Bank Client Management Documentation

Tartalom

[How to Build and Deploy the App 3](#_Toc164133437)

[Requirements 3](#_Toc164133438)

[Required runtime environments and builds tools: 3](#_Toc164133439)

[Build and Deploy (Manual) 4](#_Toc164133440)

[Launch a client-db container 4](#_Toc164133441)

[Build the client-backend microservice 4](#_Toc164133442)

[Stop the client-db container 4](#_Toc164133443)

[Use docker compose to deploy the client-backend microservice 4](#_Toc164133444)

[Successful deployment 4](#_Toc164133445)

[Build and Deploy (Automated) 5](#_Toc164133446)

[start-clean-pull-build-deploy-start-project.bat 5](#_Toc164133447)

[stop-clean-docker-build.bat 5](#_Toc164133448)

[end-clean-docker-project.bat 5](#_Toc164133449)

[Planned glitch 6](#_Toc164133450)

[How to test the App 7](#_Toc164133451)

[Switching Microservice Profiles 7](#_Toc164133452)

[Postman 7](#_Toc164133453)

[Swagger and OpenApi 8](#_Toc164133454)

[Authentication modes 9](#_Toc164133455)

[No authentication 9](#_Toc164133456)

[Basic authentication 9](#_Toc164133457)

[API-Key authentication 9](#_Toc164133458)

[TLS and mTLS 9](#_Toc164133459)

[Suggestions for further development 10](#_Toc164133460)

[Authentication (OAuth2 & SAML) 10](#_Toc164133461)

[Roles 10](#_Toc164133462)

[Secrets and Parameters 10](#_Toc164133463)

[Database 11](#_Toc164133464)

[Tools and Runtime Environments Used During Development 11](#_Toc164133465)

# How to Build and Deploy the App

## Requirements

There are 2 options for building and deploying the app: **manual** and **automated**. Both options have the same requirements.

### Required runtime environments and builds tools:

* **Java 17 (developed using OpenJDK 17.0.2)**
  + Must be included in the path environment variable. For example:
    - %JAVA\_HOME%\bin
* **Maven 3.8.x+ (developed using Apache Maven 3.9.6)**
  + Must be included in the path environment variable. For example:
    - %MAVEN\_HOME%\bin
  + On the first build, maven will download all dependencies of the project from the online maven repository. Please make sure that the public maven repository is available to the maven build tool on your computer.
  + Common maven repository problems during build:
    - An overly strict corporate VPN that blocks the public maven repository
    - Custom maven settings (.m2) that define only private upstream repositories that do not have access to the public maven repository.
* **Docker 25.0.x+ (developed using Docker version 25.0.3, build 4debf41)**
  + Must be included in the path environment variable. For example:
    - C:\Program Files\Docker\Docker\resources\bin

## Build and Deploy (Manual)

### Launch a client-db container

The microservice (client-backend) needs a database for testing during the build. Run the **start-test-client-db.bat** file (folder: **source-root/client-db**) to launch a containerized test database for building the microservice.

### Build the client-backend microservice

Go to the **source-root/client-backend** folder and execute the command **mvn clean package**.

### Stop the client-db container

Go back to the **source-root/client-db** folder and run the **stop-clean-test-client-db.bat** file in order to stop and reset the test-client-db after a (successfull) build.

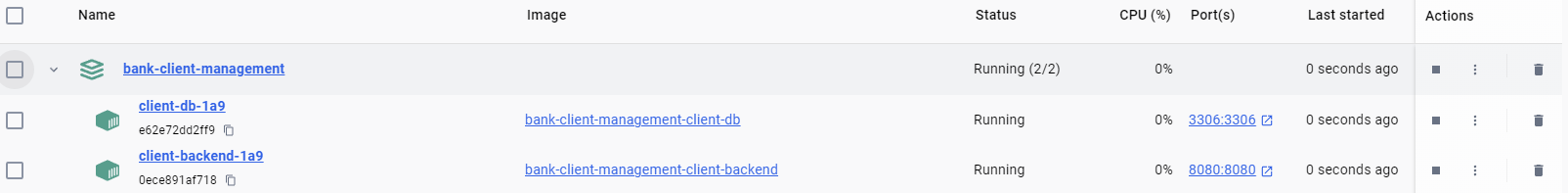
### Use docker compose to deploy the client-backend microservice

Go to the **source root** folder and execute **docker-compose up -d**. Using this command, Docker will take care of deploying the architecture contained in docker-compose.yml, including new microservice image builds and container deployments, and also the build and deployment of the database required for the microservice to run.

### Successful deployment

If all goes well, you should see something similar to the following in Docker and container logs.

**Docker Dashboard:**



**client-backend-1a9 logs ending with:**



**client-db-1a9 logs ending with:**

A képen szöveg, Betűtípus, képernyőkép látható

Automatikusan generált leírás

## Build and Deploy (Automated)

During development, I created 3 .bat scripts to automate the development lifecycle. These scripts are independent of each other. They can be executed in any order, but I prefixed them in the recommended order (**start 🡪 stop 🡪 end**) so they are easy to use.

### start-clean-pull-build-deploy-start-project.bat

Location: "source-root/devops/ start-clean-pull-build-deploy-start-project.bat"

This script first deletes all docker components associated with the actual project build, which means stopping the containers, deleting them and their images. But it leaves the base images (openjdk:17, mariadb) untouched, the images that serve as the base of the project images. This makes it easy to rebuild new custom images and containers during the next build.

Use this script to get a running build from the source code with just executing one command.

Important:

* You will find the exact instructions for using the script at the beginning of the file.
* Using the **script has some prerequisites** (Java 17, Maven 3.8+, Docker 25+) without which it will not work or will work incorrectly. You will also find a detailed description of the prerequisites in comments at the beginning of the .bat file.
* The very first build can take quite a long time (~1 minute) due to the many maven dependencies and docker base images to download. In this case, the build time is highly dependent on the network speed.Several times the .bat script may seem to freeze. Please be patient.

If the script was executed successfully you should see the following:

* Also the successful paragraph of the manual deployment section applies. Because the end result is the same.

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás

Note: Also the successful paragraph of the manual deployment section applies

### stop-clean-docker-build.bat

Location: "source-root/devops/stop-clean-docker-build.bat"

Use this script to stop and clean up the currently running build. This is sort of a sub-script (clean part) of the start-clean-pull-build-deploy-start-project.bat script.

### end-clean-docker-project.bat

Location: „source-root/devops/ end-clean-docker-project.bat”

Use this script to stop and clean up not only the currently running build (containers), but all of the Docker images (even base images) connected to this project. If you executed this script accidentally, no problem, you can use the start-clean-pull-build-deploy-start-project.bat script to recreate a running build. It will just take more time to build, because Docker will have to pull the base images (OpenJDK:17, MariaDB).

### Planned glitch

When running the provided .bat files, sometimes docker „not found” errors are seen with the containers. This is not an actual error, but is expected functionality. During development, I experienced that containers sometimes gets "stuck" in Docker between build processes. So I included more than the required amount of "clean up" commands in my script. So these commands are simply issued sometimes in a phase when the optimal execution process would be that the container should not exist. This causes the „not found” errors, but this has no effect on the normal flow. These commands just ensure that all containers and images belonging to the project are properly cleaned up before the next build.

The mentioned minor glitch from CMD:



# How to test the App

## Switching Microservice Profiles

The **application.yml** in the resource folder (**source-root/client-backend/src/main/resources**) of the client-backend microservice contains several spring (authentication) profiles. By turning these **profiles** on and off, the microservice can operate in **different authentication modes**.

Important notes about spring profiles:

* To enable the mTLS profile, you also need to enable the TLS profile. Otherwise it will not work. This is because mTLS is an "extension" on top of TLS.
* Please **always select exactly 1** from the [**auth-disabled**, **basic-auth** and **api-key**] profiles to avoid potential conflicts caused by having multiple security configs.
* If you **do not want any authentication** then the **auth-disabled profile should be active** and tls, mtls, basic-auth and api-key profiles should be disabled.
* You can **combine (m)TLS** with **basic-auth and api-key authentication modes**, because they operate in different layers of the network.
  + Although it doesn't make much sense from an authentication point of view, because mTLS is quite enough on its own . However, it is definitely useful from an encryption point of view, because it provides a secure connection. Although it may be enough to just use tls for secure connection if another authentication method is active.

**Tip:** How to test different spring profiles quickly, easily and with your own builds?

* Enable the desired spring profile(s) in the **application.yml file**
* Execute the **start-clean-pull-build-deploy-start-project.bat** (as described above)
* Wait 20 seconds for the project to build.
* Use calls from the **Postman collection** prepared for the newly actived authentication mode.

## Postman

You can use Postman to test the microservice’s API and its different authentication methods.

I have created a postman collection, with the requests sorted into folders by authentication method. Use these precreated calls to thouroughly test the API.

Before using the calls from the collection, there are two necessary configuration steps:

* Import the collection:
  + Location: **source-root/ test-the-app-in-action-with-an-http-client/ bank\_client\_management.postman\_collection.json**
  + See details: [https://learning.postman.com/docs/getting-started/importing-and-exporting/importing-data/#import-postman-data](https://learning.postman.com/docs/getting-started/importing-and-exporting/importing-data/%23import-postman-data)
* Configure keystore used for calling the microservice:
  + Location of the preconfigured keystore: **source-root/ test-the-app-in-action-with-an-http-client/** **test.http.client.p12**
  + The keystore was generated by [Keystore Explorer](https://keystore-explorer.org/) and contains the client key, client certificate and the (trusted) certificate of the client-backend microservice.
  + During the certificate configuration, fill only the localhost, PFX and passphrase fields (as you can see below on the picture), leave the rest (CRT file, KEY file) empty.
  + More information: <https://learning.postman.com/docs/sending-requests/authorization/certificates/>

A képen szöveg, képernyőkép, Betűtípus látható

Automatikusan generált leírás A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

## Swagger and OpenApi

I integrated OpenAPI and Swagger using the „**springdoc-openapi-starter-webmvc-ui**” artifact from maven.

If you disable (watch out: negation!), the **- openapi-and-swagger-disabled** profile (source-root/client-backend/src/main/resources/application.yml) the standard Swagger page is automatically available after deployment, which provides an interactive interface for testing API endpoints using the OpenAPI standard.

I **recommend** using it when **m(tls)** and **any authentication mode is not enabled**, because its authentication capabilities are limited.

Swagger is available at the following url after a successful deployment: <http://localhost:8080/swagger-ui/index.html>

# Authentication modes

When using authentication modes, it is important that exactly 1 of the auth-disabled, basic-auth and api-key spring profiles is always active. This makes sure that there is no conflict from having multiple active security profiles at the same time.

* This does not apply to TLS and mTLS, because those protocols operate at a lower level. This means that it is possible to use (m)TLS and the authentication profiles together without any problems.
* In theory, it should be possible to combine basic authentication and api-key (second factor) authentication methods, but this is not recommended due to the current implementation.

## No authentication

To use it, the **auth-disabled** profile must be active. In this mode, all requests are allowed and no authentication is performed. This mode is mainly for testing purposes.

## Basic authentication

To utilize it, the **basic-auth** profile most be enabled. In this mode, the API requires a username and a password for authentication purposes provided in the Authorization header of each request.

The Authorization request header contains the Base64-encoded username and password, seprated by a colon. When handling the request, the server decodes the login details and checks if the user can access the requested content.

In the current implementation, the users are not stored in a database, but in an InMemoryUserDetailsManager() instantiated with two hard-coded users in SecurityConfig.

## API-Key authentication

To use api-key authentication, the **api-key** profile must be active.

API-key authentication involves clients providing a unique API key as part of their request to access protected API endpoints. Upon receiving the request, the API server validates the API key against its authorized keys list to determine if the client has permission to access the requested resource.

API-key authentication can act as a second factor when used together with other authentication methods such as basic authentication.

The name of the api-key header and the value of the api-key secret can be configured in the application.yml file of the microservice.

## TLS and mTLS

To use TLS, the **tls profile** must be active. To use mTLS, the **mtls profile** must be active in addition to **tls profile**. That’s because mTLS is an extension of the TLS protocol.

Both protocols operate with the help of certificates. While TLS primarily focuses on encrypting communication between a client and a server, mTLS adds an extra layer of security by requiring both the client and the server to authenticate each other using digital certificates. So mTLS can actually be called an authentication method, TLS is not, it is only for encryption purposes.

Spring Security makes it easy to configure the keystores and truststores required for TLS and mTLS protocols via the application.yml file.

# Suggestions for further development

## Authentication (OAuth2 & SAML)

The authentication methods I implemented are not considered secure in today's world, except for mTLS. By combining Basic authentication with the api-key (second factor) some level of security can be achieved, but the static api-key can be easily leaked.

Implementing OAuth2 and SAML authentication standards in the microservice could significantly enhance security and flexibility. OAuth2 allows for delegated authorization, enabling seamless integration with third-party services (e.g. Google, Facebook, Apple) and ensuring secure access to resources.

On the other hand, SAML provides a robust framework for single sign-on (SSO) across multiple systems, simplifying user authentication and reducing the overhead of managing credentials across different platforms. The SAML protocol is considered to be a little bit legacy compared to OAuth2, but is still very popular in large enterprise systems.

## Roles

The microservice currently has only authentication implented, but no authorization. Implementing authentication without authorization in an application leaves it vulnerable to security risks and potential misuse. Without proper authorization controls, authenticated users may gain access to resources or perform actions beyond their intended privileges, leading to data breaches, unauthorized operations, and other security incidents.

By implementing authorization, access controls can be enforced, user actions can be limited to only those allowed by their roles, and risk of unauthorized access or data manipulation can be mitigated.

In the banking context, roles such as Admin (Full access), Client Service Representative(Limited access, but still possess essential rights) and Client(Most restrictive access, self-service functionalities) could be introduced as the first step towards building proper authorization.

## Secrets and Parameters

The secrets and all other parameters are currently in unencrypted form in the dockerfiles, in the microservice yaml config file and other places. This can easily lead to leakage of secrets and thus to wide range of problems such as data breaches, financial losses, reputation damage, legal consequences, and operational disruption, posing significant risks to individuals and organizations.

To use secrets safely, store and use them in encrypted form, which means that they should be encrypted both at rest and transit. Furthermore, a centralized secure secret and config manager helps to store, manage and rotate them securely.

Also, sending secrets (even if they are encrypted) should happen only over secure, encrypted channels (TLS). The [HNDL (Harvest Now, Decrypt Later)](https://en.wikipedia.org/wiki/Harvest_now,_decrypt_later) decryption strategy is becoming more and more a reality due to the rapid increase of computing capacities. This makes it particularly important to send secrets through a secure, tamper-proof channel.

## Database

During the creation of the custom MariaDB image, the user that is set as the owner of the new database is used by the microservice for database operations. Using the database owner for general database operations is considered to be a bad practice, especially in the case of microservices. Microservices should use service accounts with appropriate privileges and restrictions.

# Tools and Runtime Environments Used During Development

[OpenJDK 17.0.2](https://openjdk.org/projects/jdk/17/)

[Apache Maven 3.9.6](https://maven.apache.org/)

[Docker version 25.0.3, build 4debf41](https://www.docker.com/)

[Git version 2.44.0.windows.1](https://git-scm.com/)

[SourceTree 3.4.17](https://www.sourcetreeapp.com/)

[IntelliJ IDEA Ultimate Build #IU-241.14494.240, built on March 28, 2024](https://www.jetbrains.com/idea/)

[KeyStore Explorer 5.5.3](https://keystore-explorer.org/)

[Postman v10.24.19](https://www.postman.com/)

[HTTPie CLI v3.2.2](https://httpie.io/)

* You might ask why I used 2 HTTP clients (Postman and HTTTPie) during development? Today, many applications are built on a browser engine. This has many advantages, such as easy cross-platform development, lightweight builds, a wide range of built-in features.
* However, often it is the browser engine itself that causes application problems. So I also use another (command line) client for complex calls if I run into an error, especially for calls with utilizing certificates. This way I can make sure that my HTTP client doesn't have some protocol implemented incorrectly or some other bug.