UDC: 004.738.5 https://doi.org/10.15407/jai2024.02.049

V. Falkevych[1], A. Lisniak[2]
[1,2] Zaporizhzhia National University, Ukraine
66, Zhukovsky st., Zaporizhzhia, 69600
[1]vitaliifalkevich@gmail.com
[1]https://orcid.org/0000-0002-1114-7206
[2]https://orcid.org/0000-0001-9669-7858

CLIENT STATE MANAGEMENT USING BACKEND FOR FRONTEND PATTERN ARCHITECTURE IN B2B SEGMENT

Abstract. The article considers an architectural pattern Backend for Frontend (BFF) for developing web systems using microservice approaches. The main purpose of this article is to research aspects that existing solutions like WunderGraph cannot provide and propose a solution that enables client state management using the backend for Frontend pattern architecture specifically tailored for the B2B segment's requirements in Frontend development. During current research, there is defined a concept of API Provider Factory, Public API Gateway, Client State Management and proposes ways for their implementation. Methods of research are based on modeling, analysis, comparing, experiment, and abstracting.

Keywords: BFF, Frontend architecture, Microservices, API Provider Factory, Public API Gateway, AP, Client State Management.

I. Introduction

Backend for Frontend (BFF) is an architectural pattern that provides a dedicated backend for each specific client interface or application. Widely spreading this pattern appeared during the developing paradigm of using microservices approaches in designing backend independent systems. Microservices allow the creation of different API endpoints for use on the Frontend side. Frontend is always a whole system with end user interactions.

Splitting API microservices by entity cannot be split so easily on the Frontend side. Different components can be combined as possible, are used in different cases and pages of the Application, and should be exchanged by shared data like state. So these Frontend components use data from different API Providers (Microservices).

To solve the problem of using Several API Providers in one place there was developed a pattern of design Backend for Frontend and accompanying solutions like WunderGraph [1]. WunderGraph allows developers to create a config file for a special Frontend and build a virtual Graph or API Gateway with combined data from several API endpoints according to the Backend for Frontend pattern. It is a good solution out of the box, but some cases require more specific decisions that the architecture of existing solutions cannot offer.

The main purpose of this article is to research aspects that existing solutions like WunderGraph cannot provide and propose a solution that enables client state management using the backend for Frontend pattern architecture specifically tailored for the B2B segment's requirements in Frontend development.

Methods of research are based on modeling, analysis, comparing, experiment, and abstracting.

II. Analysis of similar decisions

As an implementation of the BFF pattern consider an existing decision WunderGraph. It is a framework that supports many technologies to implement this pattern. For example, WunderGraph provides some interface to create a Virtual graph from several combined Graphql Services and can provide one endpoint for the client that has all the necessary information from different microservices. After virtual graphs are defined, they should be imported into the configuration file for setting parameters for accessing the services. WunderGraph builds a client code for use from the Frontend. This code can include types, queries, mutations, etc. Developers can use it out of the box in their projects. Easy to use has certain restrictions in specific cases often faced in the b2b segment [2].

The first case is splitting public API and private one (Figure 1). For instance, we have

several specific microservices that provide API. Also, we have several Frontend instances that require using these microservices API. Every instance of the new Frontend is a separate project that should be worked in a different namespace. As a new instance of the Frontend is defined, we should generate new credentials to access the common microservices API within their respective namespaces. Credentials for the project should be added to the API microservices only on the server side. Additionally, we would like to hide the true API from public access. For instance, we would like to make one request with a POST method with a special body. This request could be caught on the server side and be split into several requests in GraphQL (or REST) format to the specific microservices. Splitting API into the Public API and Private API is an additional layer of the security web system that allows to filtering of some unexpected requests and confuses potential cheaters [3].

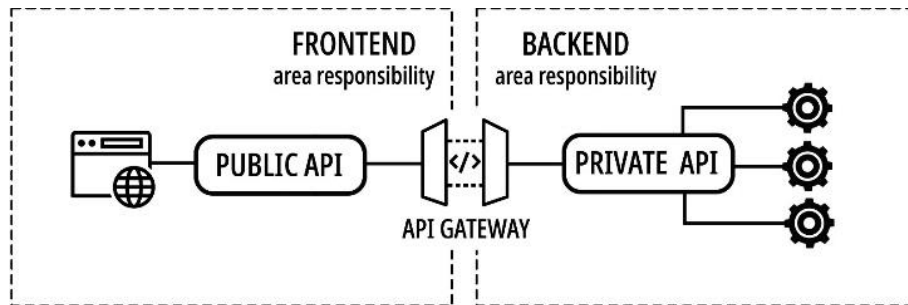

Fig. 1. Splitting API to Public and Private

A mechanism of creating a virtual graph can be difficult to support. Creating operations with composing by type only, requires searching changes and updating in every operation after a specific microservice API is changed. Considered solutions in this article offer to create API Provider Factories for using specific microservices and update only necessary API instead of searching and updating the virtual graphs in each place it appeared (Figure 2).

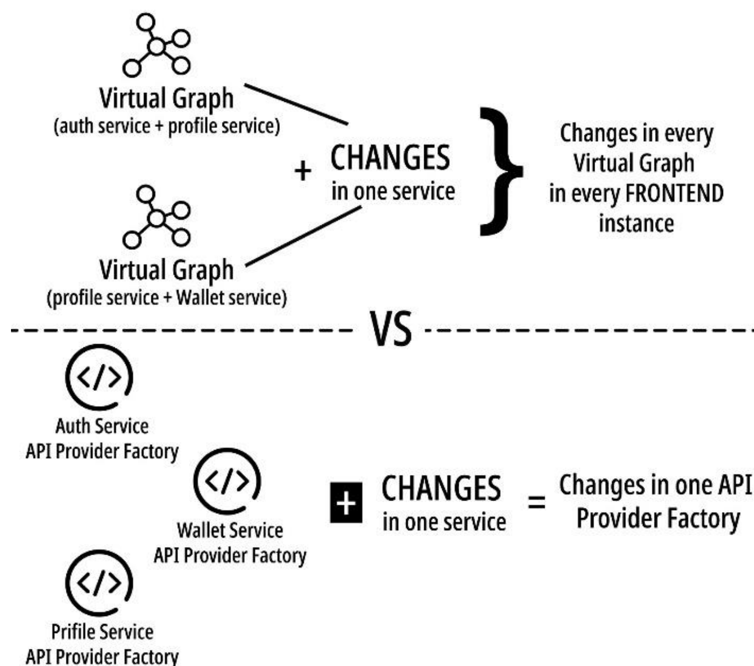

Fig. 2. Comparing Virtual Graphs support with API Providers Factories

In the WunderGraph configuration is offered to link a configuration to the one Frontend instance [1]. It is acceptable in the case when Frontend is only one instance. If we have, for instance, 20 or more Similar Frontends that use common microservices API in their namespaces, it is better to create API Provider Factories and use a simple API Gateway configuration in each Frontend. Every change in the API can be edited only in one place instead of editing every Frontend instance (Figure 3).

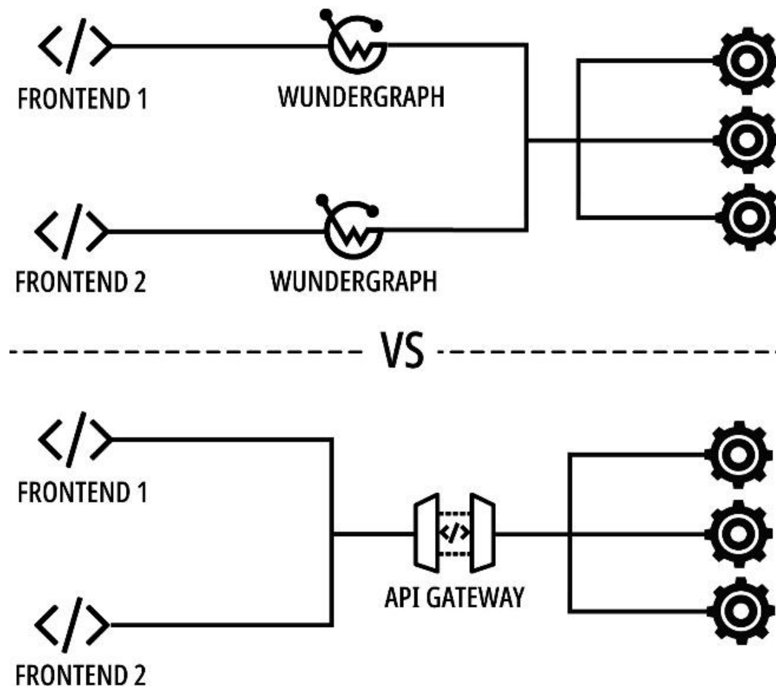

Fig. 3. Comparing Frontend scaling with WunderGraph solution

III. Defining entities and requirements

Before embarking on creating client-state management using the Backend for Frontend pattern architecture, it's crucial to define key entities and establish requirements. Consider Figure 4. There are three main entities of the research: Microservices APIs, Frontend Server, Frontend Client. The first component of the key entities is the API microservices layer. To interact with various Microservices APIs an API Provider Factory should be created. The next entity of this research is a Frontend Server. It is a part of the system of providing interaction with clients. The main responsibilities of the Frontend Server are:

- Making requests to the API and preparing results.
- Making prerender of UI layouts.
- Additional layer of security and validation data between Client and API.
- Additional logic of the system.

In the context of this research, we use Frontend Server Entity like an API Gateway for the Frontend Client.

The third component of the key entities is the Frontend client is a system that can operate on the user's device, such as a browser. In the context of usage, it could be a Frontend built on React (Angular, VueJS, or similar frameworks and libraries).

The primary purpose of this layer in the researched entities is to engage with the end user by providing a User Interface (UI). This entity should initiate requests to the API and share the state with its components. And implies a mechanism for managing the state in this layer. In the context of the research Frontend Client makes requests using Frontend Server instead of making them directly to the APIs of microservices [4].

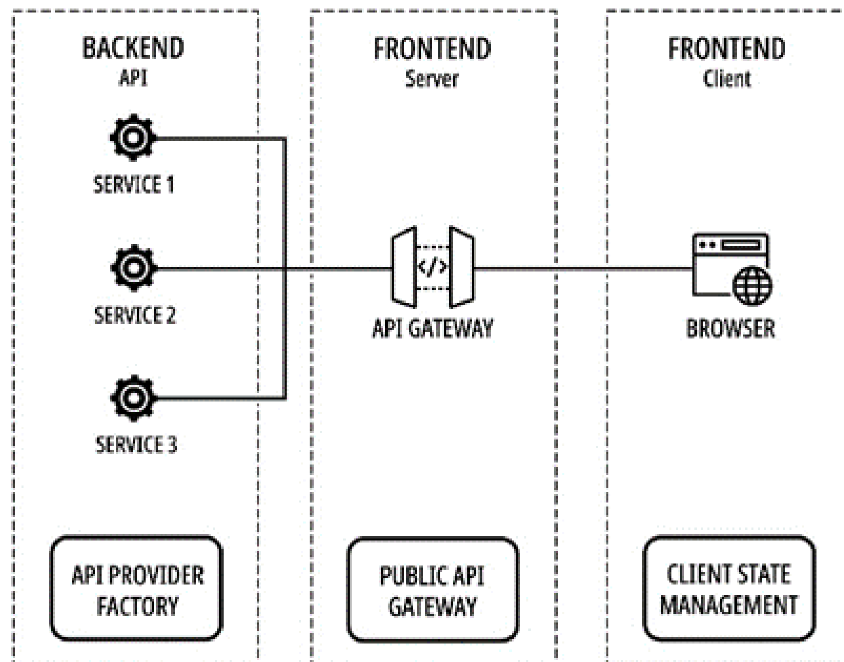

Fig. 4. Key Entities of researching

IV. API provider factories development

The first part of this research involves the development of API Provider Factories for working with the Microservices APIs. Let's take an authorization service that uses the GraphQL API and a service for working with a user profile using a REST approach. We create separate API Provider Factories for working with the API for each service [5].

Use GraphQL Code Generator library during the development of a package for working with a service of authorization [6]. This library will generate a ready-for-using GraphQL code and types. The algorithm for creating a package for working with GraphQL is as follows:

- Creation of a file of settings for code generation.
- Creation of a description of queries, and mutations.
- Generation of ready API and types for further convenient work.

An example of configuring GraphQL query generation and mutations is given in Listing 1. As can be noticed, there are defined locations of the graphql queries and mutations, URL of the schema, set-up plugins for code generation. Also, is defined output path for the generated API, is described in the queries and mutations [7].

```
const config = {
  schema: {
    [schemaUrl]: {
     },
    documents:
'./operations/**/*.graphql',
    generates: {
      [path.join(__dirname,
'src/index.ts')]: {
        plugins: [
          'typescript',
          'typescript-operations',

          'typescript-graphql-request'
        ],
    },
   },
   hooks: false,
   }
   const    output    =    await
cli.generate(config)
   output.forEach(({filename,
content}) => {
     fs.writeFileSync(filename,
content)
   })
```

Listing 1. Configuring the GraphQL API generation of the authorization service

After setting up GraphQL generation code that works as common HTTP POST requests, it is possible to generate code and get a ready-to-use package built for use via the “graphql-request” client. Then will be generated the getSdk function, which return an object with functions that are wrappers over the

queries and mutations described above. Generated interfaces are strictly typed; complex data types can be imported into the project and used when describing system functionality [8].

Develop a Provider Factory for working with the user profile through the REST API. REST defines a set of constraints and principles for designing distributed systems. The main principles of the RESTful API include the following [9]:

- Resources (objects or data, for example, users, orders, books, etc).
- HTTP methods (REST uses HTTP methods to perform operations on resources).
- Presentation (resources must be available in different formats (e.g. JSON, XML) for client interaction).

Each request to the server must contain all the necessary information for processing the request. The server should not save a client state between requests. This helps ensure system scalability. REST uses a common interface that includes standardized HTTP methods and status codes, which simplifies interaction between different systems. Each resource must have a unique identifier that identifies that resource [10]. Information exchange between the client and the server proceeds through messages that contain resources and information about what to do with those resources.

Use the Swagger or OpenAPI tool (depending on the version of Swagger - version 2, OpenAPI - version 3) for describing and generating RESTful API [11].

An important advantage of using OpenAPI is the declarative approach in programming, which allows the creation of data types, and interfaces, and wraps all API access functionality in generated functions that are described in YAML format.

The process of creating an API Provider Factory for working with a RESTful API consists of the following steps:

- Creating an API description in a YAML file.
- Setting up the code generator based on the YAML file.
- Generation of output data and package build.

The Yaml file consists of the following mandatory elements: OpenAPI version, general information, a list of tags for grouping APIs, a path for defining API routes, and components (where data types such as arguments and API response are described). An example of a YAML description for OpenAPI is shown in Listing 2. The API description should define the HTTP method for the request, the name of the function to be called through the generated client, the format of the arguments, and the format of the response [12].

The data format can be described both directly in a certain route and be a reference to the scheme (in Listing 2 $ref: '#/components/schemas/ProfilePayload' is a reference to the ProfilePayload scheme) [13].

```
openapi: 3.0.0
info:
  version: '1.0.0'
  title: Profile Frontend API
tags:
  - name: Profile
paths:
  /profile/user:
    patch:
      tags:
        - Profile
      operationId: updateProfile
      requestBody:
        content:
          application/json:
            schema:
              $ref:
'#/components/schemas/ProfilePayload
components:
 schemas:
  ProfilePayload:
   type: object
    properties:
     first_name:
      type: string
     last_name:
      type: string
```

Listing 2. An example of a YAML description for OpenAPI.

The next step in creating a Provider API Factory for working with a RESTful API is to configure the code generator. OpenAPI provides a tool written in Java that generates a ready-to-use API client from a YAML file. It is necessary to ensure the processing of source data and, the creation of a build package for use in a project. Write a BASH script, the algorithm of which will be as follows:

- Generation of the source code based on the prepared instruction in YAML through OpenAPI.
- Copying the output-generated code into a package.

- Creating a package build for further import and use directly in the project [14].

In the output data, we get classes whose name matches the tags specified in the YAML file above. Each class inherits from the BaseAPI base class. The base class has the ability to set the configuration (headers, URL API, etc.) in the constructor. Classes for using the API have methods whose name matches the "operationId" parameter specified in the YAML configuration. It is possible to use several different API access points by specifying different API URLs in the configuration. A generated API via OpenAPI makes calls through the Axios library. It is also possible to create an interceptor for preprocessing the data received from the API at the OpenAPI client level, encapsulating the area of responsibility from the main development [15].

Developed API Providers Factories have a similar interface - functions-wrappers that can be called on the client of the web system. In the case of code generation for the GraphQL service, this is the "getSDK" function, which returns an object with functions; in the RESTful approach, these are classes with methods for working with the API [16].

V. API gateway development

The second aspect of this research involves developing an API Gateway through which all API calls are routed. API Provider Factories are used within the API Gateway on the Frontend Server side.

To set up an API Gateway, there is a need for a few requirements:

- Create instances of the client for using API Provider Factories on the current Frontend.
- Implement REST API.

```
export const getAuthApi = () => {
 const store = cookies()

const token = store.get('accessToken'
)?.value
  const client = new GraphQLClient(
   process.env.AUTH_API,
     setConfigApi({token}),
   )
   return authApiSDK(client)
 }
```

Listing 3. Configuring the client for working with the authorization API

To create an instance of the client for working with an authorization service, import the API Provider Factory that contains the "getSDK" function, which creates the object with functions wrappers over the described queries and mutations. An example of configuring the client to work with the authorization API is shown in Listing 3. The example shows the creation of the "getAuthApi" function, which configures the GraphQL client for working with the API. Set a URL of the API, add settings, and get a ready client for using microservice API, which contains a complete description of the API as a returned result [12].

A similar way creates a client for working with the profile service in REST format. An example of the settings is shown in the Listing 4. A "getProfileApi" function configures the client for working with the user profile microservice. Just as in the previous case, the settings are set, and the authorization token is transferred.

```
  export const getProfileApi = () =>
{
   const store = cookies()
   const          token           =
store.get('accessToken')?.value
    return                      new
ProfileApi(setConfigApi({token}))
  }
```

Listing 4. Configuring the client for working with the user profile API

The ProfileApi class creates an object with the methods described in the REST API packages above. In both cases mentioned above, we use the same working interface: a function call with the next returning an object containing methods for making requests to the microservice API. This holds true irrespective of the chosen approach for implementing the API or the format used, be it REST or GraphQL. Establishing a uniform interface for interacting with both REST and GraphQL is feasible owing to the inherent characteristics of GraphQL, which involves standard HTTP requests using the POST method and a distinctive message format conveyed as strings. These strings are then transformed into objects (graphs) on the server side and vice versa, provided there is an applicable conversion mechanism [13]. Let's proceed to the next part of this research.

Implementation of the REST API could be different depending on the technology that

is used. For example, it can be the Server side of the framework NextJS or framework Laravel, etc. In any case, we create routes that receive requests from the Frontend Client and using prepared above instances of clients for working with microservices APIs proceed these requests like an API Gateway.

VI. Client state management

In this section, we will be developing a state management system that operates on the Frontend client side and makes requests to the API Gateway on the Frontend Server side. This system will be responsible for caching requests and facilitating state sharing among components. The presence of caching provides fast access to data using a special key. In the context of this research will take the SWR library as a basis [17].

SWR is a library for data state management in React applications, focused on data caching and automatic validation. Has a special interface for working with data coming from the API, which simplifies updating data in the system interface. The basic idea behind SWR is to use cached data to display a web page immediately and update it asynchronously from the server. SWR is based on two main principles - "caching of old data" and "asynchronous update". When data is received for the first time, it is cached and displayed on the page. Then data is updated asynchronously without blocking the user interface [18]. SWR allows applications to display old data (from the cache) instantly. Automatic and asynchronous updating of data from the server after its change usually occurs after a certain time or after events that signal the need for updating. SWR has a built-in error handling mechanism that allows correct handling of errors during data updates. Also, SWR can be configured for updating data, for example, based on time or other factors.

Move ahead and define the main abstract entities to be developed for using Client State Management:

- Query.
- Lazy Query.
- Mutation.

Query will be used to create an immediate query from the component. A lazy query is a delayed query. It can be a call to a certain action under a condition (for example, a user click, callback, etc.). Mutation is similar to a lazy query entity (deferred call of a certain action). At the same time, it involves changing the data on the server. It can be the creation, update, or deletion of data) [19].

Since this research is closely related to the React library, further code examples will be provided in the typical React style: the use of hooks, and functional programming. Let's develop abstractions for useQuery queries. This hook is the basis for writing specific implementations of other hooks responsible for working with components, calls to microservices, and communication with the SWR library. An example of the "useQuery" code is shown in the Listing 5.

```
export const useQuery = ({
  key,
  request,
  payload = {},
  ...params
}: UseClientQueryPayload = {}) => {
  const { data, error, isLoading } = useSWR(
    key,
    request(payload),
    {
      revalidateIfStale: false,
      revalidateOnFocus: false,
      revalidateOnReconnect: false,
      ...params,
    },
  )

  return useMemo(
    () => ({
      data,
      isLoading,
      error,
    }),
    [data, isLoading, error],
  )
}
```

Listing 5. Implementation of the useQuery abstraction for working with data on the client side.

The request parameter contains a function through which the microservice API is called. Additionally, it is possible to set other parameters of configuration inherent in the SWR library [20].

Data contains the data that is returned in response to a request to the microservice API (or from the SWR cache). After being called and before receiving data, loading has state "true", which allows to ensure a better user experience and to display preloaders while data is being loaded. When an error is received, "error" acquires the error value, which can be processed and, if necessary, displayed in the user interface [21].

```
interface
UseQueryBaseResponse<Response = any> {
  isLoading: boolean
  error: string
  data?: Response
}
const useUser =
():UseQueryResponse<PlayerQuery> => {
  const { data, error, isLoading } =
useQuery({
    path: USER,
  })

    return useMemo(
      () => ({
        data: data?.data,
        isLoading,
        error,
      }),
      [data, isLoading, error],
    )
```

Listing 6. Implementation of the useUser hook for working with user's data

After the implementation of the basic abstract "useQuery" hook, consider the implementation of the hook for working with user data using the example of the "useUser" hook. A code example is shown in Listing 6. There is created a call of the base useQuery hook with the "path" parameter which is a key used to cache and update data using the SWR library. useQuery also provides a "UseQueryBaseResponse" interface that takes a generic type to preserve data typing in components. An important point is that the "UseQueryBaseResponse" type is passed a generated type, which was described in created packages for working with authorization and user profile microservices in GraphQL and REST approaches. If the API is updated, it is enough to modify the description in the package. After making this change, generate a ready-to-use build of the package, and all interfaces will be updated automatically. This approach is quite flexible in use and teamwork and reduces the amount of manually written code types and interfaces [22].

Move to lazy query and mutation. As in the case of a query, it is necessary to develop a basic abstract functionality through which situational variations of specific entities will be created (Listing 7).

```
const wrapRequest = func => (key:
string, { arg }) => func(arg)
const useMutation = ({
  request,
  successAction,
  errorAction,
  keysToUpdate,
  clearKeys,
  path,
}: UseClientMutationPayload = {}) => {
  const { mutate } = useSWRConfig()
  const { trigger, isMutating, error,
data } =

useSWRMutation(path,wrapRequest(reque
st),
    { onSuccess: res => {
        successAction?.(res)
        if (keysToUpdate &&
keysToUpdate.length > 0)
          keysToUpdate.forEach((key:
string) =>
            mutate(key)
          )
        if (clearKeys &&
clearKeys.length > 0)
          clearKeys.forEach((key:
string) =>
            mutate(key, null))
      },
      onError: () => errorAction?.(),
    })
  return useMemo(
    () => ({
      mutate:
mutationWrapper(trigger),
      isSubmitting: isMutating,
      error: error?.response?.data,
      data,
    }),
    [error?.response?.data,
isMutating, trigger, data]
  )}
```

Listing 7. Implementation of the base useMutation hook

Given the similarity in the nature of lazy queries and mutations, the fundamental entity used for handling mutations will also be applied for managing deferred calls. "useMutation" accepts arguments similar to those of "useQuery", as outlined in the "UseClientMutationPayload" interface described in Listing 7. Additional arguments are "successAction", "errorAction", and "keysToUpdate", "clearKeys". The "request" argument is a function for creating a request to the API service.

The "request" argument is a function for creating a request to the API service. Depending on the specific implementation, it can be a client for working with REST or a GraphQL service. The "wrapRequest" function wraps and abstracts the request function, which is set directly in the entity hook implementation. "wrapRequest" accepts a function and passes the key as the first parameter (which was specified in the hook implementation directly) and the arguments

that are set in the component when the mutation is called (a specific example will be discussed below) [23].

"successAction" is a callback that will be executed upon receiving a successful response from the API, passing the received data as arguments to this callback. An example of a callback is closing a popup of authorization after successful login, routing to another web page, analytics event, etc.

"errorAction" is similar to the previous callback, which is executed when an error is received. This callback is used to create synchronous error handling (as opposed to the error state, which is in the errors variable).

Equally important are the keysToUpdate, clearKeys parameters. Given that storage, updating, deletion, and invalidation of data in the cache occur through the use of a key, it is imperative to develop a mechanism that facilitates the updating or complete removal of data from the cache. Therefore, keysToUpdate is an array with keys by which data should be updated upon successful mutation. After receiving a response from the API, the data management system on the client side goes through each record by key, creates a request to the microservice, and updates data by keys [24].

Consider a working example:

- A test request to retrieve user data is created using the "useUser" hook.
- Response results are stored in the cache.
- Existing user data is displayed across all components of the web system.
- The user changes his name, address, etc.
- The web system does not recognize that the data has been updated and displays outdated information.
- During creating a mutation to update user data, the USER key, formed in useUser, is specified.
- Upon updating the user's data, the system refreshes the user's information and sends a request to the profile microservice through the API.
- The data is updated in the cache, and all components display the actual information since they are subscribed to these changes.

Similarly, the processing of the clearKeys parameter functions. After a successful API response, the data specified by the keys in the clearKeys array is deleted. For example, this can be useful when checking whether a user is authenticated:

- The user clicks on the "logout" link.
- A request to the authentication service API to clear the current session is sent.
- The response comes with instructions to set up cookies, where the authorization token is missing.
- When in the "clearKeys" field key "USER" is specified, user information is cleared.
- In all signed components to the "USER" key, the data state is updated.
- The client-side middleware checks for the presence of user data in the store, and in their absence, redirects to the unauthorized user page.

Deferred requests and mutations can, in most cases, include some user-specific payload data (e.g., user's login, password, language, country, etc). The response from the API depends on the given arguments. To send data to the API, it is possible to set the data in the "payload" argument. This name is often used in other store organization solutions, for example, Redux [25].

Create a mutation "useLogin" to work with authorization. The code is shown in Listing 8.

From the given code above, the mutation accepts a "successAction" argument, which is passed in the basic useMutation hook directly from the component where this mutation is used. "successAction" is a callback that will be executed after receiving a successful response from the API authorization service (for example, redirecting to another page, closing the authorization popup, updating the data validation status, etc). This callback is given to the implementation of the component in which it is used, unlike the keysToUpdate parameter. An array with the keys "USER", "BALANCE" is passed to the "useMutation" abstraction. The data with the specified keys will be updated as soon as the authorization is successful. The "useUser" hook is implemented to obtain data about the user, where the "USER" key is specified in the path parameter. The component that uses this hook to output certain information about the user will receive updated information about the user from the corresponding API immediately after successful authorization, because the "keysToUpdate" of the "useLogin" hook is set to the "USER" key to update. In

"useLogin", the "LOGIN" key is also specified, under which the received data from the response will be stored after a successful authorization (for example, a token for working with authorized requests may come with a successful authorization, which must be stored on the client side) [26].

```
interface UseResponse extends
UseMutationResponse
 {
  login: (data?: LoginVariables) =>
void
 }

const request = async (data:
LoginVariables) => {
  const authAPI = getAuthApi()
  return authAPI.login(data)
}

export const useLogin = ({
   successAction,
}: UseClientAPIConfig = {}):
UseResponse => {
  const { mutate, isSubmitting, error
} = useMutation({
    request,
    successAction,
    keysToUpdate: [USER, BALANCE],
    path: LOGIN,
  })

  return useMemo(
    () => ({
      login: mutate,
      isSubmitting,
      error,
    }),
    [mutate, isSubmitting, error],
  )}
```

Listing 8. Implementation of the useLogin hook to work with authorization

In the useLogin hook, the request function for working with the authorization service is passed to useMutation as a parameter. This function is wrapped in useMutation by an additional wrapRequest abstraction and receives the arguments necessary to work directly with the login. Types of arguments for working with the login are described in the LoginMutationVariables type, which is generated and imported from the package for the authorization service API. The main task of the wrapRequest function in this context is to pass the second parameter to the request (user data that is set directly in the component where the useLogin hook is used) [27].

The main tasks performed by the useLogin hook:

- Abstraction of client settings for working with the API of the desired service.
- Description of data types.
- Setting the keys for cache management.

Consider an example of using the "useLogin" hook (Listing 9).

```
const {login, isSubmitting, error} =
useLogin({
    successAction: redirectDashboard,
  })
const onSubmit = ({login, password}) =>
login({
  login,
  password
})
```

Listing 9. An example using the useLogin hook in a component

After the hook "useLogin" is called the "login" function for working with the service authorization API is returned. "isSubmitting" is the status of the request at the current moment, and "error" is the error that was sent by the service.

Let's analyze what happens "under the hood" during calling "useLogin" hook in the "login" component when the "login" button is pressed:

- Calling the "login" function and passing the user credentials as parameters.
- Calling "wrapRequest" function by the SWR library, to which the key "LOGIN" of the current request is passed as the first parameter, and the object of the user`s credentials is passed as the second parameter.
- The "wrapRequest" function passes the object with credentials to the request function, which is described in Listing 7.
- Sending a request to the Frontend Server API Gateway.
- The "isSubmitting" value switches to the "true" state.
- "error" is in the null state (or undefined - depending on the settings).

- API Gateway accepts a request and makes another request to the Authorization microservice API.
- API Gateway gets a response and prepares an answer to the Frontend Client.
- After receiving a successful response from the authorization service, isSubmitting switches to the “false” state.
- Aata received from the authorization service is recorded to the SWR cache using the LOGIN key.
- Query calls are generated for the “USER” and “BALANCE” keys; these calls follow a path similar to the preceding login mutation call and influence the update of states in the components where they are utilized.
- The data retrieved from the “useUser” and “useBalance” hooks updates the existing information in the SWR cache.

To summarize, the developed abstraction of entities as queries, lazy queries, and mutations can be utilized to implement specific hooks for facilitating client-state sharing between components, caching data, and making requests to the API Gateway.

Conclusions

Client State Management using the Backend for Frontend pattern architecture approach researched in this article can be used in the B2B segment that requires deploying additional instances of Frontends and their support, as well as in B2C solutions. This solution enables flexible configuration of caching, updating data, and interacting with various API microservices by combining GraphQL and REST. It provides the ability to segment the API into Public and Private levels while adding an additional layer of security.

As a result of this research, we have reduced boilerplate code, enhanced the comfort of development, and improved various aspects of WunderGraph decision. This includes splitting the API into Public and Private components, using several Frontend instances during development, and the possibility to utilize multiple Frontend instances during development. Added the ability to work with several microservices via API Gateway, regardless of whether GraphQL or REST architecture is used.

This approach contains certain disadvantages, in particular: the complexity of implementation, and the need to conduct preliminary training of developers to work with this approach. Managing the data caching policy requires setting the policy on a case-by-case basis, which raises the threshold for getting started with this data state management solution on the client.

References

1. What is WunderGraph, August 2023, [online] Available: https://github.com/wundergraph/wundergraph.
2. Vadlamani, S.L., Emdon, B., Arts, J. and Baysal, O., 2021, June. Can graphql replace rest? a study of their efficiency and viability. In 2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP) (pp. 10-17). IEEE.
3. Falkevych, V. and Lisnyak, A., 2023, September. Internal and External Threats in Cyber Security and Methods for Their Prevention. In 2023 13th International Conference on Advanced Computer Information Technologies (ACIT) (pp. 414-419). IEEE.
4. Chemerys, H., Demirbilek, M., Bryantseva, H., Sharov, S. and Podplota, S., 2022, July. Fundamentals of UX/UI design in professional preparation of the future bachelor of computer science. In AIP Conference Proceedings (Vol. 2453, No. 1). AIP Publishing.
5. N. Vohra and I. B. Kerthyayana Manuaba, "Implementation of REST API vs GraphQL in Microservice Architecture," 2022 International Conference on Information Management and Technology (ICIMTech), Semarang, Indonesia, 2022, pp. 45-50.
6. Sklyarov, D., 2020. The Web service development with React, GraphQL and Apollo.
7. Andersson, T. and Reinholdsson, H., 2021. REST API vs GraphQL: A literature and experimental study.
8. Brito, G. and Valente, M.T., 2020, March. REST vs GraphQL: A controlled experiment. In 2020 IEEE international conference on software architecture (ICSA) (pp. 81-91). IEEE.
9. Dos Santos, J.S., Azevedo, L.G., Soares, E.F., Thiago, R.M. and da Silva, V.T., 2020. Analysis of Tools for REST Contract Specification in Swagger/OpenAPI. In *ICEIS (2)* (pp. 201-208).
10. Hagelberg, T., 2023. Development of a Serverless RESTful API.
11. Lawi, A., Panggabean, B.L.E. and Yoshida, T., 2021. Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. Computers 2021, 10, 138.
12. Wittern, E., Cha, A. and Laredo, J.A., 2018, May. Generating graphql-wrappers for rest (-like) apis. In International Conference on Web Engineering (pp. 65-83). Cham: Springer International Publishing.
13. Sferruzza, D., 2018, September. Top-down model-driven engineering of web services from extended OpenAPI models. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 940-943).
14. Carvalho, T.E.A.D., 2021. Generation of Web API Definition Files, using a single platform, accordingly to the Design First Approach (Doctoral dissertation).
15. Senger, A. and Agrawal, S., API Modeling and Description Languages.
16. Preibisch, S. and Preibisch, S., 2018. API Design. API Development: A Practical Guide for Business Implementation Success, pp.41-60.

17. Oleshchenko, L. and Burchak, P., 2023, March. Web Application State Management Performance Optimization Methods. In International Conference on Computer Science, Engineering and Education Applications (pp. 59-74). Cham: Springer Nature Switzerland.
18. Le, T., 2021. Comparison of State Management Solutions between Context API and Redux Hook in ReactJS.
19. Daishi Kato, Micro State Management with React Hooks: Explore custom hooks libraries like Zustand, Jotai, and Valtio to manage global states, Packt Publishing, 2022.
20. McFarlane, T., 2019. Managing State in React Applications with Redux.
21. Y. Yao and J. Xia, "Analysis and research on the performance optimization of Web application system in high concurrency environment," 2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference, Chongqing, China, 2016, pp. 321-326.
22. Daniel Afonso; Ricardo Mestre, State Management with React Query: Improve developer and user experience by mastering server state in React, Packt Publishing, 2023.
23. Miftachudin, Muhammad Khoirul Hasin, and Afif Zuhri Arfianto. "State Management of API Web Service using Redux on React Native App." (2023).
24. Litt, G., Schiefer, N., Schickling, J. and Jackson, D., 2023, October. Riffle: Reactive Relational State for Local-First Applications. In Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (pp. 1-16).
25. Le, T., 2021. Comparison of State Management Solutions between Context API and Redux Hook in ReactJS.
26. Tran, K., 2023. State Management in React.
27. Sapountzi, I., Progressive web apps: development of cross-platform and cross-device apps using modern web architectures and technologies.

The article has been sent to the editors 18.05.24.
After processing 15.06.24.
Submitted for printing 28.06.24

Copyright under license CCBY-SA4.0.