



CROSSMINT CHALLENGE
MANUEL
HIDALGO ROS

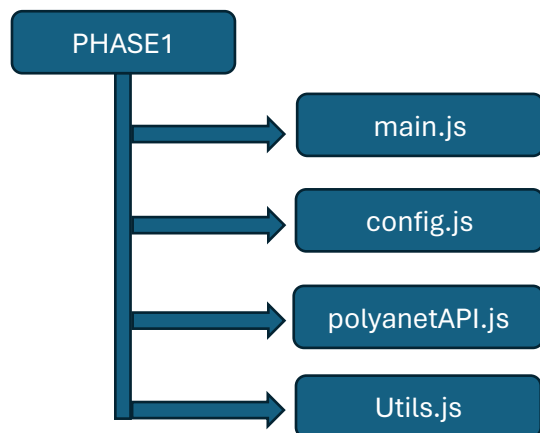
PHASE 1:

In this phase what we must do is interact with the **REST API**, to modify the map and get the *POLYanets* to form a cross in the center of the map. To achieve this, inside the *phase1* folder we find a js script, which we can execute:


```
$ node main.js
```

You will see that the planets that have been created are printed on the screen, and if you scroll to the map on the Crossmint website you will be able to see the result.

How is this phase of the challenge structured?




Explanation of each of the scripts:

 config.js

```
export const candidateId = 'dc640be1-d806-4a03-aebb-0ee7623faa6c';
export const baseUrl = 'https://challenge.crossmint.io/api';
export const N = 11; // Size of the map
export const STARTING_POSITION = 2; // Position of the first X
```

In this script, in the first two lines, we define two constants that will be useful for interacting with the API. Next, we also define two other very useful constants to be able to traverse the map and insert the X in the corresponding place.

 utils.js

```
import { N, STARTING_POSITION } from './config.js';

export const getPolyanetCoordinates = () => {
  let polyanetCoordinates = [];
  for (let i = STARTING_POSITION; i < N - STARTING_POSITION; i++) {
    polyanetCoordinates.push({ row: i, column: i });
    if (i !== N - i - 1) { // If we are in the center we don't push twice
                           // but in other cases we do
      polyanetCoordinates.push({ row: i, column: N - i - 1 });
    }
  }
  return polyanetCoordinates;
};
```

In utils we can find one of the most important functions of phase one. First of all, we must import everything that we are going to need and that is declared in `config.js`. This method returns an array with the positions in which the `POLYanets` will be located to form an (except in the center).

polyanetAPI.js

```
import { baseUrl, candidateId } from './config.js';

export class PolyanetAPI {
  constructor() {
    this.baseUrl = baseUrl;
    this.candidateId = candidateId;
  }

  async createPolyanet(row, column) {
    const response = await fetch(`${this.baseUrl}/polyanets`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        candidateId: this.candidateId,
        row: row,
        column: column
      })
    });

    if (!response.ok) {
      throw new Error(`Failed to create Polyanet at (${row}, ${column})`);
    } else {
      console.log(`Polyanet created at (${row}, ${column})`);
    }
  }
}
```

In `polyanetAPI` our goal is to create a class that allows us to establish an interaction with the API. To do this, we import `baseUrl` and `candidateId` from "`config.js`". We create a class whose constructor will set the `baseUrl` and `candidateId` values. The only method that contains the `PolyAnetAPI` class is `createPolyanet`, commented below:

In the first block of instructions within our `createPolyanet` method, we establish a connection with the API using a `POST` request. To do this, we use the `fetch` function with the base URL and the specific endpoint for creating polyanets. In the request configuration, we define the following properties:

- **method:** Specifies the type of request, which in this case is 'POST'.
- **headers:** Defines the request headers. Here, we set the content type to 'application/json'.
- **body:** Contains the data we send to the API in JSON format. This data includes `candidateId` (previously defined), `row`, and `column`.

These data are interpreted and processed by the API to insert a polyanet into our map. The API response is stored in the `response` variable. Next, we evaluate the status of this response:

- If the request was not successful (i.e., `response.ok` is `false`), we throw an error with a specific message indicating that the creation of the polyanet failed at the given row and column.
- If the request was successful, we display a message in the console indicating that the polyanet was created successfully at the specified location.

This structure allows us to effectively manage both success and error cases, providing appropriate feedback to the user about the outcome of the operation.

main.js

```
import { PolyanetAPI } from './polyanetAPI.js';
import { getPolyanetCoordinates } from './utils.js';

const createPolyanetsInXShape = async () => {
  const polyanetAPI = new PolyanetAPI();
  const polyanetCoordinates = getPolyanetCoordinates();
  for (const { row, column } of polyanetCoordinates) {
    await polyanetAPI.createPolyanet(row, column);
  }
};

// Start creating the Polyanets
createPolyanetsInXShape();
```

Finally, in `main.js`, it imports the constants, methods and classes defined in other files and contains the main function `createPolyanetsInXShape` in which the connection with the API is created through the class defined in `polyanetAPI.js` and obtains the array of coordinates to place the POLYanets. For each coordinate in the `polyanetCoordinates` array create a POLYanet entity with `createPolyanet`.

PHASE 2:

This phase is somewhat more challenging as we are dealing with a larger map, which has a peculiar and interesting shape. Faced with such a large map and with no clear logical pattern regarding some figures, the solution I have chosen to pursue involves querying the API to obtain the target map, followed by traversing it and replicating it.

To access the "phase2" folder. In order to execute the script that generates the Crossmint logo in the megaverse, run:

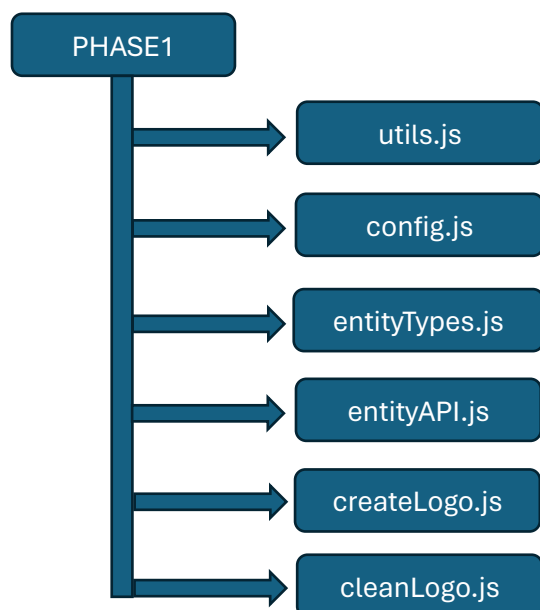
```
$ node createLogo.js
```

To clean the entire map:

```
$ node cleanLogo.js
```

You will see that the planets that have been created are printed on the screen, and if you scroll to the map on the Crossmint website you will be able to see the result.

How is this phase of the challenge structured?



Explanation of each of the scripts:

config.js

```
export const candidateId = 'dc640be1-d806-4a03-aebb-0ee7623faa6c';
export const baseUrl = 'https://challenge.crossmint.io/api';
export const N = 11; // Size of the map
export const STARTING_POSITION = 2; // Position of the first X
```

In this script, we define two constants that will be useful for interacting with the API.

utils.js

```
import fetch from 'node-fetch';
import { baseUrl, candidateId } from './config.js';

// Function to fetch goal map
export async function getGoalMap() {
  try {
    const response = await fetch(`${baseUrl}/map/${candidateId}/goal`);
    if (!response.ok) {
      throw new Error('Failed to fetch goal map');
    }
    const goalMap = await response.json();
    return goalMap; // Return the goal map array
  } catch (error) {
    console.error('Error fetching goal map:', error.message);
    return null; // Return null if fetch request fails
  }
}

// Function to set delays
export function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}
```

This class provides two useful functions for interacting with the Crossmint API and handling delays:

1. **getGoalMap():** This function makes a request to the Crossmint API to fetch the goal map. It utilizes the base URL and candidate ID provided in the configuration (`config.js`). If the request is successful, it returns the goal map in JSON format. In case of an error, it logs an error message to the console and returns null.
2. **delay(ms):** This function takes an argument `ms`, representing the number of milliseconds to wait.

In summary, this class facilitates fetching the goal map and provides a function for delay.

entityTypes.js

```
// Types of entities in goalMap
export const UP_COMETH = 'UP_COMETH';
export const DOWN_COMETH = 'DOWN_COMETH';
export const RIGHT_COMETH = 'RIGHT_COMETH';
export const LEFT_COMETH = 'LEFT_COMETH';
export const POLYANET = 'POLYANET';
export const WHITE_SOLOON = 'WHITE_SOLOON';
export const BLUE_SOLOON = 'BLUE_SOLOON';
export const RED_SOLOON = 'RED_SOLOON';
export const PURPLE_SOLOON = 'PURPLE_SOLOON';
```

In `entityTypes.js`, you can find nine constants that are used to easily identify the type of entity present in each cell.

entityAPI.js

```
import fetch from 'node-fetch';
import { candidateId, baseUrl } from './config.js';

export class EntityAPI {
  static async createPolyanet(row, column) {
    const response = await fetch(`${baseUrl}/polyanets`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({ candidateId, row, column })
    });

    if (!response.ok) {
      throw new Error(`Failed to create Polyanet at (${row}, ${column})`);
    } else {
      console.log(`Polyanet created at (${row}, ${column})`);
    }
  }
}

.....CHECK CODE IN GITHUB.....
```

In this class, the necessary methods for creating and deleting the available entities are defined. Each method takes the required arguments as parameters, and handles any errors that may arise.

createLogo.js

```
import { EntityAPI } from './entityAPI.js';
import { getGoalMap, delay } from './utils.js';
import { UP_COMETH, DOWN_COMETH, RIGHT_COMETH, LEFT_COMETH, POLYANET,
WHITE_SOLOON, BLUE_SOLOON, RED_SOLOON, PURPLE_SOLOON } from './entityTypes.js';

const createPolyanetsInCrossmintLogo = async () => {
  try {
    // Call getGoalMap() and wait for the ACK to resolve
    const goalMapResp = await getGoalMap();

    // Check if the goalMapResp is not null and assign it to an array
    if (goalMapResp !== null) {
      const goalMap = goalMapResp.goal;

      for (let row = 0; row < goalMap.length; row++) {
        for (let column = 0; column < goalMap[row].length; column++) {
          const entity = goalMap[row][column];
          switch (entity) {
            case UP_COMETH:
              await delay(1000);
              await EntityAPI.createCometh(row, column, 'up');
              break;
            case DOWN_COMETH:
              await delay(1000);
              await EntityAPI.createCometh(row, column, 'down');
              break;
            case RIGHT_COMETH:
              await delay(1000);
              await EntityAPI.createCometh(row, column, 'right');
              break;
            case LEFT_COMETH:
              await delay(1000);
              await EntityAPI.createCometh(row, column, 'left');
              break;
            case POLYANET:
              await delay(1000);
              await EntityAPI.createPolyanet(row, column);
              break;
            case WHITE_SOLOON:
              await delay(1000);
              await EntityAPI.createSoloons(row, column, 'white');
```

```

        break;
    case BLUE_SOLOON:
        await delay(1000);
        await EntityAPI.createSoloons(row, column, 'blue');
        break;
    case PURPLE_SOLOON:
        await delay(1000);
        await EntityAPI.createSoloons(row, column, 'purple');
        break;
    case RED_SOLOON:
        await delay(1000);
        await EntityAPI.createSoloons(row, column, 'red');
        break;
    }
}
}
} else {
    console.error('The result of getGoalMap() is null.');
```

En createLogo es necesario importar todo lo definido en otros scripts ya que es donde se implementa la lógica principal para crear el logo de Crossmint en el megaverso.

CreatePolyanetsInCrossmintLogo uses the getGoalMap function to obtain a goal map and then iterates through each cell of the map. Depending on the type of entity in each cell, different methods of the EntityAPI class are called to create that specific entity. Between each method call, there's a pause of 1000 milliseconds using the delay function. If any error occurs during the process, it's handled and logged to the console. Finally, the createPolyanetsInCrossmintLogo function is invoked to start creating the Crossmint logo according to the obtained goal map.

The delay time has been set based on some tests I conducted, as without delays, the API becomes overwhelmed and is unable to create the entities correctly.

clearLogo.js

```
import { EntityAPI } from './entityAPI.js';
import { delay } from './utils.js';
const MAP_SIZE = 30;

const clearAllLogo = async () => {
  for (let row = 0; row < MAP_SIZE; row++) {
    for (let column = 0; column < MAP_SIZE; column++) {
      await delay(500);
      await EntityAPI.deletePolyanet(row, column);
      await delay(500);
      await EntityAPI.deleteSoloons(row, column);
      await delay(500);
      await EntityAPI.deleteCometh(row, column);
    }
  }
};

// Start Cleaning the LOGO
clearAllLogo();
```

In `clearLogo.js`, the main logic for performing a thorough cleanup of the map is implemented to ensure that if any errors occur, we can clean up and rerun `createLogo.js`. The logic is implemented in `clearAllLogo`, and it's quite simple. It forcefully attempts to delete the three types of entities that could be located in a cell. This is achieved thanks to `EntityAPI` defined earlier.

If errors occur, it's normal. Perhaps the delay needs to be adjusted to a higher number, or as often happens, the error is detected when it can't delete an entity that isn't in that cell.

Conclusion (6/3/2024-6/4/2024):

Completing this programming challenge has been a truly fascinating experience. I had never tackled a similar task before, and I find this challenge both enjoyable and innovative. Despite my youth, at only 19 years old, I understand that accessing internship opportunities can be challenging. However, I am confident that I have completed a satisfactory and enriching piece of work.

Furthermore, this challenge has introduced me to Crossmint, a platform I was previously unfamiliar with. I would like to delve deeper into its suite of tools and explore more about its possibilities. Given my interest in computer science, programming, and blockchain technology, I believe that continuing to research topics related to this field is highly enriching.

In summary, I highly recommend undertaking challenges of this nature as they offer a stimulating and educational experience.