



POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION

Institute of Computing Science

App Documentation:

APPLICATION SECURITY

Author: Konrad Bankiewicz, 145265

Supervisor: dr inż. Michał ApolinarSKI

1. Introduction

This section presents an outline of the application architecture and important design decisions that influenced the development of the project.

2. Outline of the application architecture

Web applications are classically divided into at least two separate but closely cooperating parts, commonly known as the front-end and back-end of the application. The former is responsible for the user interface, while the latter is responsible for operations performed on the server side. The presence of the front-end part seems obvious in this case, because without interface, the user would not be able to use the application. In turn, the part operating on the server side is necessary to ensure the exchange of information between users, but also the integrity and security of the entire service.

It is no different in the case of the application that is the subject of this work. However, the idea of a classic web application¹ has been abandoned in favor of a newer and increasingly used approach in which the front-end and back-end applications are completely separate - they can even work on different physical servers. First of them functions as SPA (Single Page Application), and the other as REST API (Representational State Transfer Application Programming Interface).

The use of such an approach, on the one hand, complicates the complexity of the application as a whole to some extent, but on the other hand, it allows to significantly improve the scalability of the application, which is definitely a good investment for the future. Thanks to this, it will not be a problem to create, for example, dedicated, native mobile applications in the future, when the need arises. They would communicate with the server in the same way as a browser client. Another advantage of this approach is the increased functionality of the user interface, which becomes reactive and can be dynamically (without reloading the page) changed depending on the occurrence of certain events. This will ensure a better UX (User Experience), i.e. improve the user's experience in using the application. The initial loading time of the application will be slightly longer than in the case of a classic web application (this is due to the need to load a large amount of JS scripts and CSS styles at once), but the time of switching between individual views within the application itself will be shortened, and the lack of the need to refresh the page means that it will not unnecessarily distract the user and the behavior of the interface will be much more predictable.

This application was intended to promote cryptography, but already at the design stage it was decided that its implementation does not need to, and indeed should not, be limited to one narrow field. In the future, interested people will also be able to use it to create courses on completely different topics, even those unrelated to computer science. Widgets are the distinguishing element of application applications for various fields. Some of them may be useful to everyone, for example widgets presenting: text, video, downloadable files or test questions. The rest will be used only in specific applications, for example, a widget presenting the operation of the Playfair cipher may be included in a cryptography course, but it will not be useful in a molecular biology course. Anticipating all possible use cases and creating

dedicated widgets for various fields of science is unrealistic and pointless. Moreover, it would result in the application becoming a huge and difficult to maintain monolith .

The authors of the work considered this problem to be important and devoted a relatively large amount of time to making it possible the best way to solve it. As a result, a special abstraction layer was separated in the application that handles widget mechanisms . Widgets are not tightly integrated with the application kernel, but are modules. Thanks to this, individual instances of the application can be equipped with a different set of widgets , dedicated to specific applications. The application administrator can independently install widgets , which can also be developed by third parties who are not part of the team developing the base application.

To implement widgets, the authors use a special SDK (Software Development Kit), the preparation of which was also part of the ongoing diploma project. It provides , for example , mechanisms for storing and sharing configurations and by a classic web application, the authors of the work understand an application in which the front-end and back -end together create a monolith, and for each user request sent to the server, the generated content of the HTML document is returned support for many natural languages.

3. Back -end of the application

This part describes in detail individual aspects of implementing the application's back -end.

4. Introduction and selection of technology

The server part of the application, hereinafter referred to as the back -end, was implemented in TypeScript . It is a programming language, transpiled to JavaScript. It can be said that it is an extension of the JavaScript language with some additional functionalities, in particular static and strong typing. This allows you to eliminate many errors that would not be detected during the implementation and deployment of the application, but which would appear during execution - in short, errors that are difficult to detect. According to research [13], it accounts for up to 15% of all errors made by JavaScript programmers. Instead, these errors will appear at the compilation stage. Additionally, the use of TypeScript helps you better document your code and supports code editors in syntax suggestions. Static typing of function parameters and object fields forces the programmer to better think about the code architecture, which also has a positive impact on application development.

The back -end of the application, as mentioned earlier, plays the role of Web API. It is essentially an HTTP server that processes requests sent by the user, performing certain operations, of course after checking whether they should be performed. For example, it is verified whether the data sent by the user is correct or whether he has the appropriate permissions to perform a given operation. After processing the query, a response is returned to the client.

Therefore, a key element of the back -end is HTTP protocol support. The Express.js library was used for this purpose. It is a small framework whose purpose is to provide an interface, at the programming language level, for defining individual controllers that handle queries sent by

the user. This framework is based on the concept of middleware . This means that a query that has been matched to a specific endpoint² is then processed by a list of ranked functions assigned to that endpoint . These functions are commonly called middleware . Every middleware can modify it in some way - it can fake the original query and pass the execution on. It may also return an HTTP response, which will abort further processing of the request. The last of the middlewares , which handles the specific logic of a given endpoint , is called the controller.

Such architecture can be used in a relatively simple way, for example to authenticate the user , which was the case during the implementation of this project. If it is successful , the appropriate field is set in the query object, and the query will be processed by other middleware . If not, an HTTP response with status 401 Unauthorized is returned and further processing is interrupted.

Most requests sent by the client are thoroughly validated and sometimes also sanitized. For example, be sure to check that the e-mail address provided during registration is the correct address and not a random string of characters. In some cases by endpoint, the application authors understand a pair: the HTTP method and the pattern matched to the address to which the request was sent whitespace that the user may have accidentally entered at the beginning or end of the text is also removed. For the application described above, a small express-validator library was selected , which was developed within the project to create its own, convenient and easy-to-maintain solution that meets the requirements. It's hard to imagine this type of web application that wouldn't use a database. For the needs of this project, the non-relational database management system MongoDB was chosen . This was dictated by the better support for this database in node.js, compared to relational databases, according to the authors. Another important argument was the willingness to learn this technology. This choice has some problems, namely this database management system does not validate document structure, as it does in SQL-based databases. For the sake of data integrity, as well as convenient access to collections, represented at the code level as models, an additional, very popular Mongoose library was selected .

5. Design of the API query interface

The client application communicates with the server via the API. Such an interface had to be designed with particular care , as it should be backward compatible. Determining the details of communication within a specific API version allows you to develop both the server and the client without fear of losing the stability of communication between both elements. Please remember that the API should be clear and intuitive for the programmer. Idiomatic definition of individual endpoints , returning responses with appropriate HTTP statuses and understandable error codes allow you to significantly simplify the work of integrating the client with the server. All this contributed to the fact that the server query interface was designed based on the REST architectural style. The following paragraphs describe the impact of this decision on the final form of the interface. Dedicated methods offered by the HTTP protocol are used for individual queries: GET, POST, PUT, DELETE, HEAD. Each method determines the action to be performed when the query is executed; for example, GET retrieves a resource and DELETE deletes it. Individual methods are used in a manner consistent

with the semantics [12] defined in HTTP standards. For example, the GET and HEAD methods are called safe because they do not modify the server state. The same cannot be said about the PUT method, but this method is idempotent because executing such a query multiple times gives the same result as once. The POST method is neither secure nor idempotent. The addresses to which queries are sent represent resources and, according to REST, they do not determine the action performed when handling the query, as this is the domain of HTTP methods. This principle was taken into account when designing the API for this application. An important element of the REST standard is the statelessness of communication between the client and the server . This is because the session3 mechanism is not used to authenticate users . Instead, at the customer's request, tokens with an appropriate validity period are generated, which the customer can use to confirm his identity. Details of the implementation of this mechanism are described later in the work. Controllers handling individual requests are obliged to return HTTP statuses in response to their semantics. For example, a query that retrieves information about a specific. The session mechanism works in such a way that the server stores a certain set of information about active clients, and individual clients authenticate themselves via a cookie containing a unique ID resource may return a 200 OK response if successful, a 404 Not Found response if the resource could not be found, or a 403 Forbidden response if the logged in user does not have permission to access the resource. If the user provided an incorrect authentication token or did not provide it at all, a response with the status 401 Unauthorized will be returned . The selection of specific statuses depends on their definition in the standard . The semantic approach allows the client to easily decide whether the query was successful or take a specific action in a given situation. Ambiguous cases may arise when the same HTTP status is returned in two different situations. In this case, if the customer wants to distinguish them, additional information is needed in the response. This happens when a response returns a status of 404 Not Found . It is necessary to specify whether the returned status refers to a non-existent address (which is caused by the fault of the programmer creating the client application) or to a non-existent resource. Both the data that the user sends to the server and the data returned by the server must be saved in a structured form. The very popular JSON (JavaScript Object Notation) format was chosen for this purpose. This applies to virtually all inquiries. A slight exception to this rule is uploading files to the server. This case is described in detail later in the work.

6. User authentication

The queries sent to the server include those in which the client tries to authenticate itself in some way. Thanks to this, the server will be able to make sure that it is dealing with a specific user. There are many different authentication methods. This section will present how the authors of this work approached the implementation of this mechanism. The REST specification is based on stateless communication between the client and the server. The result is that the customer, in order to confirm his identity, must send all the required information in each query he sends. A customer who already has an account on the website must log in to use it. For this purpose, it sends a POST / auth / sessions request to the server , in which it sends access data: e-mail address and password. The controller handling this request makes sure that a user with such an e-mail address exists in the database and whether the result of the hash function saved in the database agrees with the one calculated on the

basis of the sent password and the salt set for this user. The bcrypt function is used to determine the hash . If the server determines that the provided access data are not correct, a response with the status 401 Unauthorized will be returned , which the client will have to handle appropriately by reporting incorrect login data to the user. In turn , if login is successful, the server will generate two tokens : access token token) and refresh token (refresh token). Both tokens have a certain set validity period, with the validity period of the refresh token being much longer than that of the access token . The client now authenticates its requests by sending a special HTTP Authorization header access token . As long as this token is valid and valid, the server will never return a 401 Unauthorized status . The situation changes when the access token expires . Then the client tries to refresh it, i.e. generate a new token . To do this , it must authenticate itself this time with a refresh token - such a token can only be used once . The client sends it in a PUT / auth / sessions request . If refresh token too is valid and correct, then a new pair of tokens generated by the server will be returned in response . If the token validity period has expired, which may happen when the user has not used the application for a long time, a new pair of tokens can only be obtained by logging in again.

You can use at least two different techniques to generate a credential token :

- generate a random string of characters and then save it in the database, associated with the user who can use such a token
- use a cryptographic signature mechanism, which will allow you to include certain information in the token and at the same time ensure that the token was generated by the server

Both approaches have their advantages and disadvantages. The first approach is simpler, but it is database intensive. Please remember that the access token must be verified by the server with each authenticated request.

The second approach requires the use of certain cryptographic mechanisms, but on the other hand, it does not require writing and reading the database every time. The token stores encoded information about the user associated with it and its validity period. Due to the presence of the signature, the server can be sure that these parameters have not been modified by a potential attacker, as he would have to be able to generate the appropriate signature. However, this requires knowledge of the secret (in the case of symmetric cryptography) or the private key (in the case of asymmetric cryptography).

However, the use of this type of mechanisms should be approached with great caution, if only because such tokens cannot be invalidated in any way. This poses a problem if someone's account is taken over or a device such as a laptop or phone is stolen. In theory, a mechanism based on a blacklist of revoked tokens could be used , but it would have to be saved in the database, which completely undermines the point of this solution.

Therefore, a hybrid approach that uses both approaches will be used to implement authentication support in this application. Access tokens are generated using the JWT (JSON Web Token) standard because they are more frequently used. Refresh tokens are physically

stored in the database, so it is possible to log out the user by removing the token from the database, although this will happen with a certain delay (until the access token expires).

7. User Registration

The app allows the user to create their own account in two different ways.

Registration form

The first, classic way is based on the registration form. The interested party provides, among other things, his e-mail address and password. After sending the data to the server, the password hash is determined (for the previously determined salt) using the bcrypt function , and then the user data is saved in the database. A welcome message containing a link to confirm your account is sent to the e-mail address provided during registration. It contains a special token generated by the server , associated with the newly registered user. After sending a special confirmation query `DELETE /me/ confirmation-tokens /: token` , the token is set in the user profile there is an appropriate flag. Its presence allows us to assume that the account was created by the owner of the e-mail box associated with this account. If the e-mail did not arrive or the confirmation link has expired, the user can request it to be resent using the `POST /me/ confirmation-tokens request`, but must send an access token in the header . After activating your account, the user can fully use it, of course within the limits of the rights granted to him.

OAuth2 standard

The second method is based on the use of external services, hereinafter referred to as providers, such as Google, Facebook, or Discord . From the user's perspective, creating an account on the website is as simple as clicking a button and confirming access to the above-mentioned permissions. From the technical side, the matter is much more complex.

The providers mentioned in the previous paragraph offer the OAuth2 standard as an authentication method . This allows you to gain access to the API of these services, in the form of access and refresh tokens , within a strictly defined scope. For the purposes of registering in this application, it is sufficient to obtain the e-mail address associated with the provider's account and possibly less important information, such as name and surname or avatar . Then, based on this information, a new user account is created. Please note that such an account does not use a password to authenticate the user. Instead, exactly the same mechanism is used to log in as for registration. However, in the case of the login process, there is already an account registered to this email address in the database, so a new account is no longer created, instead a pair of tokens is generated and returned to the customer, similarly to the classic login.

The implementation of such a process requires the involvement of:

- application back -end,
- application front-end,
- OAuth2 provider API.

Due to the fact that the first of the above-mentioned elements plays the greatest role in operating this process , the description is made in this part of the work.

The OAuth2 standard defines several different variants, called flows . As part of the project, the Authorization variant was used CodeGrant . This process is described in detail in the RFC standard. The process can be represented by a sequence of steps , described below . As an example, Google was selected as the provider. The entire process is additionally illustrated with a diagram in Figure.

1. The user does not have an account on the website yet. So he clicks the button Sign in with Google.
2. The front- end application sends a query to the server asking it to generate an authentication address . In this case it is a GET / auth / google / authorization - url query .
3. The server returns a special URL in response, which will vary depending on the provider. This address contains specific parameters that specify authentication details . These are primarily:

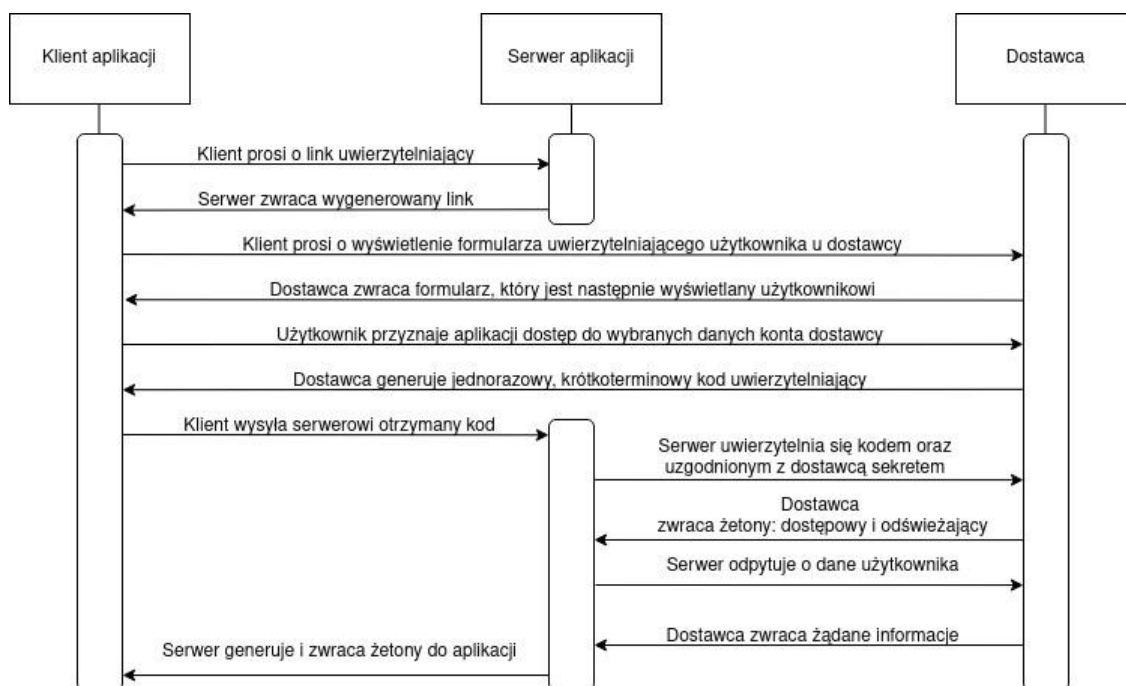


Figure: Authorization variant OAuth2 Code Grant

- response type – a parameter that defines the authentication flow, in this case code
- client id – an explicit parameter that uniquely authenticates the application requesting authentication . Its origin will be described later.

- scope – a set of permissions requested by the application. In this case, the authors only wanted to obtain information about the user's basic information, including his e-mail address. Therefore, only those permissions that are required are provided . The set of permissions differs between providers as it is not defined by a standard but depends on the specifics of the specific provider. In the case of Google, it was necessary to provide two permissions here: userinfo.email use - rinfo.profile .

- redirect uri – address to which the provider will redirect, providing the authentication code. This is an address that refers to a special view supported by the application's front-end. This will be described in more detail in the next steps.

- state – a token generated by the server , which will later be used to verify the correctness of the process. This is not a mandatory parameter, but it is recommended because it is a form of protection against CSRF (Cross Site Request) attacks. Forgers). In the case of this project, it is a JWT token with a short expiration time.

4. The client receives a generated URL from the server and is redirected to this address.

5. The provider's website is displayed to the user. It contains a message saying that the application with the indicated name is asking for a set of the listed permissions. The user can reject this request at this point - then the process is interrupted . If I accept the request, then the provider redirects me to the address indicated in the previously defined redirect parameter uri . They are passed additionally in the URL parameters: code and state . The first one is a special code that will be used in the further authentication process. The second one is the same state parameter that was generated by the server at an earlier stage of the entire process.

6. The front-end application responds to the redirection by reading the code and state parameters from the address, and then sending a POST / auth / google / sessions request to the application server , sending the read parameters.

7. In order to handle the request, the server first validates the state parameter , checking whether it has the correct value - in other words, whether it is the same token that was previously generated by it. If this value is incorrect, it indicates an incorrect process or interference by a potential attacker. In such a situation , the entire process is interrupted and the client receives a response with the status 401 Unauthorized . If this parameter is correct, the server begins to obtain the provider's tokens : access and refresh. For this purpose, an inquiry is sent to the address provided by the supplier . Exactly what parameters are transferred is defined by the OAuth2 standard. As before, the client id and redirect parameters are sent uri . Additional parameters are presented below .

- client secret – this is a secret key that only the server should know. It is used to authenticate the application when trying to obtain the provider's authentication tokens .

- grant type – this is the authorization value code denoting the type of temporary code with which the server authenticates itself.

- code – code provided by the client in the currently processed request. If the operation was successful, the server should receive a pair of tokens in response to this request : access token and refresh token , as well as a unique user ID with the selected provider. Using the obtained tokens, you can only perform operations within the scope of the permissions previously indicated in the scope parameter . Please remember that these are different tokens than those that the customer will receive later. Only the server knows these tokens , and in the case of the process just discussed, they will not be remembered at all. They will only be used to extract basic information about the user.

8. The server, now having tokens , asks the supplier for the necessary information - primarily the e-mail address. After obtaining it, it checks the database whether there is already a user with such an address. If not, it means that you need to complete the registration process. Therefore, a new user account is created in the database, in which, in addition to basic information, the user ID on the supplier's website is also saved. Additionally, a welcome e-mail address is sent.

9. Regardless of whether the user is currently registering or logging in to their account again, the process ends the same - the server generates an access token and a refresh token and returns them to the client. The client then remembers these credentials and thanks to this, the user has been logged in to the website. The above description shows that in order to carry out such a process and integrate this application with the supplier, appropriate credentials had to be generated. This process is carried out on the platform offered by the supplier. You should specify parameters such as:

- Application name
- Set of permissions – it must match the scope parameter used during authentication
- Redirect address – similarly to the above, it must match the redirect value used during the process uri After registering the application with the provider, the following credentials are generated: client id and client - secret , the use of which is described above.

8. Authorization of access to resources

The previous sections described how user authentication, i.e. confirming their identity, was implemented in this application . However, this is not sufficient, because when handling individual queries, it is often necessary to check whether the logged in user has the permissions to perform the indicated action on a specific resource. Verification of these permissions is called authorization.

To provide an authorization mechanism, the server defines certain atomic permissions . These are, for example:

- permission to read the user list,
- permission to modify users,
- permission to read the list of articles.

Each controller handling a specific type of query defines a list of necessary permissions that the user requesting to fulfill the request must have.

Each user is assigned a role, which has a name and a set of permissions. When handling a query, after the user is authenticated, the role assigned to him is checked. The set of all permissions that the user has is extracted from it. Then, it is checked whether the set of permissions necessary to fulfill the request is included in the set of permissions assigned to the user. If so, another middleware is executed . If not, an HTTP response with status 403 Forbidden is returned .

Initially, there are two built-in roles in the system: administrator and regular user. The first one has all possible permissions, and the second one only has selected ones necessary for the proper operation of the system within the functionalities intended for an ordinary user .

The way authorization is implemented in this application does not prevent you from creating new roles and assigning them a specific set of permissions. Specific endpoints in the API have been provided for this . Thanks to them, you can manage roles: create, modify, delete and read them. Of course, only a user who has the appropriate permissions can manage roles.

9. Validation and sanitization of input data

Some of the queries sent to the server contain some data in their content. For example, when creating a new user account, the following are sent: e-mail, password, duplicate password (for verification purposes) . All user-dependent parameters should be treated with great caution, because they can potentially be used by an attacker to take control of the application . You should also make sure that the given parameters meet the business logic rules. Therefore, in the case of the registration form, the server makes sure that the data sent in the email field is a correct and unique e-mail address, and that the password sent is properly complex, and also whether the duplicate password matches the original in character. To ensure maximum code transparency, the library used to support the HTTP protocol, Express.js, uses the concept of middleware . Before any use of the value sent by the user to implement the application logic, it is first validated and sanitized . Validation involves checking the data, while sanitization involves modifying it to clean it. A typical example of sanitization, also used in this application, is the cleaning of data from whitespace characters located at the beginning or end of a string of characters that the user may have accidentally entered in the form . If at the validation stage it turns out that the data format is incorrect, the following status is returned, depending on the case: 400 Bad Request , 409 Conflict or 422 Unprocessable entity . If the sent data is correct, execution is transferred to the next middleware. The validator library was used to help implement the validation and sanitization mechanism . It allows you to declaratively determine the data processing chain, i.e. what, when and how is to be checked or processed. However, this library is not used directly, although in theory it is adapted for this purpose. However, slightly more sophisticated mechanisms have been implemented that allow, among other things, the repeated use of complex validation chains without the need to redefine them , taking into account the requirements of these fields.

For example, to validate an email field, it is checked in the following order:

1. whether the field was submitted at all
2. whether the field is text and not, for example, a number
3. whether the field is empty
4. whether the field matches a regular expression describing a valid email address

If validation fails at any stage, a message related to that specific stage is returned in response, for example "The field should be a string". Subsequent steps will not be checked further. So validation is only successful if all four conditions are met. The exception are optional fields. In their case, validation succeeds even if none of these conditions are met (but if only the first one is met, the validation will fail again).

10. Database operation

As mentioned in the introduction, the database management system used in this project is MongoDB . To be able to establish a connection between the application server and the MongoDB database instance, you need the so-called controller. In this case, it is a library that supports the communication protocol with the database. It is thanks to it that the back -end can perform any operation on the database.

Moreover, you should realize that MongoDB is a non-relational database management system. The structure of data stored in such a database is not subject to restrictions, as is the case with SQL databases , for example MySQL. This is the approach he has some advantages, but there are also significant disadvantages. It is more difficult to maintain data consistency within a given collection.

Operating directly on the database and each time mapping the representation of an object at the source code level to a document stored in the database, while ensuring the completeness of the information stored in the database, is not a convenient solution, much less a safe one. There is no doubt that a certain entity is necessary, existing at the source code level, which takes responsibility for the above-mentioned elements. Mongoose library comes to the rescue , and the authors of the application decided to use it during the project. The use of this library is based on elements such as:

- diagram,
- model,
- middleware (they should not be confused with the middleware mechanism in the Express.js library, although it is undoubtedly a similar mechanism),
- plugins .

For demonstration purposes, an example is used in the following paragraphs, describing how the representation of the system user is defined in the source code. The example has been slightly trimmed to focus only on its most important elements.

At the very beginning, the schema is defined. When creating it, you must precisely specify what fields an object of a given type will have. Each field can be provided with information about the data type, requirement, uniqueness, default value, etc. In the case of text fields, it is possible, for example, to force all capital letters to be converted to lowercase and white letters before being saved to the database. Characters from the beginning or end have been removed - this happens in the case of the email field. An example of a diagram is presented in Listing 3.2.

The role field refers to a document from another collection. When the timestamps flag is set to true, the mongoose library takes care of recording additional information about the creation date and modification date of each document.

This library also provides a mechanism that allows you to execute arbitrary code just before or just after an event occurs. Listing shows the source code that generates the URL to the avatar on Gravatar4 if one has not been specified before. The Mongoose library also allows you to define a method in the diagram that can define some frequently used logic. In the case of a user, an example would be generation

token.

It is often the case that certain mechanisms may apply to a wide range of schemes, both within one and many projects. An example would be support for user authentication using a password. In such a case, Mongoose allows you to use plugins, i.e. special extensions attached to a given model. This functionality was often used in the project. Both ready-made plugins were used (for example passport-local-mongoose, which supports the local authentication mechanism using login and password), distributed in the form of libraries, and those defined internally for the purposes of this project. Gravatar is a service that allows you to associate an e-mail address with an image that acts as an avatar. The user who uses it can, for example, set his/her photo there - it will be visible as an avatar on all websites that use this service. For other users, the default image will be returned

Listing: User schema in mongoose

```
const UserSchema = new mongoose.Schema ({ email: {
  type: String, unique: true, lowercase: true, trim: true, required:
  true,
},
  roles: {
    type: mongoose.Schema.Types.ObjectId, ref: 'Roles',
    required: true, autopopulate : true,
  },
  emailConfirmed : { type: Boolean, default: false
```

```

},
avatarUrl : String,
}, {
timestamps: true,
})

```

Listing: Conditional generation Gravatar

```

UserSchema.pre ('save', function(next) { this.avatarUrl ??=
getGravatarUrl ( this.email ) next()
})

```

Listing: Definition of the method that generates the access token

```

UserSchema.methods.generateAccessToken = function() { return
jwtAccessToken.sign (
{ id: this.id },
{ expiresIn : config.accessTokenLifetimeSeconds }
)
}

```

One of the more complex and most frequently used plug -ins implemented by the authors of the application described in this work is SluggableSchema . It provides a mechanism for generating the so-called slugs , i.e. a string of characters used to identify a resource, which has a readable form and can be placed in the URL address. The defined plugin supports:

- automatic generation of a service based on another field (e.g. object name),
- detection of duplicates and automatic numbering if they occur,
- the ability to define a service explicitly, when creating or updating an object - then it is not automatically generated.

Finally, after defining the schema, methods, middleware and connecting all the plugins used with this schema , a model is defined - a representation of the schema at the database level. The model defined in this way is used throughout the entire application. With the help of the model it is possible, for example:

- downloading one or more documents from the database, based on the indicated filters,

- saving a new object to the database,
- modification of individual fields in a previously saved document,
- deleting a document from the database,
- calling the method defined in the diagram.

The specific nature of a server-side application means that certain operations are performed very frequently. Practically every resource supported by the application can be created or read by an appropriately authorized client (this applies both to selecting one specific element, for example by its database identifier, and to the complete list of elements stored in the database). In most cases, this also applies to its modification and removal. The source code supporting such operations is often very similar and contains similar error handling. For example, if a resource cannot be deleted because it does not exist, then the server returns a 404 Not Found response .

Duplicating this type of logic is, on the one hand, unsightly and makes the code difficult to read, and on the other hand, it may lead to errors and even dangers. Therefore, as part of the project implementation , certain actions were taken to eliminate this project. The middleware mechanism offered by the Express.js library was used and functions were prepared that support the previously mentioned operations. Their use is very simple and comes down to adding one line of code for each defined endpoint . Listings 3.5 and 3.6 provide examples of using the `findAll` and `createOne` functions, respectively .

Listing: Example use findAll middleware

```
router.get ('/roles', accessTokenAuth , permissions(' roles.read '),
findAll (Role),
controller(async (req, res) => { res.status (200).send({ data:
req.data })
})
)
```

Listing: Example use createOne middleware

```
router.post ('/roles', accessTokenAuth , permissions(' roles.write '),
validator( RoleForm ),
createOne (Role, ['name', 'permissions', 'slug']), controller(async
(req, res) => {
res.status (201).send({ data: req.data ,
message : req.t(' createdSuccessfully '),
})
})
)
```

)

11. Multi-language support

During the implementation of the application, support for multiple languages was also included. Thanks to this, the client sending a query to the server will receive in response a message translated into a specific language - the one selected by the user from the user interface. Importantly, supporting such a mechanism allows the server to send e-mail to the user in the appropriate language. Accept-Language header, which is sent in the request by the client, is used to recognize the language. The content of this header complies with the standard. The client specifies the user's preferences for natural language. This may be one specific language, or a list of languages with assigned weights. In practice, the client only sends the one language that was selected from the user interface anyway. The server tries to match the best supported language indicated in the query. If this operation fails, the server selects the default language. Middleware was also used to implement this mechanism. This time, each query sent to the server is processed by a function that adjusts the best language according to the user's preferences, then loads the appropriate translations and places a function translating the labels in the query object. Then, the execution is passed on to subsequent middlewares. Individual translations for different languages are stored in separate JSON objects. These objects have a simple structure in which both the key and value are of text type. The key is the label identifier and the value is the translation of this label in a specific language. Different objects representing the translation of the application into a given language have exactly the same set of keys, but different values corresponding to these keys. Adding support for a new language comes down to adding a new object containing a set of translations. In Listings presents fragments of sample objects with translations. To translate the selected label, you must have access to the object storing the query and call the req.t function. An example usage is shown in Listing.

Listing: Translations in Polish

```
{  
missingFile : 'The query is missing a file to be uploaded to the  
server', fileAlreadyUploaded : 'This file has already been uploaded',  
fileNotUploadedYet : 'This file has not been uploaded yet', }  

```

Listing: English translations

```
{  
missingFile : 'Missing uploaded file in the request',  
fileAlreadyUploaded : 'This file has already been uploaded',  
fileNotUploadedYet : 'This file has not been uploaded yet',  
}  

```

Listing: Example of using the req.t() translation function

```
throw new PublicException (req.t(' missingFile '), 400)
```


12. Uploading files to the server

The application also implements a file upload mechanism. It is used by the file upload widget , but has been designed to be usable in other situations as well.

What turned out to be so unique in this mechanism that it was decided to describe it in this work is the way in which files are transferred via the HTTP protocol. By default, when the client sends any data to the server in the request body, it does so in a JSON object. The Content-Type header is set to application / json in this case . However , this only applies to situations when the transferred values are relatively simple, as in the case of: text, number, logical value, date, etc. Uploading a file in a JSON document is not impossible, but it would be problematic when uploading files that are not text files. Then the content of the file would have to be encoded into text form, for example using Base64 encoding. However, please remember that the use of such encoding increases the size of the data and therefore the size of the query. Additionally, encoding and decoding the data can potentially be time-consuming. Both of these problems become more significant the larger the file size being transferred. Instead, the files are transferred in their original, unencoded form in a request whose Content- Type header is form/ multipart -data. This is a common practice when developing web applications. It would be possible to send the file's metadata as well. However, the app authors recognized that this would have negative consequences for the consistency of the app's API. Therefore, the file uploading process has been divided into two stages, as a result of which the client must send two queries to upload the file. The first POST / uploads query creates an entry in the system about a new file and contains metadata such as name, description and information about whether the file will be publicly visible or not. The query body is in JSON format. This query can only be performed by a logged in user who has permission to upload files highlighted in the system. In response, the user will receive, among other things, a unique file identifier. Uploads /:id/file request . The data sent in the query content is of the multipart /form-data type. To handle requests of this type, the server application uses the multer library . After verification , the file is saved in the file system, and additional metadata, such as its original name, MIME type, or file path, are saved in the database. This query, as before, can only be performed by someone with appropriate permissions. After uploading the file, you can modify its metadata, but only those sent in the first query, i.e. name, description or publication status. The remaining metadata is closely related to the stored file , so it cannot be modified. Deleting is also possible, but then both all information about the file and the file itself are deleted. The second stage cannot be performed again due to concerns about the consistency of data stored inside the application. In this case, the file should be deleted and then uploaded again, performing both steps.

It was mentioned above that information about file publication is stored in the database. As long as the file is not private, only a logged in user who has been granted appropriate permissions can download it or read information about it. However, the application needs some files to be able to be downloaded or viewed by any user. Therefore, a mechanism for publishing files was created. There are two ways to publish a file. The first one involves sending the public flag set to true while uploading the file. The second step is to change this flag from

false to true when editing metadata. When the file is published, a cryptographically safe, random string of characters is generated . This is then placed inside the address. To the client who executes the GET request / uploads /public/: publicToken , a file will be returned in response. The sender of the query does not need to be authenticated in any way. All you need to do is use this link and anyone can download the file. You can disable file publishing at any time, then the link will become inactive .

13. Mailing service

In some cases, the application sends an email to the user. This happens, for example, when the user has created an account via the registration form - then, for verification purposes, a message containing a special confirmation link is sent to his e-mail address . Clicking it allows you to make sure that the new user is the owner of the email address provided and did not accidentally enter it. Another example of an application sending an email to a user is when they try to reset their account password. on the language the user speaks, the sent message will be translated appropriately . Additionally, individual messages will contain dynamic fragments that will differ for different users - an example is a different confirmation link. The above problems had to be properly solved in order to communicate effectively with the user via e-mail. The following paragraphs describe how the authors dealt with these challenges. The most important element of the subsystem handling outgoing mail is the SMTP client. The nodemailer library was used for this purpose . It provides a factory that is based on configuration creates a transporter object that handles outgoing communication. Subsequently , it is indirectly used to send e-mails. The intermediary element is another library that supports the concept of email templates. This library is email- templates . During configuration, it must be provided with the path to the directory containing the definitions of individual templates. This library enforces the directory structure and naming of template files. It also supports support for multiple languages. Templates are defined in the special template language pug . It allows you to clearly define the HTML structure of the document and embed dynamic fragments in it. The interface provided by the email- templates library was additionally wrapped in another, easier-to-use interface in the form of the sendEmail function . This is the function that is directly used in the design to operate the entire mechanism. Mailhog was used to test e-mail sending during application development in a local development environment . This is a tool that pretends to be a real SMTP server. All e-mails sent to it are presented on the website, but none are actually forwarded to the intended recipient. In a production environment, it was necessary to use a real e-mail service, so the authors decided to use Gmail for this purpose .

14. CORS configuration

For security reasons, not every website can send a request to a selected server using the AJAX technique. In order to protect websites, browsers use the CORS (Cross- Origin Resource Sharing) mechanism . The word Origin contained in the expansion of the acronym can be translated as origin, and it means three things: protocol, domain, port. The CORS mechanism regulates , among other things, the possibility of sending asynchronous queries to addresses of different origin.

If the origin of the JavaScript script and the origin of the address to which the query is executed are the same, then the query can be executed. Otherwise, it will depend on how the server has been configured. It decides which website can send queries. Configuration is done by setting the Access-Control-Allow-Origin header and indicating the origin parameter for which it is possible to send a query. In theory, it is possible to indicate the * sign there, but this undermines the sense of this mechanism. Therefore, the CORS application was configured in such a way that queries could only be sent from the client application. For auxiliary purposes, the ready-made cors library was used, which allows you to conveniently and clearly configure the operation of this mechanism.

15. Storing articles and table of contents

The key assumption of the project was the ability for the user to create articles and then build a table of contents that should be displayed to every reader. It is true that the client was entrusted with a greater scope of responsibility for creating articles, but nevertheless, it is such an important functionality that the work also presents a detailed description of its implementation on the server side. The handling of articles and the table of contents is presented in separate sections.

Articles

In order to store articles in the database, the Article model was designed . It has the following fields:

- name – public name of the article, displayed, among others, in the table of contents
- slug – a unique (within all articles) identifier that can be displayed in the URL without any additional coding (for example , crypto-course). It can consist of lowercase letters, numbers and the hyphen -.
- public – flag (logical value) informing whether the article has been published. If set to true , any reader can view this article. Otherwise, the article is hidden and only the content creator has access to it.
- content – an encoded JSON object that stores the entire structure of the article, i.e. the widget instances located inside it, along with the indication of where they appear and their configuration.
- children – an array of references to other documents (articles), which is used when generating a nested table of contents
- parent – reference to the parent article; each article created must have it, with the exception of the article that is the root of the table of contents tree

Article management comes down to four CRUD operations. These operations are listed below, along with the queries sent to the server to perform them

- downloading an existing article – GET / articles /@: slug

- creating a new article – POST / articles
- update an existing article – PUT / articles /@: slug
- deleting an existing article – DELETE / articles /@: slug

All the queries listed above that use the @: slug parameter also exist in alternative variants with the :id parameter. To handle situations when the user wants to perform mass operations on articles, for example add several new ones, a separate mechanism has been provided, integrated with the table of contents management.

Contents

From the point of view of the server application, handling storing and generating a table of contents was a much more complex task than in the case of articles. The server does not interfere with the structure of the article and does not try to interpret it, and in the case of the table of contents the situation is different. The table of contents, like the article, is described using a JSON object. The structure of this object has been strictly defined, because both the client application and the server application must be able to interpret and process it. To illustrate it, Listing 3.10 shows the appropriate type definition in TypeScript . From the listing it can be concluded that the table of contents is a tree and must have exactly one root. Each element has attributes such as:

Listing: Definition of a type describing the structure of a table of contents in the Type language - Script

```
interface TableOfContentsEntry { name: string;
  _id: string; slug: string; public: boolean ;
  children?: TableOfContentsEntry []; [key: string]: any;
}
```

- name – name; is displayed in the table of contents
- slug – is used to generate URLs pointing to a specific article
- id – immutable identifier of the resource (article), it is used wherever it is necessary to clearly refer to the article (the slug is also suitable for this in most situations, but not on the table of contents configurator panel, because the slug can be changed from there)
- public – article publication status

Additionally, an element, like a node in a tree, can have children. Hence the presence of the children attribute , which is an array of subsequent articles. The absence of such an attribute is equivalent to an empty array. The additional marking [key : string]: any means that additional attributes can also be used in this structure, but without specifying their technical name. The table of contents, unlike articles, is not stored directly in the database, but is a resource from the point of view of the server application. This means that the client downloads the table of contents from the server and sends its modified version to it, although

the server does not actually store it directly in the database. Instead, it interprets it and translates it appropriately into the structure of the articles. Generating a table of contents was implemented in such a way that the server first queries the database for a set of articles and then, using recursive functions and information about connections stored in individual documents, builds a tree in the form of a table of contents. The object generated in this way is then returned to the server and rendered in the form of a component tree. Querying the database for all existing articles may not seem efficient, but this is a fully conscious decision by the authors. This choice was dictated by the fact that no article in the system can be an orphan (not have a reference to the parent article set in the parent field), and the application itself is adapted, in principle, to a single course. Therefore, ultimately all of these articles will be used to build the table of contents, so repeated querying of the database for individual articles or defining recursive queries is unnecessary in this case. Moreover, they could worsen the performance of the entire operation and excessively load the database server. An equally important issue as generation is the interpretation of the updated census content sent by the client. Such a request is sent when the process is completed configuration of the table of contents by the user. Therefore, after modification, the following elements may have changed :

- new article (or articles) added
- existing article (or articles) deleted
- the order of articles within the parent article has been changed
- the parent of the article has been changed (for example, the article has been moved to another chapter or to a different level of the table of contents altogether)
- the metadata of a specific article has been changed - for example, name, title or publication status

The server is obliged to detect all the changes mentioned above and influence the system state in the expected way. For this purpose, the JSON object sent by the client, representing the table of contents tree, is first decomposed into a list of articles. If the article does not yet exist in the system, a new database identifier is generated on request. This entire process is handled by a recursive procedure, the result of which is an array containing a flattened list of articles, referencing each other only by article.

Next, the list of articles is reduced to a graph and sorted topologically. Ultimately, individual articles are created or modified. In response, the user receives information about the success status of the operation.

16. Notification mechanism support

The system is also equipped with a notification mechanism. Thanks to it, a person with appropriate permissions, for example a website administrator or a person creating content, is able to send a specific message to a specific group of people. A quite obvious use case is when the course author wants to inform readers about changes, for example adding a new topic, widget or correcting a substantive error in the course content.

As part of the engineering project, this functionality was implemented in the following scope: a user with appropriate permissions (with notifications.send permission) can send a notification with specific content to all users who belong to the role (or multiple roles) indicated by the sender. . So it is possible to send a message to all readers, readers and writers or even all users in the system. However , it is not possible at this time to send a notification to one specific user - unless he is the only one assigned a specific role. The sender of the notification is excluded from the group of recipients, so he cannot receive the notification that he sent himself. To the extent that an application has been implemented for the needs of an engineering project, the user receives a notification when logging in to the website or refreshing the page. Therefore, it does not take place in real time, nor does it use the push notification mechanism , but there will be no problem implementing such a mechanism in the future, as the key foundations have already been prepared.

At the stage of designing database entities, it was decided to define two models:

- Notification – stores information about the notification sent by the sender
- Subscriber – describes the relationship between the notification recipient and the notification itself

The first model, Notification, contains the following attributes:

- content – content of the notification; mandatory field
- sender – reference to the sender (User model); mandatory field
- roles – an array of references to the roles that were selected when sending the notification; This field is for information purposes only and is not used to distribute notifications
- additionally, automatically generated createdAt and updatedAt fields are included

The second model, Subscriber , has the following fields:

- subscriber – reference to the recipient (User model); mandatory field
- notification – reference to the notification (Notification model); mandatory field
- read – a flag indicating whether the user has read the notification. The user can change this flag multiple times, so after reading it, he can mark the notification again as unread notifications.send permission can send a notification . To perform this operation, the client sends a POST / notifications request , with a JSON object included in the content, defining the content and the list of recipient roles. This object is, of course, validated for its correctness. To receive notifications by individual users, the GET /me/ notifications query has been defined . By default, it returns a list of all notifications that have been historically sent to the currently logged in user. By querying the server, the client can additionally limit the set of notifications to unread ones or those that have been previously read. This is done via the read parameter . For example, reading notifications that have not yet been read is done with the GET /me/ notifications?read =0 query. Since the GET method should be idempotent, reading the

resource should not affect its status. Therefore, the mere fact of performing a GET query on the list of notifications does not mean that all of them will be marked as read. This is done with separate PUT queries. There are two types of such queries:

- PUT /me/ notifications / read – marks all notifications received by the user as read
- PUT /me/ notifications /:-id/ read – marks one notification with the specified ID as read.

17. Application front-end

This part presents details about the implementation of the front- end part of the application, i.e. the client.

18. React

React is a free, open-source library that allows you to build dynamic user interfaces. The choice of this library turns out to be so crucial that it determines the development of the entire application, hence it can be said to be a framework . React operates on a virtual DOM (Document Object Model) tree. Thanks to the combination of HTML and JavaScript languages (and the TypeScript variant used by the authors), it allows you to build components that separate individual elements of the program logic, as well as how individual elements of the user interface are displayed . To define the structure of elements, React uses JSX5 (or TSX, respectively , for TypeScript) notation. React's declarative nature makes code more predictable and easier to debug. There are also alternative frameworks that allow for similar tasks. The leading ones , apart from React , also include Angular and Vue. React was chosen , among other things, due to its greatest popularity, which was not without significance because the choice of front- end application development technology also determines the method of creating widgets , as they are also created using the React library . An additional argument was the fact that the authors of the project had the most experience with this particular library. Theoretically, it would be possible to prepare such an application without using this type of framework . However, you should be aware that its architecture is quite complex, and the implementation of individual mechanisms (for example, ensuring reactivity or routing) would be complex and redundant. Additionally, choosing a framework that allows you to divide the code into components significantly helps in maintaining it.

19. React Router

In classic web applications, opening a subpage, both by entering its address directly in the URL bar and by clicking on a hyperlink, involves sending a GET request to the server. In response, a generated HTML document is returned. This is a very good, effective and proven method of acquiring resources, but it can be improved thanks to the single-page application (SPA) paradigm. In this approach, all requests to URLs within the same page are handled using client-side code. From the user's point of view, the application does not refresh when switching between individual views. Many modern web applications actually consist of a single HTML document, but they behave as if there were multiple pages. When you switch from one view to another, the URL changes, although the document is not actually reloaded. This functionality is provided by the React Router library used by the authors. It performs

conditional rendering of the components to be displayed, depending on the path included in the URL, without sending another request to the server to download resources. Instead, the application immediately renders a new interface. Server polling is limited 5JSX – an XML-like notation that allows the structure of HTML elements to be nested within the JavaScript code, as well as the JavaScript code within them only to download new information (in the case of this application - for example, new article content or a new table of contents). However, this is done only in strictly defined cases and at the customer's request, i.e. only when the customer really needs it. This solution makes the entire application run faster because the browser does not have to request a completely new document from the server each time. On the one hand, this increases the initial loading time of the application, because more resources must then be downloaded. On the other hand, this happens only once, and the advantages of this solution are visible all the time. It allows for a more dynamic user experience with the interface itself and with page events, such as animations. Many things happen live and it is much easier to provide the user with the appropriate UX (User Experience) The key issue of integrating our application with React Router is to place a special component associated with this library, called `<BrowserRouter>`, in the main component. This is the element that performs the entire logic of switching between subpages, so it must be located at the top level of the Virtual DOM tree. Then the component was used by the authors `<Switch>`, which is a kind of switch that allows you to display only one page at a time. Passes information about the current page to the `<BrowserRouter>` component, which displays the appropriate view. Tags are placed inside the Switch component, as per convention `Route` . These are links between components and have special path and component attributes. They denote, respectively, the path to the view (with optional parameters) and the component that renders the specific view.

The following views are defined in the application:

1. `{ path: "/login", component: Login }`
2. `{ path: "/register", component: Register }`
3. `{ path: "/auth/:provider/callback", component: OAuth2CallbackPage }`

4. `{ path: "/confirm/:token", component: ConfirmEmail }`
5. `{ path: "/reset-password/:token", component: ResetPassword }`
6. `{ path: "/forgot-password", component: ForgetPassword }`
7. `{ path: "/not-found", component: NotFound }`
8. `{ path: "/a/:slug", component: BoardSkeleton }`
9. `{ path: "/s/:query", component: SearchPage }`
10. `{ path: "*", component: () => <Redirect to="/not-found" /> }`

The order of defined views is important. The code, powered by the React Router library , iterates through the entries mentioned above and displays the first one for which the address was matched. Therefore, an entry that will match to a free path is placed at the end of the list . If it has not been matched anywhere before, it means that the path does not exist, so you are redirected to a page informing about the non-existent address.

The above steps allow you to navigate our application only by entering the URL in your browser. To make routing more useful, links are placed on the website that will redirect you to the desired path. The Link component is responsible for this. It has a special attribute to, in which the path to which the browser should go after These are the impressions that the user experiences while operating the application the user clicks on the link. When the browser detects that a given link has been clicked, it will load the appropriate view in the BrowserRouter component , according to the matched Route component , which, as previously mentioned, knows what component should be rendered . Please remember that not all views in the system have their own address. Some, such as the table of contents configurator, are presented in a modal window and cannot be accessed directly by entering a specific address in the address bar.

20. Redux

In the traditional approach to managing state in the React library , it is stored separately in each component, or more precisely - in each instance of a given component. This is a good approach in most cases, but at some point it becomes unscalable. In larger applications, it is difficult to manage state that is scattered throughout the application and isolated for individual parts of the application. During the development of the project, the authors have often encountered situations in which we need to know the state of a component that is located in a completely different part of the Virtual DOM tree. Such a case makes it necessary to create contexts or chain-add props attributes for components nested within each other. This approach significantly reduces the clarity of the code and increases the complexity of the application. The connection between the individual components becomes very large, and they are difficult to reuse within the same application. The solution to this problem is a global state management mechanism. A natural choice for a framework React is a Redux library . It is a centralized and flexible state manager. Thanks to it, in certain specific situations , instead of using the local state , oriented to one instance of the component , it is possible to use store , i.e. the main container storing the global state of the application . An example of such a situation is storing information about the currently logged in user, or whether the application is in edit or preview mode. Many different system elements need access to such information, and they are often located somewhere completely different. On the other hand, the state of individual widgets is not stored globally because there is no need for it. You should be aware that it is necessary to find a golden mean already at the application design stage . Operating on the global state is based on strictly defined rules. You can only modify state by performing actions that modify the state in a way that ensures its integrity. The reducer is responsible for performing the action, i.e. directly modifying the state . Any elements of the state can be read, as they do not negatively affect its consistency. There are many approaches to initializing a global state. For this purpose, the authors used the createStore function call . Appropriate

reducers , initial state, and possibly middleware are placed in it (in this case, the Redux Saga library is used, which will be discussed later). As mentioned earlier, an important approach in the Redux architecture is the inability to directly modify the store . Data flow starts with an action. Such an action is created by a factory function called action creator . The action created in this way is transferred to the store using a function called dispatcher . In turn, this function calls the appropriate reducer , which performs the appropriate operations on the store . Only he has the right to modify the global state. Redux sense , is only used to modify state. Sometimes, however, it is necessary to define a certain chain of operations that modify the application state, while performing side operations, called side effects. To manage them, the previously mentioned Redux Saga library was used .

An example of a complex operation that was implemented using the capabilities of this library is, for example, the user login process. A new loginUser action has been created in the system . During its initiation, parameters such as e-mail address and password are transferred using the dispatch function . The system listens to what actions are being performed and in this case, apart from overwriting the global state, side operations are also initiated - in this case, sending a query to the server . After obtaining the response, the global state is modified again, this time information about the success of the entire operation and, possibly, data of the logged in user are set. This approach allows you to extract complex logic from a single component, which is especially useful in situations where such an action could be initiated in many different ways.

21. Widget handling mechanism

An important goal that the authors set themselves during the implementation of this project was to design and implement a widget mechanism . In principle, from the point of view of the architecture of the entire system, widgets should be an entity separate from the base application that communicates with it through a strictly defined interface. This approach reduces the connection between the system core and specific widgets . This makes it easier to maintain order and organization in the source code, and above all, it is possible to develop widgets independently, also by people who are not part of the team creating the base system. The individual parts will discuss the assumptions and requirements of such a mechanism and the details of its implementation in the case of this project.

Assumptions and requirements

The desire to implement the above assumptions presented the authors with several significant problems, both of an architectural and implementation nature. First of all, it was necessary to analyze the potential use cases of individual widgets and consider what specific features distinguish individual widgets from each other. The conclusions drawn from the analysis are presented below. Each widget is associated with a certain presentation layer that the user who reads the article has direct contact with. This will henceforth be called the reader's view. If the assumptions require it, in addition to visual issues, it must also support certain interactions with the user - for example , clicking a button, editing the contents of a text field, and so on. the widgets it contains will always look the same, regardless of the circumstances. Others, in turn, will depend in some way on the defined state, hereinafter referred to as the

widget configuration . To illustrate this, you can use examples of specific widgets that were implemented as part of this diploma project. An instance of a widget that provides functionality in the form of a single- or multiple-choice question must store some configuration. This configuration contains information about the content of the question and the list of answers, including the correct ones. In this particular case, storing the configuration is important because each instance of such a widget can, or even should, present a different question. A similar situation occurs in the case of a widget that allows you to add content to an article. Everyone the article may be different, hence the need to store the configuration of each instance of such a widget. In turn, the widget that supports the Playfair encryption algorithm does not need to store additional configuration . It should look exactly the same in every place where it is placed. The above considerations led to the conclusion that in addition to the reader's view, some widgets should also have a configuration view from which it would be possible to manage the parameters of a specific instance. Widgets , apart from graphic elements, consist primarily of text. The base application supports multilingual support to some extent, so it is important that widgets can also support it, presenting appropriately translated content, both in the reader's view and in the configuration view. Widgets should also respond correctly to theme changes to maintain visual consistency with the rest of the system.

Adding a new widget

Creating a new widget is as simple as creating a subdirectory in the `src / extensions / widgets` path . The name of such a directory is , of course, unique and serves as the technical identifier of the widget. `index.tsx` file is created in this directory , where the widget is defined . Defining is done by creating a special configuration object, which is then exported from the module. This object defines elements such as:

- metadata – widget name, category list,
- initial state,
- an object defining translations of labels into different languages,
- a component supporting the reader's view,
- a component that supports the configuration view.

he `makeWidget` function is used , provided by the SDK (Software Development Kit), prepared specifically for creating widgets . An example call looks like Listing 3.11.

Listing: Example of a widget definition

```
export default makeWidget ({ translations,
metadata: props => ({ name: props.t('name'),
category: ['basic'], initialConfig : {
text: defaultText ,
},
}),
```

```
viewer : Viewer, configurator : Configurator ,
}))
```

Each widget has its own configuration object , this also applies to widgets that do not have a configuration view - in this case, the widget configuration will simply always be the same. During object definition, using the `makeWidget` function, the `initialConfig` parameter is defined , which contains the default value of the configuration object. It should be mentioned that the configuration is always a JSON object, while the specification does not define the form and fields of this object, it can even be nested multiple times. However, remember that each widget instance stores a separate configuration, so don't store a lot of information there if you don't need it. For non-configurable widgets , this configuration never changes anyway, so it's best not to store any information there. The user defines an object with translations and passes it as a parameter. The system loads them and then provides the widget with the translation function `t()`. An example object with translations is presented in Listing.

Listing: Example of translation definitions used within the widget

```
{
  "p1": {
    "name": " Markdown Content "
  },
  " en ": {
    "name": "MarkdownText"
  }
}
```

Due to the fact that a framework was used to implement the base system React is also used when creating widgets . Both the reader view and the configuration view are separate React components . The configuration parameters of individual components are defined within the SDK. The definition of the component supporting the reader view of the example widget is presented in Listing. As you can see in the example, in the `props` parameter the component receives a number of information needed to operate, such as the current configuration (`props.config`), language (`props.lang`), or theme (`props.theme`).

Listing: Example of a reader view component

```
export default (props: ViewerProps ) => {
  const text = props.config?.text ?? defaultText return (
    < React.Fragment >
    < VendorConfigProvider lang ={ props.lang } theme={ props.theme }
```

```

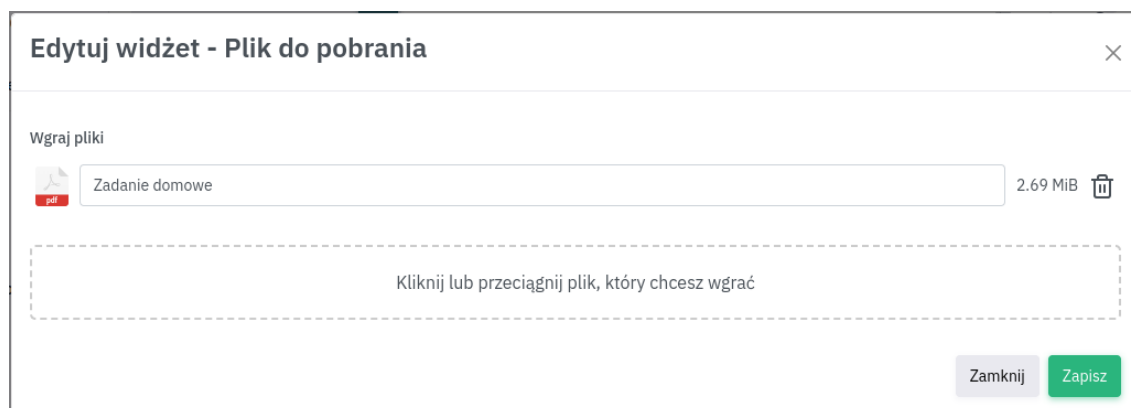
component={({locale: any, plugins: any}) =>
<Viewer value={text} plugins={plugins}
/>
}
/>
</ React.Fragment >
);
};

```

The nature of the component that handles the configuration view is slightly different. In addition to reading the information passed to it by the props argument , it also participates in modifying the configuration . However, it does not modify it directly, but stores it in the component's internal state . At the request of the parent component responsible for operating the configurator window, a temporary configuration is made available and then saved directly in the article.

To illustrate this mechanism, a screenshot is presented in Figure. Elements in the window header and footer are handled by the underlying system . The widget author only defines what is displayed in the middle. When the user presses the Save button, the parent component initiates the operation to download the modified configuration from the child component by calling the method. This operation is asynchronous, because during saving, queries may be sent to the back-end application - in this case, the files must first be uploaded to the server. Ultimately - no, this function returns a new object that the system saves in the object storing the article.

Figure: Sample configuration view



Due to the nature of React , this type of querying of a child component by a parent component requires the use of slightly more sophisticated mechanisms offered by this framework than in the case of classic communication via props . Specifically, hooks7 useRef and

useImperativeHandle were used here , as well as the forwardRef function - the author of the widget uses only the last two. The code for an example component implementing the configuration view is presented in Listing. This code has been deliberately removed from certain fragments in order to limit it only to the elements that are important at this moment. As you can see, the initial state of the component is taken from the props object . It is then updated when the configuration changes . When the Save button is clicked, the save function , defined inside useImperativeHandle, is called . It returns the current configuration, which is then saved in the article content. As you may have noticed earlier, each widget has a list of categories assigned to it. This mechanism allows you to divide the list of widgets thematically, as shown in screenshot. Thanks to this, the list is somewhat ordered and it is easier to find the widget that interests the user . Each widget can be assigned to more than one category. Some of them may already exist in your system, but certainly not all of them. If anyone would like to create a thematic group

hook – mechanism provided by the framework React . Its use comes down to calling a special function inside the component definition, such a function has a name starting with use . This mechanism is based on the mechanism offered by JavaScript closures .

Listing : Example of a component supporting the configuration view

```
export default forwardRef ((props: ConfiguratorProps , ref: any) => {
  const [config, setConfig ] = useState ( props.config )
  useImperativeHandle (ref, () => ({
    save: async() => config
  })))
```

```
function updateText (text: string) { setConfig ({ ...config, text })
}
```

```
async function uploadImages (files: File[]) {
  const uploadedFiles = await Promise.all ( files.map (file =>
    uploadPublicFile ({ name: 'Picture ${moment()}',
      public: true
    }, file)));
```

```
return uploadedFiles.map (({ url }) => ({ url }));
}
```

```

return (
  < React.Fragment >
    < VendorConfigProvider lang ={ props.lang } theme={ props.theme }
      component= {(locale: any, plugins: any) =>
        <Editor value={ config.text } plugins={plugins} locale={ locale.editor
        }
        editorConfig ={{ lineNumbers : true }} onChange ={v => updateText (v)}
        uploadImages ={ uploadImages }
        />
      }
    />
  </ React.Fragment >
);});

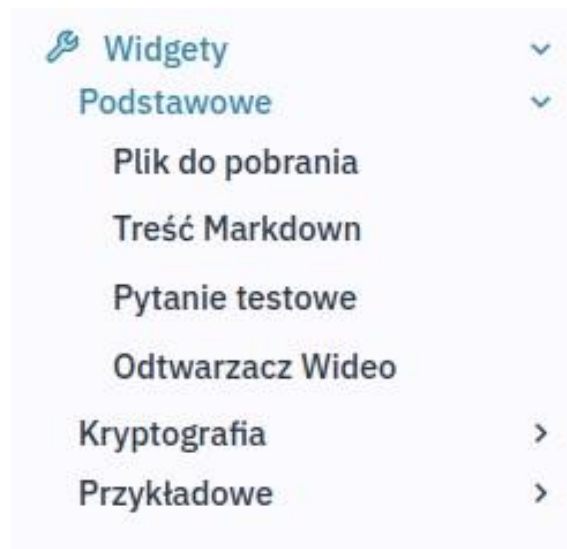
```

Widgets , you can also create your own category. Defining new categories is similar to widgets , although slightly simpler. To create a new category, create a directory in the src / extensions / categories folder . The unique name of this directory also serves as a category identifier. Using this identifier, the widget determines its affiliation to particular categories.

In this directory, you should then create an index.ts file in which a configuration object is created using the makeCategory function provided by the SDK . It consists of:

- metadata – in this case only in name,
- translation definitions – this mechanism is operated in the same way as in the case of widgets . An example category definition is presented in Listing.

Figure: List of widgets



Listing: Definition of an example category

```
export default makeCategory ({ translations,  
metadata: props => ({ name: props.t('name')  
}))  
})
```

Implementation

The implementation of the mechanism supporting widgets required, first of all, the preparation of an SDK , the use of which was presented earlier. Factory functions such as `makeCategory` or `makeWidget` communicate with the system, obtaining information needed to initialize the widget instance , such as the currently selected language or interface theme. This information is obtained from the global application state, which widgets cannot access. Then, among other things, the translator is initiated, for the translations given in the widget configuration, the widget name is translated , and the initial configuration is downloaded. This process is somewhat complex and uses the concept of higher order functions. For example, Listing defines the `makeWidget` function .

As you may have read earlier, all widgets are stored in separate sub-directories in the `src / extensions / widgets` path . To load all scripts in this directory , the functionality provided by the webpack tool , i.e. the `require.context` function, is used . Its call to load widgets is presented in Listing 3.19. Based on this list (as well as the list of categories obtained in a similar way), a widget tree is generated , from which elements are dragged to the workspace when creating an article.

This tree is only displayed if the user has appropriate permissions and edit mode has been enabled.

There is a component in the system that is the parent of a widget . It is within this component that the components defined by the widget are rendered : the reader view and the configuration view . If the user is in edit mode, when you hover over the widget , you can remove it from the article, as well as the option to edit it. After clicking on it, the presented one opens

Listing: MakeWidget function definition

```
export interface WidgetProps { theme: string;
lang : string;
}

interface CommonProps extends WidgetProps { t(label: string): any;
}

export interface ViewerProps extends CommonProps { config:
WidgetConfiguration ;
}

export interface ConfiguratorProps extends ViewerProps { onConfigSave
(configuration: WidgetConfiguration ): void;
}

export const makeWidget = ( widgetFactoryConfig : WidgetFactoryConfig
) => { const translations = translator(
widgetFactoryConfig.translations );

return ( widgetProps : WidgetProps ) => {
const translation = translations( widgetProps.lang ); const
commonProps = { ... widgetProps , t: translation };
const injectProps = (props: any) => ({ ...props, ... commonProps });

return {
```

```

metadata: widgetFactoryConfig.metadata ( commonProps ),
viewer: (props: ViewerProps ) => widgetFactoryConfig.viewer (
injectProps (props)), configurator: widgetFactoryConfig.configurator
!== undefined
? forwardRef ((props: ConfiguratorProps , ref: any) =>
React.createElement (
widgetFactoryConfig.configurator !!,
{ ... injectProps (props), ref })
)
: undefined
}
}
}
}

```

Listing: Dynamic loading of individual widget modules

```

const widgetsModules = require.context (" src /extensions/widgets",
true, /index\. ts $/) export const widgets: { [key: string]: any } =
Object.fromEntries (
widgetsModules.keys ().map(key => [ extractKey (key), widgetsModules
(key).default
])
);

```

After clicking the save button, the save operation is initiated, which is defined by the widget author .

The above describes the implementation of mechanisms only for widgets , but it is available for categories analogous, in terms of providing abstraction and dynamic loading.

22. Table of Contents Configurator

The application allows dynamic configuration of the table of contents located on the side panel. This option is only available in edit mode, for a user with appropriate permissions. On the left side of the configurator there is a menu that supports the dragging mechanism. Thanks to this , the course author is able to set the preferred order of individual sections , articles or widgets . This is a place where you can delete or add articles . When you hover the mouse over individual elements, a trash icon and a plus sign button appear. Clicking on such an icon

performs the appropriate action. On the right side there are text fields where you can set the name of the element, its URL address, and whether the fragment should be published. After finishing working on the table of contents, confirm the changes by clicking the Save button .

Figure: Panel inventory configurator

Konfigurator

Kurs kryptograf
Wprowadzenie
Szyfry historyczne
Podsumowanie
Sprawdź się
Strona Demo
Pomocje skróty
Opinie o aplikacji

Nazwa artykułu
Opinie o aplikacji

Adres URL
https://wos.komputeryk.pl/a/opinie

☒ Opublikowane


Cancel Save

23. Login page

Each person using the platform should first create an account and wait for its approval. For login purposes, a separate subpage has been created, allowing you to enter your data and go to the platform.

Construction of the login page

The login page design is divided into two main sections. The first one is the background, which is on the right and takes up most of the screen. The background has an interesting ability to display various quotes, including captions. They appear in a loop, adding charm to the entire application. The quotes concern famous people in the field of cryptography and motivation to learn, which only encourages people to use the platform. On the left side there is the actual login panel. It is divided into three main sections. The first one is the logo and name of our application, a greeting to the user and an encouragement to enter data.

 Web Crypto Center




Sign in

E-mail address


Password [Forgot password?](#)

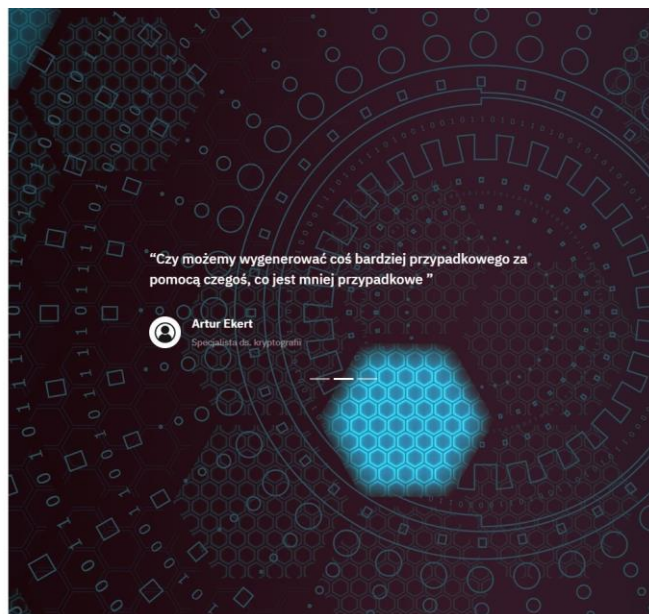
Sign in

Sign in with

Don't have an account? [Sign up now](#)

Web Crypto Center - ©2023 NMaszin 



The next section contains headers and two fields in which the user can enter his data (email and password). They are subject to strict validation, both on the client side and on the server side. In the case of an e-mail address, there must be an @ sign, and two special characters cannot appear next to each other. The password is subject to standard validation that can be found on many websites, which means that it must have at least eight characters, at least one uppercase letter, one number, and one special character. If these conditions are not met, the color of the field and header will change, and an exclamation mark and a warning will appear that the field is invalid.

Email

This field is invalid

Between the email and password fields, there is a link called [Forgot Password ?](#). It uses React Router and brings up the password recovery page. When the data has been entered correctly and no validation errors have occurred, the user can confirm the form by pressing the Log In button, which is located at the very bottom of the second section. The last section begins with buttons that allow you to log in through apps like Google, Facebook, and Discord. Once the user makes a selection regarding the application, he or she will be redirected to a separate page of the selected portal, where he or she will be authenticated. A detailed description of the procedure for logging in via external services has been described earlier. Just below, there is an encouragement to create an account. We have the Sign button at our disposal up now , which redirects you to the registration page. There the user gains access to a separate formula, where he can create an account that will be used to use the application in the future. At the very bottom of the section there is a footer containing information about the application, such as: information about the year of creation, the name of the application, and the name of the team developing the application.

24. Theme Customizations

When using a web browser, each user has his or her own habits regarding the arrangement of individual windows on the website. In order to adapt the application to a specific person, the application authors included theme configuration functionality. The appropriate panel slides out from the right side of the window after clicking the button. This panel allows you to configure most of the elements on the website. You can set, among other things:

1. color of the entire motif,
2. application width (square, full screen),
3. information whether there should be a scroll ,
4. color of the upper beam,
5. side panel size,
6. side panel color.

Additionally, the website is friendly to people with disabilities, so it allows you to access each option using only the keyboard.

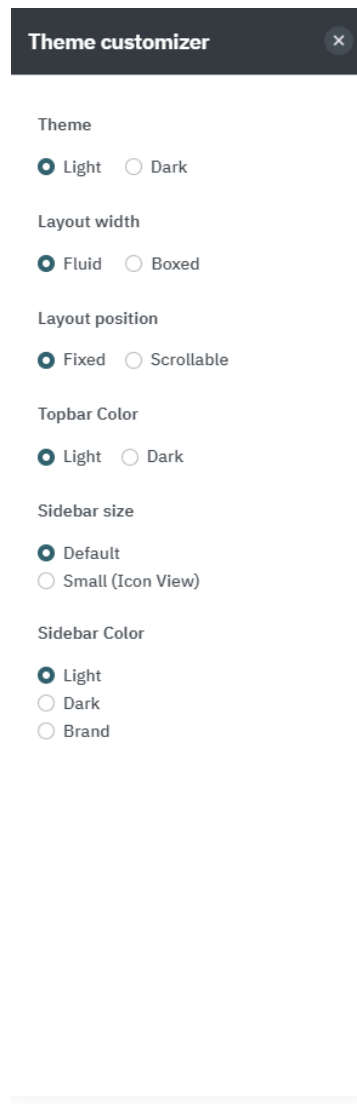
25. User management

All application users are assigned exactly one role. There are four roles built into the system by default:

- Unverified user – the user assigned to this role does not have any permissions; after logging in, an appropriate message is displayed informing him that he must wait for manual verification by the administrator
- Reader – a person who can view content published on the website
- Writer – a person who can additionally modify them, this applies to both changing the content of individual articles, as well as creating and deleting them, and managing the table of contents
- Administrator – this is a person with full administrative rights, in addition to the rights listed in the previous points, he can also manage registered users , among others .

Users with the Administrator role have access to a panel called User Management. In this panel, the administrator can view all users in the system and information about each of them. The following information about each user is displayed in table form:

- User Name
- E-mail adress



- Role
- Registration method
- Email address confirmed

When a new user is registered, he is assigned the Unverified User role by default. At the moment, he cannot use the application until he is assigned a different role. The administrator has the right to change the user role, but must first enable it edit mode in the User Management panel. When edit mode is enabled, it is possible to change the role by selecting one of the roles from the drop-down list. From this point on, the user can use the application. From this panel it is also possible to delete users, thanks to which the user loses the ability to use the application until they register again.

26. Creating and modifying articles

One of the key concepts of this application is the creation of articles, using the mechanism of dragging widgets . It allows you to create a website according to your own requirements. Widgets constitute a certain abstraction, so they can be created by anyone with basic

knowledge of the framework . React , following the rules imposed by the application. This is discussed in detail in other sections. Widgets can take various forms. On the website you can find those that contain: ciphers, cryptographic tools , tools that allow you to create new articles, write, add files, photos, test questions and many others. The program works by opening the appropriate page with the article and finding the item of interest on the right panel. After holding down the left mouse button, you can drag the element from the side panel to the appropriate place on the article. When dragging the widget over the work panel, so-called drop zones are displayed , i.e. places where the element can be embedded. The structure of the dragging mechanism defines four such areas each time: up, down, left, right. It is in such positions, relative to other widgets , that the user is able to position the element that is currently being dragged by him. All items will scale automatically and will share space equally in both rows and columns. The content of the article consists of lines within which widgets are arranged , at least one per line. The work panel takes up most of the screen. It is a workplace, both for the person creating the content and for the person using the program. It allows you to drop widgets and arrange them according to the scheme specified by the author. Dragging elements within it is only possible if editing mode is activated and for users with appropriate permissions. The content creator has a basket at his disposal, which is activated only after dragging any element from the board to its area. It allows you to remove the widget from the workspace and recalculates the space on the page to fill the empty space. The operation of deleting an element can also be performed by hovering over it and clicking the trash icon that appears. The styles in the workspace are designed so that all widgets take up a certain amount of space and do not exclude others on the page. Therefore, if, for example, the user drags three widgets into one row , each of them will occupy exactly one third of the width of the canvas. The authors of the project decided not to limit the height of the widgets because this would result in the appearance of an excessive number of sliders, which would consequently reduce the transparency of the page, worsen the overall user experience and additionally make some important information no longer immediately available to the user . react-dnd library was used . The application code defines several components that are nested within each other and are responsible for handling individual fragments of the article structure. Changing the configuration of a single widget or changing the settings of a widget within an article are treated as its modification. After modification, the content of the article is synchronized with the server. Widgets on the workspace are separated from each other by the size of the drop zone , which is 30 pixels by default. After many tests and considerations, the authors decided on this value because it is ideal from the user's point of view and does not complicate work with the program. If they are too small, they will be difficult to target when dropping the widget , while if they are too large, they will significantly reduce the aesthetics of the program and the amount of space available for use by the user.

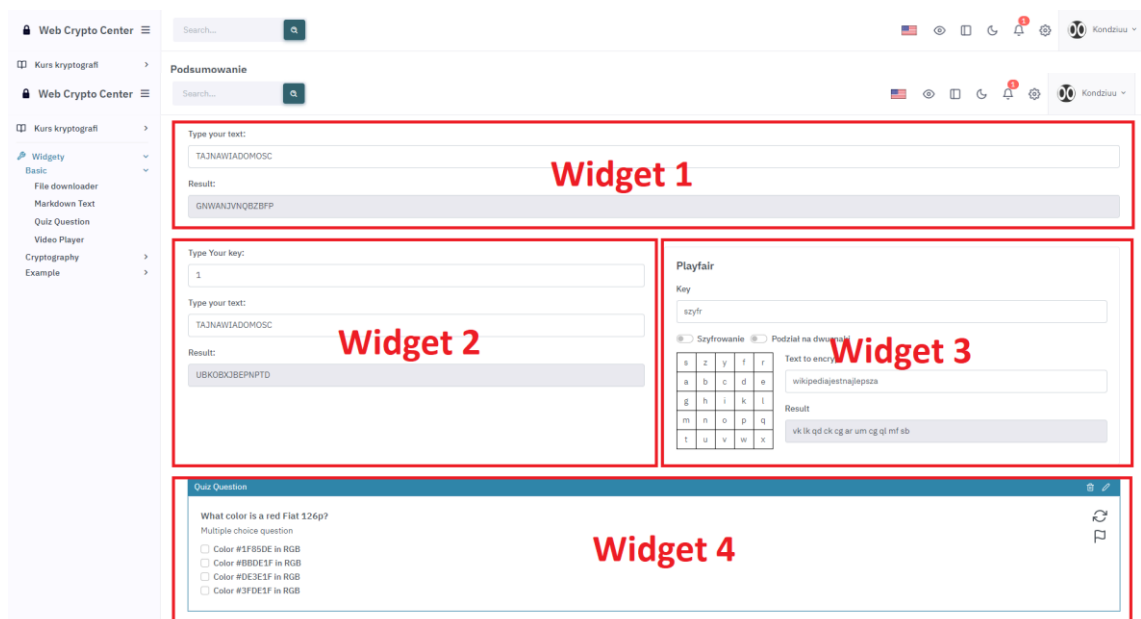


Figure: Example of an activated drop zone

Figure: Workspace

27. Multi-language support

Currently, the most commonly used language in the world is English. The entire world of IT and beyond is based on it. For this reason, in order to reach a wider audience, the authors they decided to support multiple languages, in this case English and Polish. To implement this mechanism, a library called i18next was used. It allows for the internationalization of the application and introduces a new dimension of potential application development, because it can be translated into any language, depending on market demand. It is important to note here that only user interface elements are translated. At the moment, it is not possible to define the same articles in different language versions. However, the idea was only to ensure that anyone in the world could use this application to create their own course, and not to create a course in different languages. The method of configuring the i18next library in the application is presented in Listing.

Listing: Configuration i18next libraries

```
import i18n from "i18next";

import detector from "i18next-browser-languagedetector"; import {
  initReactI18next } from "react-i18next";

import translationPL from "../locales/pl/ translation.json "; import
translationENG from " ../locales/eng/translation.json " ;

const resources: any = { pl: {
  translation: translationPL ,
```



```

},
eng : {
translation: translationENG ,
},
};

const language: any = localStorage.getItem ("I18N_LANGUAGE"); if
(!language) {
localStorage.setItem ("I18N_LANGUAGE", " en ");

i18n
.use(detector)
.use(initReactI18next)
. init ({ resources,
lng : localStorage.getItem ("I18N_LANGUAGE") || " en ", fallbackLng :
" en ",
keySeparator : false, interpolation: {
escapeValue : false,
},
});
export default i18n;

```

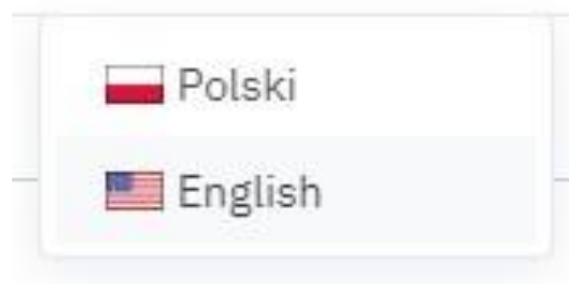
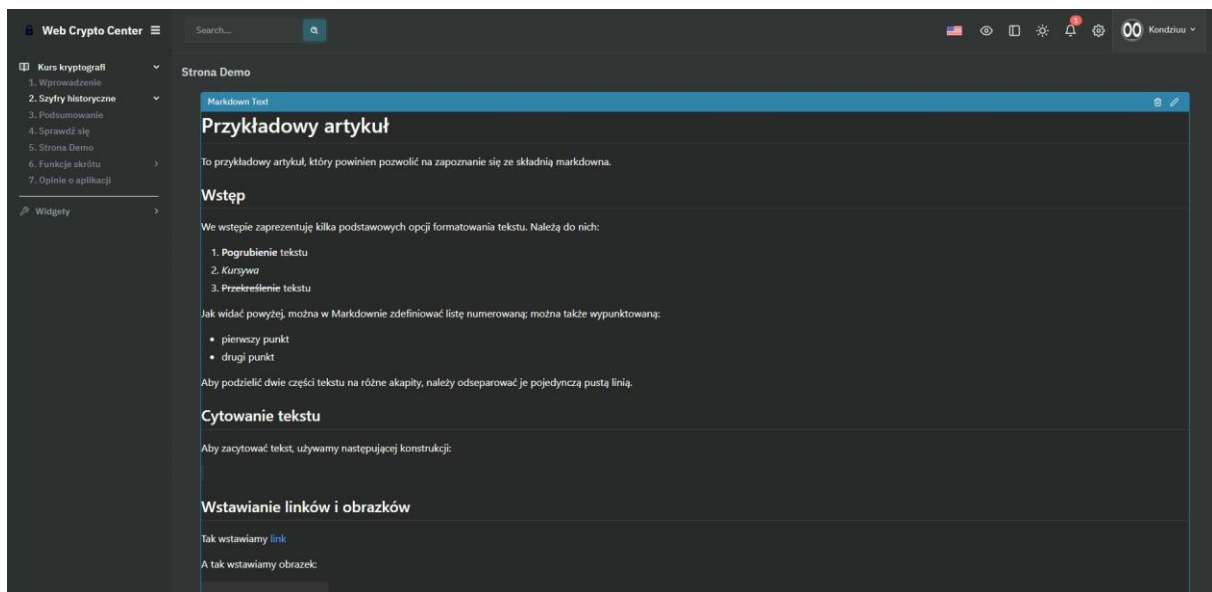


Figure: Language selection list

27. Color themes

In the era of widespread digitization and access to the Internet , the use of devices such as telephones or computers takes place constantly, regardless of the time of day or night.

Unfortunately, it can still be observed that many social networking sites, websites and



applications do not care about users' eyes. It is common knowledge that with a dark theme, the eyes are much less tired, which makes working at night more pleasant. Many developers use dark themes permanently to save their eyes from excessive glare. The authors of the application decided to give a helping hand to all those for whom the color of the theme is important. The website supports two themes: light and dark. This was achieved by declaring a completely different color palette for each of them. Everyone, regardless of their role in the system, can choose a theme that suits them better, thus increasing the potential number of recipients using our platform. Theme switching is done using the Redux library , thanks to which the application maintains a global state containing, among other things, information about the theme. Changing the theme is done by calling the dispatch function , with the action passed as an argument, so that when the theme is changed, other styles are loaded. This is shown in Listing.

Listing: Code to activate the theme change

```
const onChangeLayoutMode = ( value : any ) => { if ( changelayoutMode
) {
dispatch ( changelayoutMode ( value , layoutType ));
}
};
```

Changing the icon, however, comes down to a simple conditional operator , which indirectly contains reading the value from store . This is shown in Listing.

28. Notifications

Currently, most websites offer their users, so the authors of the work also decided to implement this mechanism and make it available to users.

Listing: Code that changes the icon, depending on the current color theme

```

{ layoutMode === layoutTheme ["DARKMODE"] ? (
<Icon name="sun" className ="icon-lg layout-mode-light" />
) : (
<Icon name="moon" className ="icon-lg layout-mode-dark" />
)}

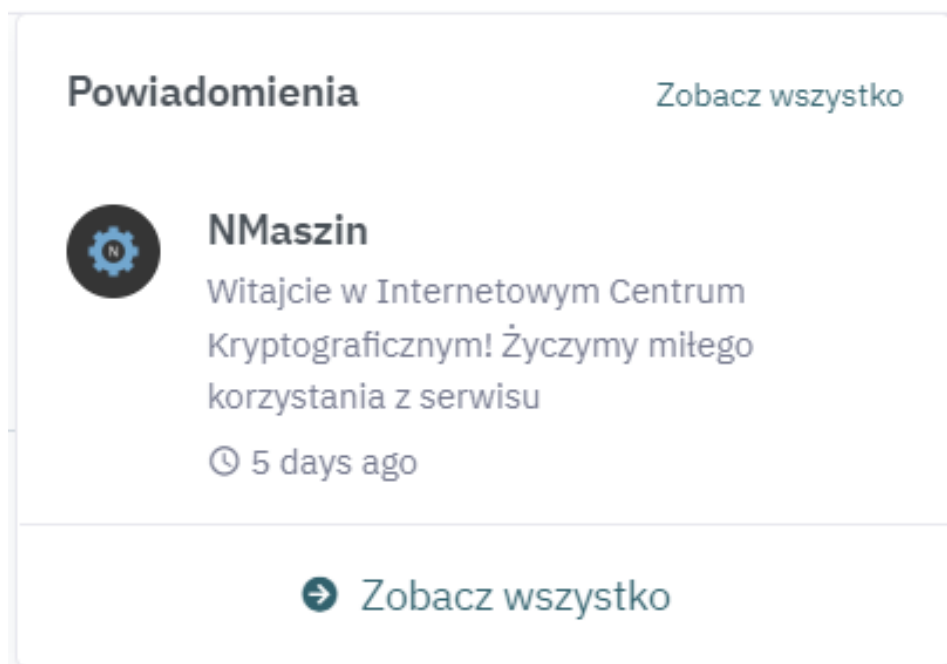
```

Notifications allow for one-way communication with groups of users, and especially with all users. The sender can include any message in the notification, usually it is short and informs about actions taken on the website, such as adding some functionality or an expected technical break. Notifications can be accessed via the icon located on the top navigation panel. After turning on notifications, a window appears with several options. It displays the last notification, information about the user who sent it and the time it was sent. Additionally, we can access the notification window containing more information by selecting the "See all" option. After going to the "See all" section, we will be transferred to a new window that provides new functionality, namely filtering. Filtering allows you to filter using 3 modes:

- all notifications,
- read,
- unread.

29. Other tools

Material UI is an open source React component library . It contains an extensive collection of components that are ready for production. Predefined components have



very important in large projects that are implemented by a very small team of programmers, in a relatively short time - such as this one. They allow you to focus on the actual features of the program and achieving the presented goals, without wasting time on obvious issues that have long been available in libraries.

ESLint is a so-called linter, a tool intended for static code analysis. It is able to detect errors that would not normally be detected at compilation time, but only at execution time, as well as inconsistencies at the level of the style of the written code.

Bootstrap is a free programming platform that allows you to create responsive websites and mobile versions. It has a very rich syntax and a mobile first approach which, as the name suggests, recommends the programmer to start creating a website from the mobile view , moving through the tablet to computers.

The main motivation for using this library is, similarly to Material UI, to speed up work and use a responsive CSS grid. Bootstrap also solves many cross-browser compatibility issues, so our platform is not problematic for users, and works on all devices and browsers.