStream({
  + metadata: {
  + contentType: req.file.mimetype
  + }
  + });
  + // END TODO
  + // TODO: Attach two event handlers (1) error
  + // Event handler if there's an error when uploading
  + stream.on('error', err => {
  + // TODO: If there's an error move to the next handler
  + next(err);
  + // END TODO
  + });
  + // END TODO
  + // TODO: Attach two event handlers (2) finish
  + // The upload completed successfully
  + stream.on('finish', () => {
  + // TODO: Make the object publicly accessible
  + file.makePublic().then(() => {
  + // TODO: Set a new property on the file for the
  + // public URL for the object
  + // Cloud Storage public URLs are in the form:
  + // https://storage.googleapis.com/[BUCKET]/[OBJECT]
  + // Use an ECMAScript template literal (`https://...`)to
  + // populate the URL with appropriate values for the bucket
  + // ${GCLOUD\_BUCKET} and object name ${oname}
  + req.file.cloudStoragePublicUrl = `https://storage.googleapis.com/${GCLOUD\_BUCKET}/${oname}`;
  + // END TODO
  + // TODO: Invoke the next middleware handler
  + next();
  + // END TODO
  + });
  + // END TODO
  + });
  + // END TODO
  + // TODO: End the stream to upload the file's data
  + stream.end(req.file.buffer);
  + // END TODO
  + }
  + // [END sendUploadToGCS]
  + // Multer handles parsing multipart/form-data requests.
  + // This instance is configured to store images in memory.
  + // This makes it straightforward to upload to Cloud Storage.
  + // [START multer]
  + const Multer = require('multer');
  + const multer = Multer({
  + storage: Multer.MemoryStorage,
  + limits: {
  + fileSize: 40 \* 1024 \* 1024 // no larger than 40mb
  + }
  + });
  + // [END multer]
  + module.exports = {
  + sendUploadToGCS,
  + multer
  + };
    - Start the application by typing npm start
    - Click Create Question.

|  |  |
| --- | --- |
| **Form Field** | **Value** |
| Author | Your Name |
| Quiz | Google Cloud Platform |
| Title | Which product does this logo relate to? |
| Image | Upload the Google-Cloud-Storage-Logo.svg file you previously downloaded |
| Answer 1 | App Engine |
| Answer 2 | **Cloud Storage** (select the Answer 2 radio button!) |
| Answer 3 | Compute Engine |
| Answer 4 | Container Engine |

* + - Return to the Cloud Console and click on Navigation menu > Cloud Storage.
    - On the Cloud Storage > Browser page, click the correct bucket (named <Project ID>-media).
    - You should see your new object named #UniqueNumber#Google-Cloud-Storage-Logo.svg.
  + Run the client application and test the Cloud Storage public URL
    - Add /api/quizzes/gcp to the end of the application's URL.
    - You should see that JSON data has been returned to the client corresponding to the Question you added in the web application.
  + Choose the correct statements about Cloud Storage.
    - Cloud Storage bucket names must be globally unique.
    - Cloud Storage object URIs must be globally unique.
    - Cloud Storage bucket names must be unique for each billing account.
    - Cloud Storage object paths within a bucket must be unique.
  + Which object and method are causing data to be sent to Cloud Storage?
    - Storage and bucket(...)
    - bucket and file(...)
    - file and end(...)
* App Dev - Storing Image and Video Files in Cloud Storage: Java 2 hours
  + git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
  + ln -s ~/training-data-analyst/courses/developingapps/v1.3/java/cloudstorage ~/cloudstorage
  + cd ~/cloudstorage/start
  + . prepare\_environment.sh
  + mvn spring-boot:run
  + Navigate to /cloudstorage/start folder using the file browser panel on the left side of the editor.
  + Select new\_question.html file in the .../src/main/resources/templates folder.
    - This file contains the Thyme template for the Create Question form. Notice how the form uses multipart/form-data as the enctype, and the two form controls for images and videos:
      * A file upload control called image
      * A hidden field called imageUrl
  + Select the .../src/main/java/com/google/training/appdev folder.
    - Java file paths are relative to this folder.
  + Select .../web/QuestionsController.java.
    - In this file the POST handler invokes a method on an image service.
  + Select the .../services/gcp/cloudstorage/ImageService.java file.
    - This is the file where you write code to save image file data into Cloud Storage.
  + Creating a Cloud Storage bucket
    - gsutil mb gs://$DEVSHELL\_PROJECT\_ID-media
    - export GCLOUD\_BUCKET=$DEVSHELL\_PROJECT\_ID-media
  + Adding Objects to Cloud Storage
  + .../services/gcp/cloudstorage/ImageService.java file.
  + Write a star import for the com.google.cloud.storage.\* package.
    - // TODO: Write a start import for Cloud Storage
    - import com.google.cloud.storage.\*;
    - // END TODO
  + Construct a Cloud Storage client named storage using the StorageOptions class.
    - // TODO: Create the storage client
    - // The StorageOptions class has a getDefaultInstance()
    - // static method.
    - // Use the getService() method to get the storage client
    - private static Storage storage = StorageOptions
    - .getDefaultInstance()
    - .getService();
    - // END TODO
  + Declare a String named bucketName, and annotate it with a Spring value annotation to retrieve the value from a property named google.storage.bucket.
    - // TODO: Get the name of the Cloud Storage bucket
    - // Use a Spring @Value annotation to get the value
    - // Get the value using ${google.storage.bucket}
    - // This references the GCLOUD\_BUCKET environment variable
    - @Value("${google.storage.bucket}")
    - private String bucketName;
    - // END TODO
  + ImageService.java
    - package com.google.training.appdev.services.gcp.cloudstorage;
    - // TODO: Write a start import for Cloud Storage
    - import com.google.cloud.storage.\*;
    - // END TODO
    - import org.springframework.beans.factory.annotation.Value;
    - import org.springframework.stereotype.Service;
    - import org.springframework.web.multipart.MultipartFile;
    - import java.io.IOException;
    - import java.util.ArrayList;
    - import java.util.Arrays;
    - @Service
    - public class ImageService {
    - // TODO: Create the storage client
    - // The StorageOptions class has a getDefaultInstance()
    - // static method.
    - // Use the getService() method to get the storage client
    - private static Storage storage = StorageOptions
    - .getDefaultInstance()
    - .getService();
    - // END TODO
    - // TODO: Get the name of the Cloud Storage bucket
    - // Use a Spring @Value annotation to get the value
    - // Get the value using ${google.storage.bucket}
    - // This references the GCLOUD\_BUCKET environment variable
    - @Value("${google.storage.bucket}")
    - private String bucketName;
    - // END TODO
  + Write code to send a file to Cloud Storage
    - Still updating ImageService.java, move to the saveImage(MultipartFile file) handler and then use the Cloud Storage client to upload a file to your Cloud Storage bucket and make it publicly available.
    - Declare a BlobInfo object and initialize it using the storage client object. Customize the BlobInfo object using its Builder. Use the option to set the Content Type and to set the ACL to allow unauthenticated Read access.
      * // TODO: Create a new Cloud Storage object
      * // Use the BlobInfo class to represent this object
      * // Use the BlobInfo.Builder to customize the Blob
      * // Set the content type from the file
      * // Set the object ACL to Public Read
      * BlobInfo blobInfo = storage.create(
      * BlobInfo.newBuilder(bucketName, fileName)
      * .setContentType(file.getContentType())
      * .setAcl(new ArrayList<>(
      * Arrays.asList(Acl.of(Acl.User.ofAllUsers(),
      * Acl.Role.READER))))
      * .build(),
      * file.getInputStream());
      * // END TODO
    - Return the public URL for the new Cloud Storage object as a string.
      * // TODO: Cloud Storage public URLs are in the form:
      * // https://storage.googleapis.com/[BUCKET]/[OBJECT]
      * // Use String concatentation to create return the URL
      * return "https://storage-download.googleapis.com/" + bucketName+ "/" +fileName;
      * // END TODO
  + ImageService.java
    - public String saveImage(MultipartFile file)
    - throws IOException {
    - // The existing code in the method creates a unique name
    - // based on the file's original name. It has a
    - // prefix generated using the current date and time.
    - // This should ensure that a new file upload won't
    - // overwrite an existing object in the bucket
    - String fileName = System.nanoTime() +
    - file.getOriginalFilename();
    - // TODO: Create a new Cloud Storage object
    - // Use the BlobInfo class to represent this object
    - // Use the BlobInfo.Builder to customize the Blob
    - // Set the content type from the file
    - // Set the object ACL to Public Read
    - BlobInfo blobInfo = storage.create(
    - BlobInfo.newBuilder(bucketName, fileName)
    - .setContentType(file.getContentType())
    - .setAcl(new ArrayList<>(
    - Arrays.asList(Acl.of(Acl.User.ofAllUsers(),
    - Acl.Role.READER))))
    - .build(),
    - file.getInputStream());
    - // END TODO
    - // TODO: Cloud Storage public URLs are in the form:
    - // https://storage.googleapis.com/[BUCKET]/[OBJECT]
    - // Use String concatenation to return the URL
    - return "https://storage-download.googleapis.com/" +
    - bucketName + "/" + fileName;
    - // END TODO
    - }
  + Start the application by typing
    - mvn spring-boot:run
  + Click the Create Question link.
  + On the Cloud Storage > Browser page, click the correct bucket (named <Project ID>-media).
    - You should see your new object named #UniqueNumber#Google\_Cloud\_Storage\_logo.png.
  + Create a Datastore entity
    - Add /api/quizzes/gcp to the end of the Quiz application's URL.
* App Dev - Storing Image and Video Files in Cloud Storage: Python 1 hour
  + virtualenv -p python3 vrenv
  + source vrenv/bin/activate
  + git clone https://github.com/GoogleCloudPlatform/training-data-analyst
  + ln -s ~/training-data-analyst/courses/developingapps/v1.3/python/cloudstorage ~/cloudstorage
  + cd ~/cloudstorage/start
  + . prepare\_environment.sh
  + python run\_server.py
  + Navigate to the /cloudstorage/start folder using the file browser panel on the left side of the editor.
  + Select the add.html file in the ...quiz/webapp/templates/ folder.
    - This file contains the template for the Create Question form.
    - Notice how the form has been modified to use multipart/form-data as the enc-type, and there are two new form controls:
      * A file upload control called image
      * A hidden field called imageUrl
  + Select the routes.py file in the ...quiz/webapp folder.
    - This file contains the route for the POST handler that receives the form data. It has been modified to get the image file from the form.
  + Select the questions.py file in the ...quiz/webapp folder.
    - This file contains the handler that processes the form data extracted in the routes.py file. You will modify this file to use a new module that is a client for Cloud Storage.
  + Select the ...quiz/gcp/storage.py file.
    - This is the file where you will write code to save image file data into Cloud Storage.
  + Create a Cloud Storage bucket
    - gsutil mb gs://$DEVSHELL\_PROJECT\_ID-media
  + export GCLOUD\_BUCKET=$DEVSHELL\_PROJECT\_ID-media
  + Import and use the Python Cloud Storage module
    - In code editor, move to the top of the ...quiz/gcp/storage.py file.
    - Get the bucket name from the GCLOUD\_BUCKET environment variable.
    - Import the storage module from the google.client package.
    - Create a Cloud Storage client.
    - Get a reference to the Cloud Storage bucket.
  + quiz/gcp/storage.py
    - # TODO: Get the Bucket name from the GCLOUD\_BUCKET environment variable
    - bucket\_name = os.getenv('GCLOUD\_BUCKET')
    - # END TODO
    - # TODO: Import the storage module
    - from google.cloud import storage
    - # END TODO
    - # TODO: Create a client for Cloud Storage
    - storage\_client = storage.Client()
    - # END TODO
    - # TODO: Use the client to get the Cloud Storage bucket
    - bucket = storage\_client.get\_bucket(bucket\_name)
    - # END TODO
  + Write code to send a file to Cloud Storage
    - Sill in storage.py, in the the upload\_file(...) function, remove the existing pass statement, then use the Cloud Storage client to upload a file to your Cloud Storage bucket and make it publicly available.
    - Get a reference to a Cloud Storage blob object in the bucket.
    - Use the blob object to upload the image.
    - Make the file public.
    - Return the blob's public URL.
  + quiz/gcp/storage.py upload\_file(...) function
    - """
    - Uploads a file to a given Cloud Storage bucket and returns the public url
    - to the new object.
    - """
    - def upload\_file(image\_file, public):
    - pass
    - # TODO: Use the bucket to get a blob object
    - blob = bucket.blob(image\_file.filename)
    - # END TODO
    - # TODO: Use the blob to upload the file
    - blob.upload\_from\_string(
    - image\_file.read(),
    - content\_type=image\_file.content\_type)
    - # END TODO
    - # TODO: Make the object public
    - if public:
    - blob.make\_public()
    - # END TODO
    - # TODO: Modify to return the blob's Public URL
    - return blob.public\_url
    - # END TODO
  + Write code to use the Cloud Storage functionality
    - In the editor, move to the top of the ...quiz/webapp/questions.py file.
    - Modify the import statement to use your storage client as well as the datastore client.
    - Move to the upload\_file(...) function. Use your storage client to upload a file, and assign the returned public URL to a variable.
    - Modify the return statement to return the public URL.
    - Move to the save\_question(...) function. Write an if test to see if the image\_file is present.
    - If it is, then call the upload\_file(...) function, and assign the public URL to a entity property named imageUrl.
    - If not, then assign an empty string to the entity imageUrl property.
  + quiz/webapp/questions.py
    - # TODO: Import the storage module
    - from quiz.gcp import storage, datastore
    - # END TODO
    - """
    - uploads file into google cloud storage
    - - upload file
    - - return public\_url
    - """
    - def upload\_file(image\_file, public):
    - if not image\_file:
    - return None
    - # TODO: Use the storage client to Upload the file
    - # The second argument is a boolean
    - public\_url = storage.upload\_file(
    - image\_file,
    - public
    - )
    - # END TODO
    - # TODO: Return the public URL
    - # for the object
    - return public\_url
    - # END TODO
    - """
    - uploads file into google cloud storage
    - - call method to upload file (public=true)
    - - call datastore helper method to save question
    - """
    - def save\_question(data, image\_file):
    - # TODO: If there is an image file, then upload it
    - # And assign the result to a new Datastore
    - # property imageUrl
    - # If there isn't, assign an empty string
    - if image\_file:
    - data['imageUrl'] = str(
    - upload\_file(image\_file, True))
    - else:
    - data['imageUrl'] = u''
    - # END TODO
    - data['correctAnswer'] = int(data['correctAnswer'])
    - datastore.save\_question(data)
    - return
  + Return to Cloud Shell to run the application:
    - python run\_server.py
  + Create Question
  + On the Cloud Storage > Browser page, click the correct bucket (named <Project ID>-media).
  + Add /api/quizzes/gcp to the end of the application's URL.
* Practice Quiz: Best Practices for Cloud Storage
  + Review the following gsutil command. What is the result when the command is executed? gsutil mb -c regional -l us-central1 gs://parent-bucket
    - The command creates a bucket called "parent-bucket" in the "us-central1" location with "regional" storage class.
  + Which of the following bucket names are valid? (Select all 2 correct answers)
    - vanilla-bucket
    - bucket\_for\_my\_web\_app
    - A bucket name can contain lowercase alphanumeric characters, hyphens, and underscores. It can contain dots (.) if it forms a valid domain name with a top-level domain (such as .com).Bucket names must start and end with an alphanumeric character
  + What are the advantages of hosting static websites on Google Cloud Storage? (Select all 2 correct answers)
    - You don't need to set up and run a Compute Engine instance.
    - You get automatic scaling with no additional effort.
* Quiz: Final Quiz - Best Practices for Cloud Storage
  + You are tasked with designing a disaster recovery system in your organization. You need to make sure that all applications recover and become available as quickly as possible. What storage class is ideal for storing backups of your data if the data is to be stored for two years and is unlikely to be accessed?
    - Archive
  + Before transferring data to you, a third-party breaks up each large data file into 15 small chunks because of network bandwidth issues. You want to use Google BigQuery to directly query Cloud Storage data. What is the best way to combine the chunks into a single file?
    - Use the "gsutil compose" command to build a composite object from smaller chunks.
  + Identify two key aspects of this build system's architecture.
    - Diagram

      Description automatically generated
    - As the build artifacts get older, they move through various storage classes to limit their retention cost.
    - The build system stores build artifacts in Cloud Storage.
  + Review the following gsutil command. What is the result when the command is executed?
    - The command creates a bucket called "parent-bucket" in the "us-central1" location with "regional" storage class.

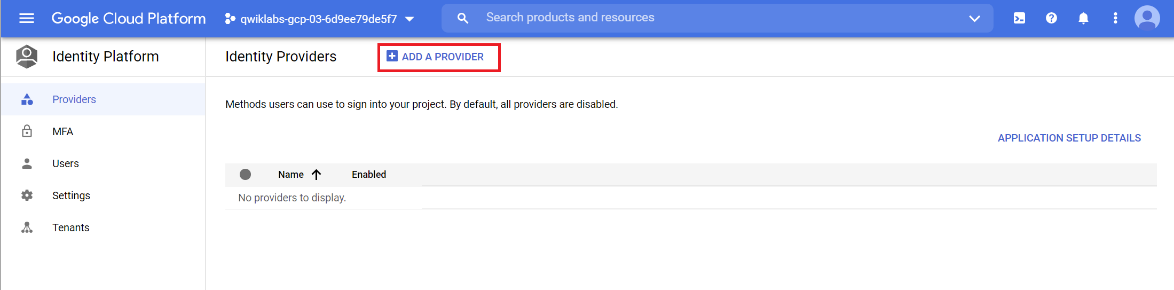
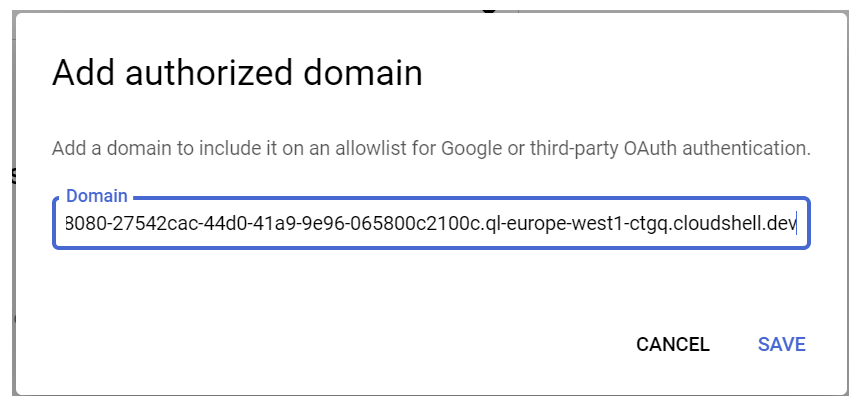
## 4. Securing and Integrating Components of your Application

* application developers learn how to design and develop cloud-native applications that seamlessly integrate managed services from Google Cloud. Participants learn how to apply best practices for application development and use the appropriate Google Cloud storage services for object storage, relational data, caching, and analytics. Learners can choose to complete labs in their favorite language: Node.js, Java, or Python.

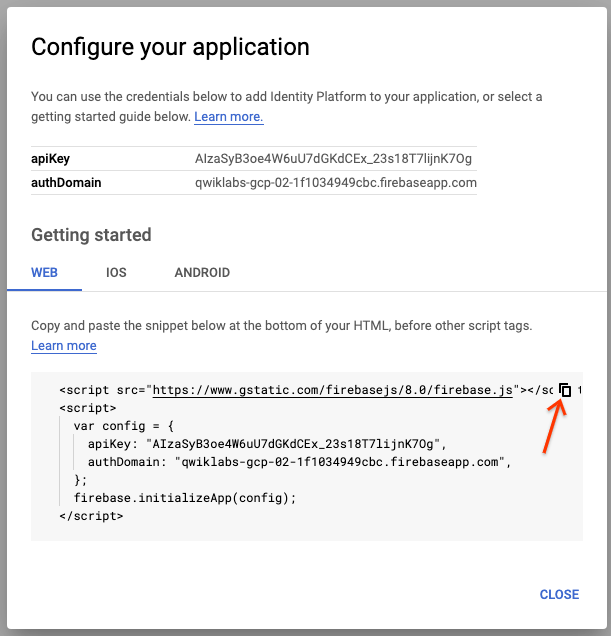
## 4.1 Handling Authentication and Authorization

* IAM Concepts 7 minutes - https://youtu.be/92zztpygZZI
* Cloud IAM Best Practices 4 minutes - https://youtu.be/lzvyFA6YhVo
* OAuth2.0, IAP, & Firebase Authentication 3 minutes - https://youtu.be/JiA1m-aimNI

### App Dev - Adding User Authentication to your Application: Node.js 2 hours

* + Overview
    - This lab shows how to add authentication to your application using Identity Platform. This authorization identifies who you are, and determines what you can do. For more information, see Authentication Overview. https://cloud.google.com/docs/authentication/
    - Identity Platform provides a drop-in, customizable authentication service for user sign-up and sign-in. Development and admin activities are made easier with a range of app SDKs (Android, iOS, and web) as well as admin SDKs (Node.js, Java, Python, and more). For more information about Identity Platform, see Identity Platform.
    - The application used in this lab is an online Quiz application. You add Identity Platform authentication, and then configure authentication to use a simple email address and password credential. Finally you ensure that users must register and log in before taking a quiz.
  + Objectives
    - Add Identity Platform configuration to a client-side web application
    - Write JavaScript code to integrate Identity Platform Authentication into a client-side web application
  + Task 1. Prepare the case study application
    - git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - ln -s ~/training-data-analyst/courses/developingapps/v1.3/nodejs/firebase ~/firebase
    - cd ~/firebase/start
    - . prepare\_environment.sh
      * This script file
      * Creates an App Engine application.
      * Creates a Cloud Storage bucket named gs://<Project-ID>-media.
      * Exports two environment variables: GCLOUD\_PROJECT and GCLOUD\_BUCKET.
      * Runs npm install.
      * Creates entities in Cloud Datastore.
      * Prints out the Google Cloud Platform Project ID.
    - npm start
    - click Web preview (Web Preview) > Preview on port 8080
  + Task 2. Examine the case study application code
    - Review the client application
    - Select the index.html file in the firebase/start/server/public/client/ folder.
      * This file is the single page in the AngularJS SPA. It contains <script></script> tags for the application libraries and code and markup where the SPA will render dynamic output.
    - Select the qiq-login-template.html file in the firebase/start/server/public/client/app/auth/ folder.
      * This file contains the AngularJS template for the Login component.
      * Notice how it contains a couple of textboxes and a button. The button has an event handler that runs code when it is clicked.
    - Select the qiq-login.js file.
      * This file contains an AngularJS component. It allows the user to log in to the application or to navigate to a registration page.
  + Task 3. Configure Identity Platform Authentication
    - In this task you confirm that billing is enabled for your cloud project. Next, you configure Identity Platform to sign in a user with an email and password. You then create a user which you can use to login to the Quiz Application.
    - Confirm that billing is enabled
      * Navigation menu > Billing.
      * Verify that a Billing account is linked to the project. You should see a message stating that the Billing account "Qwiklabs Production gcpd xx" is linked to this project.
    - Configure Identity Platform for email and password
      * Navigation > Identity Platform > Enable Identity Platform.
      * The Identity Platform page appears in the Cloud Console.
        + 
      * Click Add a Provider. In the Sign-in method window, for Select a provider, select Email / Password. Click Enabled.
      * In the Authorized Domains pane, click Add Domain.
      * Return to the running Quiz Application and copy the domain (which has the format 8080-27542cac-44d0-41a9-9e96-065800c2100c.ql-europe-west1-ctgq.cloudshell.dev).
        + 
      * Click Save. In the new identity provider window click Save.
    - Add a user
      * In the Identity Platform pane, click Users. Click Add User.
      * In the Add user dialog box, specify the following:

|  |  |
| --- | --- |
| **Email** | user1@example.com |
| **Password** | abc123! |

* + - * Click Add.
  + Task 4. Integrate a client-side web application with Identity Platform
    - In this task you apply the your Identity Platform configuration to your client-side web application.
    - In the navigation pane, click Providers.
    - Click Application Setup Details.
    - In the Configure your application dialog box, copy the Identity Platform markup.
      * 
    - Click Close.
    - Return to the Cloud Shell code editor, then open firebase/start/server/public/client/index.html.
    - Paste the copied Identity Manager configuration script markup into the blank lines just before the series of <script></script> lines at the bottom of the file.
    - Save the index.html file.
  + Task 5. Run the application
    - In this task you verify that you can login to the Quiz Application using the credentials you created in Identity Platform in a previous step. You then register a new user in the Quiz Application and verify that these credentials are added to Identity Platform.
    - Return to the Quiz application and refresh your browser.
    - In the navigation bar, click Take Test.
      * Graphical user interface, text, application, email

        Description automatically generated
    - In the navigation bar, click GCP, People, or Places.
    - Enter invalid credentials: Login will fail because the user is not registered.
    - Enter the valid credentials : [user1@example.com](mailto:user1@example.com), the first question will be presented.
    - click Logout. Click the Register link. Enter new user2 and pwd. Click Register.
    - In the Google Cloud Console, in the Identity Platform navigation pane, click Users.
    - You should see user2@example.com has been added as a user.

### App Dev - Adding User Authentication to your Application: Java 2 hours

* + Write Java code to integrate Identity Platform Authentication into a client-side web application
  + Runs mvn clean install.
  + Run the application:
    - mvn spring-boot:run
  + Task 2. Examine the case study application code
    - Navigate to the /firebase/start folder using the file browser panel on the left side of the editor.
    - In the firebase folder, notice the end folder. The end folder contains the same files as the start folder, but each file in the end folder contains the complete code required to perform this lab.
    - Select the index.html in the .../src/main/resources/static/client folder. Do not confuse with index.html in the static folder.
      * This file is the single page in the AngularJS SPA. It contains <script> </script> tags for the application libraries and code and markup where the SPA will render dynamic output.
    - Select qiq-login-template.html in the .../src/main/resources/static/client/app/auth/ folder.
      * This file contains the AngularJS template for the Login component.
      * Notice how it contains a couple of textboxes and a button. The button has an event handler that runs code when it is clicked.
    - Select qiq-login.js.
      * This file contains an AngularJS component. It allows the user to log in to the application or to navigate to a registration page.
  + Task 4. Integrate a client-side web application with Identity Platform
    - open the index.html file in .../src/main/resources/static/client/.
    - Re run
      * mvn spring-boot:run

### App Dev - Adding User Authentication to your Application: Python 2 hours

* + Write Python code to integrate Identity Platform Authentication into a client-side web application
  + Runs > pip install -r requirements.txt.
  + Creates entities in Cloud Datastore.
  + Run the application:
    - python run\_server.py
  + Navigate to the /firebase/start folder using the file browser panel on the left side of the code editor.
  + Continue navigating. Open ...quiz/webapp/static/client/. Click on the index.html file.
    - This file is the single page in the AngularJS Single Page Application (SPA). It contains <script></script> tags for the application libraries and code, and markup where the SPA will render dynamic output.
  + Select the qiq-login-template.html file in the ...quiz/webapp/static/client/app/auth/ folder.
    - This file contains the AngularJS template for the Login component. Notice how it contains a couple of textboxes and a button. The button has an event handler that runs code when it is clicked.
  + Still in this folder, select the qiq-login.js file.
    - This file contains an AngularJS component. It allows the user to log in to the application or to navigate to a registration page.
  + In the Cloud Shell code editor, open the index.html file in ...webapp/static/client/.
  + Paste the configuration markup just before the other <script></script> tags at the bottom of the page.

### Practice Quiz

* + Review the following permission: bigquery.jobs.create Which of the following statements about the permission is accurate? Select two.
    - IAM members with this permission can create new jobs in BigQuery.
      * Cloud IAM permissions have the following format: <service>.<resource>.<verb>. This permission is related to the Google BigQuery service, its "jobs" resource, and its "create" action. IAM members with this permission can create new jobs in BigQuery.
    - This permission is related to the BigQuery service, "jobs" resource, and "create" action.
      * Cloud IAM permissions have the following format: <service>.<resource>.<verb>. This permission is related to the Google BigQuery service, its "jobs" resource, and its "create" action. IAM members with this permission can create new jobs in BigQuery.
  + What mechanism should you use to authenticate your applications when invoking Google APIs?
    - Service account
    - A service account is a special Google account that belongs to your application or a VM instance, instead of to an individual end user. Use a service account to call the Google API of a service.

### Handling Authentication and Authorization Quiz

* + What is the best way to apply the principle of least privilege when granting access to Google Cloud resources?
    - Grant restricted permissionsat thetop of the resource hierarchy. Then, for specific users, grant additional granular permissions as you go down the hierarchy.
  + Your enterprise has an online expense reporting application. Employees must be able to access the application without having to log into the corporate VPN. How can you enable this type of access?
    - You can use Identity-Aware Proxy to provide application-level access.
  + Your photo-sharing application requires user login. You don't want to build a custom user authentication system that stores usernames and passwords. What is the best way to authenticate your users?
    - You can leverage federated identity management by using Firebase authentication.

## 4.2 Using Pub/Sub to Integrate Components of Your Application

* Why Pub/Sub? 3 minutes - https://youtu.be/DJBZh5XUaoc
* Pub/Sub Concepts 4 minutes - https://youtu.be/GcHUCGay4K4
* Pub/Sub Use cases 7 minutes - https://youtu.be/XFBFSJUECWE

### App Dev - Developing a Backend Service: Node.js 2 hours

* + Overview
    - In this lab, you enhance the online Quiz application by developing a backend service to process user feedback and save scores.
    - To do this you integrate several Google Cloud Platform (GCP) products and services: Cloud Pub/Sub, Cloud Spanner, and the Natural Language ML API.
  + Objectives
    - Create a Cloud Pub/Sub topic.
    - Publish messages to the topic.
    - Subscribe to the topic to receive messages in a separate worker application.
    - Use the Cloud Natural Language Machine Learning API.
    - Create a Cloud Spanner database instance.
    - Develop a Cloud Spanner database schema.
    - Insert data into a Cloud Spanner database.
  + Preparing the Case Study Application
    - git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - ln -s ~/training-data-analyst/courses/developingapps/v1.3/nodejs/pubsub-languageapi-spanner ~/pubsub-languageapi-spanner
    - cd ~/pubsub-languageapi-spanner/start
    - . prepare\_web\_environment.sh
    - This script file:
      * Creates an App Engine application.
      * Exports environment variables: GCLOUD\_PROJECT and GCLOUD\_BUCKET.
      * Runs npm install.
      * Creates entities in Cloud Datastore.
      * Prints out the GCP Project ID.
    - To run the web application, enter the following command:
      * npm start
    - To add a new Cloud Shell session, click Open a new tab (+) on the right of the Cloud Shell tab.
      * cd ~/pubsub-languageapi-spanner/start
      * . prepare\_worker\_environment.sh
      * This script file:
        + Exports environment variables GCLOUD\_PROJECT and GCLOUD\_BUCKET.
        + Creates and configures a GCP Service Account.
        + Prints out the GCP Project ID.
    - To start the worker application, enter the following command:
      * npm run worker
    - The worker application runs as a console application.
    - By the end of the lab, it will:
      * Subscribe to a Cloud Pub/Sub topic.
      * Consume message data published to the topic.
      * Use the Cloud Natural Language ML API to perform sentiment analysis on the text.
      * Save the feedback data and the sentiment score to Cloud Spanner.
    - click Web preview > Preview on port 8080 to preview the Quiz application.
      * Take Test.
      * After you answer the question, you should see a final screen inviting you to submit feedback.
    - press Ctrl+C to stop the web application.
  + Examining the Case Study Application Code
    - Review the GCP application code
      * Navigate to the pubsub-languageapi-spanner/start folder
      * Select the pubsub.js file in the .../server/gcp folder.
        + This file contains a module that allows applications to publish feedback messages to a Cloud Pub/Sub topic and register a callback to receive messages from a Cloud Pub/Sub subscription.
      * Select the languageapi.js file in the ...server/gcp folder.
        + This file contains a module that allows users to send text to the Cloud Natural Language ML API and to receive the sentiment score from the API.
      * Select the spanner.js file in the ...server/gcp folder.
        + This file contains a module that allows users to save the feedback and Natural Language API response data in a Cloud Spanner database instance.
    - Review the web application code
      * Select the index.js file in the ...server/api folder.
        + The handler for POST messages sent to the /api/quizzes/feedback/:quiz route publishes the feedback data received from the client to Pub/Sub.
      * Select the worker.js file in the ...console folder.
        + This file runs as a separate console application to consume the messages delivered to a Pub/Sub subscription.
  + Working with Cloud Pub/Sub
    - In this section, you will create a Cloud Pub/Sub topic and subscription in your GCP project, publish a message, and retrieve it.
    - Create a Cloud Pub/Sub topic
      * Navigation > Pub/Sub > Click Create Topic > For Topic ID (feedback), click CREATE TOPIC.
    - Create a Cloud Pub/Sub subscription
      * Go to the second Cloud Shell window.
      * To create a Cloud Pub/Sub subscription named cloud-shell-subscription against the feedback topic, execute the following command:
        + gcloud pubsub subscriptions create cloud-shell-subscription --topic feedback
    - Publish a message to a Cloud Pub/Sub topic
      * To publish a "Hello World" message into the feedback topic, execute the following command:
        + gcloud pubsub topics publish feedback --message "Hello World"
    - Retrieve a message from a Cloud Pub/Sub subscription
      * To pull the message from the feedback topic with automatic acknowledgement of the message, execute the following command:
        + gcloud pubsub subscriptions pull cloud-shell-subscription --auto-ack
  + Publishing Messages to Cloud Pub/Sub Programmatically
    - Open the ...gcp/pubsub.js file in the editor.
    - Load the '@google-cloud/pubsub' module.
    - Create a new Cloud Pub/Sub client.
    - Create a topic object to reference the feedback Pub/Sub topic you created earlier.
    - pubsub.js
      * const config = require('../config');
      * // TODO: Load the Cloud Pub/Sub module
      * const {PubSub} = require('@google-cloud/pubsub');
      * // END TODO
      * // TODO: Create a client object against Cloud Pub/Sub
      * // The PubSub(...) factory function accepts an options
      * // object which is used to specify which project's Cloud
      * // Pub/Sub topics should be used via the projectId
      * // property.
      * // The projectId is retrieved from the config module.
      * // This module retrieves the project ID from the
      * // GCLOUD\_PROJECT environment variable.
      * const pubsub = new PubSub({
      * projectId: config.get('GCLOUD\_PROJECT')
      * });
      * // END TODO
      * // TODO: Get a reference to the feedback topic
      * // This code assumes that the topic is already present in
      * // the project.
      * // If it isn't then you would need to handle that case by
      * // using the pubsub object's createTopic(...) method
      * const feedbackTopic = pubsub.topic('feedback');
      * // END TODO
    - Write code to publish a message to Cloud Pub/Sub
    - In the publishFeedback(feedback) function, publish a message to the feedback topic.
    - pubsub.js
      * function publishFeedback(feedback) {
      * // TODO: Publish a message to the feedback topic
      * // This runs asynchronously so you need to return the
      * // Promise that results from executing topic.publish(...)
      * // The feedback object must be converted to a buffer
      * // In addition, it's a good idea to use a consistent
      * // property for the message body. This lab will use the
      * // name dataBuffer for the message data
      * const dataBuffer=Buffer.from(JSON.stringify(feedback))
      * return feedbackTopic.publish(dataBuffer);
      * // END TODO
      * }
    - Write code to use the Pub/Sub publish functionality
    - In the .../api/index.js file, load the '../gcp/pubsub' module.
    - api/index.js
      * + // TODO: Load the ../gcp/pubsub module
        + const publisher = require('../gcp/pubsub');
        + // END TODO
    - In the post(...) handler for the '/feedback/:quiz' route, invoke the publisher.publishFeedback(feedback) method.
    - Then, return a response to the client indicating that feedback was received.
    - In the catch block for error handling, invoke the next Express middleware.
    - Here's how the complete handler might look.
    - api/index.js
      * /\*\*
      * \* POST /api/quizzes/feedback/:quiz
      * \*
      * \* Submit the quiz feedback, get a response
      * \*/
      * router.post('/feedback/:quiz', (req, res, next) => {
      * const feedback = req.body; // in the form [{id, answer}]
      * console.log(feedback);
      * // TODO: Publish the message into Cloud Pub/Sub
      * publisher.publishFeedback(feedback).then(() => {
      * // TODO: Move the statement that returns a message to
      * // the client app here
      * res.json('Feedback received'); // moved here
      * // END TODO
      * // TODO: Add a catch
      * }).catch(err => {
      * // TODO: There was an error, invoke the next middleware
      * next(err);
      * // END TODO
      * });
      * // END TODO
      * });
    - Run the application and create a Pub/Sub message
      * Preview the web application.
      * Click Take Test.
      * enter some feedback text, and click Send Feedback.
      * In the second Cloud Shell window, to pull a message from the cloud-shell-subscription:
        + gcloud pubsub subscriptions pull cloud-shell-subscription --auto-ack
  + Subscribing to Cloud Pub/Sub Topics Programmatically
    - In this section you will write the code to create a subscription to a Cloud Pub/Sub topic and receive message notifications in the worker console application.
    - Write code to create a Cloud Pub/Sub subscription and receive messages
      * Return to the ...gcp/pubsub.js file.
      * In the registerFeedbackNotification(cb) function, create a subscription called 'worker-subscription' against the feedback topic.
      * Configure the subscription options to automatically acknowledge messages.
      * Add an error handler to trap the error if the subscription already exists.
      * Using the subscription object, use the get() method to return a promise to register an event handler for message events.
      * In the promise also register an error handler for error events.
      * Add a final catch to the promise to log errors.
      * pubsub.js
        + // The worker application will pass a callback to this
        + // method as the cb argument so it is notified when a
        + // feedback PubSub message is received
        + function registerFeedbackNotification(cb) {
        + // TODO: Create a subscription called worker-subscription
        + // TODO: Have it auto-acknowledge messages
        + feedbackTopic.createSubscription('worker-subscription', { autoAck: true }, (err, feedbackSubscription) => {
        + // TODO: Trap errors where the subscription already exists
        + // Create a subscription object for worker-subscription if
        + // the subscription already exists
        + // err.code == 6 means subscription already exists
        + if (err && err.code == 6) {
        + // subscription already exists
        + console.log("Feedback subscription already exists");
        + feedbackSubscription=feedbackTopic.subscription('worker-subscription')
        + }
        + // END TODO
        + // TODO: Use the get() method on the subscription object to call
        + // the API request to return a promise
        + feedbackSubscription.get().then(results => {
        + // The results argument in the promise is an array - the
        + // first element in this array is the subscription object.
        + // TODO: Declare a subscription constant
        + const subscription = results[0];
        + // END TODO
        + // TODO: Register an event handler for message events
        + // Use an arrow function to handle the event
        + // When a message arrives, invoke a callback
        + subscription.on('message', message => {
        + cb(message.data);
        + });
        + // END TODO
        + // TODO: Register an event handler for error events
        + // Print the error to the console
        + subscription.on('error', err => {
        + console.error(err);
        + });
        + // END TODO
        + })
        + // END TODO for the get() method promise
        + // TODO
        + // Add a final catch to the promise to handle errors
        + .catch(error => { console.log("Error getting feedback subscription", error)});
        + // END TODO
        + });
        + // END TODO for the create subscription method
        + }
    - Write code to use the Pub/Sub subscribe functionality
      * In the .../console/worker.js file, load the '../server/gcp/pubsub' module.
      * In the handler(message) function, and after the existing code, log the message to the console.
      * console/worker.js
        + // TODO: Load the ../server/gcp/pubsub module
        + const subscriber = require('../server/gcp/pubsub');
        + // END TODO
        + // TODO: Load the ../server/gcp/languageapi module
        + // END TODO
        + // TODO: Load the ../server/gcp/spanner module
        + // END TODO
        + console.log('Worker starting...');
        + // The callback function - invoked when a message arrives
        + function handler(message) {
        + console.log('Message received');
        + // TODO: Log the message to the console
        + var messageData = JSON.parse(message.toString());
        + console.log(messageData);
        + // END TODO
      * At the end of the file, add the following code to register the handler function as the Pub/Sub subscription callback.
        + // TODO: Register the callback with the module
        + subscriber.registerFeedbackNotification(handler);
        + // END TODO
    - Run the web and worker application and create a Pub/Sub message
      * In the first Cloud Shell window, start the web application if it's not already running.
      * In the second Cloud Shell window, start the worker application.
        + npm run worker
      * Click Web preview > Preview on port 8080 to preview the quiz application.
      * Click Take Test.
      * enter some feedback text, and then click Send Feedback.
      * Return to the second Cloud Shell window.
      * Return to the 2nd Cloud Shell window, and press Ctrl+C to stop the worker application.
      * Return to the first Cloud Shell window, and press Ctrl+C to stop the web application.
  + Using the Cloud Natural Language API
    - In this section you will write the code to perform sentiment analysis on the feedback text submitted by the user.
    - For more information, see API documentation for the Cloud Natural Language API.
    - Write code to invoke the Cloud Natural Language API
      * In the editor, move to the top of the ...gcp/languageapi.js file.
      * Load the '@google-cloud/language module.
      * Create a new Language Service Client. This client provides access to Google Cloud Natural Language API text analysis operations such as sentiment analysis and entity recognition.
      * Move to the analyze(...) function, and create an object literal to pass the text data for analysis as a parameter to the analysis methods of the Natural Language client object.
      * Configure this parameter object with two properties: content and type.
      * Assign the feedback text to this object's content property.
      * Set the type property value to PLAIN\_TEXT.
      * Pass the parameter object to the Language Service Client's analyzeSentiment method to analyze the document.
      * The Cloud Natural Language API expects an object with the form:
        + { document: { content: 'Content to be analyzed', type: 'SOME\_TYPE' }}
      * Then, return the sentiment score from the Natural Language API.
      * This is found in sentiment.documentSentiment.score
      * Here's how your complete analyze(...) function should look.
    - languageapi.js
      * // Import the config module
      * const config = require('../config');
      * // TODO: Load the Natural Language ML API module
      * const Language = require('@google-cloud/language');
      * // END TODO
      * // TODO: Create a client object against the Language API
      * // using the Language.LanguageServiceClient function
      * // The LanguageServiceClient function accepts an options
      * // object which is used to specify which project should be
      * // billed for use of the API via the projectId property.
      * // The projectId is retrieved from the config module.
      * // This module retrieves the project ID from the
      * // GCLOUD\_PROJECT environment variable.
      * const language = new Language.LanguageServiceClient({
      * projectId: config.get('GCLOUD\_PROJECT')
      * });
      * // END TODO
      * function analyze(text) {
      * // TODO: Create an object named document with the
      * // correct structure for the Natural Language ML API
      * // TODO: Initialize object content & type properties
      * // TODO: Set content from text arg
      * // TODO: Set type to PLAIN\_TEXT
      * const document = {
      * content: text,
      * type: 'PLAIN\_TEXT'
      * };
      * // END TODO
      * // TODO: Perform sentiment detection
      * return language.analyzeSentiment({ document })
      * // TODO: Chain then
      * // When the results come back
      * // The sentiment data is the first element
      * .then(results => {
      * const sentiment = results[0];
      * // TODO: Get the sentiment score (-1 to +1)
      * return sentiment.documentSentiment.score;
      * });
      * // END TODO
      * }
  + Write code to use the Natural Language API functionality
    - In the .../console/worker.js file, load the '../server/gcp/languageapi' module.
    - In the handler(message) function, and after the existing code, perform sentiment detection on the feedback.
    - Then, log the score to the console.
    - Assign a new score property to the feedback object.
    - Return the message data.
  + console/worker.js
    - // TODO: Load the ../server/gcp/languageapi module
    - const languageAPI = require('../server/gcp/languageapi');
    - // END TODO
    - console.log('Worker starting...');
    - // The callback function - invoked when a message arrives
    - function handler(message) {
    - console.log('Message received');
    - // TODO: Log the message to the console
    - var messageData = JSON.parse(message.toString());
    - console.log(messageData);
    - // END TODO
    - // TODO: Invoke the languageapi module method
    - // with the feedback from the student
    - languageAPI.analyze(messageData.feedback)
    - .then(score => {
    - // TODO: Log sentiment score
    - console.log(`Score: ${score}`);
    - // END TODO
    - // TODO: Add a score property to feedback object
    - // and return updated feedback object
    - messageData.score = score;
    - return messageData;
    - // END TODO
    - })
    - // END TODO
    - // TODO: Pass on the feedback object
    - // to next Promise handler
    - // END TODO
    - // TODO: Add third .then(...)
    - // TODO Log feedback saved message
    - // END TODO
    - // END TODO
    - // TODO close off the promise chain with a catch() and log
    - // any errors to the console
    - // END TODO
    - }
  + Run the web and worker application and test the Natural Language API
    - Return to the first Cloud Shell window.
    - Start the web application.
    - Switch to the second Cloud Shell window.
    - Start the worker application.
    - Preview the web application.
    - Click Take Test. enter some feedback text, and then click Send Feedback.
    - Return to the second Cloud Shell window.
    - You should see that the worker application has invoked the Cloud Natural Language API and displayed the sentiment score in the console.
    - Stop the web and console applications.
  + Persisting Data to Cloud Spanner
    - In this section you will create a Cloud Spanner instance, database, and table. Then you will write the code to persist the feedback data into the database.
    - Create a Cloud Spanner instance
      * Navigation > Spanner > Create instance > For Instance name, type quiz-instance
      * Choose a configuration section, select Regional, and then select us-central1 as the region.
      * Click Create.
    - Create a Cloud Spanner database and table
      * On the Instance Details page for quiz-instance, click Create database.
      * For Name, type quiz-database.
      * Under Define your schema, type the following SQL statement:
        + CREATE TABLE Feedback (
        + feedbackId STRING(100) NOT NULL,
        + email STRING(100),
        + quiz STRING(20),
        + feedback STRING(MAX),
        + rating INT64,
        + score FLOAT64,
        + timestamp INT64 )
        + PRIMARY KEY (feedbackId);
      * Click Create.
  + Write code to persist data into Cloud Spanner
    - For more information, see the API documentation for Cloud Spanner
    - Return to the code editor, and move to the top of the ...gcp/spanner.js file.
    - Load the '@google-cloud/spanner' module.
    - Create a new Cloud Spanner client.
    - Get a reference to the Spanner instance.
    - Get a reference to the Spanner database.
    - Get a reference to the feedback Spanner database table.
    - spanner.js
      * // Import the config module
      * const config = require('../config');
      * // TODO: Import the @google-cloud/spanner module
      * const {Spanner} = require('@google-cloud/spanner');
      * // END TODO
      * // TODO: Create a client object to access Cloud Spanner
      * // The Spanner(...) factory function accepts an options
      * // object which is used to select which project's Cloud
      * // Spanner database instance(s) should be used via the
      * // projectId property.
      * // The projectId is retrieved from the config module.
      * // This module retrieves the project ID from the
      * // GCLOUD\_PROJECT environment variable.
      * const spanner = new Spanner({
      * projectID: config.get('GCLOUD\_PROJECT')
      * });
      * // END TODO
      * // TODO: Get a reference to the Cloud Spanner instance
      * const instance = spanner.instance('quiz-instance');
      * // END TODO
      * // TODO: Get a reference to the Cloud Spanner database
      * const database = instance.database('quiz-database');
      * // END TODO
      * // TODO: Get a reference to the Cloud Spanner table
      * const feedbackTable = database.table('feedback');
      * // END TODO
    - Move to the saveFeedback(...) function.
    - Create a 'reversed' email. Example Original: app.dev.student@example.org Reversed: org\_example\_student\_dev\_app
    - Create a record object to insert into Spanner using the feedback object's properties. You will need to use Spanner.float (score) to ensure that the Natural Language score has the correct format for Cloud Spanner.
    - Create a key for this record from the 'reversed' email, quiz, and timestamp.
    - Use await to insert the record into the feedback table inside a try block and catch any errors.
    - Add a catch block and log a message for any errors. Spanner uses err.code==6 to indicate a record with the same id already exists, which indicates a duplicate message from PubSub in this case.
    - spanner.js
      * async function saveFeedback(
      * { email, quiz, timestamp, rating, feedback, score }) {
      * // TODO: Declare rev\_email constant
      * // TODO: Produce a 'reversed' email address
      * // eg app.dev.student@example.org -> org\_example\_student\_dev\_app
      * const rev\_email = email
      * .replace(/[@\.]/g, '\_')
      * .split('\_')
      * .reverse()
      * .join('\_');
      * // END TODO
      * // TODO: Create record object to be inserted into Spanner
      * const record = {
      * feedbackId: `${rev\_email}\_${quiz}\_${timestamp}`,
      * email,
      * quiz,
      * timestamp,
      * rating,
      * score: Spanner.float(score),
      * feedback,
      * };
      * // END TODO
      * // TODO: Insert the record into the table
      * // use try {} catch {} and check for err.code==6 to trap
      * // insert errors caused by duplicated PubSub messages
      * try {
      * console.log('Saving feedback');
      * await feedbackTable.insert(record);
      * } catch (err) {
      * if (err.code === 6 ) {
      * console.log("Duplicate message - feedback already saved");
      * } else {
      * console.error('ERROR processing feedback:', err);
      * }
      * }
      * // END TODO
      * }
  + Write code to use the Cloud Spanner functionality
    - In the .../console/worker.js file, load the '../server/gcp/spanner' module.
    - In the handler(message) function, add a second .then(...) chained method to the first one.
    - In the body of the second .then(...) method, pass a reference to the method that saves the feedback into Spanner.
    - The second .then(...) method also returns a Promise, so add a third .then(...) chained method to the end.
    - Use an arrow function as the callback in the .then(...) method with no arguments and an empty body.
    - In the body of the arrow function, log a message to the console to say that the feedback has been saved.
    - Close off the promise chain with a catch and log any errors to the console.
    - console/worker.js
      * // TODO: Load the ../server/gcp/pubsub module
      * const subscriber = require('../server/gcp/pubsub');
      * // END TODO
      * // TODO: Load the ../server/gcp/languageapi module
      * const languageAPI = require('../server/gcp/languageapi');
      * // END TODO
      * // TODO: Load the ../server/gcp/spanner module
      * const feedbackStorage = require('../server/gcp/spanner');
      * // END TODO
      * // The callback function - invoked when a message arrives
      * function handler(message) {
      * console.log('Message received');
      * // TODO: Log the message to the console
      * var messageData = JSON.parse(message.toString());
      * console.log(messageData);
      * // END TODO
      * // TODO: Invoke the languageapi module method
      * // with the feedback from the student
      * languageAPI.analyze(messageData.feedback)
      * .then(score => {
      * // TODO: Log sentiment score
      * console.log(`Score: ${score}`);
      * // END TODO
      * // TODO: Add a score property to feedback object
      * messageData.score = score;
      * return messageData;
      * })
      * // TODO: Pass on the feedback object
      * // to next Promise handler
      * .then(feedbackStorage.saveFeedback)
      * // END TODO
      * // TODO: Add third .then(...)
      * .then(() => {
      * // TODO Log feedback saved message
      * console.log('Feedback saved');
      * // END TODO
      * })
      * // END TODO
      * // TODO close off the promise with a catch and log
      * // any errors
      * .catch(console.error);
      * // END TODO
      * }
      * // TODO: Register the callback with the module
      * subscriber.registerFeedbackNotification(handler);
      * // END TODO
  + Run the web and worker application and test Cloud Spanner
    - Save all the files, and then return to the first Cloud Shell window.
    - If the web application is not running, start it.
    - Switch to the second Cloud Shell window.
    - Restart the worker application.
    - Preview the web application.
    - Click Take Test. enter some feedback text, and then click Send Feedback.
    - Return to the second Cloud Shell window.
    - You should see that the worker application has invoked the Cloud Spanner API and displayed the message in the console window.
    - Return to the Cloud Platform Console.
    - Navigation > click Spanner > Select quiz-instance > quiz-database > Query.
    - To execute a query, for Query, type SELECT \* FROM Feedback, and then click Run.
      * SELECT \* FROM Feedback
    - You should see the new feedback record in the Cloud Spanner database, including the message data from Cloud Pub/Sub and the score from the Cloud Natural Language API
  + Bonus: Implementing the Leaderboard
    - When a student completes a quiz, their answers are submitted in an API call back to the server. Your job is to capture the student-submitted answers and the correct answers and save them into Cloud Spanner.
    - To do this you will:
      * Create a Cloud Pub/Sub topic called answers.
      * Create a Cloud Spanner table called Answers with appropriate column names and data types.
      * Post the answer data to the answers topic.
      * Subscribe to the answers topic in the console application and insert the answer data into the Answers table.
      * Create a handler and pug template in the Express application to display the data from Cloud Spanner when the user browses to the Leaderboard.
      * The details are left up to you! You can find the solution to the bonus in the lab's bonus folder.
  + Review
    - When using Cloud Pub/Sub, what method do you use to get an existing topic?
      * createTopic(topicName)
      * topic(topicName)
    - When using Cloud Pub/Sub, which subscription events do you use?
      * finish
      * error
      * message
      * subscriptionAdded
    - With Cloud Spanner, which object has a database(...) method?
      * spanner
      * instance
      * database
      * table
    - With Cloud Spanner, which table method corresponds to an INSERT SQL statement?
      * create(...)
      * insert(...)
      * add(...)
      * push(...)

### App Dev - Developing a Backend Service: Java 2 hours

* + Overview
    - In this lab, you develop a backend service for an online Quiz application to process user feedback and save scores.
    - The Quiz application has two parts, the web application that will run in the first Cloud shell window and the worker application that runs in the second Cloud Shell window.
      * Web application: manages the logic of sending the user's feedback to a pub/sub topic.
      * Worker application: listens to the feedback provided by the user to eventually perform sentiment analysis and store them in a database (Cloud Spanner).
    - This process takes advantage of Google Cloud Platform (GCP) products and services:
      * Cloud Pub/Sub: The Topic alerts and provides the subscribing worker application to new scores and feedback for analysis.
      * Cloud Natural Language: Provides sentiment analysis on the feedback.
      * Cloud Spanner: Database for the Quiz application.
  + Objectives
    - Create and publish messages to a Cloud Pub/Sub topic.
    - Subscribe to the topic to receive messages in a separate worker application.
    - Perform sentiment analysis on feedback.
    - Create a Cloud Spanner database instance and schema, then insert data into the database.
  + Prepare the Quiz application
    - git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - ln -s ~/training-data-analyst/courses/developingapps/v1.3/java/pubsub-languageapi-spanner ~/pubsub-languageapi-spanner
    - cd ~/pubsub-languageapi-spanner/start
    - . prepare\_web\_environment.sh
    - This script file:
      * Creates an App Engine application.
      * Exports environment variables: GCLOUD\_PROJECT and GCLOUD\_BUCKET.
      * Runs mvn clean install.
      * Creates entities in Cloud Datastore.
      * Prints out the GCP Project ID.
    - Run the web application:
      * mvn spring-boot:run
    - Set up the worker application
      * cd ~/pubsub-languageapi-spanner/start
      * . prepare\_worker\_environment.sh
      * This script file:
        + Exports environment variables GCLOUD\_PROJECT and GCLOUD\_BUCKET.
        + Creates and configures a GCP Service Account.
        + Prints out the Google Cloud Platform Project ID.
    - Now start the worker application:
      * mvn exec:java@worker
    - Review the Quiz application
      * click Web preview > Preview on port 8080 to preview the Quiz application.
      * Take Test & feed back.
  + Examine the Quiz Application Code
    - Review the GCP application code
    - Navigate to the /pubsub-languageapi-spanner/start folder using the file browser panel on the left side of the editor.
    - Now expand the /src/main/java/com/google/training/appdev folder. All Java code paths are relative to this folder.
    - Select the Feedback.java file in the .../services/gcp/domain folder.
      * This file contains a model class that represents the feedback submitted by quiz takers.
    - Select the PublishService.java file in the .../services/gcp/pubsub folder.
      * This file contains a service class that allows applications to publish feedback messages to a Cloud Pub/Sub topic.
    - Select the LanguageService.java file in the .../services/gcp/languageapi folder. This file contains a service class that allows users to send text to the Cloud Natural Language ML API and to receive the sentiment score from the API.
    - Select the SpannerService.java file in the .../services/gcp/spanner folder.
      * This file contains a service class that allows users to save the feedback and Natural Language API response data in a Cloud Spanner database instance.
    - Review the web and backend application code
    - Select the QuizEndpoint.java file in the .../apifolder.
      * The handler for POST messages sent to the /api/quizzes/feedback/:quiz route publishes the feedback data received from the client to Pub/Sub.
    - Select the ConsoleApp.java file in the .../backend folder.
      * This file runs as a separate console application to consume the messages delivered to a Pub/Sub subscription.
  + Working with Cloud Pub/Sub
    - Navigation menu > Pub/Sub > Topics, and then click Create topic.
    - Set the Topic ID to feedback, and then click CREATE TOPIC.
    - Create a Cloud Pub/Sub subscription
      * gcloud pubsub subscriptions create cloud-shell-subscription --topic feedback
    - Publish a message to a Cloud Pub/Sub topic
      * gcloud pubsub topics publish feedback --message="Hello World"
    - Retrieve a message from a Cloud Pub/Sub subscription
      * gcloud pubsub subscriptions pull cloud-shell-subscription --auto-ack
  + Publish Messages to Cloud Pub/Sub Programmatically
    - Publish a Pub/Sub message
    - Open the .../services/gcp/pubsub/PublishService.java file in the code editor. Update the file by adding code as directed.
    - Declare two static final strings for the PROJECT\_ID and TOPIC\_NAME.
      * // TODO: Declare and initialize two Strings,
      * // PROJECT\_ID and TOPIC\_NAME
      * private static final String PROJECT\_ID = ServiceOptions.getDefaultProjectId();
      * private static final String TOPIC\_NAME = "feedback";
      * // END TODO
    - Move to the publishFeedback(...) method. Create a TopicName object using the PROJECT\_ID and TOPIC\_NAME strings.
    - The topic name references the Cloud Pub/Sub topic you just created.
      * // TODO: Create a TopicName object
      * // for the feedback topic in the project
      * TopicName topicName = TopicName.of(PROJECT\_ID, TOPIC\_NAME);
    - Declare a Publisher object and set it to null. It will be initialized in the try block that follows.
      * // TODO: Declare a publisher for the topic
      * Publisher publisher = null;
      * // END TODO
    - Move to the try block, and initialize the publisher object using its builder.
      * // TODO: Initialize the publisher
      * // using a builder and the topicName
      * publisher = Publisher.newBuilder(topicName).build();
      * // END TODO
    - Copy the JSON serialized feedbackMessage string to a ByteString.
      * // TODO: Copy the serialized message
      * // to a byte string
      * ByteString data = ByteString.copyFromUtf8(feedbackMessage);
      * // END TODO
    - Declare a PubsubMessage object; initialize the message using its builder.
      * // TODO: Create a Pub/Sub message using a
      * // builder; set the message data
      * PubsubMessage pubsubMessage = PubsubMessage.newBuilder().setData(data).build();
      * // END TODO
    - Use the publisher to publish the message, assign the return value to the message ID future object.
      * // TODO: Publish the message,
      * // assign to the messageIdFuture
      * messageIdFuture = publisher.publish(pubsubMessage);
      * // END TODO
    - Move to the finally block and retrieve the Pub/Sub messageId from the message ID future object.
      * // TODO: Get the messageId from
      * // the messageIdFuture
      * String messageId = messageIdFuture.get();
      * // END TODO
    - Complete the publishing code by shutting down the publisher.
      * // TODO: Shutdown the publisher
      * // to free up resources
      * if (publisher != null) {
      * publisher.shutdown();
      * }
      * // END TODO
  + Write code to use the Pub/Sub publish functionality
    - In the .../api/QuizEndpoint.java file, declare a new PublishService field named publishService. Apply the Spring @Autowired annotation.
      * // TODO: Declare the publishService
      * @Autowired
      * private PublishService publishService;
      * // END TODO
    - In the processFeedback(...) method that handles POST requests to the '/feedback/:quiz' route, invoke the publishService.publishFeedback(feedback) method.
      * // TODO: Publish the feedback to Pub/Sub
      * publishService.publishFeedback(feedback);
      * // END TODO
  + Run the application and create a Pub/Sub message
    - mvn spring-boot:run
    - Take test & feed back
    - gcloud pubsub subscriptions pull cloud-shell-subscription --auto-ack
  + Subscribing to Cloud Pub/Sub Topics Programmatically
    - In this section you write the code to create a subscription and receive message notifications from a Cloud Pub/Sub topic to the worker application.
    - Write code to create a Cloud Pub/Sub subscription and receive messages
    - In the code editor open the ...backend/ConsoleApp.java file. Update the file by adding code as directed.
    - Skip over the TODO blocks for languageService and spannerService. You'll return for those later.
    - In the main()method, create a SubscriptionName object representing a new subscription named "worker1-subscription".
      * // TODO: Create the Pub/Sub subscription name
      * ProjectSubscriptionName subscription =
      * ProjectSubscriptionName.of(projectId,
      * "worker1-subscription");
      * // END TODO
    - Create a SubscriptionAdminClient object using a try block.
    - Also in the try block, use the subscription admin client to create a new subscription against the feedback topic. Don't forget the close brace.
      * // TODO: Create the subscriptionAdminClient
      * try (SubscriptionAdminClient subscriptionAdminClient =
      * SubscriptionAdminClient.create()) {
      * // TODO: create the Pub/Sub subscription
      * // using the subscription name and topic
      * subscriptionAdminClient.createSubscription(
      * subscription, topic,
      * PushConfig.getDefaultInstance(), 0);
      * // END TODO
      * }
      * // END TODO
    - Move to the code that creates a MessageReceiver, and in the receiveMessage(...) override, extract the message data into a String.
      * // TODO: Extract the message data as a JSON String
      * String fb = message.getData().toStringUtf8();
      * // END TODO
    - Use the consumer to acknowledge the message.
      * // TODO: Ack the message
      * consumer.ack();
      * // END TODO
    - After the code that initializes an ObjectMapper, deserialize the JSON String message data into a feedback object.
      * // TODO: Deserialize the JSON String
      * // representing the feedback
      * // Print out the feedback
      * Feedback feedback = mapper.readValue(
      * fb, Feedback.class);
      * System.out.println("Feedback received: "
      * + feedback);
      * // END TODO
    - After the block that creates the MessageReceiver, declare a Subscriber and initialize it to null.
    - Skip over a few TODO blocks and look for this one.
      * // TODO: Declare a subscriber
      * Subscriber subscriber = null;
      * // END TODO
    - Move to the try block, and initialize the Subscriber using its default builder. This requires the subscription and receiver.
      * // TODO: Initialize the subscriber using
      * // its default builder
      * // with a subscription and receiver
      * subscriber = Subscriber.newBuilder(
      * subscription, receiver).build();
      * // END TODO
    - Add a listener to the subscriber to display errors.
      * // TODO: Add a listener to the subscriber
      * subscriber.addListener(
      * new Subscriber.Listener() {
      * @Override
      * public void failed(
      * Subscriber.State from,
      * Throwable failure) {
      * System.err.println(failure);
      * }
      * },
      * MoreExecutors.directExecutor());
      * // END TODO
    - Start the subscriber.
      * // TODO: Start subscribing
      * subscriber.startAsync().awaitRunning();
      * // END TODO
    - Move to the finally block. Write the code to stop the subscriber, and delete the subscription.
      * // TODO: Stop subscribing
      * if (subscriber != null) {
      * subscriber.stopAsync().awaitTerminated();
      * }
      * // END TODO
      * // TODO: Delete the subscription
      * try (SubscriptionAdminClient
      * subscriptionAdminClient =
      * SubscriptionAdminClient.create()) {
      * subscriptionAdminClient.deleteSubscription(
      * subscription);
      * }
      * // END TODO
    - Run the web and worker application and create a Pub/Sub message
  + Use the Cloud Natural Language API
    - In this section you write the code to perform sentiment analysis on the feedback text submitted by the user. For more information see Google Cloud Natural Language API.
    - Write code to invoke the Cloud Natural Language API
    - Return to the editor and open the LanguageService.java file in the services/gcp/languageapi folder.
    - Move to the analyzeSentiment(...) method, and create a LanguageServiceClient object in a try block. In this step note that there is not a // END TODO in the content that you copy into the file.
      * // TODO: Create the LanguageServiceClient object
      * try (LanguageServiceClient language =
      * LanguageServiceClient.create()) {
    - Create a new Document object using its builder. Configure this object with the document content and type.
      * // TODO: Create a new Document object
      * // using the builder
      * // Set the content and type
      * Document doc = Document.newBuilder()
      * .setContent(feedback)
      * .setType(Document.Type.PLAIN\_TEXT)
      * .build();
      * // END TODO
    - Use the Natural Language client object to analyze the sentiment of the document, assigning the result to a Sentiment object.
      * // TODO: Use the client to analyze
      * // the sentiment of the feedback
      * Sentiment sentiment = language
      * .analyzeSentiment(doc)
      * .getDocumentSentiment();
      * // END TODO
    - Then, return the sentiment score from the sentiment object.
      * // TODO: Return the sentiment score
      * return sentiment.getScore();
      * // END TODO
      * }
  + Write code to use the Natural Language API functionality
    - Return to the backend/ConsoleApp.java file.
    - Move to the main(...) method.
    - In the main()method, create a SubscriptionName object representing a new subscription named "worker2-subscription".
    - This replaces the "worker1-subscription".
      * // TODO: Create the Pub/Sub subscription name
      * ProjectSubscriptionName subscription =
      * ProjectSubscriptionName.of(projectId,
      * "worker2-subscription");
      * // END TODO
    - At the point indicated by the comments, create the LanguageService instance using its static create() method.
      * // TODO: Create the languageService
      * LanguageService languageService = LanguageService.create();
      * // END TODO
    - At the point indicated by the comments, use the languageService object to perform sentiment detection on the feedback.
      * // TODO: Use the Natural Language API to analyze sentiment
      * float sentimentScore = languageService.analyzeSentiment(
      * feedback.getFeedback());
      * // END TODO
    - Then, log the score to the console and assign a new score property to the feedback object.
      * // TODO: Set the feedback object sentiment score
      * feedback.setSentimentScore(sentimentScore);
      * System.out.println("Score is: " + sentimentScore);
      * // END TODO
    - Run the web and worker application and test the Natural Language API
  + Persist Data to Cloud Spanner
    - In this section you create a Cloud Spanner instance, database, and table. You then write the code to persist the feedback data into the database.
    - Create a Cloud Spanner instance
      * Navigation > Spanner > Click CREATE INSTANCE > Set Instance name to quiz-instance
      * In the Choose a configuration section, select us-central1 from the select a configuration dropdown menu.
      * Click Create.
    - Create a Cloud Spanner database and table
      * On the Instance details page for quiz-instance, click CREATE DATABASE.
      * For Database name, type quiz-database.
      * Under Define your schema. In the text box, type the following SQL statement:
        + CREATE TABLE Feedback (
        + feedbackId STRING(100) NOT NULL,
        + email STRING(100),
        + quiz STRING(20),
        + feedback STRING(MAX),
        + rating INT64,
        + score FLOAT64,
        + timestamp INT64 )
        + PRIMARY KEY (feedbackId);
      * Click Create.
  + Write code to persist data into Cloud Spanner
    - Return to the code editor, and move to the insertFeedback(...) method in the ...services/gcp/spanner/SpannerService.java file.
    - Get a reference to Cloud Spanner.
      * // TODO: Get a reference to the Spanner API
      * SpannerOptions options =
      * SpannerOptions.newBuilder().build();
      * Spanner spanner = options.getService();
      * // END TODO
    - Get a reference to the Spanner database via the Database Id.
      * // TODO: Get a reference to the quiz-instance
      * // and its quiz-database
      * DatabaseId db = DatabaseId.of(
      * options.getProjectId(),
      * "quiz-instance",
      * "quiz-database");
      * // END TODO
    - Get a reference to the Cloud Spanner Database client.
      * // TODO: Get a client for the quiz-database
      * DatabaseClient dbClient =
      * spanner.getDatabaseClient(db);
      * // END TODO
    - Create a new List<Mutation> to reference all the changes that will be made to the database.
      * // TODO: Create a list to hold mutations
      * // against the database
      * List<Mutation> mutations = new ArrayList<>();
      * // END TODO
      * // END TODO
    - Add the Mutation that represents an insert against the feedback table, using data from the feedback object.
      * // TODO: Add an insert mutation
      * mutations.add(
      * // TODO: Build a new insert mutation
      * Mutation.newInsertBuilder("Feedback")
      * .set("feedbackId")
      * .to(feedback.getEmail() + '\_' +
      * feedback.getQuiz() + "\_" +
      * feedback.getTimestamp())
      * .set("email")
      * .to(feedback.getEmail())
      * .set("quiz")
      * .to(feedback.getQuiz())
      * .set("feedback")
      * .to(feedback.getFeedback())
      * .set("rating")
      * .to(feedback.getRating())
      * .set("score")
      * .to(
      * feedback.getSentimentScore())
      * .set("timestamp")
      * .to(feedback.getTimestamp())
      * .build());
      * // END TODO
    - Use the database client to write the mutations.
      * // TODO: Write the change to Spanner
      * dbClient.write(mutations);
      * // END TODO
  + Write code to use the Cloud Spanner functionality
    - Move to the main(...) method in the backend/ConsoleApp.java file.
    - In the main()method, create a SubscriptionName object representing a new subscription named "worker3-subscription".
    - This replaces the "worker2-subscription".
      * // TODO: Create the Pub/Sub subscription name
      * ProjectSubscriptionName subscription =
      * ProjectSubscriptionName.of(projectId,
      * "worker3-subscription");
      * // END TODO
    - At the point indicated by the comments, create the SpannerService instance.
      * // TODO: Create the spannerService
      * SpannerService spannerService = SpannerService.create();
      * // END TODO
    - At the point indicated by the comments, use the spannerService object to insert the feedback into the database and print out a message to the console.
      * // TODO: Insert the feedback into Cloud Spanner
      * spannerService.insertFeedback(feedback);
      * System.out.println("Feedback saved");
      * // END TODO
  + Run the web and worker application and test Cloud Spanner
    - Return to the Console. Click Navigation menu > Spanner.
    - Select quiz-instance > quiz-database and click Query from the left pane .
    - To execute a query, in the Query dialog, type SELECT \* FROM Feedback, and then click Run.
      * SELECT \* FROM Feedback

### App Dev - Developing a Backend Service: Python 2 hours

* + Overview
    - Google App Engine lets you manage resources from the command line, debug source code in production and run API backends. This lab concentrates on the backend service, putting together Pub/Sub, Natural Language, and Spanner services and APIs to collect and analyze feedback and scores from an online Quiz application.
  + Objectives
    - Create and publish messages to a Cloud Pub/Sub topic.
    - Subscribe to the topic to receive messages in a separate worker application.
    - Use the Cloud Natural Language Machine Learning API.
    - Create and configure a Cloud Spanner database instance, then insert data into the database.
  + Run a script to configure the web application:
    - Creates and configures a GCP Service Account to provide access to GCP services for your application.
    - Creates an App Engine application.
    - Exports environment variables GCLOUD\_PROJECT, GCLOUD\_BUCKET, and GOOGLE\_APPLICATION\_CREDENTIALS.
    - Creates a virtualenv isolated Python environment for Python 3 and activates it.
    - Updates pip and runs pip install -r requirements.txt.
    - Creates entities in Cloud Datastore.
    - Prints out the Google Cloud Platform Project ID.
    - . prepare\_web\_environment.sh
  + Run the web application:
    - python run\_server.py
    - . run\_worker.sh
  + This script file:
    - * Exports environment variables GCLOUD\_PROJECT, GCLOUD\_BUCKET, and GOOGLE\_APPLICATION\_CREDENTIALS.
      * Activates a virtualenv environment for Python 3.
      * Runs the worker application python -m quiz.console.worker.
  + Examine the Quiz application code
    - Review the GCP application code structure
      * /pubsub-languageapi-spanner/start/quiz/gcp/pubsub.py file
      * This file contains a module that allows applications to publish feedback messages to a Cloud Pub/Sub topic and register a callback to receive messages from a Cloud Pub/Sub subscription.
      * Select the languageapi.py file in the .../quiz/gcp folder.
      * This file contains a module that allows users to send text to the Cloud Natural Language ML API and to receive the sentiment score from the API.
      * Select the spanner.py file.
      * This file contains a module that allows users to save the feedback and Natural Language API response data in a Cloud Spanner database instance.
      * Review the web application code
    - Select the api.py file in the .../quiz/api folder.
      * The handler for POST messages sent to the /api/quizzes/feedback/:quiz route publishes the feedback data received from the client to Pub/Sub.
      * Select the worker.py file in the .../quiz/console folder.
      * This file runs as a separate console application to consume the messages delivered to a Pub/Sub subscription.
  + Work with Cloud Pub/Sub
  + Import and use the Python Cloud Pub/Sub module
    - In this section, you'll update ...quiz/gcp/pubsub.py to do the following:
    - Open the ...quiz/gcp/pubsub.py file in the editor.
    - Load the pubsub\_v1 module from the google.cloud package.
    - quiz/gcp/pubsub.py
      * # TODO: Load the Cloud Pub/Sub module
      * from google.cloud import pubsub\_v1
      * # END TODO
    - Construct a Cloud Pub/Sub Publisher client.
      * # TODO: Create a Pub/Sub Publisher Client
      * publisher = pubsub\_v1.PublisherClient()
      * # END TODO
    - Get the fully qualified path referencing the feedback Pub/Sub topic you created earlier.
      * # TODO: Create Topic Object to reference feedback topic
      * topic\_path = publisher.topic\_path(project\_id, 'feedback')
      * # END TODO
    - Write code to publish a message to Cloud Pub/Sub
    - In the publish\_feedback(feedback) function, publish a message to the feedback topic.
    - quiz/gcp/pubsub.py, publish\_feedback(feedback) function
      * # TODO: Publish the feedback object to the feedback topic
      * payload = json.dumps(feedback, indent=2,
      * sort\_keys=True)
      * data = payload.encode('utf-8')
      * future = publisher.publish(topic\_path, data=data)
      * return future.result()
      * # END TODO
    - Write code to use the Pub/Sub publish functionality
    - In the .../quiz/api/api.py file, load the pubsub module from the quiz.gcp package.
    - quiz/api/api.py
      * # TODO: Add pubsub to import list
      * from quiz.gcp import datastore, pubsub
      * # END TODO
    - In the publish\_feedback(feedback) function, remove the placeholder pass statement, invoke the pubsub.publish\_feedback(feedback) function, and return a response to the client indicating that feedback was received.
    - quiz/api/api.py, publish\_feedback(feedback) function
      * # TODO: Publish the feedback using your pubsub module,
      * # return the result
      * result = pubsub.publish\_feedback(feedback)
      * response = Response(json.dumps(result, indent=2,
      * sort\_keys=True))
      * response.headers['Content-Type'] = 'application/json'
      * return response
      * # END TODO
    - Run the application and create a Pub/Sub message
  + Subscribe to Cloud Pub/Sub Topics Programmatically
    - In this section you write code to create a subscription to a Cloud Pub/Sub topic and receive message notifications in the worker console application.
    - Write code to create a Cloud Pub/Sub subscription and receive messages
    - Return to the ...quiz/gcp/pubsub.py file.
    - Declare a Cloud Pub/Sub Subscriber Client.
      * /quiz/gcp/pubsub.py
      * # TODO: Create a Pub/Sub Subscriber Client
      * sub\_client = pubsub\_v1.SubscriberClient()
      * # END TODO
    - Get the fully qualified path referencing the 'worker-subscription'.
      * # TODO: Create a Subscription object named
      * # worker-subscription
      * sub\_path = sub\_client.subscription\_path(project\_id, 'worker-subscription')
      * # END TODO
    - Move to the pull\_feedback(callback) function, and remove the placeholder pass statement.
    - Use the subscriber client to subscribe to the worker subscription, invoking the callback when a message is received.
    - /quiz/gcp/pubsub.py, pull\_feedback(callback) function
      * # TODO: Subscribe to the worker-subscription,
      * # invoking the callback
      * sub\_client.subscribe(sub\_path, callback=callback)
      * # END TODO
    - Write code to use the Pub/Sub subscribe functionality
    - In the ...quiz/console/worker.py file, load the pubsub module from the quiz.gcp package.
    - console/worker.py
      * # TODO: Load the pubsub, languageapi and spanner modules from
      * # the quiz.gcp package
      * from quiz.gcp import pubsub
      * # END TODO
    - In the pubsub\_callback(message) function, acknowledge the message
    - console/worker.py, pubsub\_callback(message) function
      * # TODO: Acknowledge the message
      * message.ack()
      * # END TODO
    - Log the message to the console.
      * # TODO: Log the message
      * log.info('Message received')
      * log.info(message)
      * # END TODO
    - In the main() function, register the handler function as the Pub/Sub subscription callback.
    - console/worker.py, main() function
      * # TODO: Register the callback
      * pubsub.pull\_feedback(pubsub\_callback)
      * # END TODO
    - Run the web and worker applications and create a Pub/Sub message
  + Use the Cloud Natural Language API
    - In this section you write the code to perform sentiment analysis on the feedback text submitted by the user.
    - Write code to invoke the Cloud Natural Language API
    - In the editor, move to the top of the ...quiz/gcp/languageapi.py file.
    - Load the language module from the google.cloud package.
    - quiz/gcp/languageapi.py
      * # TODO: Import the language module
      * from google.cloud import language\_v1
      * # END TODO
    - Create a Cloud Natural Language client object.
      * # TODO: Create the Language API client
      * lang\_client = language\_v1.LanguageServiceClient()
      * # END TODO
    - Move to the analyze(text) function, and create a Document object to pass to the Natural Language client, assigning the feedback text to the content parameter.
    - quiz/gcp/languageapi.py, analyze(text) function
      * # TODO: Create a Document object
      * doc = language\_v1.types.Document(content=text,
      * type\_='PLAIN\_TEXT')
      * # END TODO
    - Use the Natural Language client object to analyze the sentiment of the document.
      * # TODO: Analyze the sentiment
      * sentiment = lang\_client.analyze\_sentiment(
      * document=doc).document\_sentiment
      * # END TODO
    - Return the sentiment score from the Natural Language API.
      * # TODO: Return the sentiment score
      * return sentiment.score
      * # END TODO
    - Write code to use the Natural Language API functionality
    - In the .../quiz/console/worker.py file, add languageapi to the import statement.
    - console/worker.py
      * # TODO: Load the pubsub, languageapi and spanner modules from
      * # from the quiz.gcp package
      * from quiz.gcp import pubsub, languageapi
      * # END TODO
    - In the pubsub\_callback(message) function, and after the existing code, perform sentiment detection on the feedback.
    - console/worker.py, pubsub\_callback(message) function
      * # TODO: Use the languageapi module to
      * # analyze the sentiment
      * score = languageapi.analyze(str(data['feedback']))
      * # END TODO
    - Log the score to the console.
      * # TODO: Log the sentiment score
      * log.info('Score: {}'.format(score))
      * # END TODO
    - Assign a new score property to the feedback object.
      * # TODO: Assign the sentiment score to
      * # a new score property
      * data['score'] = score
      * # END TODO
    - Run the web and worker application and test the Natural Language API
  + Persist Data to Cloud Spanner
    - Write code to persist data into Cloud Spanner
    - Return to the code editor, and move to the top of the .../quiz/gcp/spanner.py file.
    - Load the spanner module from the google.cloud package.
    - quiz/gcp/spanner.py
      * # TODO: Import the spanner module
      * from google.cloud import spanner
      * # END TODO
    - Construct a Cloud Spanner client.
      * # TODO: Create a spanner Client
      * spanner\_client = spanner.Client()
      * # END TODO
    - Get a reference to the Spanner instance.
      * # TODO: Get a reference to the Cloud Spanner quiz-instance
      * instance = spanner\_client.instance('quiz-instance')
      * # END TODO
    - Get a reference to the Spanner database.
      * # TODO: Get a referent to the Cloud Spanner quiz-database
      * database = instance.database('quiz-database')
      * # END TODO
    - Move to the saveFeedback(data) function.
    - Create a database.batch object using a with block. This can be used to perform multiple operations against a Spanner database.
    - quiz/gcp/spanner.py, save\_feedback(data) function
      * # TODO: Create a batch object for database operations
      * with database.batch() as batch:
      * # END TODO
    - Create a key for the feedback record from the email, quiz, and timestamp properties from the data. The reverse\_email(...) function takes the input email and creates a reversed string that is used as a primary key. For example, app.dev.student@example.com is converted to com\_example\_student\_dev\_app.
      * # TODO: Create a key for the record
      * # from the email, quiz and timestamp
      * feedback\_id = '{}\_{}\_{}'.format(
      * reverse\_email(data['email']),
      * data['quiz'],
      * data['timestamp'])
      * # END TODO
    - Use the batch object to insert a record, using a set of columns and values.
      * # TODO: Use the batch to insert a record
      * # into the feedback table
      * # This needs the columns and values
      * batch.insert(
      * table='feedback',
      * columns=(
      * 'feedbackId',
      * 'email',
      * 'quiz',
      * 'timestamp',
      * 'rating',
      * 'score',
      * 'feedback'
      * ),
      * values=[
      * (
      * feedback\_id,
      * data['email'],
      * data['quiz'],
      * data['timestamp'],
      * data['rating'],
      * data['score'],
      * data['feedback']
      * )
      * ]
      * )
      * # END TODO
    - Write code to use the Cloud Spanner functionality
    - In the .../quiz/console/worker.py file, load the spanner module.
    - quiz/console/worker.py
      * # TODO: Load the pubsub, languageapi and spanner modules
      * # from the quiz.gcp package
      * from quiz.gcp import pubsub, languageapi, spanner
      * # END TODO
    - After the existing code in the pubsub\_callback(message) function, save the feedback into Spanner.
    - quiz/console/worker.py, pubsub\_callback(message) function
      * # TODO: Use the spanner module to save the feedback
      * spanner.save\_feedback(data)
      * # END TODO
    - Log a message to the console to say that the feedback has been saved.
      * # TODO: Log a message to say the feedback
      * # has been saved
      * log.info('Feedback saved')
      * # END TODO
  + Run the web and worker application and test Cloud Spanner
  + See the spanner result also.

## 4.3 Adding Intelligence to Your Application

* Adding Intelligence 5 minutes - https://youtu.be/JItKZRMNJ-I
* Practice Quiz: Using Pub/Sub to Integrate Components of Your Application
  + Review the following diagram. Which of the following statements is accurate? Select three.
    - Diagram

      Description automatically generated
    - The Orders service publishes orders to the Orders topic.
    - Multiple instances of the Orders service are publishing unique orders.
    - Subscribers can process orders at a reasonable pace because the Orders topic acts as a buffer to hold data that is coming in rapidly.
      * Cloud Pub/Sub can reliably receive and store large amounts of rapidly incoming data. The downstream service can then act as a subscriber and consume this data at a reasonable pace.
  + Review the following diagram. Which of the following statements is accurate? Select two.
    - Timeline

      Description automatically generated
    - Each subscriber processes every order in the Orders topic.
    - The architecture solves the problem of point-to-point connections between applications.
      * The diagram shows a message fan out scenario. With Cloud Pub/Sub, your application can fan out messages from one publisher to multiple subscribers. Applications avoid making point-to-point connections with each other.
  + How does Cloud Pub/Sub enable you to build scalable and reliable applications? Select two.
    - Cloud Pub/Sub enables you to scalably and reliably ingest large volumes of data.
    - By using Cloud Pub/Sub, you can avoid brittle point-to-point connections between applications.
      * Cloud Pub/Sub enables you to design loosely coupled applications.

### Final Quiz: Using Pub/Sub to Integrate Components of Your Application

* + When a new employee joins your organization, HR needs to notify the security, facilities, and training teams so that those teams can perform their tasks related to new employees. You need to design an application architecture that notifies all teams promptly and reliably. Select the four steps that create the most effective design for this scenario.
    - Create a Cloud Pub/Sub topic called NewEmployee.
    - Create a separate subscription for security, facilities, and training.
    - Create SecuritySubscriber, FacilitiesSubscriber, and TrainingSubscriber services that subscribe to messages in the NewEmployee topic.
    - Create an HRPublisher service that publishes messages to the NewEmployee topic.
  + Sales data is published to a Cloud Pub/Sub topic called SalesTopic. The Finance application subscribes to the topic and begins receiving sales data messages. Later, the Inventory team creates another subscription to the topic to receive the Sales data as well. However, the Inventory team's data does not tally with the Finance team's data. What could the reason be?
    - The Inventory team's subscriber only receives messages that are published after the subscription was created.
  + Why is Cloud Pub/Sub used in the architecture of an application that performs real-time data analysis of Twitter data?
    - Diagram

      Description automatically generated
    - Cloud Pub/Sub is used to buffer the high volume of incoming tweets.

## 4.4 Using Cloud Functions for Event-Driven Processing

* Cloud Functions Concepts 5 minutes - https://youtu.be/QZIzlp\_k83U
* Writing, Deploying, and Monitoring Cloud Functions 2 minutes - https://youtu.be/FhDBz909H4w
* Demo: Invoke Cloud Functions through direct request-response 5 minutes - https://youtu.be/l5GmB7RgpAE

### App Dev - Processing Cloud Pub/Sub Data using Cloud Functions: Node.js 2 hours

* + Overview
    - In this lab, you enhance the online Quiz application by creating a Cloud Function to process Cloud Pub/Sub messages.
    - This process harnesses several GCP products in a serverless environment: Cloud Pub/Sub, Cloud Natural Language API, and Cloud Spanner.
  + Objectives
    - Create a Cloud Function that responds to Cloud Pub/Sub messages.
    - Deploy multiple files to a Cloud Function.
  + Preparing the case study application
    - git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - ln -s ~/training-data-analyst/courses/developingapps/v1.3/nodejs/cloudfunctions ~/cloudfunctions
    - cd ~/cloudfunctions/start
    - . prepare\_environment.sh
      * This script file:
        + Creates an App Engine application.
        + Exports the environment variables GCLOUD\_PROJECT and GCLOUD\_BUCKET.
        + Runs npm install.
        + Creates entities in Cloud Datastore.
        + Creates a Cloud Pub/Sub topic.
        + Creates a Cloud Spanner Instance, Database, and Table.
        + Prints out the Google Cloud Platform Project ID.
    - To run the web application, enter the following command:
      * npm start
  + Working with Cloud Functions
    - In this section, you create a Cloud Function in your Google Cloud Platform project that is triggered by publishing a message to Cloud Pub/Sub, runs the application, and monitors the Cloud Function invocation.
    - Create a Cloud Function
      * Navigation > Cloud Functions > If necessary, click Enable API > click Create function.

|  |  |
| --- | --- |
| **Property** | **Value** |
| **Function name** | **process-feedback** |
| **Trigger type** | **Cloud Pub/Sub** |
| **Topic** | Select **projects/[PROJECT\_ID]/topics/feedback** |

* + - * Click Save, and then click Next.
      * Review the provided function implementation, and then click Deploy.
    - Run the web application, Take test and send feedback
    - View Cloud Function monitoring and logs
      * Cloud Functions page > Click process-feedback > Click Logs
      * Go to Cloud Shell to also see your feedback.
  + Examining the case study application code
    - Navigate to cloudfunctions/start.
    - Select the index.js file in the ...function folder.
      * This file contains the same code as the sample from the Cloud Functions window in the Cloud Platform Console, with one change: because the function you will write returns a Promise, the callback argument has been omitted.
    - Select the package.json file.
      * This file contains the list of dependencies that this function needs to run.
      * Cloud Functions automatically install the dependencies.
    - Select the languageapi.js file.
      * This file contains code to process feedback text and return the Natural Language ML API sentiment score.
    - Select the spanner.js file.
      * This file contains code to insert a record into a Cloud Spanner database.
  + Coding a Cloud Function
    - Write code to modify a Cloud Function
      * Return to the ...function/index.js file.
      * Load the languageapi and spanner modules. These modules are in the same folder as the index.js file.
      * In the subscribe() method, after the existing code that loads the Cloud Pub/Sub message into a buffer, convert the PubSub message into a feedback object by parsing it as JSON data.
      * Return a promise that invokes the languageapi module's analyze method to analyze the feedback text.
      * Chain a .then(...) method to the end of the return statement.
      * Supply an arrow function as the value of the callback.
      * In the body of the arrow function, log the Natural Language API sentiment score to the console.
      * Add a new property called score to the feedback object.
      * Complete the arrow function body by returning the feedback object.
      * Chain a second .then(...) method to the end of the first one. This one uses the spanner module to save the feedback.
      * Write a third .then(...) chained method, including an arrow function with no arguments and an empty body as the value of the callback.
      * In the body of this callback, log a message to indicate that the feedback has been saved, and return a success message.
      * Attach a .catch(...) handler to the end of the chain, which logs the error message to the console.
    - Here's how your function should look when you're done.
    - function/index.js
      * // TODO: Load the ./languageapi module
      * const languageAPI = require('./languageapi');
      * // END TODO
      * // TODO: Load the ./spanner module
      * const feedbackStorage = require('./spanner');
      * // END TODO
      * exports.subscribe = function subscribe(event) {
      * // The Cloud Pub/Sub Message object.
      * // TODO: Decode the Cloud Pub/Sub message
      * // extracting the feedbackObject data
      * // The message received from Pub/Sub is base64 encoded, and
      * // the data submitted by students is in a data property
      * const pubsubMessage = Buffer.from(event.data, 'base64').toString();
      * let feedbackObject = JSON.parse(pubsubMessage);
      * console.log('Feedback object data before Language API:' + JSON.stringify(feedbackObject));
      * // END TODO
      * // TODO: Run Natural Language API sentiment analysis
      * // The analyze(...) method expects to be passed the
      * // feedback text from the feedbackObject as an argument,
      * // and returns a Promise.
      * return languageAPI.analyze(feedbackObject.feedback).then(score => {
      * // TODO: Log the sentiment score
      * console.log(`Score: ${score}`);
      * // END TODO
      * // TODO: Add new score property to feedbackObject
      * feedbackObject.score = score;
      * // END TODO
      * // TODO: Pass feedback object to the next handler
      * return feedbackObject;
      * // END TODO
      * })
      * // TODO: insert record
      * .then(feedbackStorage.saveFeedback).then(() => {
      * // TODO: Log and return success
      * console.log('feedback saved...');
      * return 'success';
      * // END TODO
      * })
      * // END TODO
      * // TODO: Catch and Log error
      * .catch(console.error);
      * // End TODO
      * };
    - Package and deploy the Cloud Function code
  + Return to Cloud Shell and stop the web application by pressing Ctrl+C.
    - To change the working directory to the Cloud Function code, enter the following command:
      * cd function
    - To zip up the files needed to deploy the function, enter the following command:
      * zip cf.zip \*.js\*
      * This generates a zip archive named cf.zip that includes all the JavaScript and JSON files in the folder.
    - To stage the zip file into Cloud Storage, enter the following command:
      * gsutil cp cf.zip gs://$GCLOUD\_BUCKET/
      * This copies the zip archive into the Cloud Storage bucket named after your project ID with a -media suffix.
    - Cloud Functions > Select the process-feedback function > Click Edit > Click Next.
      * Under Source code, select ZIP from Cloud Storage.
      * For Cloud Storage location, Click Browse, select the cf.zip file in the bucket named after your GCP project ID with the -media suffix, and click Select.
      * In the Entry point field, type subscribe. Click Deploy.
  + Testing the case study application
    - Run the web application
      * Change the working folder back to the start folder for the cloudfunctions lab.

cd ..

* + - * To start the web application, execute the following command:
        + npm start
      * Preview the web application, take test and send feedback
    - View Cloud Function monitoring and logs
      * Cloud Functions > click process-feedback > click logs
    - View Cloud Spanner data
      * Navigation > Spanner > Click Quiz instance > click quiz-database > select Query.
        + SELECT \* FROM Feedback
  + Bonus: Storing student answers using a Cloud Function
    - When a student completes a quiz, their answers are submitted in an API call back to the server. Your job is to capture the student-submitted answers and the correct answers and save them into Cloud Spanner. You should see that a new record has been added to the Feedback table.
    - To do this:
      * Create a Cloud Pub/Sub topic called answers.
      * Create a Cloud Spanner table called Answers with appropriate column names and data types.
      * Post the answer data to the answers topic.
      * Subscribe to the answers topic in the console application and insert the answer data into the Answers table.
      * The details are left up to you! You can find the solution to the bonus in the lab's bonus folder.
  + Review
    - Which triggers can be used with Cloud Functions?
      * Cloud Pub/Sub
      * Cloud Spanner
      * Cloud Storage
      * HTTP
    - With a Cloud Function triggered by Cloud Pub/Sub, how is the message delivered?
      * Base64 encoded
      * CSV encoded
      * Tarred and Gzipped
      * Zipped
    - What is the maximum execution time for a Cloud Function?
      * 60 seconds
      * 540 seconds
      * Unlimited

### Practice Quiz: Adding Intelligence to Your Application

* + What API would you use in an Expense Report application to extract text from images of receipts?
    - Cloud Vision API
      * Cloud Vision API can perform optical character recognition (OCR).
  + What API would you use to transcribe audio into text?
    - Cloud Speech API
      * Cloud Speech API enables developers to convert audio to text. It handles over 100 languages and variants to support your global user base.
  + Which of the following statements about the Vision API are accurate? Select two.
    - The Vision API can detect landmarks, logos, faces, and explicit content.
    - The Vision API can categorize objects under labels and perform optical character recognition (OCR).

### Final Quiz: Adding Intelligence to Your Application

* + Your support database containsfeedbackdata from customers. You want to analyze customer sentiment for the last quarter. What property and pre-trained machine learning API can you use to gauge customer sentiment?
    - Sentiment score and magnitude from Cloud Natural Language API
  + Identify two ways to invoke the pre-trained machine learning APIs such as the Vision API or Natural Language API in your application?
    - Use the REST API.
    - Use Cloud Client Libraries when available for production use.
  + What pre-trained machine learning APIs would you use in this image processing pipeline?
    - * Graphical user interface, application

        Description automatically generated
    - Cloud Vision API, Cloud Speech API, Cloud Video Intelligence API

## 4.5 Managing APIs with Cloud Endpoints

* Cloud Endpoints Concepts 4 minutes - https://youtu.be/rRWOroprRzc
* Cloud Endpoints for REST APIs 11 minutes - https://youtu.be/P3-ibPNfF3c

### App Dev - Deploying an API for the Quiz Application: Node.js 2 hours

* + Overview
    - In this lab, you deploy the Quiz application API into Google Compute Engine and leverage Cloud Endpoints to provide monitoring functionality.
  + Objectives
    - Create an Open API specification from the existing Quiz application REST API.
    - Deploy the specification as a Cloud Endpoint.
    - Provision a Compute Engine instance with the Extensible Service Proxy to host the Cloud Endpoints API.
  + Preparing the Case Study Application
    - git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
    - ln -s ~/training-data-analyst/courses/developingapps/v1.3/nodejs/cloudendpoints ~/cloudendpoints
    - cd ~/cloudendpoints/start
    - . prepare\_environment.sh
      * This script file
        + Creates an App Engine application.
        + Creates the Datastore database.
        + Exports environment variables GCLOUD\_PROJECT and GCLOUD\_BUCKET.
        + Runs npm install.
        + Creates entities in Cloud Datastore.
        + Installs an Open API generator tool, api2swagger.
        + Prints out the Google Cloud Platform Project ID.
    - npm start
    - In Cloud Shell, click Web preview > Preview on port 8080 to preview the quiz application.
      * Modify the URL by adding /api/quizzes/places to the end of the hostname.
      * Replace ?authuser=0 with /api/quizzes/places.
        + You should see JSON data returned from the quiz application API corresponding to the questions in the places quiz.
      * Make a note of the complete API URL to use later in this lab.
      * Save the url so you can paste it in a command in the next section.
        + The URL will look similar to: https://8080-cs-\*\*\*.cloudshell.dev/api/quizzes/places
  + Generating an Open API Specification
    - In this section, you will generate an Open API specification that will form the basis for deploying your Cloud Endpoint.
    - Create an Open API Specification
    - Open a second Cloud Shell window.
      * cd ~/cloudendpoints/start/endpoint
    - Update the PATH environment variable to set the path for the api2swagger command:
      * export PATH=$PATH:`npm root -g`/api2swagger/bin
    - To generate the Open API specification, execute the following command:
      * api2swagger -e https://8080-5da1bb2f-9b67-453d-86ba-d1ea76ff1ddb.ql-europe-west1-chpf.cloudshell.dev/api/quizzes/places -o ./quiz-api.json
        + Replace [API\_URL\_FROM\_CLIPBOARD] with the API URL saved in the previous step of form https://8080-cs-.../api/quizzes/places.
      * The api2swagger tool will make the request against the API and display a series of prompts.
      * If you don't complete the proper replacement, you'll see an Invalid hostname error.
    - Enter responses to the api2swagger prompts using the following table:

|  |  |
| --- | --- |
| **Prompt** | **Response** |
|  |  |
| Title of Swagger Spec ? | Quite Interesting Quiz API |
| Description of Swagger Spec ? | An API for the Quite Interesting Quiz |
| Terms of Service URL | (Press enter key) |
| Version of your API Program ? | (Press enter key) |
| Contact Name? | (Press enter key) |
| Contact URL ? | (Press enter key) |
| Contact Email ? | (Press enter key) |
| License Name ? | (Press enter key) |
| License URL ? | (Press enter key) |
| Does your API support http ? | Yes |
| Pick Base Path from your API ? | Use the up and down arrows to select:  **❯ /api/quizzes**  (Press enter key) |
| A verbose explanation of the operation behavior ? | Gets questions for a quiz |
| A short summary of what the operation does ? | Get quiz questions |
| Additional external documentation for this operation ? | (Press enter key) |
| Unique string used to identify the operation ? | getQuizQuestions |
| A list of tags for API documentation control ? | (Press enter key) |
| API Path has any dynamic parameters ? | Y (Press enter key) |
| Choose Dynamic Params in URL ? | Use the up and down arrows to move to:**❯◉ places**  Then **press the space bar to select places**, then press enter |
| Name of URL Param ? | quizName |
| Description of URL Param ? | The name of the quiz |
| Type of query param ? | Use the up and down arrows to select:**❯ string**  (Press enter key) |

* + Navigate to cloudendpoints/start/endpoint/quiz-api.json.
    - Replace the value for the "host" key with a hostname using a string in the form quiz-api.endpoints.[Project-ID].cloud.goog.
      * endpoint/quiz-api.json
      * {
      * "swagger": "2.0",
      * "host": "quiz-api.endpoints.qwiklabs-gcp-XX-XXXX.cloud.goog",
      * "schemes": [
      * "https",
      * "http"
      * ],
  + Deploy the API Specification to Cloud Endpoints
    - To deploy the Open API specification as a Cloud Endpoint, execute the following command in the Cloud Shell window:
      * gcloud endpoints services deploy quiz-api.json
    - If you see an error like PERMISSION\_DENIED: Ownership for domain name, make sure your host in the quiz-api.json file matches the suggested format including the correct your project-id.
    - To view the Open API configuration name, execute the following command:
      * gcloud endpoints configs list --service="quiz-api.endpoints.$GOOGLE\_CLOUD\_PROJECT.cloud.goog"
    - Notice the project-id has been inserted. You can copy the service from the output of your deploy command.
    - The service configuration will be returned, including the CONFIG\_ID and SERVICE\_NAME.
    - You will need the both values in the next section.
  + Deploying the API Backend
    - In this section, you will provision a Compute Engine instance to run the API implementation and the Cloud Endpoints Extensible Service Proxy.
    - Create a Compute Engine Instance
      * Navigation > Compute Engine > click Create Instance.

|  |  |
| --- | --- |
| **Property** | **Value** |
| Name | endpoint-host |
| Region | us-central1 |
| Zone | us-central1-a |
| Boot disk | Make sure **Debian GNU/Linux 10 (buster)** is selected |
| Identity & API access > Access scopes | **Allow full access to all Cloud APIs** |
| Firewall | **Allow HTTP traffic** |
| Networking, Disks, Security, Management, Sole-Tenancy > Management > Metadata > Add item | Add the following keys and values |
| **Metadata Key** | **Value** |
| endpoints-service-config-id | The CONFIG\_ID for the endpoint in the form:yyyy-mm-ddr0 |
| endpoints-service-name | The SERVICE\_NAME for the endpoint in the form:  quiz-api.endpoints.<Project-ID>.cloud.goog |

* + - * Click Create.
  + Install and run the API Backend
    - Return to the Cloud Shell window.
      * cd ~/cloudendpoints/start/endpoint/
    - To copy the application source files from Cloud Shell to endpoint-host:
      * gcloud compute scp ./quiz-api endpoint-host:~/ --recurse --zone=us-central1-a
    - You may be prompted to create an SSH key. You can press the ENTER key for each prompt including the Y to create the .ssh directory.
    - Return to the Cloud Platform Console.
    - On the Compute Engine page, to connect to the endpoint-host virtual machine, click SSH.
    - In the endpoint-host SSH window, install the software prerequisites:
      * curl -sL https://deb.nodesource.com/setup\_14.x | sudo -E bash -
      * sudo apt-get install -y nodejs
      * curl -L https://npmjs.org/install.sh | sudo sh
    - To run the quiz API application, execute the following commands within the SSH session:
      * export GCLOUD\_PROJECT="$(curl -H Metadata-Flavor:Google http://metadata/computeMetadata/v1/project/project-id)"
      * export GCLOUD\_BUCKET=$GCLOUD\_PROJECT-media
      * export PORT=8081
      * cd ~/quiz-api
      * npm install
      * npm start
    - Return to the Cloud Platform Console, which should still display the Compute Engine VM instance list.
    - Establish a second SSH connection to endpoint-host.
    - In the second endpoint-host SSH window, to install the Cloud Endpoints Extensible Service Proxy by executing the following commands:
      * export CLOUD\_ENDPOINTS\_REPO="google-cloud-endpoints-jessie"
      * echo "deb http://packages.cloud.google.com/apt $CLOUD\_ENDPOINTS\_REPO main" | sudo tee /etc/apt/sources.list.d/google-cloud-endpoints.list
      * curl --silent https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
      * sudo apt-get update && sudo apt-get install google-cloud-sdk && sudo apt-get install endpoints-runtime
    - Still in the second endpoint-host SSH window, execute the following command to edit the nginx configuration file:
      * sudo nano /etc/default/nginx
    - To modify the file, add the following declaration after the existing configuration:
      * PORT=80
    - Save the file (press Ctrl+O then Enter) and exit nano (press Ctrl+X).
    - To restart nginx, execute the following command:
      * sudo service nginx restart
  + Testing and Modifying Cloud Endpoint
    - In this section, you will verify the deployment of the endpoint and modify the configuration to allow serving via a hostname instead of an IP address.
    - Invoke the Cloud Endpoint via the IP address
      * Return to the Cloud Platform Console > Compute Engine.
      * To launch a browser window, click on the endpoint-host virtual machine's External IP address.
        + A JSON message reporting Method does not exist will be displayed in the new browser window. This is expected behavior, because you have not associated a method with "/". You do not need to take corrective action.
      * Add /api/quizzes/places to the end of the URL.
        + You should see JSON data returned from the quiz API.
      * Navigation menu, click Endpoints, and then click on your deployed endpoint.
        + You should see a monitoring page for the quiz-api endpoint.
        + After a few minutes, you should see the request that you made against the endpoint.
    - Modify and redeploy the Cloud Endpoint configuration
      * Return to the Cloud Shell code editor, and select the quiz-api.json file.
      * To allow the API to be accessed via the Cloud Endpoint hostname instead of the IP address, add the "x-google-endpoints" key and value shown in the following file fragment:
      * endpoint/quiz-api.json
        + {
        + "swagger": "2.0",
        + "host": "quiz-api.endpoints.<Project-ID>.cloud.goog",
        + "x-google-endpoints": [ {
        + "name": "quiz-api.endpoints.<Project-ID>.cloud.goog",
        + "target": "<endpoint-host-EXTERNAL-IP-ADDRESS>" } ],
        + "schemes": [
        + "https",
        + "http"
        + ],
      * Change the values of <Project-ID>, and <endpoint-host-EXTERNAL-IP-ADDRESS> then save the file.
      * To redeploy the API, return to the Cloud Shell window, and execute the following command:
        + gcloud endpoints services deploy quiz-api.json
      * Open a new browser tab, and navigate to: http://quiz-api.endpoints.<Project-ID>.cloud.goog/api/quizzes/gcp
        + You should see JSON data from the GCP quiz.
  + Monitoring the API
    - In this section you will review the monitoring output from your API.
    - Inspect the API in the Console
      * Return to the Cloud Platform Console, and on the Navigation menu, click Endpoints, and then click on your deployed endpoint.
        + You should see an overview page for the Quiz API.
    - Take a few minutes to review the Requests, Latency, and Error graphs.
      * You should see that the monitoring graphs allow you to observe the performance of your API as it executes requests from clients.

### Practice Quiz: Using Cloud Functions for Event-Driven Processing

* + Cloud Functions is ideal for which of the following use cases?
    - Light-weight, event-driven, serverless, microservices
      * Cloud Functions are ideal for microservices that require a small piece of code to quickly process data related to an event.
  + You can view the output from your console.log and console.error messages in which of the following services?
    - CloudLogging
      * You can view the output from your console.log and console.error messages inCloudLogging.
  + Identify two ways of invoking Cloud Functions.
    - Invoke asynchronously when triggered by an event.
    - Invoke synchronously as a web hook.
      * External systems can synchronously invoke functions as web hooks in response to events in those systems. Cloud Functions can also be triggered asynchronously in response to events inGoogle Cloudservices.

## 5. App Deployment, Debugging, and Performance

* application developers learn how to design and develop cloud-native applications that seamlessly integrate components from the Google Cloud ecosystem. learn how to create repeatable deployments by treating infrastructure as code, choose the appropriate application execution environment for an application, and monitor application performance.

## 5.1. Deploying Applications

* Deploying Applications Using Cloud Build, Container Registry, and Terraform 4 minutes
* Why use containers for deployment? 3 minutes
* Build Pipelines with Cloud Build and Terraform 8 minutes
* App Dev - Deploying the Application into Kubernetes Engine: Node.js 2 hours
* App Dev - Deploying the Application into Kubernetes Engine: Java 2 hours
* App Dev - Deploying the Application into Kubernetes Engine: Python 2 hours
* Practice Quiz: Deploying Applications
* Final Quiz: Deploying Applications

## 5.2. Execution Environments for Your Application

* Introduction to Execution Environments in Google Cloud 1 minute
* Dataflow 3 minutes
* Cloud Functions 2 minutes
* App Engine Flexible Environment 5 minutes
* Google Kubernetes Engine 7 minutes
* Cloud Run 2 minutes
* Compute Engine 3 minutes
* App Dev - Deploying the Application into App Engine Flexible Environment: Node.js 2 hours
* App Dev - Deploying the Application into App Engine Flexible Environment: Java 2 hours
* App Dev - Deploying the Application into App Engine Flexible Environment: Python 2 hours
* Practice Quiz: Execution Environments for Your Application
* Final Quiz: Execution Environments for Your Application

## 5.3. Debugging, Monitoring, and Performance Tuning

* Google Cloud's Operations Suite - a Multi-Cloud Service 1 minute
* Debugging Your Application 10 minutes
* App Dev - Debugging Application Errors: Node.js 2 hours
* Logging 1 minute
* Monitoring and Tuning Performance 4 minutes
* Identifying and Troubleshooting Performance Issues 5 minutes
* Operations Features Demo 5 minutes
* App Dev - Harnessing Cloud Trace and Cloud Monitoring: Node.js 2 hours
* Practice Quiz: Debugging, Monitoring, and Performance Tuning
* Final Quiz: Debugging, Monitoring, and Performance Tuning