Coding Exercise: Conway's Game of Life API

Target Language/Framework:

C# using .NET 8.0 (net8.0)

Objective:

Implement a RESTful API for Conway's Game of Life. Your solution should be designed with production readiness in mind. Reference:
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Functional Requirements:

The API should include (at a minimum) the following endpoints:

x, y
x, y
x, y
x, y

Serialize/deserialize
Text

1. Upload Board State Post
   - Accept a new board state (2D grid of cells).
   - Return a unique identifier for the stored board.

2. Get Next State get
   - Given a board ID, return the next generation state of the board.

3. Get N States Ahead get _
   - Given a board ID and a number N, return the board state after N generations.

4. Get Final State delete
   - Return the final stable state of the board (i.e., when it no longer changes or cycles).
   - If the board does not reach a stable conclusion within a reasonable number of iterations, return a suitable error message.

Non-Functional Requirements:

- The service must persist board states so they are not lost if the application is restarted or crashes. DB
- The code should be production-ready:
  - Clean, modular, and testable
  - Includes appropriate error handling and validation
  - Follows C# and .NET best practices
- You do not need to implement authentication or authorization.

API
Service
DB

Evaluation Criteria:

- Correctness – Does the API behave as described?
- Code Quality – Is the code clean, well-structured, and maintainable?
- Design & Architecture – Are design decisions well thought out? Is the code extensible?
- Production Readiness – Is the service robust and resilient?

- Discussion Readiness – Be prepared to walk us through your design and decisions in a follow-up discussion.

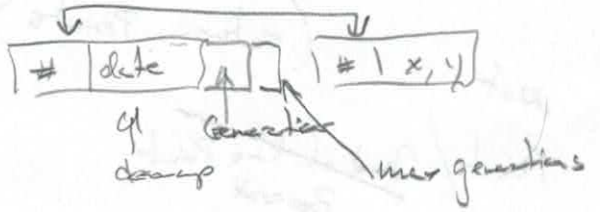
Estimated Time:

This exercise may take 4–5 hours. Manage your time appropriately. We are more interested in quality and thorghtful design than in a perfect or overly complex implementation.

Once you've completed the exercise, please upload your code to a GitHub repository (or a similar platform like GitLab or Bitbucket) and share the link with us. You're also welcome to include any notes or documentation you'd like us to review.

- Base project
- Controller ← swagger / Auth ???
- Service Layer ← DI
- Data layer

DB design Readme.com



Services
load DB
Save DB

Live Cells
Dead Cells

34 million

~~Process point~~

Count neighbours
live cell transition
dead cell transition

tick or return generation(ticks) ← test for still life
beehive
Loaf
Boat
tub
blinker
Toad
Beacon
etc.

✓ Points

✓ Fix Expires In X Days

✓ Finish Doc

✓ Fix Error Messages

Refactor Services

Project

Conways Game Of Life

Build Basic

~~Dummy Controller~~

– Add dummy service

– Swagger

– DB, Data

- Discussion Readiness – Be prepared to walk us through your design and decisions in a follow-up discussion.

Estimated Time:

This exercise may take 4–5 hours. Manage your time appropriately. We are more interested in [illegible] than in a perfect or overly complex implementation.

Once you've completed the exercise, please upload your code to a GitHub repository (or a similar platform like GitLab or Bitbucket) and share the link with us. You're also welcome to include any notes or documentation you'd like us to review.

Point

GameOfLife

Point

method CountNeighbors (~~Dead~~ Potential Live Cells) also resets Potential ~~Dead~~ Dead Cells ~~Reset~~ ~~Potential~~ ~~Dead cells~~

Init Seeds (Livecells) resets (Dead cells)

Transition only works if CountNeighbors executed will do this automatically

(return Points)

Point / ~~Game of Life Point~~ Board .Net

Ozero or max # not exceeded.

- Configuration for Service
- Test for population change to end with point test???
- Test w/Acorn

R-pentomino

1, 0
0, 2
1, 2
1, 3
2, 3

End

Breaking Points

R-pentomino 1103

Diehard

1,0 5,0 6,0 7,0
0,1 1,1
6,3

8 1
6 2
8 1
6 2

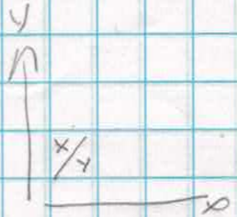
Acorn

0,0 1,0 4,0 5,0 6,0
3,1
1,2

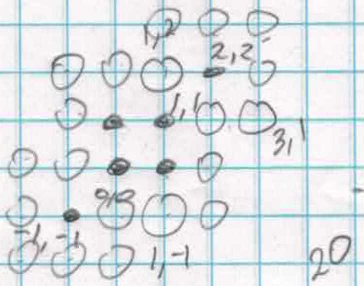
initial
Count

Single step
2
3
4
8

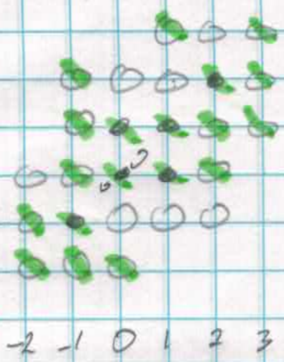
Previous Population[] { }



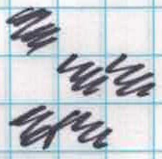
- Scan for all 8 surrounding cells
    - count live cells
    - add dead cells to list
        to scan later
    - based on live cell count
        < 2 remove cell (underpopulat
        > 3 remove cell (overpopulatio



- scan



Glider

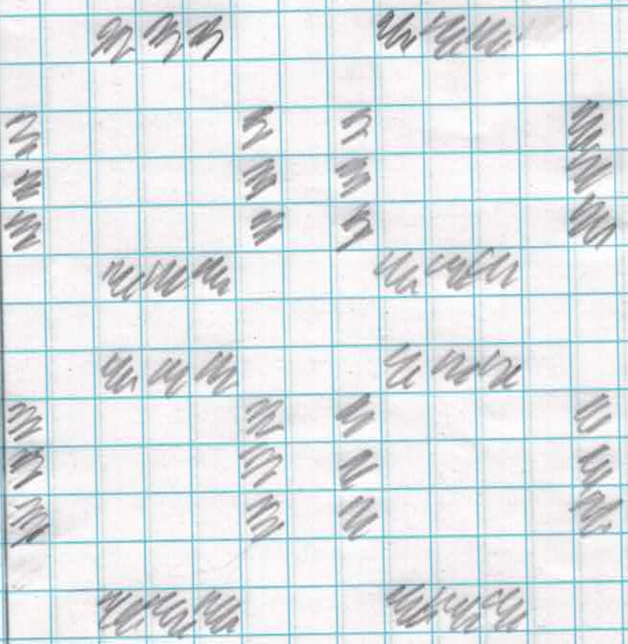


Testing Count

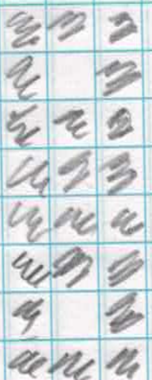
```
1 2 3          -1/1 0/1 1/1
8 ▨ 4      < 2 -1/0 0/0 1/0
7 6 5          -1/-1 0/-1 1/-1

1 1 2
4 ▨ 2      = 2      for -1 to 0
4 3 3               for
                      -1 to 0
41 1 12               not 0/0
4 ▨ 2      = 3
4
3 3 3²

1 1 1
2 ▨ 1      > 3
2 2 2
```

also test >, 4, 5, 6, 7,

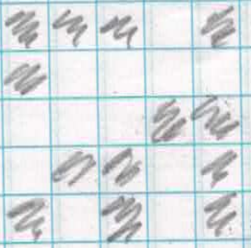
Pulsar



Pentadecathlon



-infinite-



or