Context

An integration engineer's responsibilities include processing third-party files that come in various formats, and turning them into clean data digestible by in-house tools.

For the sake of this assignment we'll assume our destination is an endpoint in a hypothetical in-house REST API. The endpoint documentation is found below.

API Documentation

Request URI: `POST /` [redacted] *(don't worry about it for now)*

Request sample:

```
{
  "uuid": "52ab9466-5a2b-4d93-9425-62171387c2a7",
  "fname": "file_name.txt",
  "errors": [
    "This is a sample error text!"
  ],
  "payload": [
    {
      "employeeCode": "000000aa",
      "action": "hire",
      "data": {
        "first_entity.first_field": "string value",
        "first_entity.second_field": 0.0,
        "second_entity.third_field": "string value"
      },
      "payComponents": [
        {
          "amount": 1000,
          "currency": "USD",
          "startDate": "2019-01-01",
          "endDate": "2019-01-01"
        }
      ]
    }
  ]
}
```

Request format:

Parameter	Required	Type	Description
uuid	yes	string	Unique identificator, must be in UUID format.
fname	yes	string	File name
errors	no	array of strings (:error)	
:error	no	string	An error text
payload	yes	array of :payloadElement	
:payloadElement	yes	object	An object that represents all the information about one person.
:payloadElement.employeeCode	yes	string	A unique identifier assigned to the person by their employer. Basically Employee ID in an organization.
:payloadElement.action	yes	string Allowed values: hire, change, terminate	A fixed value code that indicates what type of action is to be performed with this person's data: 1) hire a new employee 2) change existing employee's data 3) terminate an existing employee

Parameter	Required	Type	Description
:payloadElement.data	yes	object	Map of entity fields and their values to be persisted.
data::key	*yes, at least one*	string	The keys represent entities and their respective fields where the information will go.
data::key→value	yes	string, integer, decimal or boolean	Values can be of any allowed type and show what the given field should be filled with. Correctness is assumed and any unfit data results in process interruption.
:payloadElement.payComponents	no	array of :payComponent	
:payComponent	no	object	An object that represents a component of an employee's salary.
:payComponent.amount	yes	decimal	The amount being paid
:payComponent.currency	yes	string	The currency of payment
:payComponent.startDate	yes	string, format "yyyy-MM-dd"	Start of the period for which a given pay component is paid.
:payComponent.endDate	yes	string, format "yyyy-MM-dd"	End of the period for which a given pay component is paid.

Business rules and compromises

The hypothetical client that provided us with the input file (CSV) has specified some business rules as well as agreed to several conditions from our side. These agreements further specify how our integration team should process incoming data and what to do if any information comes in differently than expected. These are as follows:

1. All dates and timestamps are to be standardized by the integration team. The input format can be whatever, as long as it's consistent across all fields. Any inconsistent data can be discarded.

2. Our (in-house) software system has certain fields that are mandatory. The client has agreed to provide all such values and if they miss any by accident, their entire data entry (i.e. a row in a CSV file) will be discarded by us.

3. Our integration allows clients to hire new people, update their information (including adding new salary items for new time periods) and terminate existing employees. This is all specified by *action codes* that on our side are "hire", "change" and "terminate".
    - The client may send their own interpretation of actions, such as "add", "update" and "delete". Integration will map them accordingly, as long as the custom codes are obvious enough to match ours 1:1.
    - The custom codes must be consistent across all input but not necessarily in case-sensitive way (uppercase/lowercase variants allowed).

4. When hiring a new person…
    - the input can include their salary elements but doesn't necessarily have to.
    - any missing mandatory fields will result in this person's data being discarded.
    - If the new hire comes without an employee code, we will generate one based on their first work day (6 digits: yymmdd) and an order number between 0-255 in hexadecimal format (2 characters) concatenated as a string.
    - If the new hire comes without both employee code and first day of work, their data will be discarded.

5. When changing an existing person's data…
    - all valid new values will be applied, while faulty ones will be ignored.
    - all valid salary components will be added, while all faulty ones discarded.
    - their employee code is used to find the target whom to change. If no code is provided, the change will be ignored.

6. When terminating an existing employee…
    - it is sufficient to provide their employee code and the termination date.
    - If the date is not provided, the date of request processing will be used instead.
    - Without the employee code, the termination will not be carried out.

7. The input file can include redundant data. If we have nothing to do with a column, we'll simply ignore it.

The Assignment

Hint: read the entire thing before starting to code or you may end up doing redundant work.

The input CSV file is provided as a separate attachment “input_01.csv”.

The Rules

- We prefer the submitted code to be written in **Kotlin**. Also we use Maven but that’s not as important unless you want to be better prepared.
- You can use external tools/libraries as long as you comment (in-line or in an external summary) what each function you used does.
    - A written summary/explanation of your solution is welcome either way.
    - So are any comments about your struggles and/or description of your journey for finding the solutions.
- **We appreciate communication!** You can email your additional questions to

    - They will gladly explain anything that’s confusing for you or point you in the right direction if you’re stuck. That’s how we do it during real work, too.
- **You don’t need to do all the tasks.** Pick the one that matches your skill level: difficulty increases with each task No. and so does the tech stack you’re expected to use.

Task Level 1

You are to parse the provided CSV file and transform the data into JSON format, in a way that the API endpoint described above would accept it.

We expect you to submit your app code and the resulting JSON. See section “Submission Format” for more details.

Points are granted in order of priority for:

- Code quality and cleanliness
- Abstractions and design
- Understanding of the data models
    - Yes, we did not explain what goes under “data”. At this level it is up to your intuition and there are no *wrong* answers (as long as they make sense).
- The validity of end result (JSON)
- Adherence to business rules and agreements (see above)

Task Level 2

You are to parse the provided CSV file and transform the data into JSON format, in a way that the API endpoint described above would accept it.

Among the attachments you will find an SQL dump. That is the representation of the target database, where the in-house REST API would deposit given data. You can import it and see what the models will look like.

The “data” object in the JSON request should contain key-value pairs that correspond to the tables and columns in the database. Like this:

```
"data": {
        "table_name.column_name": "{{the value}}",
        "table_name.different_column_name": "{{the value}}",
        "different_table_name.other_column_name": "{{the value}}"
      }
```

We expect you to submit your app code and the resulting JSON. See section “Submission Format” for more details.

Points are granted in order of priority for:

- Same points as in Task 1 but with “data” correctness being graded according to SQL schema.
    - Note that some DB columns are mandatory and this means the data is to be expected in the input file. If anything’s missing, handle the error according to business rules (see above).
- (optional) bonus points for an SQL script that inserts the same data as the hypothetical REST API would based on the JSON.
    - Again, no wrong answers as long as they make sense.
- (optional) more bonus points for an SQL script that builds a meaningful view of people and their salaries after the data has been inserted. Use your imagination and the database sample provided!

Task Level 3

You are to parse the provided CSV file and transform the data into JSON format, in a way that the API endpoint described above would accept it.

Among the attachments you will find the file dynamic_config.json. Study it in your preferred json/text editor and make your application take into account what the configuration says about the entities, fields and their values.

In addition, you will find the directory Models attached. Inside there are Kotlin code files that represent the data models useful for parsing and applying the configuration programmatically. Feel free to include them in your solution as-is! Note that they make use of the Kotlin module for the external Java library com.fasterxml.jackson.

We expect you to submit your app code and the resulting JSON. See section “Submission Format” for more details.

Points are granted in order of priority for:

- Same points as in Task 1 but with “data” correctness being graded according to the provided configuration file.
- The way you handle the configuration will also be looked at from the code quality and design perspective.
- The input is supposed to adhere to the provided configuration. Meaning, if any CSV column has the wrong value type or other issues, it should not be included in the output JSON.
    - And if this results in an empty mandatory field, that disqualifies the entire entity.
- Your solution must work even with a slightly different input file as long as it adheres to the provided configuration.
- (optional) bonus points if your solution will work even on a slightly different configuration than we provided (still same data models as in Kotlin files) and corresponding CSV input files.

Extra bonus points

This can be done along with a task of any level.

You may have noticed that the JSON sample includes an “errors” array. You get bonus points if you identify invalid data based on business rules (see above) and write it down as errors in a meaningful way. Once again, there are no wrong answers as long as they make sense.

Submission format

You can submit your code whichever way you’re most comfortable with. A zip archive is fine. But if you are familiar with version control tools, you probably know the benefits of setting up a repository, keeping your commit history and providing us with a link where we can leave feedback right in your code.

As for any additional scripts, documents or your draft-and-test files (yes, please!), include them in a subdirectory.

Your submission must also include either the final output file (JSON) or the instruction on how to get it from your app.

Also please include an informative README.md where you *at the very least* describe what your app (“basically”) does and how to start it. All sorts of interesting (but reasonable) bootstrapping setups are a bonus!