```
const generateResponse = (chatElement) => {
    const API_URL = "https://api.openai.com/v1/chat/completions";
    const messageElement = chatElement.querySelector("p");

    // Define the properties and message for the API request
    const requestOptions = {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            "Authorization": `Bearer ${API_KEY}`
        },
        body: JSON.stringify({
            model: "gpt-3.5-turbo",
            messages: [{role: "user", content: userMessage}],
        })
    }

    // Send POST request to API, get response and set the reponse as
paragraph text
    fetch(API_URL, requestOptions).then(res => res.json()).then(data => {
        messageElement.textContent =
data.choices[0].message.content.trim();
    }).catch(() => {
        messageElement.classList.add("error");
        messageElement.textContent = "Oops! Something went wrong. Please
try again.";
    }).finally(() => chatbox.scrollTo(0, chatbox.scrollHeight));
}
```

Challenge: Figuring out what was the right way to define the properties and message for the API request.
Solution: Had to look more into the documentation along with some research.

```
document.addEventListener('DOMContentLoaded', function() {
    const loginContainer = document.querySelector('.login-container');
    const registerContainer =
document.querySelector('.register-container');
    const registerLink = document.getElementById('registerLink');
    const backToLoginLink = document.getElementById('backToLoginLink');

    registerLink.addEventListener('click', function(e) {
        e.preventDefault();
```

```
        loginContainer.classList.add('hidden');
        registerContainer.classList.remove('hidden');
    });

    backToLoginLink.addEventListener('click', function(e) {
        e.preventDefault();
        registerContainer.classList.add('hidden');
        loginContainer.classList.remove('hidden');
    });

    // Other code
});
```

Challenge: Handling the switch between the login and register forms when the corresponding links are clicked.

Solution: I discovered that adding event listeners to the "register" and "back to login" links allows me to toggle the visibility of the login and register containers by adding or removing the "hidden" class. This way, only one form is visible at a time, providing a smooth user experience.

```
document.addEventListener('DOMContentLoaded', function() {
    // Previous code for handling form switch

    document.getElementById('loginForm').addEventListener('submit', async
(e) => {
        e.preventDefault();
        const username = document.getElementById('loginUsername').value;
        const password = document.getElementById('loginPassword').value;
        const response = await fetch('/login', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ username, password })
        });
        if (response.ok) {
            alert(await response.text());
            window.location.href = 'index.html'; // Redirect to homepage
after login
        } else {
            alert('Login failed');
        }
    });
```

```
    document.getElementById('registerForm').addEventListener('submit',
async (e) => {
        e.preventDefault();
        const username = document.getElementById('registerUsername').value;
        const password = document.getElementById('registerPassword').value;
        const response = await fetch('/register', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify({ username, password })
        });
        if (response.ok) {
            alert(await response.text());
            registerContainer.classList.add('hidden');
            loginContainer.classList.remove('hidden');
        } else {
            alert('Registration failed');
        }
    });
});
```

Challenge: Sending login and registration data to the server using the fetch API, and handling the server's response.

Solution: By studying the fetch API documentation and examples, I learned how to make asynchronous POST requests to the server with the appropriate headers and request body containing the username and password. I also used async/await to handle the response from the server, displaying success or failure messages to the user and updating the UI accordingly.

```
// Register endpoint
app.post('/register', async (req, res) => {
    const { username, password } = req.body;

    try {
        // Check if username already exists
        const existingUser = await usersCollection.findOne({ username });
        if (existingUser) {
            return res.status(400).send('Username already exists');
        }
```

```
        // Hash password and save user to database
        const hashedPassword = await bcrypt.hash(password, 10);
        await usersCollection.insertOne({ username, password:
hashedPassword });

        res.send('Registration successful');
    } catch (err) {
        console.error('Error registering user:', err);
        res.status(500).send('Registration failed');
    }
});

// Login endpoint
app.post('/login', async (req, res) => {
    const { username, password } = req.body;

    try {
        // Find user by username
        const user = await usersCollection.findOne({ username });
        if (!user) {
            return res.status(401).send('Invalid username or password');
        }

        // Compare passwords
        const passwordMatch = await bcrypt.compare(password,
user.password);
        if (!passwordMatch) {
            return res.status(401).send('Invalid username or password');
        }

        res.send('Login successful');
    } catch (err) {
        console.error('Error logging in user:', err);
        res.status(500).send('Login failed');
    }
});
```

Challenge: Handling Password Security
Solution: Use a well-established hashing algorithm like `bcrypt` with an appropriate cost factor (e.g., 10 in our case) to balance security and performance.

```
// Front-end
function createForum() {
    const titleInput = document.getElementById('forumTitle');
    const descriptionInput = document.getElementById('forumDescription');
    const title = titleInput.value;
    const description = descriptionInput.value;

    fetch('/forums', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ title, description })
    })
    .then(response => response.text())
    .then(message => {
        alert(message);
        titleInput.value = '';
        descriptionInput.value = '';
        fetchForums(); // Refresh the list of forums
    })
    .catch(error => console.error('Error creating forum:', error));
}

// Back-end
app.post('/forums', (req, res) => {
    const { title, description } = req.body;
    fs.readFile(forumsFilePath, (err, data) => {
        if (err) {
            res.status(500).send('Error reading forum data.');
            return;
        }
        const forums = JSON.parse(data);
        const newForum = { id: forumId++, title, description, messages: []
};
        forums.push(newForum);
        fs.writeFile(forumsFilePath, JSON.stringify(forums, null, 2), (err)
=> {
            if (err) {
                res.status(500).send('Error saving forum data.');
                return;
            }
            fs.writeFile(forumIdFilePath, forumId.toString(), (err) => {
```

```
                if (err) {
                    console.error('Error updating forumId file:', err);
                }
            });
            res.send('Forum created successfully.');
        });
    });
});
```

The code allows users to create new forums and display existing ones. The createForum function is triggered when a user clicks a button, sending a POST request to the server to create a new forum. The server handles this by saving the new forum in a JSON file.

Challenge: Ensuring forums are created and displayed effectively.
Solution: The structured approach where the front-end communicates with the server to create and display forums ensures a smooth user experience. The `fetchForums` function refreshes the list of forums after a new one is created, addressing any potential issues with stale data.

```
// Front-end
function postComment(forumId) {
    const commentInput =
document.getElementById(`comment-input-${forumId}`);
    const commentsContainer =
document.getElementById(`comments-container-${forumId}`);

    if (!commentInput.value.trim()) {
        alert('Please write something before submitting.');
        return;
    }

    fetch(`/forums/${forumId}/messages`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ message: commentInput.value })
    })
    .then(response => response.text())
    .then(message => {
        alert(message);
        commentInput.value = ''; // Clear input after posting
        fetchComments(forumId, commentsContainer); // Refresh comments
    })
```

```
    .catch(error => console.error('Error posting comment:', error));
}

// Back-end
app.post('/forums/:id/messages', (req, res) => {
    const { message } = req.body;
    const { id } = req.params;
    fs.readFile(forumsFilePath, 'utf8', (err, data) => {
        if (err) {
            res.status(500).send('Error reading forum data.');
            return;
        }
        let forums = JSON.parse(data);
        const forum = forums.find(f => f.id == id);
        if (forum) {
            if (!forum.messages) {
                forum.messages = [];
            }
            forum.messages.push(message);
            fs.writeFile(forumsFilePath, JSON.stringify(forums, null, 2),
(err) => {
                if (err) {
                    res.status(500).send('Error updating forum data.');
                    return;
                }
                res.send('Message added successfully.');
            });
        } else {
            res.status(404).send('Forum not found.');
        }
    });
});
```

The code enables users to add comments to forums. The `postComment` function captures user input and sends it to the server, which appends the comment to the corresponding forum. The server routes handle this process, while the front-end dynamically updates the display.

Challenge: Handling user comments reliably and displaying them under the corresponding forum.
Solution: Implementing functions to post and display comments with proper error handling. This setup ensures reliable communication between the front-end and back-end, while providing feedback to users on the success of their actions. And implemented a forumID.json that helps tracks the forums ID and would be updated whenever a new forum is created.

```
const forumsFilePath = path.join(__dirname, 'data', 'forums.json');
const forumIdFilePath = path.join(__dirname, 'data', 'forumId.json');

if (!fs.existsSync(path.join(__dirname, 'data'))) {
    fs.mkdirSync(path.join(__dirname, 'data'));
}

let forumId = 1;
if (fs.existsSync(forumIdFilePath)) {
    forumId = parseInt(fs.readFileSync(forumIdFilePath, 'utf8'));
} else {
    fs.writeFileSync(forumIdFilePath, forumId.toString());
}
```

The server uses JSON files to store forum data and maintain a unique ID for each forum. This approach ensures data persistence across server restarts and allows unique identification when creating new forums.

Challenge: Maintaining persistent storage for forums and comments.
Solution: Using JSON files to store forum data and unique IDs to provide a simple and effective way to manage the forums. However, it might need to adopt a more robust data storage solution, such as a database.

```
let latestVersion = '';
let championsData = [];

fetch('https://ddragon.leagueoflegends.com/api/versions.json')
    .then(response => response.json())
    .then(versions => {
        latestVersion = versions[0];
        const championsUrl =
`https://ddragon.leagueoflegends.com/cdn/${latestVersion}/data/en_US/champi
on.json`;
        return fetch(championsUrl);
    })
    .then(response => response.json())
    .then(data => {
        championsData = Object.values(data.data);
        populateChampions(championsData);
    })
    .catch(error => console.error('Error fetching champions data:',
error));
```

The code fetches the latest version of game data from Riot Games' API and displays the available champions. It starts by fetching the latest game version, then retrieves the corresponding champion data. The `populateChampions` and `displayChampions` functions manage the display of the fetched data.

Challenge: Handling asynchronous data fetching while maintaining code readability and avoiding callback hell.
Solution: The use of JavaScript's fetch API with promise chaining ensures that asynchronous calls are handled sequentially and cleanly. This approach prevents callback hell and keeps the code more maintainable. By handling errors in a centralized `.catch` block, the code effectively manages potential issues in the data-fetching process.

```
function openPopup(champ) {
    const championDetailsUrl =
`https://ddragon.leagueoflegends.com/cdn/${latestVersion}/data/en_US/champi
on/${champ.id}.json`;

    fetch(championDetailsUrl)
        .then(response => response.json())
        .then(data => {
            const detailedChampData = data.data[champ.id];
            const abilitiesHtml = detailedChampData.spells.map((spell,
index) => {
                const skillType = ['Q', 'W', 'E', 'R'][index];
                return `<img
src="https://ddragon.leagueoflegends.com/cdn/${latestVersion}/img/spell/${s
pell.image.full}" alt="${spell.name}" class="skill-icon"
data-index="${index}" data-skilltype="${skillType}"
data-skillname="${spell.name}">`;
            }).join('') + `<img
src="https://ddragon.leagueoflegends.com/cdn/${latestVersion}/img/passive/$
{detailedChampData.passive.image.full}" alt="Passive:
${detailedChampData.passive.name}" class="skill-icon" data-index="passive"
data-skilltype="Passive"
data-skillname="${detailedChampData.passive.name}">`;

            const popupOverlay = document.createElement('div');
            popupOverlay.className = 'popup-overlay';
            popupOverlay.innerHTML = `<div class="popup-content"><span
class="close-btn">&times;</span><div class="popup-inner"><div
```

```
class="champion-image"><img
src="https://ddragon.leagueoflegends.com/cdn/img/champion/splash/${champ.id
}_0.jpg" alt="${champ.name}"></div><div
class="champion-details"><h2>${champ.name} -
${champ.title}</h2><p>${detailedChampData.lore}</p><div
class="abilities-icons">${abilitiesHtml}</div><div
class="skill-description"></div></div></div></div>`;
            document.body.appendChild(popupOverlay);
            popupOverlay.style.display = 'flex';

            // Setup close button and outside click functionality
            const closeBtn = popupOverlay.querySelector('.close-btn');
            closeBtn.addEventListener('click', () => {
                popupOverlay.style.display = 'none';
                document.body.removeChild(popupOverlay);
            });
            popupOverlay.addEventListener('click', (e) => {
                if (e.target === popupOverlay) {
                    closeBtn.click();
                }
            });

            document.querySelectorAll('.skill-icon').forEach(icon => {
                icon.addEventListener('click', () => {

document.querySelectorAll('.skill-icon').forEach(otherIcon => {
                        otherIcon.classList.remove('skill-icon-selected');
                    });
                    icon.classList.add('skill-icon-selected');

                    const index = icon.getAttribute('data-index');
                    const skillType = icon.getAttribute('data-skilltype');
                    const skillName = icon.getAttribute('data-skillname');
                    let description;
                    if (index === 'passive') {
                        description =
detailedChampData.passive.description;
                    } else {
                        description =
detailedChampData.spells[index].description;
                    }
                    const skillDescriptionDiv =
document.querySelector('.skill-description');
```

```
                    skillDescriptionDiv.innerHTML =
`<p><strong>${skillType} - ${skillName}</strong>: ${description}</p>`;
                });
            });
        })
        .catch(error => console.error(`Error fetching details for
${champ.id}:`, error));
}
```

When a user clicks on a champion, the `openPopup` function displays a detailed view with the champion's abilities and lore. This function fetches specific data for the selected champion and dynamically creates a popup overlay to present the information.

Challenge: Displaying complex data in an organized manner while managing popup interactions.
Solution: The openPopup function dynamically creates a popup overlay for each champion, organizing the data into distinct sections for details and abilities. The function also includes event listeners to handle popup closing and skill selection.

```
window.searchChampions = function() {
    const searchTerm =
document.getElementById('searchBar').value.toLowerCase();
    const filteredChampions = championsData.filter(champ =>
        champ.name.toLowerCase().includes(searchTerm)
    );
    displayChampions(filteredChampions);
};
```

The code includes a search function, `searchChampions`, allowing users to filter champions based on their names. This improves user experience by enabling quick access to specific champions.
Challenge: Filtering large datasets efficiently without affecting performance.
Solution: The code leverages JavaScript's filter method to process the `championsData` array, ensuring the search is efficient. Using the `toLowerCase()` method on both the search term and the champion names ensures case-insensitive matching.

3 more