Brandon Mullins

CS-230

Module Four Journal: Software Application requirements

Southern New Hampshire University

A REST API was developed to function as an intermediary between the client browser and the server. This architecture provides numerous advantages by clearly delineating the responsibilities of each component. By establishing a server-client relationship, the workload becomes distributed, allowing the client to operate independently without sharing resources with the server. The client initiates data requests, which the server processes and returns, facilitating efficient communication.

Despite these benefits, the server-client model also presents certain drawbacks. The separation of functions can create distinct vulnerabilities; for instance, viruses on the client could potentially compromise the server, while Denial of Service (DoS) attacks on the server could disrupt communications back to the client. Additionally, threats such as phishing and Man-in-the-Middle attacks can intercept sensitive information, including login credentials. There is also a risk that data transmission can be compromised through spoofed packets or modifications, highlighting the importance of robust security measures.

The advantages of this model have led to its widespread adoption. It enables centralized data management, which simplifies storage and access. It allows the server and clients to scale and evolve independently, accommodating changes in technology and user needs. The architecture also supports language-agnostic communication, making it versatile across various platforms.

When it comes to online data communication, two primary approaches are commonly utilized: SOAP and RESTful APIs. SOAP is a formal protocol governed by the W3C Consortium, whereas RESTful APIs represent a more flexible architectural style. While SOAP provides built-in security and compliance for transactions, it can be more cumbersome and slower to implement compared to RESTful APIs.

The RESTful API architecture is characterized by six essential attributes. First, there is a clear separation between clients and servers. Second, each request is stateless, meaning that it contains all the necessary information without relying on previous interactions. Third, caching is expected to enhance response times. Fourth, the architecture includes a uniform interface that ensures predictable API behavior, allowing for unique identification of resources and a common vocabulary for manipulation. Fifth, any additional layers—such as caching and security—should be transparent to both clients and servers. Finally, the server may optionally return executable code, like JavaScript, as part of its response.

The REST API designed for this project focuses on authenticating and authorizing various user accounts. Following a well-defined pattern, as described in a tutorial on Dropwizard security, the primary object involved in security is the user whose credentials are provided in the API request. The Authenticator class is tasked with verifying the username and password, while role-based authentication is implemented through the Authorizer class, which determines if a user is permitted to perform specific actions.

Dropwizard leverages these classes to manage authentication and authorization for users accessing the API. By using the @AUTH annotation alongside the @RolesPermitted annotation, access can be restricted to specific roles, contrasting with the @PermitAll annotation that allows universal access.

This flexible implementation pattern can be adapted to various client types. Since RESTful APIs are not bound by specific protocols, they can seamlessly integrate across different platforms. For instance, if the API were to be deployed on gaming consoles like XBOX or PS4, the server would remain unchanged as long as those platforms can construct requests and handle responses appropriately.

However, enhancements to the gameauth service are necessary. The existing usernames for guests, users, and admins are insufficient, necessitating the development of an interface to allow the addition of new usernames and their corresponding roles. Furthermore, the process of adding users and assigning roles must be carefully controlled to prevent unauthorized access, such as a guest creating an admin account. While these functionalities could be implemented through an API, they present potential security risks. A more effective approach might involve creating a user interface for backend management of enrollment and role assignments, secured by multiple layers of protection.