Porting console applications to microservice architecture: XTide use-case

Domagoj Bazina
Faculty of Science, University of Split
Split, Croatia
dbazina@pmfst.hr

Toni Mastelic
Ericsson Nikola Tesla
Split, Croatia
toni.mastelic@ericsson.com

***Abstract*—Older applications are frequently neglected by users due to their seemingly complicated interfaces and inability to integrate them with other applications, while they are still in use due to their irreplaceability. Their lack of integrability and compatibility is due to a lack of an application programming interface (API). However, an API can be realized by building a wrapper around the console interface of the application making it integrable into microservice architecture. In this paper, a procedure for building wrapper applications around console interfaces is defined considering five main features, namely functions, output data format, errors, logs and configuration Furthermore, a console application XTide for tide predictions is integrated and deployed as a microservice with an REST API using Python and Docker container, thus demonstrating the applicability of the defined procedure.**

***Index Terms*—XTide, console interface, wrapper, microservice architecture, REST interface, Docker container**

I. Introduction

Today, there are numerous applications and software solutions that are made using older technologies, and they are still utilized regardless of the fact that they are harder to use and integrate into other systems [2]. These applications have been in use for several decades, and the outcome of that utilization is the increase of application effectiveness and error resistance. Mentioned improvements along with the lack of quality successor are major reasons why are these applications still valuable and usable [4]. Mentioned applications usually include several interfaces, such as graphical, console and occasionally web interface. The principal flaw of these interfaces is that they are not compatible with other applications, as opposed to an API that enables communication between applications and therefore their integration into other systems, e.g., solutions based on microservice architecture. Focus of this paper are console applications as they are most similar to API.

There are different approaches to resolve mentioned drawbacks, such as possibility to refactor and rebuild the entire application. However, that could be time-consuming, and it may result in a new set of errors. Furthermore, there is an option to import a source code of an application as a library into other programming languages. Nonetheless, this option is limited by interoperability between used technologies. Finally, there is an alternative to build a new application that will behave as a wrapper around the console application. Wrapper makes simple "calls" to the console interface, captures the output and converts it to the acceptable format.

Wrapper solution is the most suitable of all mentioned options as it can be easily transformed into API that enables integration of an application into microservice architecture, one of the most rapidly expanding architectural paradigms in commercial computing today [1]. Use case selected to achieve this approach is XTide, software that provides tide predictions in a wide variety of formats [3].

Rest of the paper is structured as follows. Section II defines five main features of console applications and uses them to define a procedure for building a wrapper application. The procedure as applied on XTide application as a use case in Section III. Finally, Section IV gives a conclusion.

II. Five steps for integrating console application into microservice architecture

To enable integration of a console application into microservice architecture, two conditions have to be fulfilled. The first condition is to create an environment that is used as a mediator between the console interface and requests made to it. The second one is merging the wrapper with the API, as well as deployment of the application as a microservice into the containerized environment.

A. Mapping console application features into wrapper features

The key part of the mediator is the wrapper which is based on mapping (Figure II-A) five main features of the console application:

1) **Functions** - console interfaces commonly provide several functionalities to perform certain tasks. Any function that is used should be matched with function in the wrapper. Later, those functions may be aggregated into a single function.
2) **Output data format** - console interfaces may provide different output formats, most of which are not designed for data interchange. Therefore, they should be converted into more suitable format.
3) **Errors** - from time to time every application will run into errors. To prevent applications from collapsing it is necessary to develop an environment that is able to handle potential errors. Although these applications are

not prone to errors, the wrapper should be prepared to "catch" potential errors, resolve them and inform the user about the cause of the error.

4) **Logs** - logs are small textual records describing events that happen inside of the application. These records are extremely useful when it comes to debugging or learning about sequence of events in the background of the application. Along with logs of the wrapper application, it is important to map logs of the console application, otherwise it would be extremely hard to debug the system.
5) **Configuration** - configuration defines behaviour and technical parameters of the application. These parameters may include network parameters, database parameters, information about users, etc. By mapping the configuration into the wrapper, startup of the application is significantly simplified as all those parameters may be sent when the wrapper application is executed, therefore the whole process may be automated.

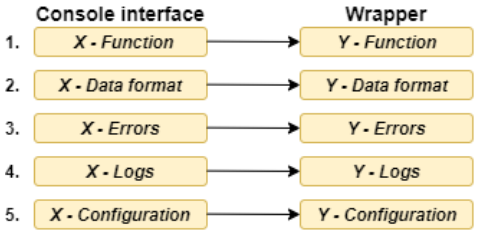

Fig. 1. Mapping console application features with wrapper features

B. Merging wrapper features with API and deployment into microservice architecture

Microservice architecture (Figure II-B) is the architectural pattern for developing applications based on microservices, i.e., small and autonomous components that mutually interact. Each component has its own specific task, and composing those tasks results in a purposeful application. A significant aspect of microservice architecture is the communication between microservices, and it can be achieved through an API that precisely defines the procedure of the communication and format of data interchange between microservices. Users or applications may access API features through an URL that is paired with the function defined in the wrapper. Whenever request is made to the specific URL, wrapper functions is executed and its output is returned as a response to the requester.

Additionally, each microservice should be deployed in an environment that packages all mandatories required for running the service, for instance Docker container. Mandatories may include libraries, dependencies, binaries and configuration files. Deployment of a microservice into a container additionally isolates and encapsulates the microservice, which leads to increase of independence and autonomy.

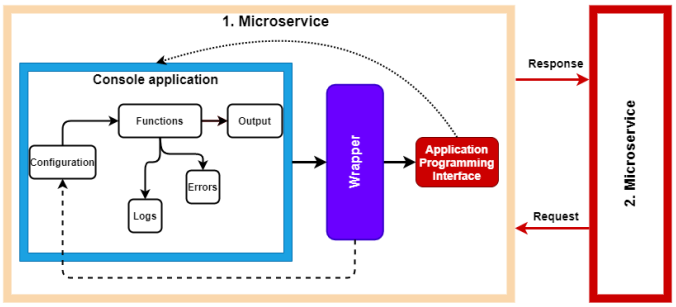

Fig. 2. Structure of microservice built around console application

III. Integration of XTide application into microservice architecture

XTide is a software that provides tide predictions based on the algorithm that is defined by the National Ocean Service [5] in the United States. The application consists of an interactive graphical (xtide), non-interactive console (tide) and a web (xttpd) interface. These interfaces are considered as user interfaces, not APIs. XTide consists of 3 main features that are mapped to a wrapper, and they include XTide functions, output data format and errors. Mapping XTide logs and configuration is skipped as the application does not implement those features.

A. Mapping XTide features into wrapper

XTide wrapper is built using Python and libraries such as Pandas and Regular Expression (Regex). Python is capable of executing Linux commands, and mentioned libraries are suitable for conversion of data formats.

1) **XTide functions**
There are several functions in XTide used for retrieving data, where two of them are used and matched with Python functions defined in the wrapper. First function retrieves tide data based on query parameters, which may include location and time period. There are 4 combination of query parameters that define what data is retrieved:
   - Location parameter sent; by default XTide generates data for time period of 120 hours.
   - Location parameter sent along with start date parameter; by default XTide generates data for a time period of approximately 96 hours, using query date as an initial date.
   - Location parameter sent along with end date parameter; XTide generates data within a time period, where query date will be used as a start date and end date will be the date when the query request was made.
   - Location parameter sent along with start and end date parameters; XTide generates data within the given time period.

   The second function retrieves detailed information about the requested location, also known as metadata. Meta-

data includes parameters like coordinates, time zone, country, state, native unites and type of tide station.

2) **XTide output data format**
One of the features of XTide is that it can output data in multiple textual formats, and the most useful of them is CSV as it is widely used and easily convertible to other formats, such as JSON. The goal of this step is to convert the format from XTide output to JSON, a textual format used for data interchange. JSON is structured in key/value pairs, and it is the preferred format in terms of communication with APIs.
Comma-separated values or CSV is type of textual format that uses any character as delimiter and structures data in form of rows and columns, i.e. it is used in a function for retrieving tide data that contains 5 different values: location, date, time, height of tide and type of tide. Pandas is used to convert CSV into JSON. Second function retrieves data in a textual format using key/value pairs, delimited by two spaces. Regex is used to convert this format into JSON.

3) **XTide errors**
There are several types of errors that can occur while fetching data from XTide and most common errors are caused by invalid parameters sent by a user. These errors include non-existent location, invalid time format and small time span. Mapping of errors in XTide is achieved by using a generic class for error handling, which attributes include error message and status code used for the API. For each occurrence of an error, a new instance of the error class is created with an error message captured from XTide and the appropriate status code.

B. Merging XTide wrapper into REST API and deployment into Docker platform

Matching XTide wrapper features with REST API features results in autonomous web service for tide predictions. REST API is an interface that fulfills Representational State Transfer (REST) constraints that simplify communication between systems, and it is implemented using Flask, a Python micro web framework. Matching wrapper functions and outputs with the REST API is achieved by assigning wrapper functions with URL paths. Inside URL, users define query parameters used for retrieving and filtering data. There are 2 different URLs that are matched with wrapper functions.

- */data/<location>/tidedata*
- */data/<location>/metadata*

First URL is matched with wrapper function for retrieving tide data. There is also an option to send date parameters inside the URL path. Format of date parameters is yyyy-MM-dd HH:mm. Other URL is matched with a wrapper function for retrieving metadata. Response from the REST API is consisted of a wrapper output data in JSON format and a response status code.

Finally, XTide microservice is deployed using Docker, an open source containerization platform that enables developers

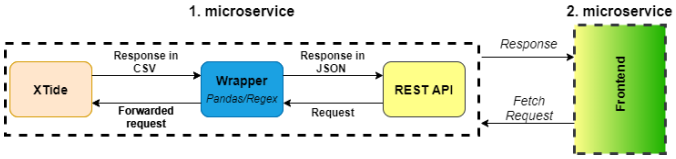

Fig. 3. Structure of microservice built around XTide console interface

to package applications as container—standardized executable components. Along with dependencies, i.e., libraries defined in a Docker image, there is a feature to define a network interface and port on which a microservice will be running. This is achieved using environment variables. Such deployment significantly automates the process of installation and usage of microservices, as well as it increases isolation, independence and portability of mentioned service.

IV. Conclusion

Outdated applications are still in use due to shortage of other solutions that could replace them. The goal of this paper was to demonstrate that upgrading these applications could overcome drawbacks such as lack of interactivity or compatibility. Upgrading applications with console interfaces may be done by building the wrapper around it, that is later merged into an API. Integration of XTide application into microservice architecture demonstrated that these outdated applications shouldn't be discarded, rather the effort should be directed into building extensions over its core functionalities.

References

[1] Kyle Brown and Bobby Woolf. Implementation patterns for microservices architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*, pages 1–35, 2016.

[2] Liliana Favre. Modernizing software amp; system engineering processes. In *2008 19th International Conference on Systems Engineering*, pages 442–447, 2008.

[3] David Flatter. Xtide manual: Harmonic tide clock and tide predictor. https://flaterco.com/xtide/disclaimer.html, 1992. Accessed: 2021-07-15.

[4] Ruben Lif and Wilhelm Handberg. Converting outdated software, 2021.

[5] National Ocean Service. https://tidesandcurrents.noaa.gov/PageHelp.html. Accessed: 2021-07-15.