Principles of securing RESTful API web services developed with python frameworks

Principles of securing RESTful API web services developed with python frameworks

D V Kornienko, S V Mishina, S V Shcherbatykh and M O Melnikov

Bunin Yelets State University, 28, Kommunarov, 399770, Yelets, Russian

E-mail: dmkornienko@mail.ru

Abstract. This article discusses the key points of developing a secure RESTful web service API for keeping a student achievement journal. The relevance of using web services has been analyzed. The classification of web applications is given. The features of the Single Page Application architecture were considered. Comparative characteristics of architectural styles of application programming interfaces are given. Requirements to be met by RESTful API services are considered. The basic principles of API security were analyzed. A list of the main vulnerabilities that may appear during the development of the REST API is given. An overview of popular authentication schemes (methods) is given. Comparative characteristics of web frameworks of the Python programming language are given. The main tools used in the development of web API applications are listed. The process of creating a secure prototype of a RESTful web service API in Python using the Flask microframework and a tool for describing the Swagger specifications is presented. The process of configuring the application was examined in detail. The main recommendations for securing a web application, database and web server settings are listed. The key points of ensuring the protection of the developed web application are considered. The results obtained were analyzed.

1. Introduction

In the context of the global digitalization of the modern society, the trend of a massive transition of business solutions to online is becoming more and more noticeable [1]. Previously developed software is being rebuilt for the format of interaction on the Web. When creating new services, web solutions are preferred over mobile or desktop platforms. This transition is motivated by a more convenient and accessible mechanism of interaction between the software and the end customer [2]. Online resources do not impose strict hardware requirements on the user (storage memory, RAM size, central processing unit (CPU) power), do not tie the interaction with a specific type of the platform (personal computer, smartphone, specialized device). They organize the work exclusively through the Internet browser. In addition, the share of directly online businesses is growing, which also leads to increased attention and demand for websites and web applications [3].

However, the complexity of the logic of websites is constantly increasing. Therefore, ordinary static websites or dynamic solutions based on content management systems (Wordpress, Joomla, 1C-Bitrix) can no longer meet the needs of customers when it comes to non-linear platforms or e-commerce segment, and not trivial landing landings. Many questions arise both in functional terms and in the efficiency of the process of development, expansion, protection and maintenance of a software product. As a result, static websites have been almost completely replaced by fully functional web applications.

A web application is no longer a pre-assembled collection of HTML pages, style files, scripts, and media files. Web applications are based on a client-server architecture based on the separation of a functional block and appearance). The business logic of the application is performed on a special dedicated server (back-end), while the client interacts through the user interface (UI) in his Internet browser (front-end). This approach allows to generate HTML markup content in real time, with a minimum number of reloads on the viewed web page. Almost all modern e-commerce sites, social networks, instant messengers, online programs, forums, and search engines are web applications [4].

By the type of main components, web applications can be classified into several types: backend, frontend and SPA.

Backend applications (server-side applications) collect the main functionality and logic of the application, interact with the database. They are developed using a number of programming languages, the most popular of which are: Python, Ruby, PHP, C # (.NET), Node.js. However, it is almost impossible to implement production-ready solutions without the use of special tools called web frameworks. Each of the programming languages listed above has its own set of frameworks for creating web applications: Python (Django, Flask, FastAPI, Tornado), PHP (Laravel, Simfony, Yii2), Ruby (Ruby on Rails, Sinatra), Node.js (Express .js), C # (ASP .NET Core).

Frameworks take on some of the developer's responsibilities and help in ensuring application security, working with typical redirection, authentication, registration and other functionality. The application is deployed on a specially configured web server and independently generates the HTML representing pages (however, the browser will reload the page in most cases).

Frontend applications (client-side applications) work directly in the user's web browser. They are developed using JavaScript, TypeScript and Vue.js, ReactJS, AngularJS programming languages. They are applicable in cases when the application does not work with the database and does not store client information for longer than one session. They are fully responsible for the presentation of the HTML code.

SPA (Single Page Application) is a combination of solutions, dividing a web application into two (frontdend and backend) and organizing interaction between them. The backend part deals with logic, processes request, works with the database; the frontend part forms an external view based on the data received from the backend and displays the result in the browser. The exchange of information between the client and the server parts of the application, as a rule, occurs through a specially developed application programming interface (API). Almost all modern web applications are based on the SPA concept.

API is a set of methods and rules by which applications exchange messages and transfer data. As a rule, these are classes, functions, structures of one program, interacting with other programs. API not only helps organize interactions; it also plays an important role in securing that interaction. Such an interface hides the specifics of the service implementation and works within the framework of what the developers provide. Using the program interface of the application, the operations that are available to the user are defined, restrictions are imposed, input and output data are indicated [5]. In addition, API has a number of architectural styles that reflect certain aspects of the interface work.

Applications of this kind have complex logic and consist of many components, which noticeably affect the security of the system [6]. There are many factors to consider when organizing security. In addition to the front and back-ends security, one should focus on developing a secure API. Intrusion or an accidental client error can result in service unavailability, data loss, or complete application operability failure. This can critically affect the user experience, and as a result, bring massive losses to business. Therefore, ensuring security is a topical issue when developing any web service.

This article is devoted to the development of a secure RESTful API web service for generating a list of student achievements using the Python3 programming language and the Flask web microframework.

2. Materials and methods

The API implementation is based on the use of protocols and software specifications that build the syntactic and semantic rules for transmitted messages. Such specifications shape the architecture of API.

There are many styles containing standardization schemas and data exchange rules. The most common architectural styles are RPC (gRPC), REST, SOAP, and GraphQL.

The architectural style, represented by the RPC (Remote Procedure Call) specification, allows to call for a function remotely in the context of HTTP API. The client gets most of the capabilities when calling a particular procedure remotely, through personal formation of a special message for the server. The server deserializes the received message [7], starts the requested function and returns the result. gRPC is a new version of RPC designed to address the accumulated problems of the previous version of the specification. gRPC also adds a wide range of features like load balancing, message trace, and others. In addition, the new standard has started using the protobufs concept and runs on top of HTTP / 2. Protobuf is a proto file, which is a dictionary on the basis of which data is encoded and decoded in a special binary format. That is, it is a kind of new serialization format (alternative to JSON / XML) for the exchange of information between the client and the server [8]. The innovation allows to maintain high performance in an environment of heavy data exchange in real time. It is also possible to use SSL / TLS, OAuth 2.0, tokens for authentication, or write your own implementation of secure authentication.

Benefits of RPC (gRPC):

- simple interaction mechanism;
- the ability to add functions with subsequent conversion to the end point;
- high performance;
- the ability to interact with security tools.

Disadvantages of RPC (gRPC):

- low level of abstraction, which leads to difficulty in reuse;
- difficulties in scaling;
- problems with introspection of queries;
- reduced security due to the inability to completely hide the details of the API implementation.

RPC (gRPC) is the most applicable in the development of high-performance microservices [9] for internal communication, as well as in the creation of APIs for remote control systems, for example, in the field of IoT.

GraphQL is another specification introduced by the developers at Facebook. GraphQL is focused on data represention in a graph format. Unlike the creators of a tabular view from relational databases, the developers of GraphQL started from the document structure borrowed from document-oriented databases like MongoDB. The implementation is based on a graph of entities (nodes) that refer to each other (edges).

First of all, a schema is built that describes (using the special syntax of the SDL language) the exact queries and data types in the expected responses. A client with a drawn up scheme, even before sending a query, can make sure that the server has an answer to it. On the backend, the operation is interpreted according to the scheme and sends a response in JSON format, with the requested data. As a result, we have a technology that, based on the query scheme, can receive data from multiple endpoints and issue a single response in a specific form determined by the sender of the query. In addition, GraphQL supports a subscription mechanism that allows to receive information in real time from a web server.

Benefits of GraphQL:

- one query is enough to retrieve data of any complexity;
- availability of a typed query data schema;
- works effectively with linked data;
- detailed error log;
- flexible customization of queries (format definition, portioning, single version).

Disadvantages of GrapthQL:

- possible low performance if there are too many nested fields in one query;
- difficulties with caching;
- high learning curve, it is necessary to deal with SDL.

GraphQL is a great choice when you need to encapsulate the complex logic of the systems on which the API is developed, which is important when designing bulky projects and microservices. In addition, GraphQL provides an opportunity to work more efficiently with data on mobile devices (mobile API) by optimizing the payload (payload) of a single message.

SOAP or "Simple Object Access Protocol" is a protocol specification for building web communication. It is a W3C industry standard and uses the XML format as the underlying interoperability format. SOAP implements a stateful message passing that is used in transactions involving multiple parties or in complex secure transactions. Works with HTTP, SMTP and FTP protocols. SOAP integrates with WS-Security protocols, which guarantees the integrity and confidentiality of processed transactions with additional encryption [10]. By using XML, SOAP is the most verbose API style. Despite this, the XML interchange format has many implications that prevent SOAP from being a universal architectural style.

Benefits of SOAP:

- the ability to work with various transport protocols;
- implementation of built-in error handling (Retry messages);
- security enhancements.
- detailed message format (lots of metadata).

Disadvantages of SOAP:

- support for a single XML format;
- the need for a large service bandwidth due to the heavy weight of XML;
- requires deep knowledge of protocols and data transfer rules;
- a strict scheme slows down the addition and removal of properties in messages;
- high complexity of data parsing.

Thus, SOAP is most often used to provide reliable access within a company or directly with a client. Strong security capabilities make SOAP the preferred choice in the corporate and financial sectors.

REST (View State Transfer) is a style of API architecture for creating web services and services. It was presented in 2000 in a doctoral dissertation [11] by one of the founders of the HTTP protocol, Roy Fielding. REST is a simple interface for transferring information that does not use third-party software layers. That is, when sending data, there is no conversion stage, the information is delivered in its original form, which has a beneficial effect on the client load, but adds a load on the network part.

Working with data is organized in JSON or XML formats. A web service that meets all the requirements and conditions of the REST architecture is called a RESTful web service.

RESTful architectural requirements (Fielding Constraints):

- Not to contain state (stateless): the state for processing a query can only be in the query itself, so the server does not store any session information.
- Caching.
- Common Interface: Allows a consistent, application-independent interaction with the web server.

It imposes a number of sub-restrictions: manipulating resources through their representations, identifying resources, "self-sufficiency" of messages, working with hypermedia.

- Focus on client-server architecture.
- Ability to deliver executable code to the client side.
- Independence from the number of levels / layers of the application.

In REST, all communication is based on the use of HTTP methods: GET, POST, PUT, PATCH, and DELETE.

REST is most often used as a management API for CRUD (Create, Read, Update, Delete) to implement interaction with resources in lightweight scalable services. A resource is usually a data model object or a database table.

Benefits of REST:

- openness of interaction;
- simple implementation;
- data caching at the HTTP level;
- work with several formats of data presentation;
- stability due to the high level of abstraction.

Disadvantages of REST:

- lack of a unified standardized structure;
- high load on the network;
- excessive or insufficient information, which can lead to the need to send an additional query.

Of all the specifications, REST implements the highest level of abstraction and is best suited for developing simpler CRUD API. It maintains a balance of reliability and ease of use. The shift to the client-server, statless, and predisposition to caching (important for listing achievements, sorting, and filtering) are also advantages for services like the one discussed in this article. The large, in comparison with other specifications, the load on the network is leveled by the specifics of the problem being solved (one student with a low degree of probability will have thousands of achievements and add them to the platform in batches of several thousand. At the same time, the development of a separate mobile application is not planned). This is the reason why; it is REST that forms the basis of the architectural style of the project prototype being developed.

Creation of secure RESTful APIs also imposes certain standard requirements:

- Using the HTTPS protocol: cryptooperation is required to ensure the integrity of the transmitted data.
- Rate-limits: It is necessary to monitor the load on the API. Dropping requests in case of overload
- or connecting additional systems for balancing.
- Authentication: User / application / device identification.
- Audit log: Recording actions by creating an entry in the log file.
- Control of access rights (authorization): Determination of access rights for working with
- resources.
- Access to the business logic of the application.

When working with the REST architecture, it is customary to distinguish two levels of security, since the REST API is an interface for network interaction with a web application:

- the first level - getting access to API;
- second level - getting access directly to the application.

Protection of the API level implies proper organization of authentication, authorization, registration and separation of access rights. The developer must ensure that only authorized clients can use the API and only have access to authorized operations.

Application layer security includes vulnerability checking of service endpoints - the URL addresses responsible for interacting with the interface.

The OWASP (Open Web Application Security Project) community has identified (at various levels) about ten API development vulnerabilities. The most dangerous are:

- Broken Object Level Authorization: No separation and enforcement of the resource access
- control.
- Broken User Authentication: Vulnerabilities related to user authentication.
- Lack of Resources & Rate Limiting: Mismatch of checks and limits.
- Broken Function Level Authorization: Lack of the access control.
- Mass Assignment: Vulnerabilities related to deserialization of received objects.
- Security Misconfiguration: Errors in working with application security settings.
- Insufficient Logging & Monitoring: Lack or incorrect vision of the logs file and system monitoring.

First of all, it is necessary to pay attention to the problems associated with access control, authentication and authorization [12]. By design, REST API is stateless, so the access is controlled through local endpoints.

There are several the most common methods (authentication schemes): Basic Authentication, API Key, JSON Web Tokens, OAuth 2.0, Token-Base Authentication, Cookie-Based Authentication, and OpenID.

Basic Authentication (BA) is the simplest HTTP-based authentication scheme. The client or application forms an HTTP query, the header of which contains the "Authorization" field. A string of the form: Basic <username: password> (encoded in Base64) is passed as the value of this field. To ensure security and preserve the confidentiality of data, in combination with Basic Authentication (and all subsequent schemes), it is always necessary to use secure HTTPS / TLS protocols [13], since the user ID and password enter the Network in the form of plain text encoded in Base64 (which just amenable to decoding).

Cookie-Based Authentication is a method based on checking the content of cookies, inside which all the necessary information about the session is stored. The user initiates a login request. In case of successful login, the server sends a response, in the header of which the Set-Cookies field is included, containing the name of the cookie-field, value, cookie expiration, and so on. The next time the user needs to access the API, he will pass the value of the saved Cookie-field JSESSIONID with key "Cookie" in the request header.

API Key - a key in the form of a character string that the user passes along with the request to the server. The server will confirm the identity of the client if its key is contained in the application database. The key itself is issued by the application when registering a user. This scheme is used to protect against unauthorized access and allows to impose a limit on API calls. API key can be passed: as a query parameter, in the query header, as a cookie value.

The Token-Base Authentication or Bearer Authentication scheme is based on the use of a "token signed by the server". This token is then sent to the server inside the Authorization query header. However, unlike Basic Authentication, the "Basic" keyword is replaced with "Bearer": Authorization: Bearer <token>. After receiving the token, the server validates it (does a user exist with such a token, the validity period has not expired, etc.). Token-Base Authentication can be used independently, when

the server itself generates tokens for new users, or be part of a more complex authentication mechanism such as OAuth 2.0 or OpenID Connect. The difference between tokens and API Key is that the key can only provide access to API methods without the ability to get the user's personal data.

JSON Web Tokens (JWT) is an authentication mechanism based on the use of a special type of token - JWT token. It is a JSON data structure. Such a token contains a header with general information, a body with a payload (user-id, group, data) and a cryptographic signature. This scheme is one of the most secure mechanisms for transferring data between two parties, therefore it is considered the preferred method for controlling REST API access. The user for working with the API, when sending a request, adds to it the personal JWT previously issued by the server.

OAuth 2.0 is a more complex protocol responsible for user authorization. Allows a web application to acquire rights to use client resources on another service. This makes it possible to provide a third-party application with limited access to the controlled resources of the user without directly transferring the login and password to this application. OAuth can be used on any platform, as the protocol relies on the underlying web technology stack, namely HTTP queries, responses, URL redirects [14], etc.

This approach is the most complex authorization option. Only when it is used the service is able to uniquely identify the application calling for authorization. Otherwise, authorization is done entirely on the client side. The schema is often used when developing REST APIs that interact with third-party services.

OpenID is a standardized method for a decentralized authentication scheme. A distinctive feature is the ability to create a single account for authentication on several services at once (creating a unique digital identifier) through the services of an intermediary service. In terms of the mechanism of operation, this method is similar to OAuth 2.0, but OpenID is intended solely for user authentication. The new version of OpenID Connection was transformed into an authentication add-on over OAuth 2.0, received a reliable encryption mechanism and digital signatures.

It was decided to use JWT as the main access control scheme for the developed application, since the service does not use third-party resources in its work, and such tokens are easy to use, have a convenient data description format and do not create unnecessary network load. The use of the HTTPS protocol in conjunction with a cryptographic signature provides an optimal level of protection for the service.

When securing a web service, input control deserves special attention. You need to make sure that all data that the application will subsequently operate on is valid and conforms to the API specification.

The developer community has formed a number of recommendations when validating the input data:

- to conduct data validation both on the client side and on the server side;
- not to use your own validation mechanisms, you should use the built-in functions for
- checking a programming language or framework;
- it is always necessary to check the content type, size and length of the query;
- not to create manual queries to the database on the backend, use only parameterized queries;
- to use allow lists on the server;
- to keep logs of errors, attempts to enter data, etc.

Choosing the right tools for the backend is an equally important task when developing a REST API. The Python programming language has long established itself in the web development market. A lot of high-quality modules, packages, frameworks and utilities were created for "python", which greatly simplifies the development process. Over the years, the industry developed a rich code base and a huge international community of developers. An important factor is the relatively low learning curve, the simple and concise syntax of the language. This and much more affected the fact that the development of application prototypes, proof-of-concept and minimum viable products (MVP) is faster and most efficiently done in Python.

There are many powerful and versatile frameworks for developing services and web applications in the Python ecosystem. Flask is the most popular framework for developing API services.

Flask is a WSGI micro-framework that provides minimal basic functionality with the ability to add modules as needed. Due to the modular approach, the programmer can collect only those tools that will definitely be used in the development of the project [15]. The flexibility of the architecture and the presence of a large number of ready-made extensions make it possible to use Flask both in simple projects and in the development of complex platforms. In the developer community, it is often used to create RESTful web service APIs. All of this became a determining factor in choosing Flask as the basis for the development of the application prototype.

API Flask provides a number of loadable modules [16]:

- Flask-Login and Flask_simplelogin for managing user sessions. Thanks to these extensions,
- registration, authentication and authorization mechanisms are integrated.
- Flask-WTF– module for creating secure forms (validation, csrf tokens, etc.).
- Flask_jwt - extension for working with JWT authentication scheme.
- Flask_restful / Flask_restplus - Sets of helper objects and functions that simplify REST API
- development.
- Flask-Security - a package for managing users roles and groups.

The disadvantage of many developed services is that the description of the API itself is rigidly tied to the implementation in a specific language or its framework [17]. The open-source tool Swagger is used to solve this problem and create a flexible unified API.

Swagger is a framework independent from a programming language and technology stack for describing the specification and documentation of RESTful APIs. It allows to configure REST API without access to the source code of the application, through a special file with the yml or json extension [18]. A documentation package is also automatically generated on this file basis.

3. Results

The server on which the developed web application will be launched is also a potentially vulnerable point for attackers. To reduce the risk of hacking, you need to disable root logins of the linux server by editing the file / etc / ssh / sshd_config. In the PermitRootLogin option, change the value from yes to no.

The next step is installation and configuration of the firewall. It is necessary to close access to all ports except 80 (HTTP), 443 (HTTPS) and 22 if we work with server via SSH: sudo ufw allow ssh hhtp 443 / tcp –force enable.

In addition, you should replace the standard Flask application server with a more advanced WSGI server, gunicorn. It is a robust production server that many developers take as a standard when developing Python web applications. Configure the service to wait for queries from private port 8000: gunicorn -b localhost: 8000 -w 4 restapiservice: app. However, application web traffic can only go through ports 80 and 443 open on the firewall. At the same time, to ensure the security of the application, all traffic arriving on HTTP port 80 must be redirected to encrypted port 443 (HTTPS). To work with the HTTPS protocol, you need to obtain an SSL security certificate (for example, Let’s Encrypt): sudo Apt install python-certbot-nginx / sudo certbot –nginx -d <application_domain> -d www. <application_domain>

An Nginx proxy will be used to redirect traffic. Listening port settings, HTTPS connection, and traffic redirection are done by setting the configuration file / etc / nginx / sites-enabled / default [19].

The default SQLite database is also not an acceptable solution. The application will use the reliable and efficient PostgreSQL DBMS, which is the de facto standard on the Python + Django / Flask technological stack.

However, no DBMS can efficiently service the large number of independent processes that call for it. Too many queries (from real users or those created by an intruder) will lead to the failure of the database. To solve this problem, we will use the PgBouncer connection puller. The puller proxies all incoming queries in small chunks, creating specialized queues.

Initially, when developing a web application, you need to pay attention to the form of storing confidential application data: keys, addresses, database passwords, etc. The simplest and most reliable option is to use environment variables. The programmer creates a bat / bash script that, when launched on the server, will write the necessary data to the variables of the current environment. You can then use the os python module to read the required information and configure your Flask application. Thanks to this, secret data does not appear openly in the code.

When creating a user model, it is important to pay attention to the work with the "Password" field. Passwords should not be explicitly stored in the database. For security reasons, it is not the passwords themselves that are used to record and compare passwords, but their hashes. To do this, inside the set_password and check_password methods of the User model, we will use the functions of the security package - generate_password_hash and check_password_hash.

Next, let's look at the method responsible for adding a new achievement. In this case, the user must be authorized.

Getting the contents of the fields of the model should be done through a call to the get method, and in the absence of the corresponding field, the value of the variable is set to None. In addition, in order to ensure security, the work with the database is carried out through the objects of the built-in ORM, and not by direct SQL queries.

The description of the API query specification itself, the setting of the handler and security restrictions are moved to the swagger.yml file. In this file, you need to specify the authentication scheme that the application will use, as well as a special function that contains the implementation of this scheme in Python. In the case of a developed web service, the schema handler (jwt_auth function) is located in a separate auth.py file.

In the swagger.yml file, each query to the API is represented by a separate block, which contains all the service options in a declarative style. In the specification block, you must specify: URL route, type of HTTP query, description of the response from the API (status codes, error messages, etc.) and the authentication scheme.

A description of the specification for adding achievements is shown in figure 1.

```
paths:
  /achievement:
    post:
      operationId: "achievements.create"
      tags:
        - "achievement"
      summary: "Create a achievement"
      description: "Create a new achievement. Added in achievement list"
      parameters: "..."
      responses:
        201:
          description: "Successfully created achievement in list"
          schema:
            properties:
              title:
                type: "string"
                description: "Title of the achievement"
        406:
          description: "achievement already exists"
      security:
        - bearerAuth: []
```

Figure 1. Specification of a query to API to add a new achievement.

Now, when trying to resort to the API POST / achievement method, the system will ask the user to log in (figure 2).

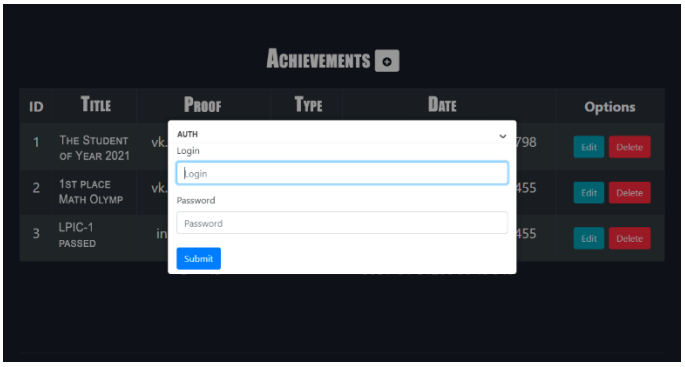

Figure 2. User authorization for access the secured API.

The rest of the API methods are created in the same way.

After completing all the procedures for securing the server side, you must also take into account a number of considerations related to securing the client side of the application.

A web application developed and configured in this way closes most of the vulnerabilities of the frontend / backend side and allows the client to safely use all the functionality of the service.

4. Discussion

The basic rules and recommendations for securing web applications have been formed on the experience of commercial development of thousands of programmers. These are generally accepted norms in the community. Standards govern their key points that engineers, software testers, and DevOps engineers rely on. Much more controversy arises when choosing a technology stack for developing web applications.

In 2018, Sebastian Ramirez developed a Python web framework called FastAPI. In his opinion, FastAPI should have become the best framework for developing REST API services and a more performant analogue of Flask. Ramirez's framework is completely asynchronous and is easily integrated with OpenAPI-schema, which makes it easier to work with Swagger and ReDoc. Despite the relatively short term in the market, FastAPI established itself as a quality tool. Many large companies started using it to develop their APIs. So, in the project discussion on GitHub, Microsoft Lead Engineer Khan Kabir posted the following comment: "I use FastAPI very often in recent days. I definitely plan to use it for all ML services of my team at Microsoft."

Also, as an alternative to Flask, many developers recommend considering another relatively "young" framework - Sanic. It was also conceived as a multi-threaded, asynchronous analjgue of Flask. It is a micro-framework with a modular structure and a number of features such as basic routing of static files, special extensions for creating CRUD, mechanisms of non-blocking operations, and others.

The Python language has also been criticized by some developers for its relatively poor performance due to the language's interpretability. As an alternative, Google engineers offer their own development - the Golang programming language. Its main advantages are strong typing, the presence of a compiler, and goroutines are an efficient analogue of threads. In addition, Go provides the ability to work with static typing when developing web applications, which makes it much easier to test and maintain the product in the future.

5. Conclusion

Thus, APIs have become a key element in the development of modern web and mobile applications. They allow to organize interactions with applications, services, and software platforms. Thanks to their versatility and ease of use, REST architectural style interfaces account for about 80% of all public and proprietary APIs.

In the course of the research, a prototype of the protected RESTful API web service was developed, with the help of which students will be able to keep track of personal individual achievements in educational, scientific, creative, social and sports activities. The Python programming language and its Flask web microframework significantly increased the efficiency, quality and speed of MVP

development, and the Swagger framework made it possible to achieve independence of the developed REST API from the technology stack.

The project has a high potential for expansion and scaling by adding new functionality, replacing the used database with a more flexible No-SQL document-oriented version (for example, MongoDB), implementing advanced caching based on the in-memory Redis database, authorization through social networks. In addition, it is possible to switch to a more efficient asynchronous backend framework, refine and optimize the UX / UI of the client side of the application.

References

[1] Kornienko D V, Shcherbatykh S V, Mishina S V and Popov S E 2020 The effectiveness of the pedagogical conditions for organizing the educational process using distance educational technologies at the university *J. Phys.: Conf.Ser.* **1691(1)** 012090
[2] Kornienko D V 2020 Organization of a system of digital education practices in the municipal sphere of general education *J. Phys. Conf. Ser.* **1691(1)** 012108
[3] Shcherbatykh S V and Mishina S V 2019 Development of professionally significant qualities of future economists by means of the hidden curriculum Desarrollo de cualidades profesionalmente significativas de futuros economistas mediante el currículum oculto *Utopia y Praxis Latinoamericanat* **24(Extra6)** 387-95
[4] Mishina S V 2020 Strategies for students' lean competencies formation in the educational process of the university *J. Phys.: Conf. Ser.* **1691(1)** 012213
[5] Shevat A, Jin B and Sahni S Designing 2018 *Web APIs: Building APIs That Developers Love* (United States: O'Reilly Media)
[6] Jiao J, Yang Y, Feng B, Wu S, Li Y and Zhang Q 2017 Distributed rateless codes with unequal error protection property for space information networks *Entropy* **19(1)** 38
[7] Abbaspour E, Fani B and Heydarian-Forushani E 2019 A bi-level multi agent-based protection scheme for distribution networks with distributed generation *Int. Jour. of Electrical Power and Energy Systems* **112** 209-20
[8] Cong W, Zheng Y, Zhang Z, Kang Q and Wang X 2017 Distributed Storage and Management Method for Topology Information of Smart Distribution Network *Dianli Xitong Zidonghua Automation of Electric Power Systems* **41(13)** 111-8
[9] Bonér J 2016 *Reactive Microservices Architecture: Design Principles for Distributed Systems* (United States: O'Reilly Media)
[10] Song X, Zhang Y, Zhang S, Song S, Ma J and Zhang W 2018 Active distribution network protection mode based on coordination of distributed and centralized protection *Proc. of 2017 China Int. Electrical and Energy Conf.* 180-3
[11] Fielding R 2000 *Architectural Styles and the Design of Network-based Software Architectures* (California: University of California)
[12] Tong B B, Zou G B and Shi M L 2013 A distributed protection and control scheme for distribution network with DG *Advanced Materials Research* **732-3** 628-33
[13] Zhong S, Liu C, Yang Z and Yan D 2009 Privacy protection model for distributed service system in converged network *Int. Conf. on E-Business and Information System Security* (Piscataway: Institute of Electrical and Electronics Engineers)
[14] Maximov R V, Ivanov I I and Sharifullin S R 2017 Network topology masking in distributed information systems *CEUR Workshop Proc.* **2081** 83-7
[15] Grinberg M 2016 *Development of web applications using Flask in Python* (Moscow: DMK Press)
[16] Dwyer G 2016 *Flask By Example* (UK: Pack Publishing Ltd)
[17] Percival G 2018 *Development based on testing* (Moscow: DMK Press)
[18] *Swagger* Retrieved from: https://swagger.io
[19] Melnikov M O and Igonina E V 2021 Configuring a Debian server for developing Python and Django web applications *Modern Science* **2(2)** 389-98