COMARCH

Improving User Experience with Technical Documentation for Microservices: How They Even Relate?

WHITEPAPER

COMARCH

Table of contents

Introduction

User Experience is a term that in the last few years became hugely important and popular with sales strategy and development managers in Finance. Remote and digital client services are now the first, basic, sometimes even the only choice. Thus it's vital that the way the service goes is kept simple, fun, professional, but most importantly safe.

Solutions have to be comprehensive, fulfill basic tasks, i.e. provide financial services, while at the same time being intuitive, easy to use, and – most importantly – arouse interest and sympathy among users, making them loyal consumers. Currently, the trend is to build wide range solutions, the so-called Digital Banking Engagement Platforms. According to the Forrester report (Now Tech: Digital Banking Engagement Platforms, Q1 2021), comprehensive solutions should at the very minimum:

- **Provide basic banking products in an omnichannel way (web, mobile, h2h);**
- **Include analytical tools that allow to continuously measure business efficiency and analyze the behavior of platform users (clients, but also customer service employees on the bank side);**

→

- **Provide the possibility of free contact adjusted to preferences (e-mail, chat, video chat);**
- **Support active sales by handling various campaigns targeted at individual users.**

Financial institutions taking the next step in digital transformation and wanting to build ecosystems for serving customers are facing the choice of the technology that will allow them to meet current requirements and ensure continued future growth. The requirement of complexity and variety of solutions must assume the possibility of cooperation of many entities providing individual tools and their seamless integration. It is often the integration of technologically different solutions. An increasingly clearer trend is the desire to independently develop applications by using own IT resources, thanks to the buy&build model. This additionally requires the applications to be open.

An increasingly clearer trend is the desire to independently develop applications by using own IT resources

The answer is choosing a platform with a microservice based architecture, that assumes the system distribution, built of many small, independent applications communicating through API, and responsible for providing a narrow range of mutually used services. Each microservice is a separate entity that performs its tasks without exerting direct influence on other microservices.

Microservice based architecture allows for integrating many solutions from various providers within one efficient system. Moreso, each microservice can be developed independently by a separate team or vendor. The pace of its development, or a possible break in operation, resulting e.g. from a technological upgrade or failure, will not affect the operation of the entire system. This is especially important for banking transaction systems.

The multitude of microservices and teams working in multiple parallel streams as well as the openness to various models of cooperation require appropriate and systematic documentation.

Antora — a tool for microservice documentation

Documentation? What for?

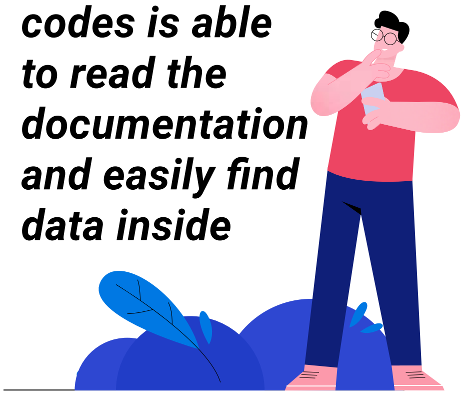
One of the main advantages of microservice based architecture is great autonomy of teams, which is achieved through an independent service life cycle. One of the steps in a way to achieve such autonomy is often breaking the system down into many separate projects, sometimes even hundreds of them. This kind of approach can affect the documentation development aspect in a rather unexpected way. The websites and libraries we produce make up a system in which it is common to build new functionalities on the basis of pre-existing code. But how do we know how the code we want to use works? How a programmer that is only getting into the project is supposed to get all the project information? A self-documenting code is a kind of a mythical creature — a unicorn, virtually unattainable. Moreover, there is also some information that does not match Javadoc, but is useful nonetheless, and thus needs to be salvaged. Each project should contain at least a basic documentation, so that there's no need for reanalyzing it each time. There's an approach to use here, in which every microservice or library contains documentation that is stored together with a source code. The developer does not need to leave IDE to update it. This approach promotes creating and updating the documentation.

An alternative is to keep the documentation on a dedicated wiki page or in a tool like Confluence. Here, however, there's a much greater risk of developers not being so eager to update the documentation, as well the potential risk of inheriting the problems of such external tool, e.g. slow search engine. We know from experience that plain text files that are being kept together with code are read, written and updated more often, and — after a little bit of syntax learning — cause less editing problems. Still, the questions remain, such as:

- **What if we want to provide documentation of the microservices and libraries we create to other teams or clients?;**
- **What if we want to be able to easily find information in the entire documentation, no matter where it's located?;**
- **What if the documentation is to be used by testers or analysts? Do they need to have access to the source code?**
- **The above problems can be solved using a tool called *Antora*. We will now present it based on a way we used it in our current project.**

The multitude of microservices and teams working in multiple parallel streams as well as the openness to various models of cooperation require appropriate and systematic documentation.

Even someone who doesn't have access to codes is able to read the documentation and easily find data inside



The basics, or how to start

The most common way to documenting projects is to put the README.md file in the root directory of the project. It's a simple solution, but it doesn't necessarily work well in a system consisting of a large number of projects. What if we don't know exactly which project contains the mechanism we're looking for?

This is where *Antora* comes in handy, allowing to aggregate documentation from multiple git projects. After collecting text files from git projects, the resulting html is created, which can be now put on a specific server. Even someone who doesn't have access to codes is able to read the documentation and easily find data inside. To start working with *Antora*, you need to install Node.js and create a package.json file in accordance with the official documentation. The next step would be to create an antora-playbook.yaml file, in which you can start configuring the tool. Here is an example of such file, which is a bit more elaborate than in the official documentation:

```yaml
sources: # 1
  - url: ${GITLAB_URL}/some-repo.git
    start_path: docs
  ...
antora:
  extensions: # 2
    - './../antora/extensions/maven-version-parser.js'
output: # 3
  dir: ./aggregated
  destinations:
    - provider: fs
clean: false
runtime:
  cache_dir: /tmp/antora-cache/ # 4
site:
  title: Your application name
  start_page: some-project::index.adoc
```

- The sources section contains a list of git repositories that can be authenticated to in **various ways**. Each of these projects must have an *antora.yaml file* with the configuration of a given project in the directory indicated by the **start_path** field.
- *Antora* allows to create extensions in javascript, which can influence the documentation generating process. In our project, there was a need to correlate the version of the published documentation with the version of the artifact from the pom.xml file. Such a plugin worked perfectly in this case.
- In this section we can define the save location of the resulting documentation html file. *Antora* allows for using a regular directory, ZIP archive file, as well as attaching your own code publishing mechanism, implemented in javascript.
- If you implement your own plugin that changes the documentation generating method, it may be useful to indicate the directory in which *Antora* should download the projects indicated in the **sources** section.
- Default page configuration, which can come from any project indicated in the **sources** section.

It's important to pinpoint here that the files in which the information is stored are not markdown, but asciidoc. Both solutions are quite similar, but **Asciidoc** offers much more than what can be done in the markdown file. This is a slight drawback when using md files before moving to *Antora*. It requires rewriting the existing documentation to a different syntax, but also taking your time to learn it, above all else.

Appearance customization

If you need to adapt the appearance of your documentation to company standards or customer requirements, you can do it by downloading the templates, from which *Antora* generates the resulting html and changing selected fragments. After unpacking the files, indicate their location:

```
ui:
 bundle:
  url: ./../antora/ui # 1
  supplemental_files: ./../antora/supplemental-ui # 2
antora:
  extensions:
   - '@antora/lunr-extension' # 3
asciidoc:
 attributes:
  source-highlighter: highlight.js #4
```

- Directory of the downloaded templates. Here you can also store them as a package on a remote server.
- Additional static files used in the generated html, e.g. **lunr search library**.
- Registering extensions installed with npm install xxx. In this case it's a documentation search plugin.
- Enabling syntax highlighting (more on that in a moment).

Having downloaded the templates, you can freely modify them. It's a good idea to configure the code snippet syntax highlighting here, using the **highlightjs** library. This can be done by adding a template fragment, from which the script import will appear in the resulting html, and calling the function that initiates the highlighting mechanism, e.g. *the footer-scripts.hbs* file:

```
<script src="{{{uiRootPath}}}/js/site.js"></script>
<script src="{{{uiRootPath}}}/js/vendor/highlight.js"></script>
{{#if env.SITE_SEARCH_PROVIDER}}
{{> search-scripts}}
{{/if}}

<script>hljs.highlightAll();</script>
```

Diagrams as code

It is good practice to create and store diagrams and technical drawings as a code. It makes editing them easier — everyone has a text editor after all, but a graphic file editor — not necessarily. One of the tools that allows to create such diagrams using your own language is **PlantUML**. Unfortunately, *Antora* will not be able to render such diagrams by default. For this the tool **Kroki** can be used, which generates diagrams, supporting various source file formats. This tool is available online, but for security reasons, you should not use this option. In this case, the easiest way is to run your own server locally and connect to it. The fastest option is to use a **docker image** or the **helm chart** to run the Kroki server inside your infrastructure. After starting the server in the *antora-playbook.yaml* file, you should indicate your server address:

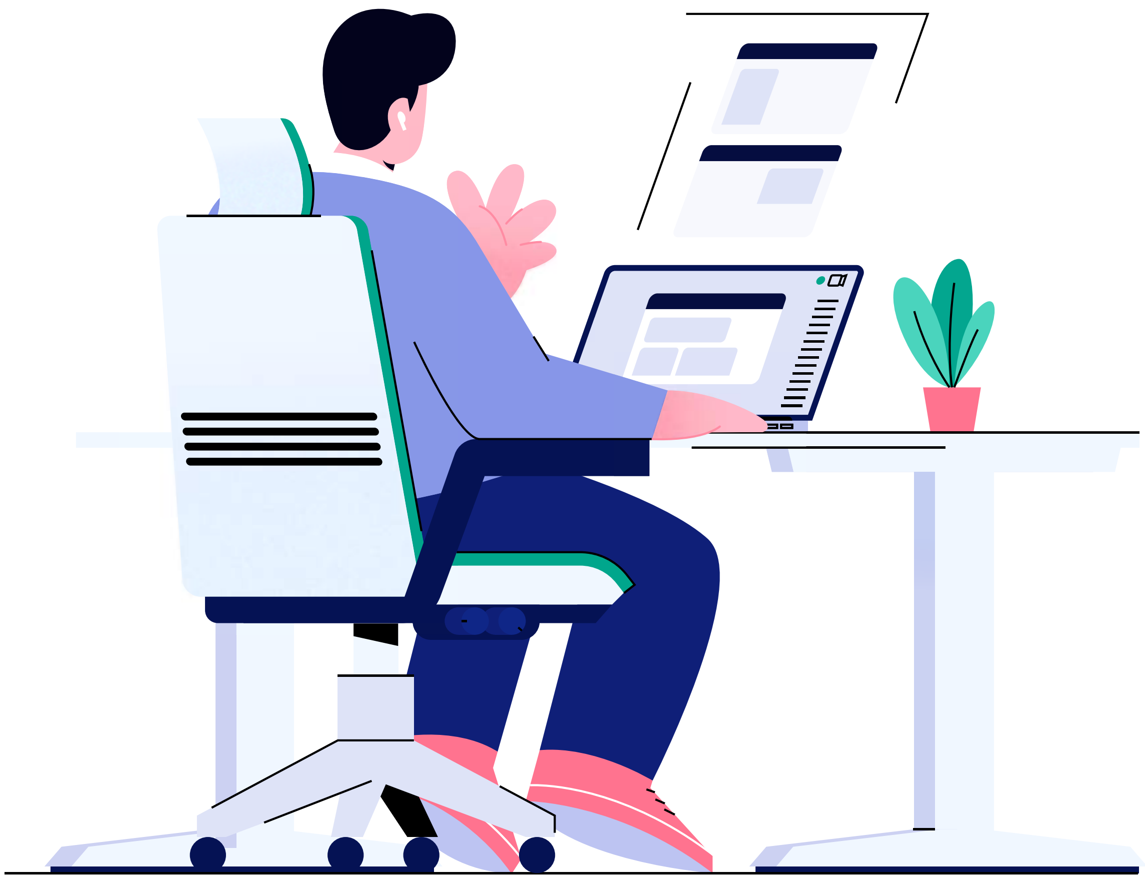
```
asciidoc:
  extensions:
    - asciidoctor-kroki # 1
  attributes:
    kroki-fetch-diagram: true
    kroki-server-url: <URL> #2
    kroki-http-method: adaptive
    kroki-plantuml-include: https://<URL>/-/raw/master/puml-theme.puml #3
    source-language: asciidoc@
    table-caption: false
```

- Enabling the extension that supports generating diagrams with the use of the Kroki server. **This plugin** must be installed beforehand.
- Address location at which the server will be available.
- If you want to have globally styled diagrams, in accordance with the appearance of your documentation or an internal corporate standard, here you can indicate the address of the **style configuration for PlantUML**.

With *Antora* configured, you can create diagrams in the code. The recommended approach is that the files with the diagram code are kept in a dedicated directory (*Antora* enforces its name: partials) and the diagram is attached to the content of the adoc file as follows:

```
[plantuml]
----
include::partial$some-diagram.puml[]
----
```

Thanks to this, you will maintain greater file readability and it will be easier to edit them.



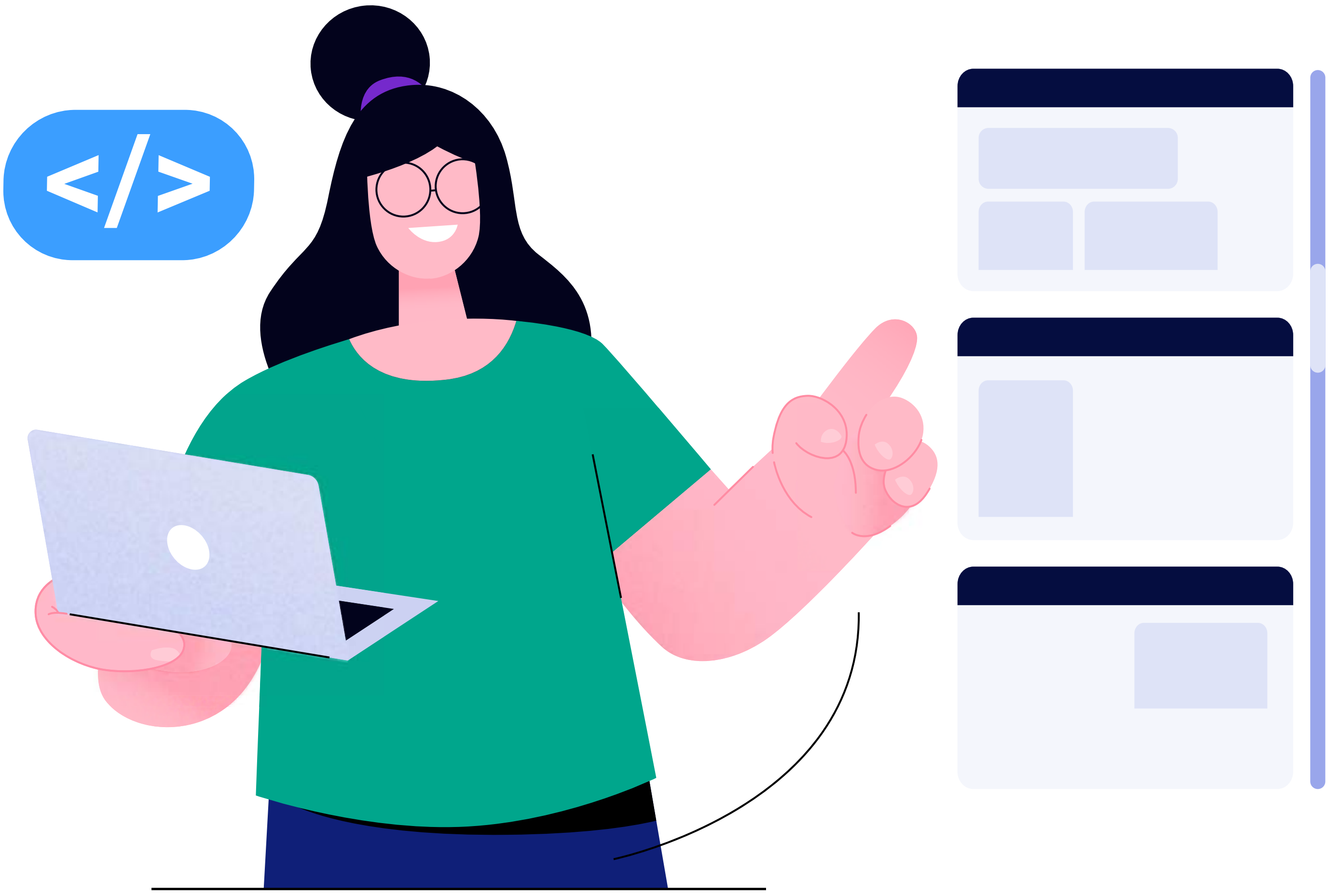
Building documentation

With *Antora* configured, the last step is to automate the building and implementation of documentation in the environment. In this case, we used the approach in which our pipeline runs a simple script in the docker image, which in turn starts building *Antora*:

```
const fs = require('fs');
const fsExtra = require('fs-extra');
const spawn = require('child_process').spawn;
const os = require('os');
const tempDir = os.tmpdir();
const yaml_config = require('node-yaml-config');
const zipper = require('zip-local');
const replace = require('replace');

if (fs.existsSync('target')) {
  fsExtra.removeSync('target')
}

fs.mkdirSync('target');
fsExtra.copySync('./antora/playbook', 'target')

const antoraBuildEnv = process.env;
antoraBuildEnv.DOCSEARCH_ENABLED = 'true';
antoraBuildEnv.DOCSEARCH_ENGINE = 'lunr';
antoraBuildEnv.DOCSEARCH_INDEX_VERSION = 'latest';
antoraBuildEnv.FORCE_SHOW_EDIT_PAGE_LINK = 'true';
antoraBuildEnv.CACHE_DIR = tempDir + '/antora-cache/';

# start building documentation
const npxProcess = spawn('npx',
        ['antora', '--fetch', '--stacktrace', 'target/antora-playbook.yml'],
        {env: antoraBuildEnv}
);
```

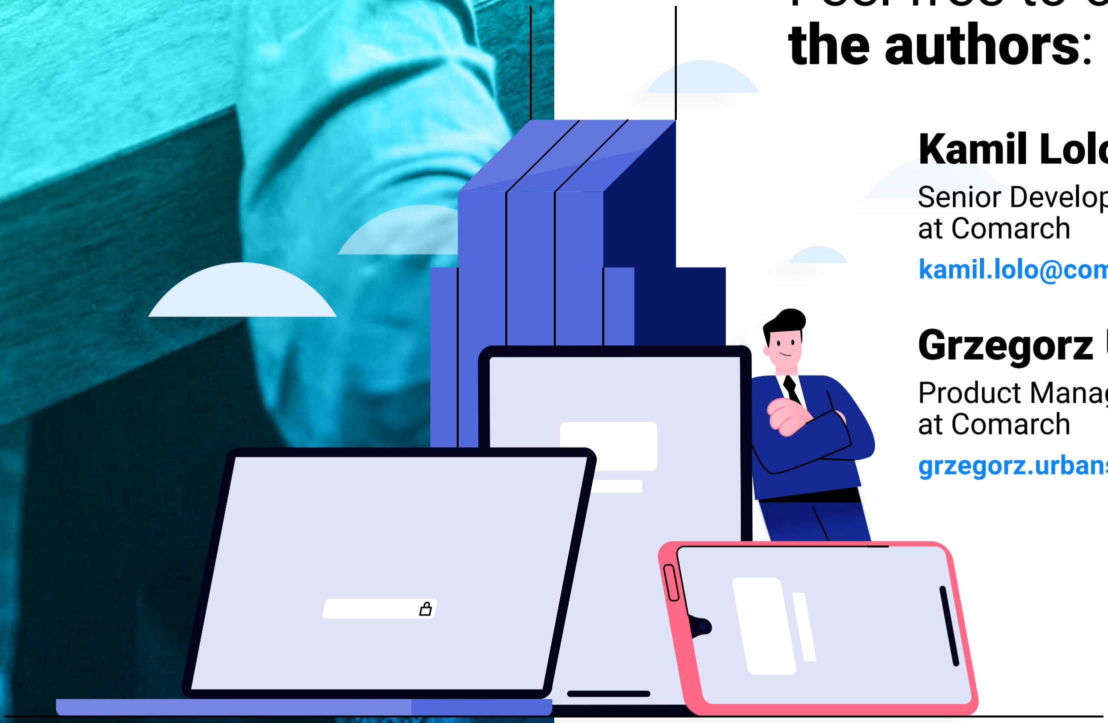
```

npxProcess.stdout.on('data', data => console.log(data.toString()));
npxProcess.stderr.on('data', data => console.log(data.toString()));
npxProcess.on('error', err => console.log(err.toString()));
```

Here of course the building process can be solved in many different ways. This particular one is optimal for us and allows to add additional steps necessary in our CI process from the script level.



Summary

As presented, *Antora* is a proper solution for distributed documentation projects that need to provide one central point of access to the documentation. It allows to easily aggregate documents and modify the appearance of the target document page. As if that wasn't enough, *Antora* allows to use extensions that increase its capabilities. There are many available community-created plugins, and if these turn out insufficient, you can easily add your own in javascript. All this makes *Antora* and indispensable tool in the process of creating technical documentation for all types of projects. Fine documentation means smoother development of your platform using microservice based architecture, resulting in better user experience and customer service.



Feel free to contact **the authors**:

Kamil Lolo
Senior Developer
at Comarch
kamil.lolo@comarch.com

Grzegorz Urbański
Product Manager
at Comarch
grzegorz.urbanski@comarch.com

COMARCH