Altar.io

Full-Stack Exercise **#1223**

Exercise Description

The exercise is composed by two parts:
a) frontend application (SPA);
b) backend application;

Some non-functional requirements:

- All the code must be written in TypeScript.
- You're free to apply any patterns, methodologies and styling languages.
- Frontend should be made using Angular Framework, but you can also use other frameworks if you feel more comfortable
- Backend should run on Node.js JavaScript runtime 12+.

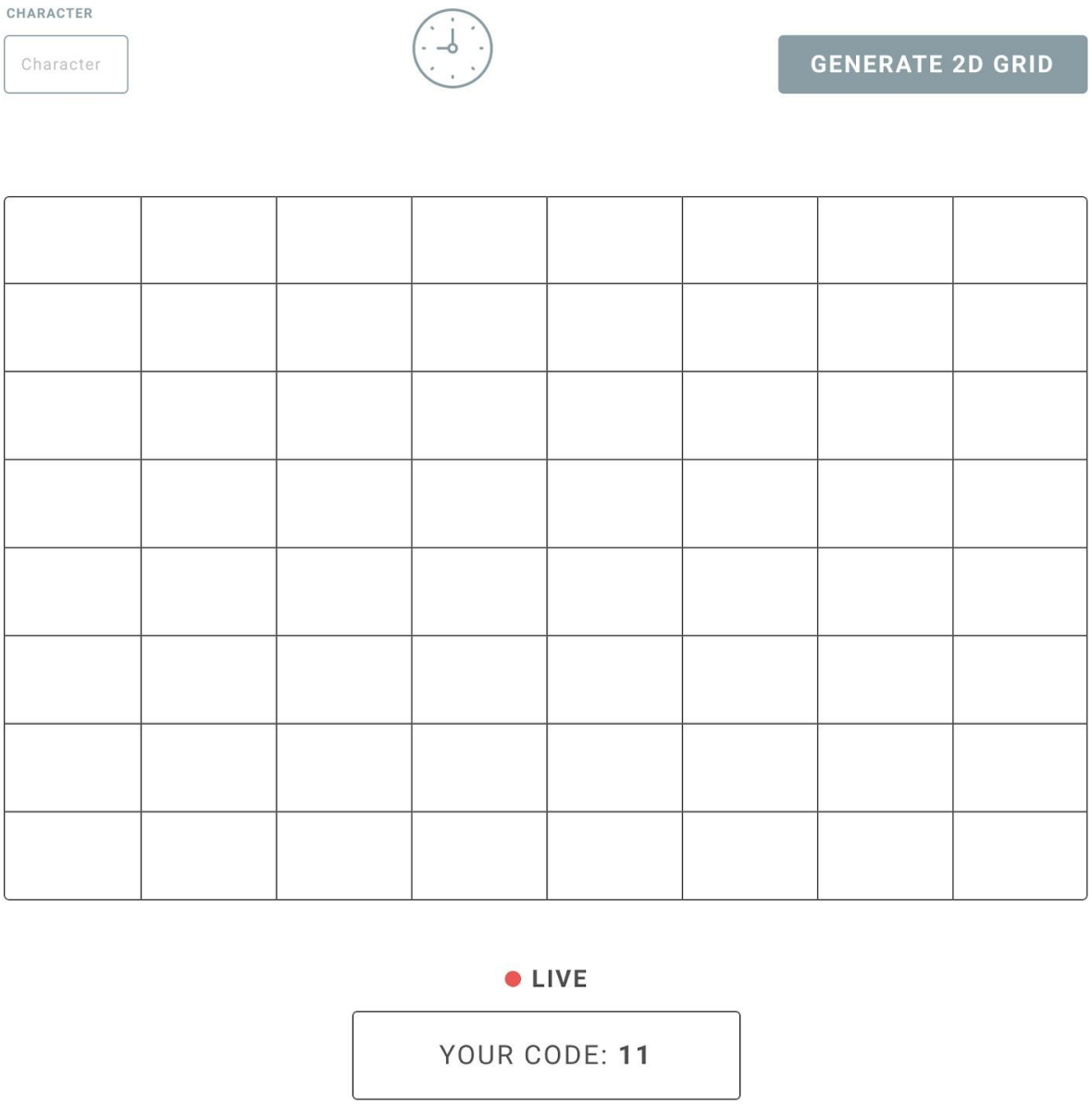
You will be creating **1 mandatory page** with a couple of interactive widgets.

How to deliver the exercise

1. Create a project on your GitHub account
2. Bootstrap your foundation framework
3. Create a branch with your name (ex: feat/john-doe)
4. Develop under that branch
5. Submit a Pull Request on your Repo
6. Send us an email with your Repo

Generator Page

Here is a high fidelity prototype of the only screen that your frontend application should have.



Features

Grid

You'll have a **10x10** grid filled with random alphabetic characters, refreshed/updated every second.

In the page you have a **button** to start the "generator".

After clicking on this button the grid will be filled with random alphabetic characters (a-z), like so coordinates [0,1] will have a "b" and [8,7] will have a "k".

The grid calculation should be made on the backend and exposed over an endpoint; the frontend should just consume and display

Code/Secret

There is a display field underneath the table with the 2 digit code updated at the same time the grid gets refreshed.

You will use the *host API system clock* and the grid together to **generate** a 2 digit code which will show behind the grid (see the high fidelity prototype).

This code should be consumed via an endpoint

To compute the code, the following trivial algorithm needs to be followed:

1. Get the 2 digit seconds from the system clock, like so: 12:40:**36**
2. Get the matching grid cell characters for the positions [**3,6**] and [**6,3**], like so: "v" and "c".
3. Count the occurrences of "v" and "c" on the entire grid, like so: v = 7, c = 9.
4. Exception: If the count is larger than 9, divide the count by the lowest integer possible in order to get a value lower or equal to 9.
5. Done! That is your code: **79**

Bias/Weighting factor Input

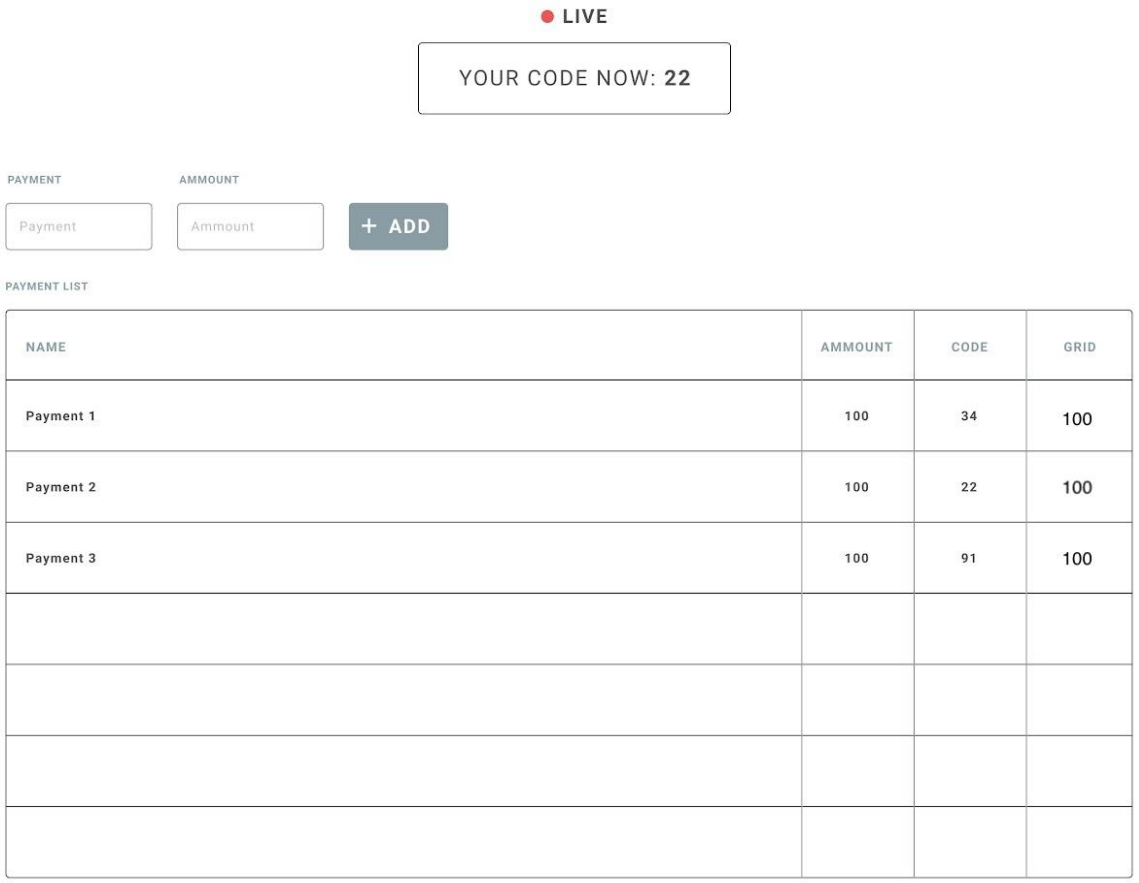
You also find an **input** field which allows the user to enter an alphabetic character (a-z) and this character will be used as a weight constant of 20% when filling the grid, like so: If a character is entered and it's a "z", means that 20% of the grid cells (random positions) will be filled with "z" and the remaining 80% ones with usual random characters.

The user is only allowed to enter a character once every 4 seconds, i.e. a user cannot type a random character.

The bias is sent to the backend.

Note: Every 2 seconds the grid needs to be refreshed automatically and a different code will be generated.

Bonus 1: Payments Page



In this page you will show the updated code on the top (don’t forget, every 2 seconds we have a new code).

There are 2 simple form fields to add a payment name and amount and a button to add to the payments list.

Every entry on the grid will have the current code assigned to it, together with a copy of the grid (yes, all 100 cells).

This payments list should be saved and loaded from the API.

The user should be able to navigate between the 2 pages whilst not losing any information i.,e., still see the payments list.**API (optional)**
You can also create an API (BFF w/ CRUD) to support storing the payments.

Bonus 1: Real-Time Data Synchronization

Description:

Implement a real-time data synchronization feature for the grid and payment list across multiple clients.

Features:

Use WebSockets or a similar technology to establish a real-time bi-directional communication channel between the client and the server.
Whenever a user starts the generator or adds a payment on one client, all connected clients should see these changes in real-time without needing to refresh the page.
Implement conflict resolution strategies on the server-side to handle scenarios where multiple users attempt to add or modify data simultaneously.
Provide a visual indicator on the client side to show that the grid or payments list is being updated in real time.

Technical Considerations:

Ensure that the server can handle multiple connections efficiently.
Secure the WebSocket connection and implement proper authentication to prevent unauthorized access

Bonus 2: Operations (optional)

You can create packaging instructions and deployment templates for CI/CD

How to Deliver the Bonus:

Extend the existing GitHub project with additional branches for each bonus feature (e.g., feat/real-time-sync).
Develop the features under their respective branches.
Update the README.md with instructions on how to use the new features and any necessary documentation.
Submit a Pull Request for each new feature into the main branch of your repository.
Update the CI/CD pipeline to include deployment of the new features.