Tuesday, 11 March 2025

Research Document: Interactive Digital Notice Board API

Part 1: RESTful APIs (KU1)

RESTful API stands for REpresentational State Transfer and is an interface that two computer systems use to exchange information securely over the internet. Most business applications have to communicate with other internal and third party applications to perform various tasks.
RESTful APIs use HTTP methods to perform CRUD operations. CRUD stands for Create, Read, Update, Delete). These operations are designed to be scalable and simple. The main principle is that each request from a client to a server must contain all the information the server needs to fulfil the request. The server should never store any of the information about the client in between the requests.

Core Principles

The client server architecture is a system where resources and services are hosted, delivered, and managed by the server in response to client requests. In this setup, all interactions and services are transmitted over a network, and it is also known as the network computing model or client server network.

Client server architecture, also known as the client server model, is a network application that distributes tasks and workloads between clients and servers, which may either reside on the same system or be connected through a computer network.

Typically, this architecture involves multiple users' devices, such as workstations, PCs, or other devices, connected to a central server via the internet or another network. The client sends a data request, and the server processes and responds by sending the necessary data packets back to the requesting user.

This model is also referred to as a client server network or network computing model.
How it works:

- First, the client submits their request through a network enabled device.
- Next, the network server receives and processes the user's request.
- Lastly, the server sends the response back to the client.

Client server architecture offers the ideal framework that modern organisations require to tackle the challenges of an ever changing IT landscape.
Client and server machines usually need different hardware and software resources and are often provided by different vendors. The network supports horizontal scalability, which increases the number of client machines, and vertical scalability, where the workload is shifted to more powerful servers or a multi server setup.
A single computer server can offer multiple services at once, though each service requires its own server program. Both client and server applications interact directly with a transport layer protocol, which facilitates communication and allows data to be sent and received between them.
Both client and server computers require a full protocol stack. The transport protocol relies on lower layer protocols to transmit and receive individual messages.

Examples of Client Server Architecture

- Email servers: Email has replaced traditional postal mail as the main way businesses communicate. Email servers, along with different types of software, help send and receive emails between people.
- File servers: When you store files on cloud services like Google Docs or Microsoft Office, you're using a file server. File servers are central places where files are stored and can be accessed by many users.

Statelessness

Every API request must include all the information the server needs to understand the request. This means the server does not store any session data and cannot remember anything from previous requests.

Cache ability

The server's responses should clearly specify whether they can be cached or not. This helps boost performance by reducing the need to make repeated requests for the same resource.

Uniform Interface

RESTful APIs have a consistent interface, making it easier for clients and servers to communicate. This standardisation allows clients to interact with various services in a uniform way.

How Making a Request to an API Works

API call follows three basic steps:

- Client makes a request: The client sends an HTTP request (GET, POST, PUT, PATCH, DELETE) to the specified URL, formatted according to the API endpoint's protocol and schema.
- Server processes the request: The server receives and processes the request, such as querying a database or performing an action.
- Server sends a response: The server responds with data (usually in JSON or XML), or an error message with an appropriate HTTP status code if there is an issue.

HTTP Methods

GET is the most commonly used HTTP method, mainly for retrieving data from the server. It is used to request resources like:

- A webpage or HTML file
- An image or video
- A JSON document
- A CSS or JavaScript file
- An XML file

GET is considered a safe operation because it should not change the state of any resource on the server.

POST:

The HTTP POST method sends data to the server for processing. The data sent is usually in the form of:

- Input fields from online forms
- XML or JSON data
- Text data from query parameters

The developer decides how the server should process the data. Common uses of POST include:

- Posting a message to a bulletin board
- Saving form data to a database
- Calculating results based on submitted data

POST is not a safe operation because it can change the server's state and cause side effects. It is also not idempotent, meaning repeated POST requests can change the server's state differently each time.

PUT:

The HTTP PUT method is used to fully replace a resource at a given URL.
Key rules of the PUT method:

1. A PUT request always includes a payload with a new version of the resource to be saved on the server.
2. The PUT request uses the exact URL of the resource being replaced. If the resource exists, it is fully replaced; if it doesn't, a new resource is created.
3. The payload can be in any format the server understands, with JSON and XML being the most common.

Idempotent and unsafe:

- PUT is considered unsafe because it changes the resource on the server.
- It is idempotent, meaning multiple identical PUT requests will result in the same state. For example, updating a flight status to "ontime" multiple times will always leave the status as "ontime."

Unlike PUT, POST operations are not idempotent.

DELETE:

The HTTP DELETE method is how you tell a web server to remove (delete) something at a specific URL. DELETE changes data on the server (so it’s considered “unsafe”), if you send the same DELETE request more than once you end up in the same state, the resource is gone. For example:
if you send a DELETE request to `https://api.example.com/users/123`, you’re asking the server to remove user #123. If the deletion succeeds, the server usually replies with “204 No Content.” If you send that same DELETE request again, you’ll typically get a “404 Not Found,” because the user was already deleted.

PATCH:

The HTTP PATCH method is used when you want to change just part of a large resource instead of sending the entire thing back to the server. Introduced in RFC 5789, PATCH takes a small JSON “diff” that describes only the fields you want to update.
The server applies that single change and usually responds with **204 No Content** (success with no body) or **200 OK** (success with the updated resource). Because you’re only sending the fields that change, PATCH is more efficient than PUT when working with large objects.

Tuesday, 11 March 2025

QueryString Parameters and Request Body

Query string parameters are key value pairs added to the URL to filter or modify the request. They typically appear after a question mark (`?`) and are separated by an ampersand (`&`) if there are multiple parameters.
Examples:

1. **/users?age=25** - Filters users to show only those who are 25 years old.
2. **/products?category=electronics&sort=price** - Displays electronics products sorted by price.

The request body is used to send data to the server, mainly in POST and PUT requests. It contains the information that the client wants to create or update on the server, often in formats like JSON or XML.

Part 2 – Authorisation (KU2)

OAuth 2.0, which stands for "Open Authorisation," is a standard that allows a website or application to access resources hosted by other web apps on behalf of a user. It replaced OAuth 1.0 in 2012 and has become the industry standard for online authorisation. OAuth 2.0 ensures consented access and limits the actions the client app can perform on resources for the user, without sharing the user's credentials.

Benefits of OAuth 2.0:

- Security: OAuth reduces the risk of credential theft by eliminating the need for users to share their passwords with third party services.
- Granular Permissions: OAuth 2.0 provides detailed control over the level of access granted to third party applications, allowing users to limit what data can be accessed.
- Revocable Tokens: Access tokens can be revoked at any time without impacting other services or requiring the user to change their password.

How Does OAuth 2.0 Work?

1. **Scopes**: These define the level of access the third party application will have. For example, `read:user` grants read only access to user data, while `write:user` allows modifying user information.
2. **Access Tokens**: Issued by the authorisation server, these tokens authenticate API requests and represent the client's authorisation to access specific resources on behalf of the user.
3. **Client ID and Client Secret**: The Client ID identifies the application requesting access, and the Client Secret serves as a password to authenticate the client to the authorisation server.

Process Flow:

- The client sends an authorisation request to the Authorisation server, providing its Client ID, secret, requested scopes, and a redirect URI for sending the Access Token or Authorisation Code.
- The Authorisation server authenticates the client and checks if the requested scopes are allowed.
- The resource owner interacts with the Authorisation server to grant or deny access.

- The Authorisation server responds by redirecting the client with either an Authorisation Code or Access Token (depending on the grant type). A Refresh Token may also be included.
- Using the Access Token, the client requests access to resources from the Resource server.

Part 3: Security Considerations (KU3)

What is the OWASP API Security Top 10?

The organisation's flagship project is the OWASP Top 10 list, which highlights the most critical web application vulnerabilities and provides mitigation strategies to help web developers address these security risks. In 2019, OWASP introduced the API Security Top 10 list in their annual reports. This list serves as an important awareness tool for software developers, highlighting the key security issues they need to address when building and maintaining APIs.

Latest Threats:

API1:2023 - Broken Object Level Authorisation: This occurs when an API allows unauthorised access to objects or data, which could expose sensitive information. Attackers exploit this by manipulating API requests to access unauthorised resources.
API2:2023 - Broken Authentication: APIs that do not properly authenticate users or manage sessions are vulnerable to attacks where attackers impersonate legitimate users to access sensitive data.
API3:2023 - Excessive Data Exposure: APIs that expose more data than necessary can be exploited by attackers, leading to leakage of sensitive information.
API4:2023 - Lack of Resources & Rate Limiting: APIs that fail to limit the number of requests a client can make within a specified time frame are susceptible to denial of service (DoS) attacks.
API5:2023 - Broken Function Level Authorisation: Similar to broken object level authorisation, this occurs when users can access or perform actions that they are not authorised for.
API6:2023 - Mass Assignment: This happens when an API allows bulk modification of multiple fields without sufficient validation, potentially enabling data manipulation or injection attacks.
API7:2023 - Security Misconfiguration: APIs with poorly configured security settings (e.g., default credentials or misconfigured permissions) are vulnerable to attacks.
API8:2023 - Injection: APIs that fail to properly sanitise user input are vulnerable to injection attacks, where malicious data is inserted into queries or commands.
API9:2023 - Improper Assets Management: APIs that do not manage assets (such as endpoints or configurations) properly can expose unnecessary or sensitive resources to attackers.
API10:2023 - Insufficient Logging & Monitoring: Without proper logging and monitoring, attackers can exploit vulnerabilities without being detected.

Mitigating Security Threats in Our API

For the Interactive Digital Notice Board API, we will address the following security concerns:

1. **Broken Authentication**: To mitigate this threat, we will implement OAuth 2.0 for secure and authorised access. We will also enforce strong password policies and use multi factor authentication (MFA) to enhance the security of user logins.
2. **Excessive Data Exposure**: We will implement proper data filtering and ensure that only authorised users can access certain types of data. For example, users will only

be able to access the notice boards relevant to their organisation, ensuring that sensitive information is not exposed to unauthorised individuals.

By focusing on these key security measures, we aim to develop a secure, efficient, and user friendly API for our Interactive Digital Notice Board system, ensuring both robust protection against potential vulnerabilities and a seamless experience for users.

References

1. **RESTful APIs (KU1)** — https://shorturl.at/c6MHb
2. **Examples of Client-Server Architecture** — https://shorturl.at/X8SX9
3. **HTTP Methods** — https://shorturl.at/vg3vu
4. **How Does OAuth 2.0 Work?** — https://shorturl.at/1NJVq