```
import * as functions from "firebase-functions";
import admin from "firebase-admin";
const db = admin.firestore();

import { BigQuery } from "@google-cloud/bigquery";
const bigquery = new BigQuery();

import { DataTable, ActivityData, TrainerData } from "./interfaces";

/**
 * Guidelines
 * ===
 *  1. Only the necessary libraries;
 *  2. Catch and log all exceptions properly, response with
appropriated error code;
 *  3. Clean self-descriptive code;
 *  4. splitting the application into logic files
 */

//
==========================================================================
================

/**
 * Start new training session
 * ---
 * Prepare the training session by setting the state in the latest (if
exists) DataTable document
 * to closed and create a DataTable document based on the DataTable
interface where activityType
 * is set to 'training'.
 *
 * @memberOf Training
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 * @property {'post'} method - HTTP method
 */
export const startTraining = functions.https.onRequest((request,
response) => {
  const trainingKey = request.get("Training-Key");

  /*
```

```
   *  1. Check if training key in request **header** exist in a Device
document in Firestore. If not
   *     reponse with failure 401 {success: false,  message:
"Unauthorized, check training key"}
   *  2. Close previous DataTable document if exist by setting the
activityTableState value to 'close'
   *  3. Create a new DataTable document and set activityType to
'training'
   *  4. Response with 200:
   *     {success: true,  message: "Successfully created Training
session", deviceId: "abcd1234"}
   */

  let dataTable: DataTable = {
    organisationId: "[from device document]",
    deviceId: "[from device document]",
    workerId: "[from device document]",
    modelId: "[from device document]",
    creationTimestamp: new Date(),
    activityType: "training",
    dataSet: "[from device document => organisationId]",
    activityTable: `activity_{deviceId}_{timestamp}`,
    activityState: "waiting",
    activityRecordTimestamp: null,
    trainingTable: `training_{deviceId}_{timestamp}`,
    trainingState: "waiting",
    trainingRecordTimestamp: null,
    labeledTable: `labelled_{deviceId}_{timestamp}`,
    labeledTableState: "waiting",
    outputTable: "output",
    outputTableState: "na",
    notes: "",
  };

  db.collection("")
    .add({
      dataTable,
    })
    .then(() =>
      response.status(200).json({
        success: true,
        message: "Successfully created Training session",
        deviceId: dataTable.deviceId,
```

```
      })
    );
});

/**
 * Sync training session
 * ---
 * On both devices the sync button will be clicked simultaneously. Both
exact timestamps will be stored
 * in the Training DataTable document. The difference between both
timestamps will be the input for
 * timestamp correction.
 *
 * @memberOf Training
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 * @property {'post'} method - HTTP method
 */
export const syncTraining = functions.https.onRequest((request,
response) => {
  const deviceId = request.get("Device-ID");

  /*
   *  1. Get the latest DataTable document where the device id matches
with the device id in the
   *     request **header** and the activityType value is 'training'.
If no match reponse with
   *     failure 401:
   *     {success: false, message: "Unauthorized, check device id"}
   *  2. Chech the device type in the **header** (Device-Type: 'watch'
| 'phone'). If not valid reponse with
   *     failure 406:
   *     {success: false, message: "Device type not valid"}
   *  3. Update DataTable document:
   *      - when device type is equal to 'watch' update the
'activityRecordTimestamp' timestamp with
   *        the request **header** timestamp (Device-Timestamp).
   *      - when device type is equal to 'phone' update the
'trainingRecordTimestamp' timestamp with
   *        the request **header** timestamp (Device-Timestamp).
   *  3. Response with 200:
   *     {success: true,  message: "Successfully updated
synchronization timestamp"}
```

```
    */

  response.status(200).json({
    success: true,
    message: "Successfully updated synchronization timestamp",
  });
});

/**
 * End training session
 * ---
 * Ends the training session when done by setting the
[activityTableState] to 'done'.
 *
 * @memberOf Training
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 * @property {'post'} method - HTTP method
 */
export const endTraining = functions.https.onRequest((request,
response) => {
  const deviceId = request.get("Device-ID");
  /*
   *  1. Get the latest DataTable document where the device id matches
with the device id in the
   *     request **header** and the activityType value is 'training'.
If no match reponse with
   *     failure 401:
   *     {success: false, message: "Unauthorized, check device id"}
   *  2. Set the value of activityState and trainingState on 'closed'
   *  3. Response with 200:
   *     {success: true,  message: "Successfully closed training
session"}
   */

  response.status(200).json({
    success: true,
    message: "Successfully closed training session",
  });
});

/**
 * Activity recording
```

```
 * ---
 * Activity data handling. The functions checks if the latest DataTable
document of this device
 * is set to the activity training. If so, the data is stored directly
in BigQuery. If not, the
 * function will check the creation data of the latest DataTable
document. When older than 15 min,
 * close the current DataTable, create a new DataTable document and
store the data to BigQuery.
 *
 * @memberOf Recording
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 * @property {'post'} method - HTTP method
 */
export const activityRecording = functions.https.onRequest(
  (request, response) => {
    const deviceId = request.get("Device-ID");
    const body: ActivityData[] = JSON.parse(request.body);

    /*
     *  1. See visual: Handel data.png
     */

    const timestamp = new Date();
    let newDataTable: DataTable = {
      organisationId: "[from device document]",
      deviceId: "[from device document]",
      workerId: "[from device document]",
      modelId: "[from device document]",
      creationTimestamp: timestamp,
      activityType: "regular",
      dataSet: "[from device document => organisationId]",
      activityTable: `activity_{deviceId}_${timestamp.getTime()}`,
      activityState: "waiting",
      activityRecordTimestamp: null,
      trainingTable: `training_{deviceId}_${timestamp.getTime()}`,
      trainingState: "waiting",
      trainingRecordTimestamp: null,
      labeledTable: `labeled_{deviceId}_${timestamp.getTime()}`,
      labeledTableState: "waiting",
      outputTable: "output",
      outputTableState: "na",
```

```
      notes: "",
    };

    response.status(200).json({
      success: true,
      message: "Successfully stored the activity recording",
    });
  }
);

/**
 * Training recording
 * ---
 * Store training data to BigQuery table.
 *
 * @memberOf Recording
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 * @property {'post'} method - HTTP method
 */
export const trainingRecording = functions.https.onRequest(
  (request, response) => {
    const deviceID: string = request.get("Device-ID");
    const body: TrainerData[] = JSON.parse(request.body);

    /*
     *  1. Get the previous DataTable document where the device id
matches with the device id in the
     *     request **header** and the activityType value is 'training'.
If no match reponse with
     *     failure 401:
     *     {success: false, message: "Unauthorized, check device id"}
     *  2. Create Dataset when not existing (dataset name:
{OrganisationId}).
     *  3. Create Table when not existing (table name:
activity_{deviceId}_{startTimeStamp}, startTimeStamp in UTC
milliseconds).
     *  4. Store the data from the json-payload into the table.
     *  5. Response with 200:
     *     {success: true,  message: "Successfully stored the training
recording"}
     */
```

```
    response.status(200).json({
      success: true,
      message: "Successfully stored the training recording",
    });
  }
);

/**
 * Training recording List
 * ---
 * Return all training recordings.
 *
 * @memberOf Recording
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 * @property {'post'} method - HTTP method
 */
export const trainingList = functions.https.onRequest((request,
response) => {
  const key = request.get("Key");

  /*
   *  1. Check if the user id in request **header** exist as an user in
the user
   *     collection and check the userType of the concerning user is
set to 'dev'.
   *     If not reponse with failure 401:
   *     {success: false, message: "Unauthorized, check user id"}
   *  2. Get all the documents where the activityType is set to
'training'.
   *  3. Response with 200:
   *     {success: true,  message: "Successfully stored the training
recording",
   *     data: [ ...DataTable documents... ],}
   */

  response.status(200).json({
    success: true,
    message: "Successfully collected all the training recordings",
    data: [
      // DataTable documents
    ],
  });
```

```
});

/**
 * labeled recording data
 * ---
 * Return all training recordings.
 *
 * @memberOf Recording
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 * @property {'post'} method - HTTP method
 */
export const labelledRecording = functions.https.onRequest(
  (request, response) => {
    const key = request.get("Key");
    const dataTableID = request.get("DataTable-ID");
    const page = request.get("Page");

    /*
     *  1. Check if the user id in request **header** exist as an user
in the user
     *     collection and check the userType of the concerning user is
set to 'dev'.
     *     If not reponse with failure 401:
     *     {success: false, message: "Unauthorized, check user id"}
     *  2. Get the DataTable document with the matching datatable Id
from the **header**
     *     (DataTable-ID).
     *  3. Get the labeled table from BigQuery order by the timestamp
and limited to
     *     10.000 records.
     *     Maximum size below the 10MB => 10.000 records.
     *  4. Convert to csv format.
     *  4. Response with 200: complete csv
     */

    response
      .status(200)
      .header({
        Page: page,
        Pages: "5",
      })
      .send
```

```
      // CSV data
      ();
  }
);

/**
 * Generate labeled table
 * ---
 * When DataTable document is updated and both the activity state and
training state are
 * set to 'closed' the data of both tables will be joined. In this
query the timestamp of
 * the training data will be auto corrected based on the record
timestamp difference between
 * both devices.
 *
 * @memberOf Recording
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 */
export const labelingRecording = functions.firestore
  .document("DataTable/{deviceId}")
  .onUpdate((change, context) => {
    /*
     * 1. check if the activityTableState and the trainingTableState is
set to 'closed' and
     *    labeledTableState is not yet set to 'closed'.
     * 2. Construct query and execute the query abd dictly save the
table to the destination
     *    table.
     * 3. Set the labeledTableState value to 'closed' when successful
execution
     */

    const newValue = change.after.data();

    if (
      newValue.activityTableState === "closed" &&
      newValue.trainingTableState === "closed" &&
      newValue.labeledTableState !== "closed"
    ) {
      const deviceId = newValue.deviceId;
      const timestamp = new Date().getTime();
```

```
      const destination = {
        projectId: process.env.GCLOUD_PROJECT,
        datasetId: newValue.dataSet,
        tableId: `labelled_${deviceId}_${timestamp}`,
      };
      const activityDatasetName = "";
      const activityTableName = "";
      const trainingDatasetName = "";
      const trainingTableName = "";
      const activityTrainingOffset = ""; // is in hours should be
milliseconds (maximum milliseconds to add or extract is 12 hours)

      const sqlQuery = `SELECT a.*, t.*  EXCEPT (added, timestamp,
timestamp_end, userId)
      FROM \`growatch.@activityDatasetName.@activityTableName\` as a
      LEFT JOIN
      (SELECT *, row_number() over (order by timestamp) as label_row
      FROM \`growatch.@trainingDatasetName.@trainingTableName\`) AS t
      ON
      t.userId = a.userId AND
      a.timestamp BETWEEN TIMESTAMP_ADD(TIMESTAMP_ADD(t.timestamp,
INTERVAL @activityTrainingOffset HOUR), INTERVAL -12 MILLISECOND) AND
      TIMESTAMP_ADD(TIMESTAMP_ADD(t.timestamp_end, INTERVAL
@activityTrainingOffset HOUR), INTERVAL +12 MILLISECOND)
      ORDER BY a.timestamp, elapsed_realtime`;

      const options = {
        query: sqlQuery,
        // Location must match that of the dataset(s) referenced in the
query.
        location: "US",
        params: {
          activityDatasetName,
          activityTableName,
          trainingDatasetName,
          trainingTableName,
          activityTrainingOffset,
        },
        destination,
      };

      return bigquery
        .createQueryJob(options)
```

```
        .then(([job]) => change.after.ref({ labeledTableState: "closed"
}))
        .catch((error) => console.log(error));
    }
  });

/**
 * Generate activation key
 * ---
 * Autogeneration of a device activation key when new device is created
 *
 * @memberOf Device
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 */
export const activationKeyDevice = functions.firestore
  .document("Devices/{diviceId}")
  .onCreate((snap, context) => {
    const chars = "abcdefghijklmnopqrstuvwxyz0123456789";
    const charsLength = chars.length;

    let newKey = (length) => {
      let result = "";
      for (var i = 0; i < length; i++) {
        result += chars.charAt(Math.floor(Math.random() *
charsLength));
      }
      return result;
    };
    return snap.ref.set(
      { activationKey: newKey(4), trainingKey: newKey(4) },
      { merge: true }
    );
  });

/**
 * Validate device key
 * ---
 * Validate if the device key is matching with a device key in a device
document.
 * The matching id is returned back once.
 *
 * @memberOf Device
```

```
 * @param {object} request - HTTP request object
 * @param {object} response - HTTP response object
 */
export const validateKeyDevice = functions.https.onRequest(
  (request, response) => {
    const key = request.get("Key");

    /*
     * 1. Get the Device document where the activationKey is the same as the request.
     *    If there is no matching activationKey response with an error 401:
     *    {success: false,  message: "No matching activation key"}
     * 2. Set the activationKey value to null to avoid double registration
     * 3. Response with 200:
     *    {success: true, message: "Successfully found the device key, 'id' is the
     *    matching device id", id: 'x'}
     */

    const id = "x";

    response.status(200).json({
      success: true,
      message:
        "Successfully found the device key, 'id' is the matching device id",
      id,
    });
  }
);
```