ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 7, Num. 2, 2022

FEATURES OF DEVELOPMENT AND ANALYSIS OF REST SYSTEMS

Bohdan Marii, Ivan Zholubak

Lviv Polytechnic National University, 12, Bandera Str, Lviv, 79013, Ukraine.
Authors' e-mails*: marii.bohdan1@gmail.com, IvanZholubak7@ukr.net*

https://doi.org/10.23939/acps2022.__.___
Submitted on 01.10.2022

***Abstract*: The paper analyzes and presents the architecture of REST systems construction. What is the REST API and why it should be used? It describes the basic principles for what the system could be called Restful. Examples of REST-like systems, their comparison, advantages, and disadvantages of REST, and why this particular architecture was chosen, have been given. It aims at which technologies can be used for the REST system, etc. A description of all technologies used during the development of this system, with all the advantages and disadvantages of using certain technologies and the system in general have been considered. A description of the development environment and some of its functions have been provided. Implementation of the REST system is based on the web application of the forum.**

***Index Terms*: client-server system, database, server, client, REST.**

I. INTRODUCTION

REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems on the Internet, making it easier for systems to communicate with each other [1]. You can build a working application, but it must also have good architecture, as it is critical to support future growth. The architecture is a plan to support future growth that may arise from increased demand, future interoperability, and increased reliability requirements. In 1999, the API environment was free for all. At the time, most developers had to deal with SOAP (Simple Object Access Protocol) to integrate APIs. And the "simple" part of this abbreviation should not be taken literally. To make the call, developers had to manually write an XML document with the RPC call in the body. But a small group of expert developers have realized the true potential of Web APIs. Due to this small group led by Roy Fielding, REST was created and the API landscape changed forever. In 2000, Roy Fielding and his colleagues had one goal: to create a standard so that any server could communicate with any other server in the world. Here's what he came up with in his PhD thesis: "I had input from more than 500 developers, many of whom were distinguished engineers with decades of experience, and I had to explain everything from the most abstract notions of web interaction to the smallest details of HTTP syntax. This process refined my model to a core set of principles, properties, and constraints that are now called REST." With the help of REST, the organizational design of the web application architecture defines exactly how the application will function [2]. Some features include:

- Providing persistent data over HTTP that can be understood by client code and vice versa
- Ensuring that queries contain valid data
- Authentication for users
- Restrict user visibility based on permissions
- Creating, updating, and deleting resources.

The resource [3] shows the REST API design. Here are a few attributes required for a good web application architecture: solving problems consistently and evenly, maximum simplification, support of the latest standards, including A/B testing and analytics, fast response time, security standards, easy scaling, easy creation of known data, errors registered in a user-friendly way, automatic deployment. The above factors are necessary because, with the right attributes, you can create a better app. Not to mention, by supporting horizontal and vertical growth, software deployment is much more efficient, convenient and reliable.

According to the resource [4], the REST architecture allows for the above build requirements. REST-compliant systems often referred to as RESTful systems, are characterized by being stateless and separating client and server concerns. REST is easy to understand and implement, so it helps improve the productivity of your development team. There are 2 main reasons why REST helps make your application more scalable: stateless and caching. As we'll see in the following sections, one of the fundamental principles of REST is that it is stateless on the server side, so each request will be processed independently of previous ones. In the resource [5], the caching of REST services is explained. Caching is easier with REST. Caching is a critical factor in the scalability and performance of a modern web application. A well-tuned cache mechanism (with the best possible hit rates) can dramatically reduce your server's average response time. Since the server is stateless and each request can be handled individually, GET requests should generally return the same response regardless of the previous ones. REST is flexible. By flexible, I mean that it's easy to modify, and it can respond to many clients that may request different types of data (XML, JSON, etc.). The client can specify the type using the Accept header, and the REST API can return different responses depending on that.

The article further shows the construction of a REST system based on the forum [6]. An Internet forum is an online discussion site where people can have conversations in the form of posted messages. Forums make it easy to organize conversations by specific categories or topics and browse trending or recent content. Forums are good for building a REST system because there will be many requests for data, topic updates, many similar requests that can be cached, etc.

II. OVERVIEW OF KNOWN METHODS AND TOOLS FOR SOLVING PROBLEMS

Currently, web applications running on remote APIs or services, service-oriented applications or service compositions, mobile applications built on cloud services, and similar web technologies are the most advanced. The resource [7] explains the opportunities for REST to be used for automation. Two main types of remote programming resources have emerged over the years: SOAP/WSDL web services and REST APIs. While the former can rely on a very rich set of standards and reference specifications, and developers know well how to use WSDL to describe a service and SOAP to communicate with clients, REST APIs have not experienced this kind of standardization (we specifically refer to JSON/XML API). The resource [8] shows the basic principles of the REST architecture:

• Addressing the resources. APIs manage and expose resources representing domain concepts; each resource is uniquely identified and addressed by a corresponding Uniform Resource Identifier (URI).

• Presentation of resources. Clients do not directly know the internal format and state of resources; they work with resource representations (such as JSON or XML) that represent the current or expected state of the resource. Declaring content types in HTTP message headers allows clients and servers to handle representations correctly.

• Unified interface. Resources are accessed and managed using standard methods defined by the HTTP protocol (GET, POST, PUT, etc.). Each method has its own expected, standard behaviour and standard status codes.

• Stateless. The interaction between the client and the API is stateless, meaning that each request contains all the information needed to process the API; the server does not store the state of the interaction.

• HATEAOS (Hypermedia as the Engine of Application State). Resources as concepts of the subject area can be related to other resources. Relationships between resources (embedded in their representations) allow clients to discover and navigate relationships and maintain an interaction state.

According to the resource [9], about two-thirds of REST APIs (64 %) have from 2 to 50 operations. Approximately half of the analyzed services (56.2 %) provide no more than 20 operations, and about a third offer from 2 to 10 operations. In the analyzed set, 5.2 % provided a huge number of transactions (more than 200), ranging from social platform services such as Facebook.com to trading sites such as ebay.com Sell API. REST web service APIs typically provide a relatively small number of operations. The output format selection function refers to how the output format of the service is selected. The results show that almost half (45 % - 223 APIs) of the 500 APIs support only one output format and do not require any output format selection. In other cases, the most popular way to select the output type is in the HTTP header field (24.0 %). Then 2.4 % use the URI path, 13.6 % use the URI suffix, and 15.4 % use the query parameter to specify the format.

The resource [10] lists the different stages of API development:

• Design — The main goal here is to define the shape of the API, document interfaces, and provide stub endpoints. Platforms such as Swagger and Apiary are used for design.

• Testing - Here we perform functional testing of the API by sending a request and analyzing the response at different levels of visibility, namely application, HTTP and network. Postman, Wireshark, cUrl, Burp suite and others are usually used.

• Web Hosting - When deploying to the web, there are HTTP tools to help host APIs for better performance, security, and reliability. HTTP Caching, DNS, and TLS are used.

• Productivity. Before we go into production, we use API performance testing tools that tell us what kind of load the API can handle. Loader.io and others are suitable for performance measurement.

• Monitoring — After API deployment, testing ensures the overall health of live APIs and alerts us if any issues arise. For this, we use logging services: Splunk, ELK, Zipkin, Prometheus and others.

• Management. Finally, we'll look at some tools for API management activities like traffic shaping, blue-green deployment, and more. We use API Gateway.

According to the resource [11] to get baseline API performance, you can run different kinds of load tests with increasing load, measured in requests per second, to find performance metrics quantified by errors and response times:

• The default test is an average load over a long period, for example, running for 48 hours at 1 request per second. This will reveal any memory leaks or such other hidden bugs.

• Load test - peak load, e.g. run 2k requests per second with 6 API instances.

• Stress test — exceeding the peak load, for example, execution of 10,000 requests per second for 10 minutes.

It also allows us to determine the infrastructure that will allow us to deliver the API with the desired performance metrics and whether our solution will scale linearly. Once the API is deployed, it doesn't mean we can forget about the API. Deployment to production initiates another phase of testing—production testing, which can uncover issues that went undetected in previous phases. Production testing includes a set of activities grouped as observability that includes logging, monitoring, and tracing. The tools for these activities will help us diagnose and solve problems found in production.

In the resource [12] the implementation of logging in the ASP .NET Core is explained. Logging - Developers should explicitly log using their preferred logging structure and logging standard. For example, one log statement for every 10 or more lines of code if the code is complex with log levels split as - 60 per cent DEBUG, 25 per cent INFO, 10 per cent WARN, and 5 per cent ERROR.

Monitoring - Monitoring is done at a higher level than logging. While logging tells us what's happening with the API,

monitoring ensures the overall health of the API using general metrics provided by the platform and the API itself. Metrics are typically accessed by an agent deployed on a server or may be part of a solution and periodically collected by a monitoring solution deployed remotely.

In the resource [13] the tracing of web API is explained. Diagnostic endpoints can be included in the solution to tell us about the overall health of the API. Tracking - Zipkin is a distributed tracking system. This helps collect the timing data needed to troubleshoot latency issues in microservices architectures. Enabling centralized logging covers both logging and tracing. For monitoring, metrics of interest can be stored in a time series repository like Prometheus and visualized with Grafana. Further information is taken from the resource [14].

REST uses HTTP methods - methods or actions that are available to interact with resources on the server. The limited number of verbs in RESTful systems is confusing and frustrating for people unfamiliar with the approach. Further information is taken from the resource [15]. What appears to be arbitrary and unnecessary restrictions are intended to encourage predictable program-independent behaviour. By clearly defining the behaviour of these methods, customers can make independent decisions in the face of network outages and failures.

There are four main HTTP verbs used by developed RESTful systems.

GET

The GET request is the most common method on the Internet. A GET request transfers a representation of a named resource from the server to the client. Although the client does not know anything about the resource it requests, the request returns a stream of bytes with metadata tags indicating how the client should interpret the resource. This is usually displayed on the web as "text/HTML" or "application/xhtml+xml". One of the key points about a GET request is that it doesn't have to change anything on the server side. This is essentially a secure request. GET requests must also be idempotent. This means that submitting a request more than once will have no consequences. This is an important property in a distributed network infrastructure. If the client is interrupted while performing a GET request, it should be authorized to resend it via verb idempotency. This is an extremely important point. In a well-designed infrastructure, it doesn't matter what the client requests from which application. There will always be characteristic application behaviour, but the more we can dive into non-application behaviour, the more resilient and lightweight our systems will be.

POST

The situation becomes a little less clear if we consider a set of POST and PUT requests. Based on their definitions, both appear to be used to create or update a resource from the client to the server, but they serve different purposes. POST is used when the client cannot predict the ID of the resource it is requesting to create. When we hire people, place orders, submit forms, etc., we can't predict what the server will call these resources we create. That's why we publish the resource view to the handler. The server will accept the input, validate it, verify the user's credentials, etc. After successful processing, the server will create the resource.

POST can also be used to update a known resource, such as adding a new shipping address to an order or updating the quantity of an item in the cart. Because of this potential for partial updates, POST is neither secure nor idempotent. A final common use of POST is to send requests. Either the request representation or URL-encoded form values are passed to the service to interpret the request. It is usually fair to return results directly from this type of POST, as there is no identification associated with the request.

PUT

Many developers largely ignore the PUT method because HTML forms do not currently support it. However, it serves an important purpose and is part of the overall vision of RESTful systems. A client can send a PUT request to a known URL to pass the view back to the server to perform an overwrite operation. This distinction allows a PUT request to be idempotent in a way that a POST update is not. If a client is in the process of overwriting a PUT and it is interrupted, it may feel empowered to do it again because the overwrite action can be re-executed without consequence; the client is trying to control the state, so it can simply override the command. PUT can also be used to create a resource if the client can predict the identity of the resource. This is usually not the case, as we discussed in the POST section, but if the client controls the information spaces on the server side, it is reasonable to allow it.

DELETE

The DELETE method is not widely used on the public network, but for information spaces that you control, it is a useful part of the life cycle of a resource. DELETE requests must be idempotent. A DELETE request may be interrupted by a network failure. Regardless of whether the request was successfully processed on the first request or not, the resource should return a 204 (No Content). Additional processing may be required to track previously deleted resources and resources that never existed (in which case the API should return 404 Not Found). Some security policies may require that 404 be returned for non-existent and deleted resources so that DELETE requests do not miss resource availability information.

According to the resources [16], REST assumes a client/server architecture—a computing model in which a server hosts, supplies, and manages most of the resources and services consumed by a client. This type of architecture has one or more client computers connected to a central server via a network or the Internet. This system shares computing resources. Also, the resource [17] shows how the client-server architecture needs to be built. A client/server architecture is also known as a network computing model or a client/server network because all requests and services are delivered over a network.

III. THE GOAL OF THE WORK

The purpose of the work is to speed up the processing of requests in forums by implementing them based on REST requests, using the example of the implementation of a student discussion forum. The advantages of the REST system are given. Apart from that, the goal is to research and analysis of REST systems. Analysis of the architecture and principles of

construction of such systems. A Restful system must support a unified interface, client-server architecture, multi-layer system, caching, and non-constancy.

IV. AN OVERVIEW OF THE REST ARCHITECTURAL STYLE, ITS ADVANTAGES AND DISADVANTAGES

REST systems must satisfy the 6 main guidelines or constraints of the RESTful architecture. These are:

• Unified interface: By applying the principle of commonality to the component interface, we can simplify the overall system architecture and improve the visibility of interactions. Various architectural constraints help achieve a uniform interface and control the behaviour of components. The following four constraints can achieve a uniform REST interface:

1. Resource identification – the interface must uniquely identify each resource involved in the interaction between the client and the server.

2. Manipulation of resources using representations - resources must have unified representations in the server response. API consumers must use these views to change the state of resources on the server.

3. Self-descriptive messages - Each resource representation must contain enough information to describe how to process the message. It should also provide information about additional actions that the client can perform on the resource.

4. Hypermedia as an application state mechanism - the client should only have the initial URI of the application. The client application must dynamically manage all other resources and interactions using hyperlinks.

• Client-server architecture. Both the client and the server have different sets of challenges. The server stores and/or manipulates information and effectively provides it to the user. The client takes this information, displays it to the user, and/or uses it to fulfil subsequent requests for information. This separation of concerns allows both the client and the server to evolve independently, as it only requires the interface to remain the same.

• Indecisiveness. This means that the communication between the client and the server always includes all the information necessary to fulfil the request. There is no session state on the server, it is stored entirely on the client side. If authentication is required to access the resource, then the client must authenticate itself with each request.

• Caching. The cacheable constraint requires that the response implicitly or explicitly designates itself as caching or non-caching. If the response is cached, the client application gets the right to reuse the response data later for equivalent requests and a certain period.

• Multi-layered system. Individual components cannot see beyond the immediate layer with which they interact. This means that a client connecting to an intermediate component, such as a proxy, is unaware of what lies beyond it. This allows the components to be independent and thus easily replaced or extended.

• Code on demand (optional). REST also allows to extend the functionality of the client by loading and executing code as scripts. Downloaded code simplifies the customer experience by reducing the number of features that need to be implemented upfront. Servers can provide some of the functionality that is provided to the client as code, and the client only needs to execute the code.

Advantages of REST API:

• REST API is easy to understand and learn due to its simplicity.

• With the REST API, you can organize complex applications and facilitate the use of resources.

• High load can be managed with HTTP proxies and cache.

• REST APIs are easy to explore and discover.

• It makes it easier for new customers to work with other applications, whether or not it is purpose-built.

• Use standard HTTP procedure calls to retrieve data and requests.

• REST API depends on code, and can synchronize data with websites without any complications.

• Users can access the same standard objects and data model compared to SOAP-based web services.

• It provides format flexibility by serializing data in XML or JSON format.

• It allows to use standard security using OAuth protocols to validate REST requests.

Disadvantages of REST API:

• The biggest problem with REST APIs is the multi-endpoint nature. They require clients to make multiple round trips to retrieve their data.

• REST is a lightweight architecture, but it is not suitable for working in complex environments.

• REST requests (especially GET) are not suitable for large amounts of data.

• Oversampling is a waste of network and memory resources for both the client and the server.

V.COMPARISON OF REST ARCHITECTURAL STYLE WITH OTHER ARCHITECTURES

Alternative technologies to REST for building SOA-based systems or building APIs for calling remote microservices include XML over HTTP (XML-RPC), CORBA, RMI over IIOP, and Simple Object Access Protocol (SOAP) [9], [10]. In general, each technology has advantages and disadvantages. REST insists that the best way to implement networked web services is to use the underlying design of the network protocol itself, which in Internet terms is HTTP. This is an important component because REST is not just for the web; rather, its principles are intended to apply to all protocols, including WebDav and FTP. Fig. 1 shows an illustration of a sample client-server architecture.

Two competing styles for implementing web services are REST and SOAP. The fundamental difference between them lies in the philosophical approach to distant challenges. REST takes a resource-based approach to web interactions. With REST, you find a resource on the server and decide to update that resource, delete it, or get some information about it. With SOAP, the client does not decide to interact with a resource directly but instead calls a service, and that service provides access to various objects and resources behind the scenes. SOAP has also spawned a large number of HTTP-based

frameworks and APIs, including the Web Services Description Language (WSDL), which defines the structure of the data that is passed back and forth between a client and a server. Some problem domains are well served by being able to explicitly define the message format or can benefit from using various SOAP-related APIs, such as WS-Eventing, WS-Notification, and WS-Security. There are times when HTTP cannot provide the level of functionality that an application may require, and in these cases, using SOAP is preferable. Fig. 2 shows the difference between SOAP and REST [11].

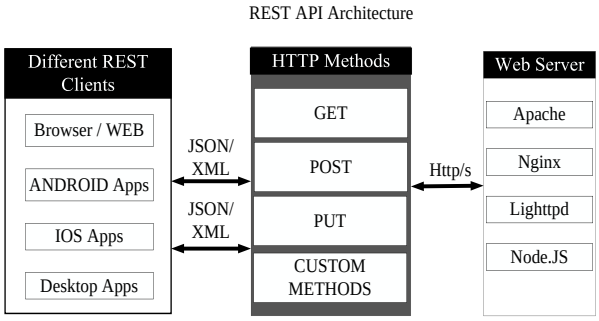

Fig. 1. The working principle of the REST architectural style

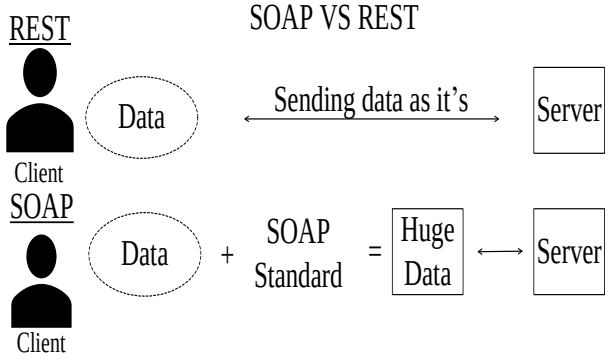

Fig. 2. Difference between REST and SOAP

1) Difference between REST and gRPC.

gRPC stands for Google Remote Procedure Call and is a variant based on the RPC architecture. This technology follows the RPC API implementation, which uses the HTTP 2.0 protocol, but HTTP is not exposed to either the API developer or the server. So, there is no need to worry about how RPC concepts map to HTTP, which reduces complexity. In general, gRPC aims to make data transfer between microservices faster. It is based on the approach of defining a service, setting methods and appropriate parameters to enable remote invocation and return types.

Now that we have an overview of gRPC and REST, let's look at their main differences.

HTTP 1.1 vs HTTP 2

REST APIs follow a request-response communication model that is typically built on top of HTTP 1.1. Unfortunately, this means that if a microservice receives multiple requests from multiple clients, the model must handle each request one at a time, which therefore slows down the entire system. However, REST APIs can also be built on top of HTTP 2, but the request-response communication model remains the same, preventing REST APIs from taking full advantage of HTTP 2's benefits, such as streaming and bidirectional support. gRPC faces no such obstacles. It is built on top of HTTP 2 and instead follows a client response-based communication model. These conditions support bidirectional communication and streaming due to gRPC's ability to receive multiple requests from multiple clients and process those requests concurrently, continuously streaming information. In addition, gRPC can also handle " unnecessary" interactions, such as those built on HTTP 1.1.

Browser support

This aspect is probably one of the main advantages of the REST API over gRPC. On the one hand, REST is fully supported by all browsers. On the other hand, gRPC is still quite limited when it comes to browser support. Unfortunately, conversion between HTTP 1.1 and HTTP 2 requires gRPC-web and a proxy layer.

Payload data structure

As it has been mentioned earlier, gRPC uses a protocol buffer by default to serialize data. This solution is easier because it provides a highly compressed format and reduces the size of messages [12]. Also, the Protobuf (or protocol buffer) is binary; thus, it serializes and deserializes structured data to transfer it. In other words, strongly typed messages can be automatically converted from Protobuf to client and server programming languages. In contrast, REST primarily relies on JSON or XML formats to send and receive data. In fact, despite not requiring any structure, JSON is the most popular format due to its flexibility and ability to send dynamic data without necessarily following a strict structure. Another important advantage of using JSON is its level of readability, which Protobuf cannot yet compete with.

Features of code generation

Unlike gRPC, the REST API does not provide built-in code generation functionality, which means that developers must use third-party tools such as Swagger or Postman to generate code for API requests. In contrast, gRPC has code generation capabilities due to its protocol compiler, which is compatible with several programming languages. This is especially useful for microservice systems that integrate different services developed in different languages and on different platforms.

2) Difference between GraphQL and REST

The main difference between GraphQL and REST API is that GraphQL is a specification, a query language, while REST is an architectural concept of network software. GraphQL is great for being well-typed and self-documenting based on schema types and descriptions and integrates with code generator tools to reduce development time. Looking at one of the most well-known differences – the difference in expected responses to requests – in very simple terms, we can think about the process of ordering hamburgers. Although the GraphQL meme-burger has been around for a while, the clarification it provides still makes the concepts easier to understand.

Imagine you walking into a hamburger restaurant and ordering a cheeseburger. No matter how many times you order (call your RESTful API), you get every ingredient in that double cheeseburger every time. It will always be the same shape and size (which is returned in the RESTful response). With GraphQL, you can "make it your own" by describing

exactly what you want that cheeseburger to be. Now you can make a cheeseburger (answer) as a bun on top, followed by a patty, pickles, onions and cheese, without the bottom bun. A REST API is an "architectural concept" of networking software. GraphQL, on the other hand, is a query language and set of tools that work on a single endpoint. Additionally, REST has been used to build new APIs over the past few years, while GraphQL's focus has been on optimizing for performance and flexibility. Fig. 3 shows the difference between REST and GraphQL.

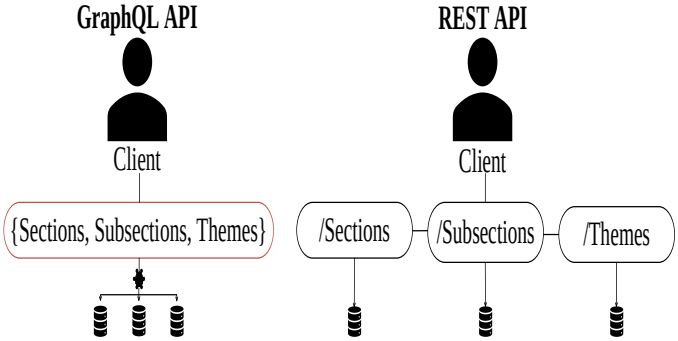

Fig. 3. Differences between REST and GraphQL

VI. AN EXAMPLE OF REST SYSTEM IMPLEMENTATION ON THE EXAMPLE OF A STUDENT FORUM

All forums are built according to approximately the same scheme. They will include certain sections where users communicate, and exchange information and opinions on a certain topic. The general structure of the forums looks something like this: Sections -> Topics -> User Replies. Consider three main types of forums:

• discussion - discussion forums are the most traditional type of forum: after creating a thread on the forum, other participants can reply and continue the conversation, enter text in the reply field and click the "Publish" button to participate.

• Questions (Q&A) – The Q&A format represents the initial posting of a thread as a question that requires an answer and provides respondents with a field to answer and a suggestion as an answer checkbox.

• QA and Discussion – Both questions and traditional discussions are enabled for this type of forum thread.

Most forums provide functionality for user registration, logins, roles, moderation, etc. However, the main difference between all forums remains mainly the topic. Moreover, it can be diverse from medicine to electronics. The most popular topics are development and technology (Adobe, StackOverflow, Subsekt, etc.), medicine (Doctor's Lounge, Health24, eHealth, Patient.Info, etc.), gaming (Steam Discussions, GameFAQs, IGN Boards, GameSpot Boards, etc.) and also forums on various topics that allow users to create topics of interest to them, for example, Reddit, Quora and others. According to the topic, forums can also add their specific functionality. For example, medical forums, communication with a doctor, the location of pharmacies, etc., or technical forums that allow to collect the necessary equipment, repair equipment, etc. Role support for forums is also widespread: administrator, moderator, OP (opener of the post) and regular user, etc. Fig. 4 shows the typical forum structure. Fig. 5 shows the possible additional functions of the forum.

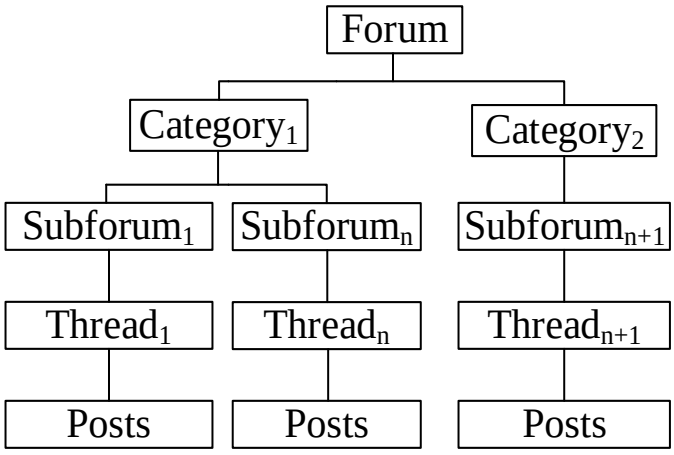

Fig. 4. Typical forum structure

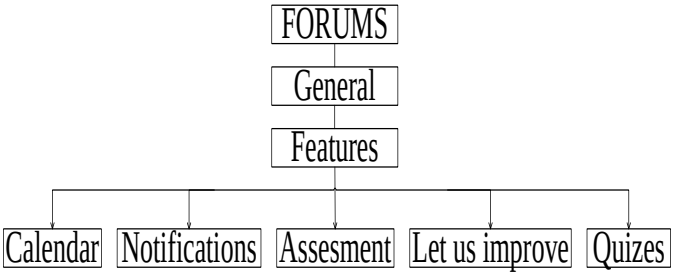

Fig. 5. Some of the possible additional functions of the forum

A special feature of the developed forum is its topic - student discussions, which is quite relevant in the conditions of distance learning. It looks something like this: Sections-> Subsections-> Topics-> Posts. Roles are also available: teacher, student, user, and admin. Additional functionality is a calendar in which a student or teacher can set the necessary events, receive notifications, and give access to other people. There is also the possibility to organize ZOOM conferences using the ZOOM API. Fig. 6 shows the structure of the developed forum.

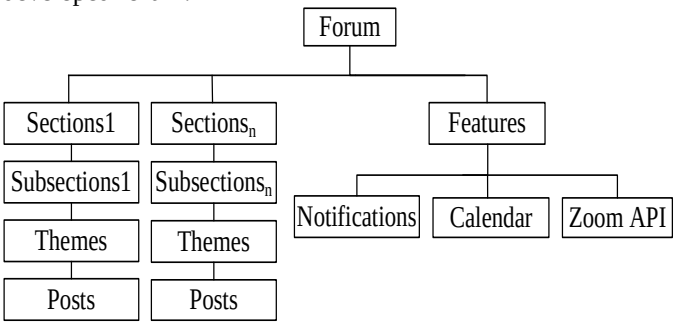

Fig. 6. Structure of the developed forum

Creating a database for the REST system

Both SQL and NoSQL databases are suitable for REST systems [13]. From SQL databases you can use MSSQL, Oracle, Sybase, MySQL, Postgres and others, from NoSQL

suitable Aerospike, Google Cloud Bigtable, MarkLogic, Couchbase, Amazon DynamoDB, DreamFactory, AnyChart and others. To create a database, we use the MS SQL relational database. Using Entity Framework Core is a lightweight, extensible, cross-platform version of the popular Entity Framework data access technology. EF Core can serve as an object-relational mapper (O/RM) that allows .NET developers to interact with the database using .NET objects. It eliminates the need for most of the data access code that would normally need to be written. With EF Core, data is accessed using a model. The model consists of entity classes and a context object representing a database session. A context object allows to query and store data. After successfully creating the database, it needs to be connected to the server using the connection thread. The Microsoft Identity library is used to store user data, login and registration. The structure of the database is indicated. The query methodology is based on CRUD - Create, Read, Update, Delete.

Creation of the server part of the application

NodeJs, Spring, Django, Larabel, Ruby on Rails, ASP.NET and other frameworks are suitable for building APIs. The architecture of the server part is shown in Fig. 7.

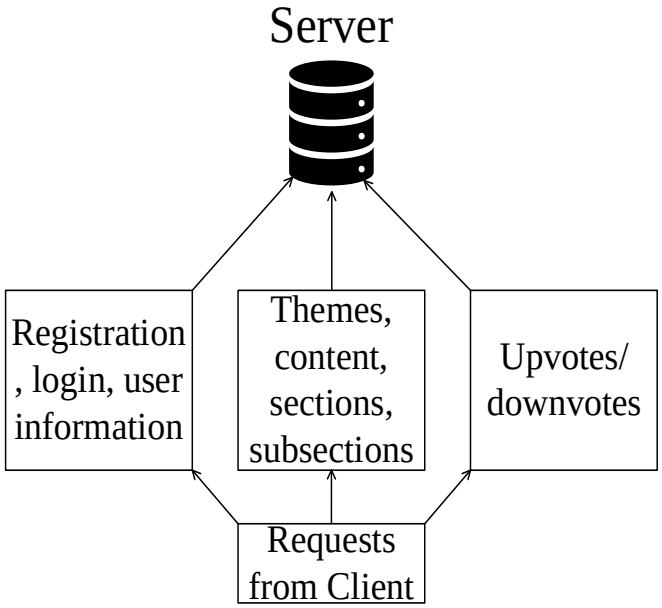

Fig. 7. Architecture of the server part

Configure the server and dependencies as follows [14], [15]. We create the server using ASP.NET Core Web API technology to create a full-featured REST API. Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic to the same machine. We add the model class and the database context. Then we assemble the controller using CRUD (Create, Read, Update, Delete) methods. After that, we configure routing, URL paths, and return values. The server starts on port 44381, based on the IIS (Internet Information Services) web server. Fig. 8 shows the architecture of the server part.

The system is implemented based on CRUD operations (Create, Read, Update, Delete) [16]. Let's consider queries using topics as an example. To obtain a theme, a GET theme/{id: int} request is used, where it is the theme identifier, to add, respectively, a POST theme/ with a request body that specifies the model of the theme that the user is adding. For modification, PUT with the request body specifying the modified model of the topic that the user is adding, and DELETE with the topic ID accordingly. Accordingly, there is a client/server separation where the client and server can operate independently of each other without knowing about each other. This means that client-side code can be changed at any time without affecting server performance, and server-side code can be changed without affecting client performance.

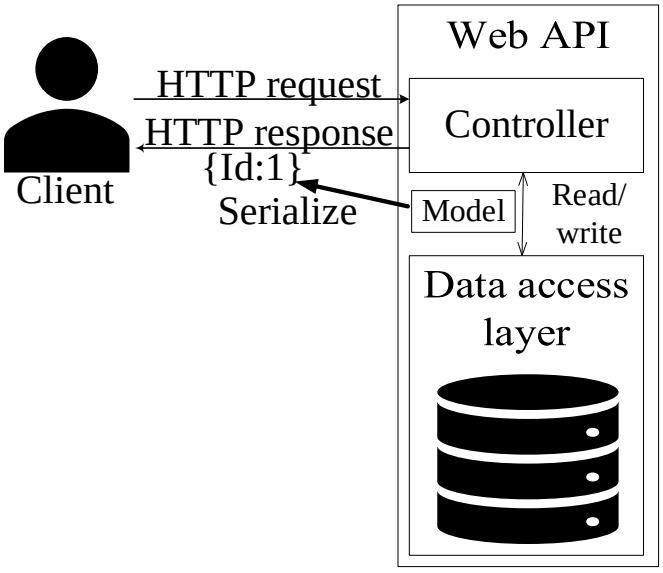

Fig. 8. Application design

Resources have the same representation in the server response. API consumers use these models to change the state of resources on the server. The client only has the application's initial URI, the client dynamically manages all other resources and interactions through hyperlinks. The interface uniquely identifies each resource involved in the client and server interaction. Each request from the client to the server contains all the information necessary to understand and execute the request. The server cannot use any previously stored context information on the server. For this reason, the client program completely preserves the session state. Caching is present, which implies that the server's response implicitly or explicitly marks itself as cached or uncached. If the response is cached, the client application gets the right to reuse the response data later for equivalent requests and a certain period. The system is divided into three layers: Business Logic Layer (BLL), Data Access Layer (DAL) and Presentation Layer (PL). Components cannot see beyond the immediate layer they interact with. The Business Logic Layer is a layer that manages the communication between the end-user interface and the database, the main components of which are work processes and business rules. A DAL is a layer that provides simplified access to data stored in some persistent storage, such as an object-relational database. The presentation Layer

is the highest level of the program where the user performs his activities. Consider an example of any application where the user needs to fill out a form. This form is nothing more than a presentation layer. Basically, at this level, user input is checked and rules are processed.

Testing of the developed REST system and comparison with others.

Testing was done by sending a certain number of requests to our REST system and the systems of other architectural styles and comparing the results. Requests are processed by the server and then a response is generated. Processing the request means that the server parses the requested URL, retrieves the file name from it, looks for that file in the directory, and generates a response code according to the result, i.e. 404 for file not found and 200 if the file is found. After finding a file in the directory, it is written to a socket, which is later displayed in browsers. During testing, the number of concurrent requests sent to the server and the results was noted and compared to the SOAP architecture. Results are shown in Tables 1 and 2 and displayed in Fig. 9.

Table 1.

REST web service performance analysis

Number,*i*	Number of requests	Number of responses	Number of successful responses	The percentage of failed requests, %	Average time of execution (ms), a_i
1	60	60	60	0.0	30
2	100	100	100	0.0	33
3	130	130	130	0.0	35
4	150	150	150	0.0	36
5	170	170	170	0.0	45

Table 2.

SOAP web service performance analysis

Number,*i*	Number of requests	Number of responses	Number of successful responses	The percentage of failed requests, %	Average time of execution (ms), b_i
1	60	60	60	0.0	31
2	100	100	100	0.0	36
3	130	130	130	0.0	41
4	150	150	150	0.0	47
5	170	170	170	0.0	52

The formula for looking at how much our developed REST system is faster than a similar SOAP in percentage:

$$Cs = \left| 100 \times \frac{\frac{1}{n}\sum_{i=1}^{n} a_i}{\frac{1}{n}\sum_{i=1}^{n} b_i} - 100 \right| . \quad (1)$$

Using (1), where a_i is the average REST request execution time for test sequence number (i), b_i is the average SOAP request execution time for test sequence number (i), and n is the download sequence number, we calculate the percentage of how much our developed REST system is faster than a similar SOAP system – C_s. After calculating the result, we get a value of 13.5 %, which means that, on average, the average execution time in our developed REST system is 13.5 % faster than in a similar SOAP system.

Due to the large size of XML files, SOAP services require a lot of bandwidth. The ability to use REST with different file types is an advantage of this architectural style. In particular, JSON (JavaScript Object Notation) is useful because it is a lightweight data exchange format, easy for humans to read and write, and easy for machines to parse, serialize, and generate.

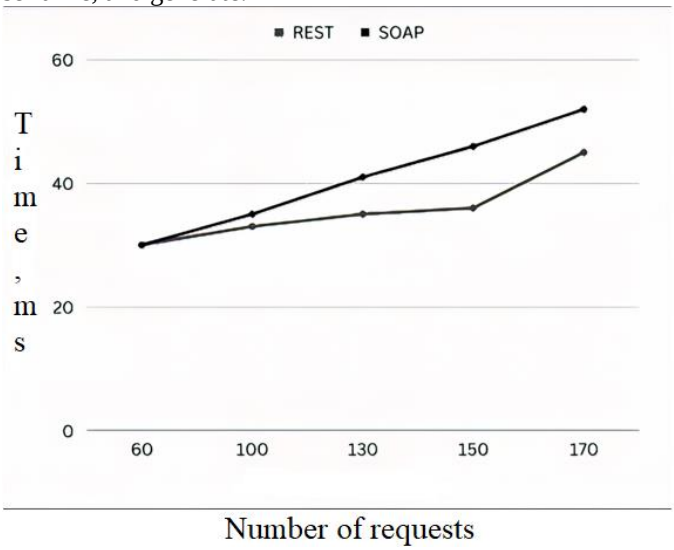

Fig. 9. Display of performance results

VII. CONCLUSION

The processing of requests in forums was accelerated by implementing them based on REST requests, using the example of implementing a student discussion forum. The advantages of the REST system were given. In addition, REST systems were studied and analyzed. An analysis of the architecture and principles of construction of such systems was conducted. The Restful system supports a unified interface, client-server architecture, multi-tier system, caching and volatility. The advantages and disadvantages of the REST architectural style compared to other styles were listed. Features of the forum were introduced to the REST system. The speed of architectural styles for the implementation of web applications was studied. The implementation of REST web applications was demonstrated in the example of a student forum. A forum based on the REST system was implemented. By and large, the average execution time in our developed REST system was 13.5 % faster than in a similar SOAP system, which is quite a high result if we care about the overall performance of the system. Also, this value will be even larger for larger response file sizes, as JSON is potentially smaller and lighter than XML in SOAP. The REST architectural style supports a large number of file types and is easy to learn. The developed REST system of the student discussion forum also implemented features that distinguish the system from others. A calendar was implemented that allows teachers and students to add and monitor important events, it works with CRUD (CREATE, READ, UPDATE, DELETE) operations. A notification system using RabbitMQ was also created, which would process and send notifications to forum users regarding their specified events in the calendar, including using the ZOOM API for organizing meetings. Simple survey systems using GET, POST requests, and rating systems were implemented. Analysis and research of REST systems were conducted. It was considered what REST API

was, why it was worth using this architecture, its principles, and the nuances of building such systems. The advantages and disadvantages of the REST architectural style and a comparison with other styles were given. A Restful system must support a unified interface, client-server, multi-layered system, caching and restlessness. It was described what technologies could be successfully used for the REST system, etc. Features and stages of building a REST system were provided. An example of a REST system implementation based on a forum web application was described. A description of the tools used during the development of this system was given, with all the advantages and disadvantages of using certain technologies and the system in general. A description of the development environment and some of its features were provided. The peculiarity of this system was that the issue of education and discussion, finding the necessary information online, currently occupied an important place. So, based on the purpose of the work, it can be said that the task that had to be performed was completed, which means that this system is relevant in this field.

References

[1] Lokesh Gupta, (2022). What is REST? Restful API, pp.1–5. Available at https://restfulapi.net/ (Accessed: 27 October 2022).

[2] Saifulfiul Tarek, (2020). What is a RESTful API (REST API) and How Does it Work? Namespace, pp. 1–4. Available at https://namespaceit.com/blog/what-is-a-restful-api-rest-api-and-how-does-it-work (Accessed: 27 October 2022).

[3] L. Li and W. Chou, (2011). "Design and Describe REST API without Violating REST: A Petri Net Based Approach,". IEEE International Conference on Web Services, pp. 508–515, DOI: 10.1109/ICWS.2011.54.

[4] Randhir Singh, (2019). Developing REST APIs. DZONE, pp. 1–9. Available at https://dzone.com/articles/developing-rest-apis (Accessed: 27 October 2022).

[5] Jamie Kurtz, Brian Wortman, (2014). Designing the Sample REST API. ASP.NET Web API 2: Building a REST Service from Start to Finish, pp. 21–29. DOI: 10.1007/978-1-4842-0109-1_3

[6] L. Li, W. Chou, W. Zhou and M. Luo, (2016). "Design Patterns and Extensibility of REST API for Networking Applications," in IEEE Transactions on Network and Service Management, vol. 13, no. 1, pp. 154–167, March 2016, DOI: 10.1109/TNSM.2016.2516946.

[7] Brian Sletten, Chase Doelling, (2019). Foundations of RESTful Architecture. DZONE, pp. 1–
9. Available at https://dzone.com/refcardz/rest-foundations-restful?chapter=1#section-4 (Acces-sed: 27 October 2022).

[8] S. Stoudenmier and A. Olmsted, (2017). "Efficient retrieval of information from hierarchical REST requests,", 12th International Conference for Internet Technology and Secured Transactions (ICITST), pp. 452–454, DOI: 10.23919/ICITST.2017.8356445.

[9] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali & Gianraffaele Percannella, (2016). REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. Lecture Notes in Computer Science vol. 9671, pp. 21–39. DOI: 10.1007/978-3-319-38791-8_2.

[10] Christian Nagel, (2018). Web API. Professional C# 7 and .NET Core 2.0, vol. 32, pp. 1039–
1080. DOI:10.1002/9781119549147.ch32.

[11] L. Li, W. Chou, W. Zhou and M. Luo, (2016). "Design Patterns and Extensibility of REST API for Networking Applications," in IEEE Transactions on Network and Service Management, vol. 13, no. 1, pp. 154–167, DOI: 10.1109/TNSM.2016.2516946.

[12] H. Garg and M. Dave, (2019). "Securing IoT Devices and SecurelyConnecting the Dots Using REST API and Middleware," 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU), pp. 1–6, DOI: 10.1109/IoT-SIU.2019.8777334.

[13] S. M. Sohan, F. Maurer, C. Anslow and M. P. Robillard, (2017). "A study of the effectiveness of usage examples in the REST API documentation,", IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2017, pp. 53–61, DOI: 10.1109/VLHCC.2017.8103450.

[14] A. Hasibuan, M. Mustadi, I. E. Y. Syamsuddin and I. M. Anis Rosidi, (2015). "Design and implementation of modular home automation based on the wireless network, REST API, and WebSocket," International Symposium on Intelligent Signal Processing and Communication Systems (PACS), pp. 362–367, DOI: 10.1109/ISPACS.2015.7432797.

[15] K. Boonchuay, Y. Intasorn and K. Rattanaopas, (2017). "Design and implementation a REST API for association rule mining,", 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), pp. 668–671, DOI: 10.1109/ECTICon.2017.8096326.

[16] Artur Britvin, Jawad Hammad Alrawashdeh, Rostyslav Tkachuck, (2022). Client-Server System for Parsing Data from Web Pages in Advances in Cyber-Physical Systems, vol. 7, no. 1, pp. 8–14. DOI: https://doi.org/10.23939/acps2022.01.008

[17] Andy Neumann, Nuno Laranjeiro, Jorge Bernardino, (2021). An Analysis of Public REST Web Service APIs. IEEE Transactions on Services Computing, vol.14, pp. 1–8. DOI:10.1109/TSC.2018.2847344.

Ivan Zholubak is a Senior Lecturer of the Computer Engineering Department at Lviv Polytechnic National University, Ukraine. He graduated from Lviv Polytechnic National University with an engineering degree in Computer Engineering in 2013. In 2016 he graduated from Postgraduate courses at Lviv Polytechnic National University, Department of Computer Engineering. He has scientific, academic and hands-on experience in the field of algorithms for hardware data protection in cryptography, robotic systems and AI. He is the author of 7 scientific papers.

Bohdan Marii received his B.S. degree in Computer Engineering at Lviv Polytechnic National University, Ukraine, in 2022. His research interests include the architecture, patterns and development of web applications, SQL databases, and programming and technologies for web programming.