IGP job application task v4

The application

You'll be building a small social **web application** called *bSocial*. The application will allow the users to register, follow other people, leave posts and comments.
The app will consist of the following components:

- Main application needs to be written in either Node.js or Go, or combination. If written in Node.js, please use *express* web framework. If written in Node.js, please use *express*. If written in Go, please use *fiber* framework.
- The Frontend web application is not required. However, if you don't write the frontend application, you need to give some documentation on how to access the backend application API
- All application data should be stored to a SQL database (e.g. SQLite, MySQL) and to Elasticsearch (where required by task). NoSQL use is strictly forbidden.

User registration

The main backend service has to handle user registrations. Users should be able to input first and last name, username, e-mail, password, and password confirmation.

Upon registering, the service needs to send a message to the Kafka, which will indicate that the user has registered. This message needs to contain:

- User data
- The date of registration

Log-in

When a user visits a *bSocial* web application, he/she will be presented with a login form if he/she doesn't have a valid session. The user should log in with a username (e-mail) and password. The server should respond with some form of a session (JWT, session cookie, or something similar).

Bonus tasks

1. Log-out implementation (if you opt for JWT, there's no need to check if token is invalidated due to logout)

Adding posts and comments

Every user can publish posts and add comments to posts (two HTTP endpoints to handle that need to be implemented). The user can only see posts that he/she publishes or that one of his/her followers publishes. Posts or comments don't have to be editable.
Upon creating a post or comment, the application should send a Kafka message.

Kafka message for the **post** should include the following:

- Username
- Email
- User ID (if exists)
- Timestamp
- Post ID
- Message content

Kafka message for every comment should include the following:

- Sender username
- Sender email
- Sender ID (if exists)
- Timestamp
- Post ID
- Comment ID
- Comment content

Posts feed for every user should be accessible via HTTP Rest endpoint (user should get only his/her posts and posts from his/her followers). Post feed needs to have some form of paging and doesn't need to return comments.
Implement HTTP endpoint which returns all the comments for a given post.

Write a microservice that will consume comments feed from Kafka and send notifications to the post publishers whenever someone comments their post.
Implementing notifications can be done in two ways:

1. If your application has a frontend (UI), you could implement notifications via WebSockets (it's a bonus task as well)
2. Or, create an HTTP endpoint that will return all unsent notifications. Every time the endpoint gets called, the user should receive **only new** (undelivered) notifications.

Bonus tasks:

1. Notifications implemented via Websockets

Following people

Implement an HTTP endpoint that will allow a user to follow another user. When a user starts following another user, he/she can see all his/her posts.
Following users is one-way relation (it's not symmetric). If user A starts to follow user B, this doesn't mean that user B will start to follow user A.

Small telemetry system

Write a microservice that will consume all messages sent to Kafka, and write them to Elasticsearch.

Then write the following elasticsearch queries (please store them to some file(s) on the Git repository):

1. A query that returns the count of logged-in players per day
2. A query that returns the top 10 posts (by the number of comments) per day in the last 10 days (the results don't have to be 100% accurate, but the general idea needs to be followed).

Bonus query task:

3. Write a single query that returns an all-time top post from a given user (the post with the highest number of comments), as well as the worst post from the same user (with the lowest number of comments). If there is more than one top/worst post for a specified player, it's not important which one will be returned.

Docker

Write at least one Dockerfile, for the main service.

General bonus tasks

1. Frontend doesn't need to be nice but it's definitely a bonus if it is. To save time it's completely fine if you use a component library. The design doesn't need to be ground-breaking, but it needs to work.
2. Pack all of the services to Docker containers. Use docker compose or kubernetes to keep things simple and connected.
3. Set up any type of testing for some of the services and some parts of the code. Not looking for too much coverage, just as a proof of concept.

4. Since your application code needs to be pushed to some remote (like Bitbucket) anyway, it'd be nice if you could be able to use GitFlow concept while building your application. Make at least one or two feature branches, and merge them back after they are finished. If you opt for setting up some sort of CI (which would be a huge bonus) it's fine to use an alternative to GitFlow.

Requirements

All code has to work. There can be bugs (of course), but every feature should be completed before submitting the application.
Requirements marked as "*Bonus tasks*" are features that are nice to have, but not strictly required. They can show your dedication to what you're doing.
All code has to be committed and pushed to some remote server (Bitbucket, GitHub, etc.) The code/application reviewer must receive access to the code after the application has been developed.

Best of luck!