Bachelor's Thesis

Czech
Technical
University
in Prague

FEL

Cloud-Native Application Development

Alena Suvorova

Supervisor: ing. Martin Komárek
Field of study: Software Engineering and Technology
January 2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Suvorova**	Jméno: **Alena**	Osobní číslo: **467560**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Softwarové inženýrství a technologie**		

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Rozšíření ukázkové cloud-native aplikace

Název bakalářské práce anglicky:

Cloud-Native Application Development

Pokyny pro vypracování:

Rozšiřte o nové funkce a zabezpečte stávající ukázkovou cloud-native aplikaci “Rezervace lístků”[1] implementaci několika vzorů/technologií typických pro architekturu mikroslužeb a cloud-native architektur[2]. Zejména:
1. Integrujte systém Keycloak (Identity a Access Management) do aplikace. Využijte vzory/technologie: AOuth2, OIDC, JWT tokens, RBAC (Role based access control).
1.1 Zabezpečte back-end aplikace.
1.2 Implementujte zabezpečení front-end aplikace, včetně adaptivního UI na základě stavu uživatele a jeho role.
2. Přidejte podporu pro správu uživatelů a jejích rezervací (například vytváření účtu, přístup k vlastním rezervacím atd.). Využijte vzory/technologie: service decomposition, Event-Driven Architecture (Kafka), Command Query Responsibility Segregation (CQRS), containerization (Docker), ETags.
Při vývoji používejte relevantní mikroservisní vzory a postupujte iterativně. Vše průběžné testujte, nasazujte na Value Stream Delivery Platformu CodeNOW[3] a dokumentujte.

Seznam doporučené literatury:

[1] Vávra Robin. Vývoj cloud native aplikací v praxi. B.S. thesis, České vysoké učení technické v Praze. Výpočetní a informační centrum., 2022. Dostupné z: http://hdl.handle.net/10467/101026.
[2] Chris Richardson. Microservices patterns: with examples in Java. Manning Publications, 2018. Dostupné z: https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/.
[3] CodeNOW. CodeNOW Documentation, 2022. Dostupné z: https://docs.codenow.com/.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Komárek kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **09.02.2023** Termín odevzdání bakalářské práce: **09.01.2024**

Platnost zadání bakalářské práce: **22.09.2024**

Ing. Martin Komárek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Studentka bere na vědomí, že je povinna vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání Podpis studentky

Acknowledgments

I would like to thank my supervisor,
ing. Martin Komárek for his guidance and patience throughout my academic journey and the creation of this bachelor's work.
I also thank my colleague and fellow CTU student, Robin Vávra, for his valuable advice and thoughtful consultations.

Declaration

I hereby declare that the presented thesis is my own work and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of a university thesis.

In Prague on January 09, 2024

Keywords

Keycloak, CodeNOW, Web security, OAuth2, Microservice, Microservice patterns, Spring Boot, Spring Security, CI/CD, Cloud Native, DevOps, Docker

Abstract

This bachelor's work researches the landscape of web application security, offering practical tutorials through the development of a demo application for the CodeNOW delivery platform. The main goals are to immerse in real-world DevOps practices, enhance web security knowledge, and contribute to CodeNOW documentation.

Focused on CodeNOW's Ticket Reservations demo app development, the project integrates Keycloak as the security provider and extends user management functionalities with a good level of UI/UX in mind.

The text of the implementation part is intended to be a practical guide interwoven with personal insights. It aims to provide a reader with a complete set of code snippets and the necessary configuration to implement all mentioned functionality.

In several development iterations, the Ticket Reservation demo application underwent substantial enhancements, such as obtaining security features on the back end and front end, the basis for user management, some bug fixes, and was deployed locally and on the CodeNOW platform.

While the application is still on track to become production-ready software, a comprehensive groundwork has been done. Valuable insights and proofs of concept documented in the Implementation Part Chapter have the potential to serve as a resource for future CodeNOW guides and tutorials.

Contents

Charter I

Introduction

In today's software landscape, it’s rare to find a product with no concerns about security. The spectrum of application security spans from simple password hashing and input validation to tackling protocols’ intricacies and navigating company policies. The evolution of this realm doesn’t seem to stop, consistently deepening in both scope and complexity. That is why it is important to gain at least a basic orientation in this field for anyone who steps into the world of software development.

This work explores the basics of modern web application protection and provides some practical tutorials that can be used in various projects. The practical aspect of this work is rooted in developing a demo application for CodeNOW, a Czech company, on its software delivery platform.

1.1 Goals

The upper-level objective of this project is familiarisation with real-world work in a DevOps environment within a tech company. The workflow involves practicing the agile approach and operating in iterative cycles. Another goal is to acquire knowledge and a basic understanding of the web security field, subsequently applying it in the development of an existing application. The final product of this work is expected to be an overall improved demo application enriched with security features and user management. The practical insights obtained during this work will contribute to the improvement of CodeNOW documentation.

Charter II

Theoretical Background

This section explores the foundational aspects of securing web applications, aiming to equip the reader with the essentials for understanding the topic and the following Practical Part chapter.

2.1 History

At the early stages of web history, gaining access to 3rd party resources was a non-trivial task. Websites and applications would ask you to give them your credentials from 3rd party resources (like e-mail accounts) in order for them to work. The only protection from malicious intentions was your strong faith in humanity.
Fortunately, new approaches and technologies were invented to address the task of accessing and sharing resources. The emphasis in this work lies on the protocols of the OAuth family. The OAuth takes care of so-called **access delegation** and its release in 2007[1] was an important milestone in web security history.
In 2012 the OAuth2 replaced its predecessor and now is an industry standard[2].
In 2014 OpenID Connect was introduced as a layer on top of the OAuth2, that closes the need for authentication requirements[3].

At the time of the creation of this work (2023-2024), the OAuth2.1 protocol is being developed "*to consolidate and simplify the most commonly used features of OAuth 2.0*"[4].

2.2 Authentication and Authorisation

"***Authentication** is a term that refers to the process of proving that some fact or some document is genuine. In computer science, this term is typically associated with proving a user's identity. Usually, a user proves their identity by providing their credentials, that is, an agreed piece of information shared between the user and the system*"[5]

"***Authorization** is the process of giving someone the ability to access a resource*"[6]

2.2.1 Important Terminology

The security realm is full of jargon. For a comprehensive understanding of this thesis, it is important to be familiar with the specific terminology..
Most of the definitions below are a paraphrase from articles[7] and glossary[8] by Octa.

- **resource** - anything that can be viewed by an end-user (file, web page, etc.)
- **client** - a system that requires access to resources.
- **resource server** - a server hosting protected resources.
- **permission** - an action that can be performed on a resource.
- **role** - a collection of permissions assignable to a user.
- **privilege** - permission that was assigned to someone.
- **scope** - an action that is asked to be performed by a client. Often part of a JWT token.
- **attribute** - a feature of an identity (name, address…). Often part of a JWT token.
- **claim** - an attribute packaged in a security token.

- **access token** - a credential that can be used by an application to access an endpoint. Popular formats are opaque strings and JSON Web Tokens (JWT). They should be transmitted to the API as a Bearer credential in an HTTP Authorization header.

2.2.2 Authorization Strategies

- **Role-Based Access Control** (RBAC) - "*restricts network access based on a person's role within an organization"*[9].
- **Attribute-Based Access Control** (ABAC) - "*a computer system defines whether a user has sufficient access privileges to execute an action based on a trait (attribute or claim) associated with that user*"[6].
- **Relationship-Based Access Control** (ReBAC) - "*relationship can come via a user attribute, such as being a member of a role group related to the object, or having a direct relationship, such as being shared on a document*"[6].

There are other strategies, not mentioned here. The most important strategy in the context of this thesis is the RBAC. It is the strategy used in the Practical part as it is robustly supported by both Keycloak and Spring Security.

2.2.3 OAuth Protocols

OAuth stands for "Open Authorisation". It was the pioneer in securing API access. Created out of the necessity to grant third-party applications *limited* access to user accounts (like logging in with a Google account on a third-party website), OAuth introduced a standardized flow for **authorization**.

2.2.3.1 Client types

OAuth defines two types of clients: confidential (private) clients and public clients. From a technical point of view, confidential clients have credentials (like client secrets) and public clients do not.

"***Confidential clients*** *are applications that are able to securely authenticate with the authorization server, for example being able to keep their registered client secret safe.*

Public clients *are unable to use registered client secrets, such as applications running in a browser or on a mobile device.*"[10]

2.2.3.2 Grant Types

A grant type is a framework mechanism by OAuth protocols, that a client application uses to obtain an access token from an authorization server.

Authorization Code

This grant is a multi-step process. One of the most widely-used grant types.
A user is redirected to an authorization server, which provides credentials. After the user returns to the client via the redirect URL, the application will obtain the authorization code from the URL and use it to request an access token from an authorization server[11].

PKCE

"*PKCE (RFC 7636) is an extension to the Authorization Code flow to prevent CSRF and authorization code injection attacks.*"[12]
It is often recommended to use this version of the Authorization Code grant type.

Client Credentials

"The Client Credentials grant type is used by clients to obtain an access token outside of the context of a user. This is typically used by clients to access resources about themselves rather than to access a user's resources."[13]

Device Code

"*The Device Code grant type is used by browserless or input-constrained devices in the device flow to exchange a previously obtained device code for an access token.*"[14]

Refresh Token

"*The Refresh Token grant type is used by clients to exchange a refresh token for an access token when the access token has expired.*"[15]

Implicit Flow

It is a legacy from OAuth, not recommended for use anymore.
"*The Implicit flow was a simplified OAuth flow previously recommended for native apps and JavaScript apps where the access token was returned immediately without an extra authorization code exchange step.*"[16]

Password Grant

"*The Password grant type is a legacy way to exchange a user's credentials for an access token. Because the client application has to collect the user's password and send it to the authorization server, it is not recommended that this grant be used at all anymore.*"[17]

2.2.4 OpenID Connect

While OAuth 2.0 is about authorization, OIDC is about user **authentication**.

"*OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It enables Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.*"[18]

2.2.4.1 ID Token

Official OIDC specification defines an ID token as follows:
"*The primary extension that OpenID Connect makes to OAuth 2.0 to enable End-Users to be Authenticated is the ID Token data structure. The ID Token is a security token that contains Claims about the Authentication of an End-User by an Authorization Server when using a Client, and potentially other requested Claims. The ID Token is represented as a JSON Web Token (JWT).*"[18]

Specific Claims

The list is not full. For more details see the OpenID Connect specification[18].

- **iss** - REQUIRED. Issuer Identifier for the Issuer of the response. The *iss* value is a case-sensitive URL.
- **sub** - REQUIRED. Subject Identifier. A locally unique and never-assigned identifier within the Issuer for the end user, which is intended to be consumed by the Client.
- **aud** - REQUIRED. Audience(s) that this ID Token is intended for. It MUST contain the OAuth 2.0 client_id of the Relying Party as an audience value.
- **exp** - REQUIRED. Expiration time on or after which the ID Token MUST NOT be accepted by the RP when performing authentication with the OP.
- **iat** - REQUIRED. Time at which the JWT was issued. Its value is a JSON number representing the number of seconds from 1970-01-01T00:00:00Z as measured in UTC until the date/time.
- **auth_time** - Time when the End-User authentication occurred.
- **nonce** - String value used to associate a Client session with an ID Token, and to mitigate replay attacks.

2.2.5 Tokens

In a microservice architecture, there are basically 2 types of tokens to be used by the API gateway to pass user information to a server: opaque and transparent.[19]

2.2.5.1 Opaque Token

An opaque token is usually a cryptographically strong random number. A client does not have enough data to obtain the user's information by a token itself.
"*The downside of opaque tokens is that they reduce performance and availability and increase latency. That's because the recipient of such a token must make a synchronous RPC call to a security service to validate the token and retrieve the user information.*"[19]

2.2.5.2 Transparent Token: JWT

Another option is using so-called transparent tokens. Such tokens are self-contained - a client can obtain information (such as the user's identity and roles) directly from a token, which eliminates the call to the security service.

A widely-used standard for transparent tokens is the JSON Web Token (JWT).
"*It's signed with a secret that's only known to the creator of the JWT, such as the API gateway and the recipient of the JWT, such as a service. The secret ensures that a malicious third party can't forge or tamper with a JWT*"[20]

One issue with JWT is that there's no practical way to revoke an individual JWT that has fallen into the hands of a malicious third party. "*The solution is to issue JWTs with short expiration times, because that limits what a malicious party could do. One drawback of short-lived JWTs, though, is that the application must somehow continually reissue JWTs to keep the session active*"[20]. This problem is tackled by OAuth2 (see *2.2.3.2 Grant Types*: *Refresh Token*)

2.3 Cloud Native Applications

The interpretation of the term "cloud native" has still does not have a strict definition. The most comprehensive approach for defying a cloud-native app is through 12 factors by Adam Wiggind[21].

- **Codebase** - centralized orchestration of changes to the program (e.g. utilizing Git platforms) and deploying the same code base to the different environments (dev, prod…).
- **Dependencies** - provide a consistent deployment environment by utilizing virtual environments (e.g. venv, Docker).
- **Config** - configuration is externalized in a separate file and set as environment variables.
- **Backing Services -** changing a service implementation should not affect the app code, except for configuration.
- **Build, release, run** - separating development phases.
- **Processes** - implement stateless processes, using an external cache (like Redis) to share data..
- **Port Biding** - an application is listening to a specific port, rather than to a specific web server.
- **Concurrency** - use horizontal scaling and load balancers.
- **Disposability** - an application should complete all the tasks before shutting down.
- **Dev/prod parity** - dev and prod processes are kept as close as possible (using CI/CD).
- **Logs** - .lods are written to a standard output/JSON and then processed by a centralized solution.
- **Admin processes** - administrative tasks run as separate processes.

Some platforms, such as CodeNOW (see *2.4.3 CodeNOW - The Cloud Software Delivery Platform*) by design shape applications into a cloud-native, by providing external configuration, code management with git, environments support, etc.

2.4 Software Solutions

This section describes some of the main software solutions relevant to the Practical Part.

2.4.1 Keycloak as 3rd Party Security Solutions

In a microservice environment using a 3rd party solution is widespread practice. Using a 3rd party security provider is a cost-effective solution.

2.4.1.1 Keycloak Overview

Keycloak is an open-source Java-based Identity and Access Management (IAM) solution developed by Red Hat. It has a Community Free version and Red Hat SSO Commercial Offering.

2.4.1.2 Why Keycloak

While the choice of an identity and access management (IAM) solution depends on specific requirements and use cases, Keycloak has certain advantages that make it suitable for a wide range of projects:

- Open Source and Vendor-Neutral
- Cloud Native - Keycloak was accepted to CNCF on April 10, 2023, at the **Incubating** maturity level[22].
- Wide Range of Protocol Support - OAuth 2.0, OICS, SAML 2.0
- Scalability and High Availability
- Robust set functionality - SSO, Multi-Factor Authentication, Social Login, Directory Services, Customization, and Extensible…

2.4.1.3 Structure

- **Realm** - the main unit of Keycloak structure, encapsulating clients, users, roles, etc.
    - **Clients** - the applications to secure, managed by Keycloak
        - **Client roles** - roles specific to the client
        - **Groups** - sets of users with common attributes and roles.
    - **Realm roles** - roles common for all clients
    - **Users** - entities that are able to log into the system.
        - attributes
    - **Themes -** customization of UI and emails
    - **User Federation** - external user storages (Kerberos, LDAP).
    - **Identity provider** (IDP) - external authentication providers (social login, e.g. GitHub, Facebook..)
    - **Keys -** private keys.

2.4.1.4 Keycloak Architecture

The main governing tool of the Keycloak is the **Admin Console** running on the Keycloak server. With it, it is possible to create and configure realms, clients, users, etc.
Account Console provides an interface for users to manage their accounts.
Keycloak server exposes several important HTTP endpoints, that give access to Admin and Account consoles front end and back end. Particularly important is **Admin Rest API**, which allows management of a Keycloak realm through Admin Console UI, directly from a browser or with **Admin Client**. This client allows accessing realm back-end endpoints programmatically from within managed apps.
Keycloak provides tools for generating or listening to events (e.g. logins, user registrations…).
A Keycloak server is initialized with an **embedded relational database called H2**. Later the server can migrate towards more prod-friendly solutions like PostgreSQL, Oracle Database, etc.

The illustration[23] by Thomas Darimont *Figure 1. Keycloak architecture* shows details of a Keycloak server architecture.

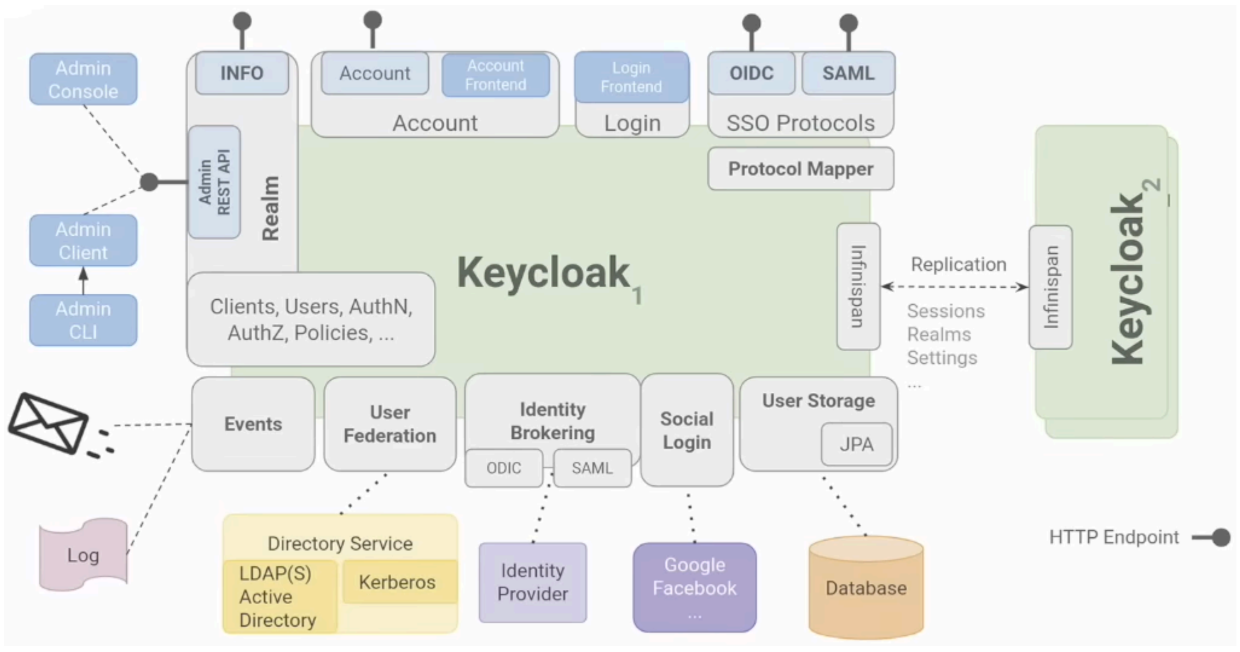


Figure 1. Keycloak architecture

Keycloak allows the creation of a clustered setup for providing higher system availability and stability. Keycloak instances distribute cache via JGroups[24] and Infinispan[25], distributing SSO sessions, volatile configuration information, etc.

2.4.2 Spring Framework

"*The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. ...Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments*"[26]

Spring Framework provides infrastructural support for a comprehensive list of application concerns, such as cloud distribution, data serialization, load balancing, and, of course, **security**. This support can be amplified by using Spring Boot:

"*Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run"... Most Spring Boot applications need minimal Spring configuration.*"[27]

2.4.2.1 Spring Security

"*Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications*"[49]

Here is the description of the main Spring Security interfaces and important classes:

- The web security is based on Servlet *Filters*, that are layered in a particular order. This order can be changed. To register a custom filter usually a bean customizing ***HttpSecurity*** class is used.
- The main strategy interface for authentication is ***AuthenticationManager***. The most commonly used implementation of *AuthenticationManager* is *ProviderManager*. Spring Boot provides a reasonably configured default global *AuthenticationManager*.[28]
- The next important interface is ***AuthenticationProvider***.

"*An Authentication request is processed by an AuthenticationProvider, and a fully authenticated object with full credentials is returned. The standard and most common implementation is the DaoAuthenticationProvider, which retrieves the user details from a simple, read-only user DAO, the UserDetailsService.*"[29]

- The ***Authentication*** interface represents the currently authenticated user. It contains principal, credentials, and authorities (roles and scopes)
- ***SecurityContextHolder*** - the core of Spring Security's authentication model. It contains the ***SecurityContext***. *SecurityContext* keeps track of current Authentication.[30]

2.4.3 CodeNOW - The Cloud Software Delivery Platform

"*CodeNOW® is a* ***Cloud Software Delivery Platform***"[31]. This platform aims to aid over the full software delivery lifecycle, from creating an application, through tests and deployment on various environments, to aftercare.

2.4.3.1 Platform Structure

- **Cluster** - an instance of a CodeNOW.
  - **Applications** - microservice-based software, set of components.
    - **Components** - microservices.
      - **Connected services** - a contract for managed or external services (see *2.4.3.2 Services*). A specific instance of a service is set in the Deployment configuration.
      - **Source code** - code hosted in a GitLab repository.
      - **Builds** - releases and previews of specific git branches of the component.
    - **Packages** - a group of specific versions of application components
      - **Components' releases** - specific version of a component.
    - **Deployment configurations**
      - ***.yaml*** **files** - a set of configuration files that contain environment variables for components.
      - **Runtime configuration** - resource limits and number of replicas.
      - **Target connected services** - specific instances of managed or external service.
    - **Deployments**
      - **Target environment** - the environment the app will be deployed on.
      - **Target package** - application version.
      - **Target configuration -** environment and package-specific configuration.
      - **Domain name** - configuration of custom DNS.
    - **Documentation** - Swagger Editor+draw.io documentation for application components deployed in a separate git repository.
  - **Environments** - a place where applications are deployed (e.g. dev, prod, UAT…)
    - **Deployed managed services Deployed application packages**
  - **Managed Services -** third-party technological services managed by CodeNOW.
    - **Target environment** - a place where resources for the service are allocated.
  - **External Services**
    - **Template** - reusable interface defying variables.
  - **Libraries** - Java, .NET, or TypeScript/Angular9 reusable code, hosted in GitLab repositories, managed by CodeNOW.
  - **CI/CD pipelines** - set of tasks for a component build

2.4.3.2 Services

There are two types of services on CodeNOW: managed services and external services.
Managed Service is a 3rd party application, deployed, configured, and managed by CodeNOW. It aims to save time in installing and configuring infrastructure surrounding an application. A managed service typically provides a variables list to add in the **Deployment Configuration** *.yaml* file, for CodeNOW to autocomplete it during a component build.
At the time of the thesis creation (2023-2024), CodeNOW supported the following services: **Keycloak** (v 10.0.1), **Apache Kafka** (v 2.8.0), **PostgreSQL** (v 12), **CockroachDB** (v 21.1.6), **MariaDB** (v 10.6.7), **Redis** (v 5.0.3), **RabbitMQ** (v 3.8.9)

External Service is a set of variables for connecting and configuring a 3rd party application, that is not presented in the list of Managed Services. It can hold any custom variables, and as in the Managed Service case, CodeNOW auto-completes them during a component build.

2.4.3.3 Other functionality

CodeNOW provides a broad spectrum of functionalities, that can not be listed here due to the thesis format limitations. The full platform overview is in the official CodeNOW Documentation[32]. The most noticeable tools are for analysis and quality management. After an application is released, its endpoint is accessible in *Swagger* as well as a health check. Logging and tracing can be configured to be displayed with embedded *Logger*, *Jaeger*, and *Grafana*. Static code analysis helps to create a bug-free application with minimum security issues.

2.5 Related Technologies

There are short descriptions of some technologies relevant to the understanding of the Practical Part.

2.5.1 CQRS Pattern

"*The command query responsibility segregation (CQRS) pattern separates the data mutation, or the command part of a system, from the query part. You can use the CQRS pattern to separate updates and queries if they have different requirements for throughput, latency, or consistency.*"[33]

2.5.2 Etag Header

"*The ETag (or entity tag) HTTP response header is an identifier for a specific version of a resource. It lets caches be more efficient …, as a web server does not need to resend a full response if the content was not changed.*"[34]

2.5.3 CORS Protocol

"*Cross-Origin Resource Sharing* *(CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request...*"[35]

2.6 Conclusion

This chapter provided the context for the Practical Part of this thesis. The list of software solutions described here is not complete and appropriate notes and references will be provided later in the implementation context.

Charter III

Practical Part

This section describes the phases of the demo application development starting from the basic project setup. It discusses the encountered implementation challenges and the corresponding solutions.

3.1 Stylistics

This work aims to create a basis for a comprehensive guide that can be used by other people and subsequently transformed into codeNOW documentation. To achieve this, I use an **imperative stylistic** approach, accompanied by step-by-step tutorial sections. Also, as it is still a description of my implementation journey, it is intertwined with **personal implementation notes and insights**, as well as descriptions of the decision process to provide a well-rounded narrative.

The code and configuration snippets are marked with `the blue monospaced font`. The error snippets are marked with the `red monospaced font`. References to the chapters are written in *italics*.

3.1.1 CodeNOW Deployment Disclaimer

During the local application development, the changes were committed via GIT into CodeNOW GitLab repositories. A detailed description of it is not provided here, as most of the instructions can be found on the CodeNOW documentation website - docs.codenow.com. The functionality of the deployed application was tested with Swagger and debugged using Logging and Tracing tools provided by the CodeNOW platform.
Any CodeNOW-specific errors and intricacies are mentioned in the relevant sections.

3.2 Introduction

Ticket Reservations is an application being developed by CodeNOW for demonstrative purposes.
In this project, I have two main implementation goals:

- Integrate Keycloak into the app, securing the front end and back end.
- Extend the demo app with relatable user management functions

The secondary goals are providing an acceptable level of UI/UX, getting familiar with diverse technologies, and acquiring insights that can be used in CodeNOW documentation.

3.2.1 Technologies and Patterns

Quick overview of the technologies and patterns used in this project.

Deployment

Development

Testing

Deployment

- CodeNOW platform
- Docker - all the infrastructure for demo-app components during local development is deployed in containers.
- Git

Development

- Keycloak
    - OAuth2, OpenID, JWT tokens
- Spring Boot
    - Spring Security
    - Spring Data JPA
- PostgreSQL + pgAdmin
- Redis
- Kafka + Kafdrop
- React + TypeScript
    - Axios
    - MUI

Testing

- Postman
- Swagger (on the CodeNOW platform)
- Web browsers (Firefox, Chrome)

Further patterns and technologies

- CQRS
- API Gateway
- ETag
- Externalized configuration (by CodeNOW)

3.2.2 As-Is State

Ticket reservation app is a single-page web application, where a customer can choose from several destination points and reserve tickets.

Demo-app is a growing application serving as an example of CodeNOW platform usage and good practices in software development. It has a variety of modules, but in this work, only relevant ones will be mentioned and used.

Initially, the demo app has a simple user interface functionality:

- Choose destinations and departure dates
- Choose from available tickets
- Create a reservation using first name, second name, and email.

The initial version of the front end lacks views of created reservations, login flow, and an account-dependent interface. Although the backend provides a foundation for CRUD operations and email notifications, it requires further enhancements and modifications.

The demo app already implements the CQRS pattern, using asynchronous communication via Kafka, and has a basement for web traffic optimization using ETag. Those qualities are to be reckoned with and they greatly influenced the development process.

3.2.2.1 As-Is Architecture

Demo-app has a **microservice** architecture. It has further components:

- Frontend (**FE**) - based on React + TypeScript.
- Backend (**BL**) - based on Spring Boot **(v 2.4.4)**, connected to PostgreSQL.
- **API** - based on Spring Boot **(v 2.4.4)**, using Redis for cache
- **Schedules** - based on Spring Boot **(v 2.4.4)**, connected to PostgreSQL

The visualization is in *Figure 2. As-Is abstract architecture*.

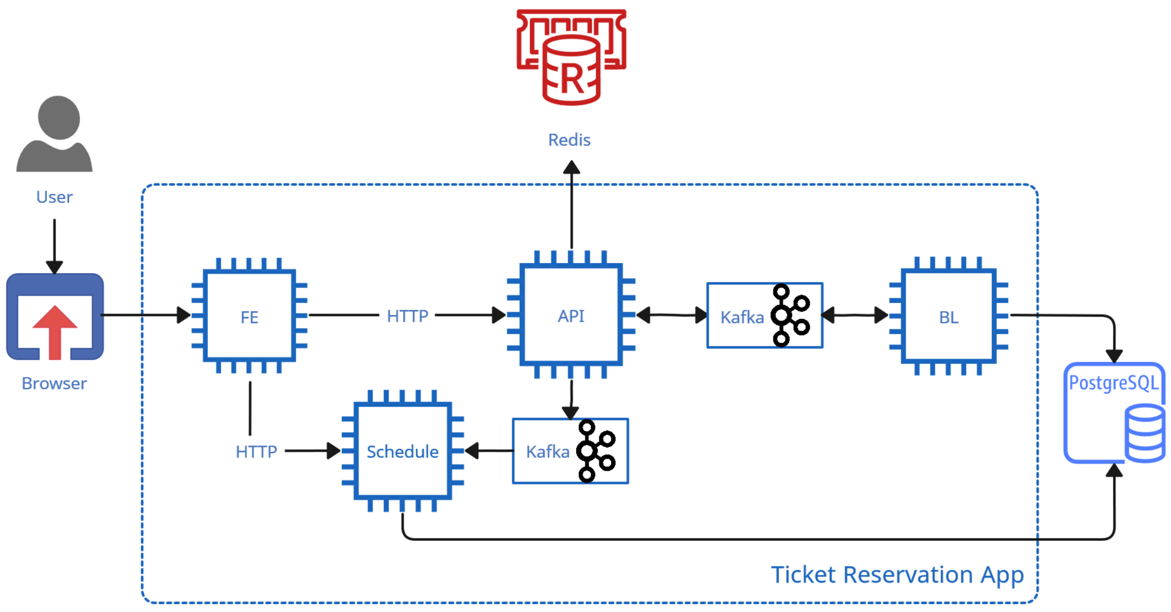


Figure 2. As-Is abstract architecture

The BL and Schedules components communicate with API via Kafka. Frontend uses Axios and communicates with API and Schedules via HTTP protocol.

In *Figure 3. As-Is submit sequence diagram* you may see the process of the reservation (front-end communication with Schedule service is omitted).

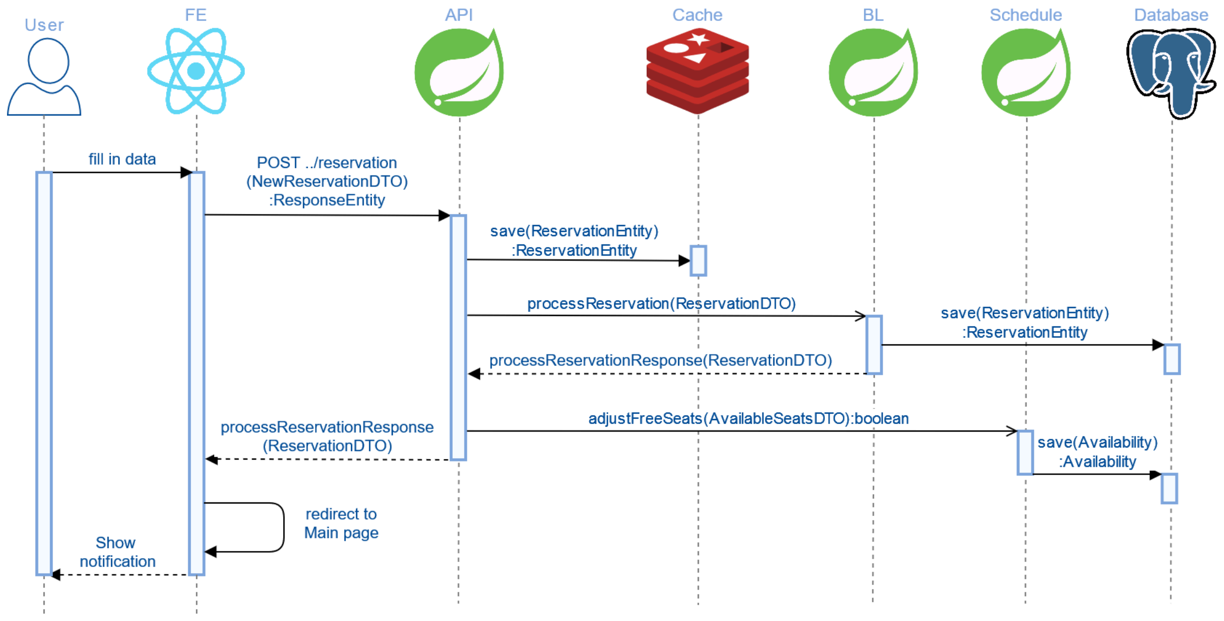

User
FE
API
Cache
BL
Schedule
Database
fill in data
POST ../reservation (NewReservationDTO) :ResponseEntity
save(ReservationEntity) :ReservationEntity
processReservation(ReservationDTO)
save(ReservationEntity) :ReservationEntity
processReservationResponse(ReservationDTO)
adjustFreeSeats(AvailableSeatsDTO):boolean
processReservationResponse (ReservationDTO)
save(Availability) :Availability
redirect to Main page
Show notification

Figure 3. As-Is submit sequence diagram

FE component requests Schedule services view of available destinations. Then a user fills in his data and reserves a ticket: the POST request is sent to API, where 3 things happen:

1. The new reservation is saved with the status PENDING into the cache (synchronously)
2. A request to process the reservation is sent to BL service (asynchronously)
3. A request is sent to Schedule service via Kafka to adjust free seats (asynchronously)

Subsequently, on the BL component the reservation is saved to the database with the status ACTIVE. The response is sent back to API via Kafka and the Cache is refreshed with the processed reservation with status ACTIVE. Some details, such as creating and checking an ETag, are omitted for now and will be discussed further in the text.

Ports and Aliases

For future reference, it is useful to mention the ports bindings (on the local machine) and the alternative names the components may use:

- FE - ticket-reservation-dev-fe, port **3000**
- API - ticket-reservation-dev-api, port **8082**
- BL - ticket-reservation-dev-bl, port **8091**
- Schedules - ticket-reservation-dev-schedules, port **8092**
- Kafka - BOOTSTRAP_SERVERS, port **9092**

3.2.2.2 As-Is Architecture on CodeNOW

On CodeNOW the Ticket reservation uses managed services (e.g. Kafka, Redis…) that are configured and maintained by CodeNOW. Those services are connected to the individual components. An illustration of that is in *Figure 4. As-Is CodeNOW architecture*.

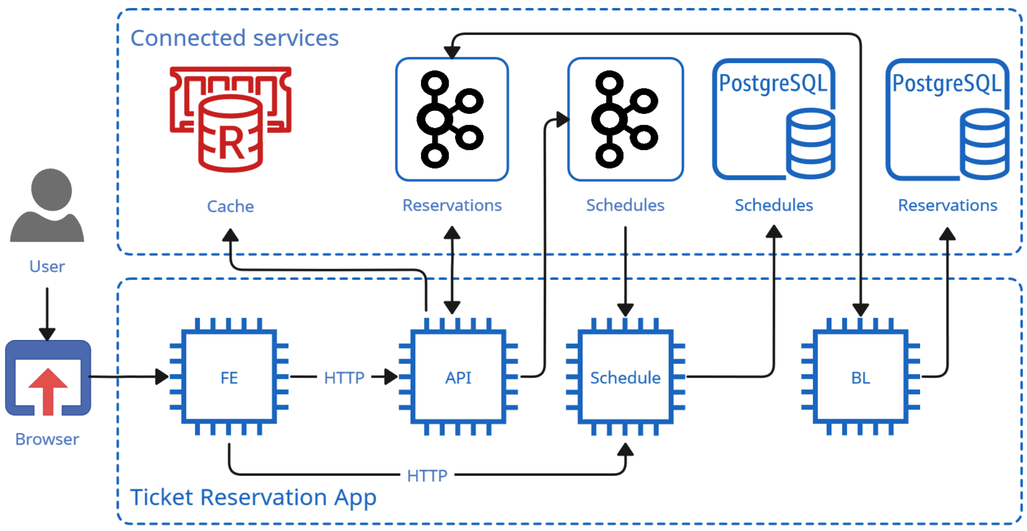


Figure 4. As-Is CodeNOW architecture

3.2.2.3 API REST Controller

This section describes the implementation of endpoints already available in the application.

Create reservation endpoint

```
@PostMapping
private ResponseEntity<ReservationDTO> createReservation(
@RequestBody NewReservationDTO newReservationDTO)
```

Get all reservations endpoint

```
@GetMapping
public ResponseEntity<?> getViewOfReservations(@RequestHeader(required =
false, value = "If-None-Match") String etag)
```

Cancel reservation endpoint

```
@PutMapping("/cancel/{id}")
 public ResponseEntity<ReservationDTO> cancelReservation(@PathVariable UUID id)
```

3.2.3 To-Be State

The demo-app will be integrated with Keycloak and will be enriched with additional functionality to demonstrate the effects of this integration.

3.2.3.1 Main and Side Goals

- **MG1: Keycloak integration**
    - **Authorization and role-based access to Backend endpoints** - access to certain endpoints will be granted based on login status and role
    - **Authorization and role-based access to Frontend endpoints** - access to certain endpoints will be granted based on login status and role
    - **Account-based user interface** - the interface looks different based on login status, role, and user info.
- **MG2: User realm in demo-app** - add functionality for user management
    - **Creating users** - creating in the system, binding with reservations, etc.
    - **Set up user account** - changing user's data, sending emails…
- **MG3: Support functionality** - additional functionality to complete previous goals and better demonstrate the Keycloak functionality.

- **Login** - add login flow on the front end.
- **View of reservations** - add flows to get reservation views on FE.
- **Frontend cache** - add cache support on the front end using ETag.

There are a couple of business rules to respect:

- A user does not have to register to make a reservation
- A reservation can be made only for the mail of a logged user

3.2.3.2 User Workflow Study

There were several possible scenarios of how to manage user account creation and account setup. The final version of registration was shaped through iterative discussions and clarifications with supervisors. The requirements for the system are:

- Good user experience
- User can change their mail
- Registered users can not change the email in a reservation

Note, that this section uses activity diagrams in BPMN notation[36].

Option 1: the account is not created and a user tracks his reservations by a unique ID.

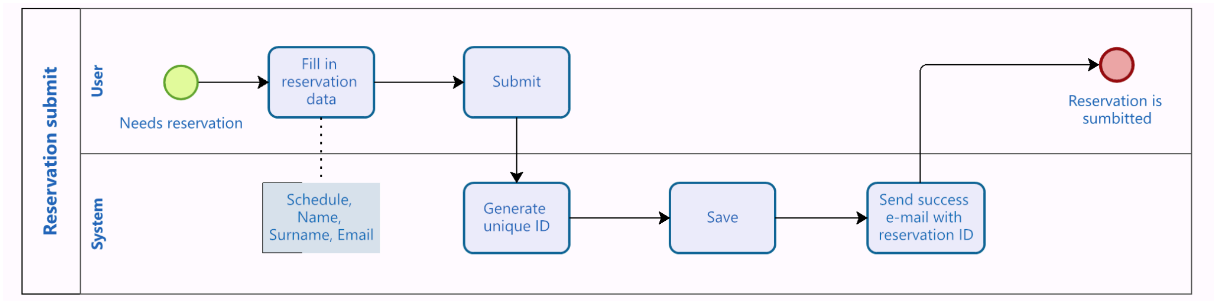


Figure 5. Opt.1 Submit activity diagram

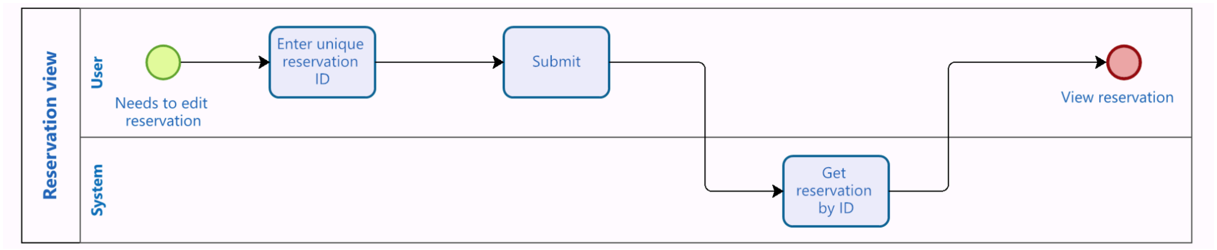


Figure 6. Opt. 1 view activity diagram

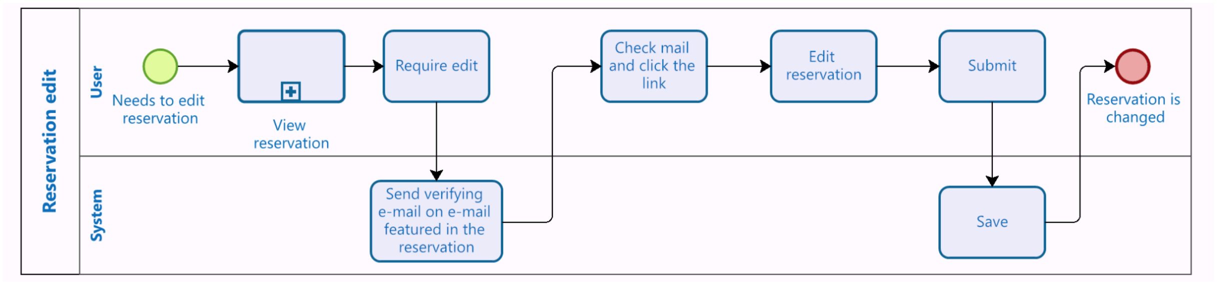

Reservation edit
User
System
Needs to edit reservation
View reservation
Require edit
Check mail and click the link
Edit reservation
Submit
Reservation is changed
Send verifying e-mail on e-mail featured in the reservation
Save

Figure 7. Opt. 1 edit activity diagram

- **Pros**: only small tweaks in the current implementation, minimalistic concept, easy flow for a user.
- **Cons**: no central reservation history - the reservations are accessible only one by one, the user needs to remember several IDs and keep emails, otherwise the reservations are lost.

Option 2: A user has to register before creating a reservation and create a reservation being logged in.

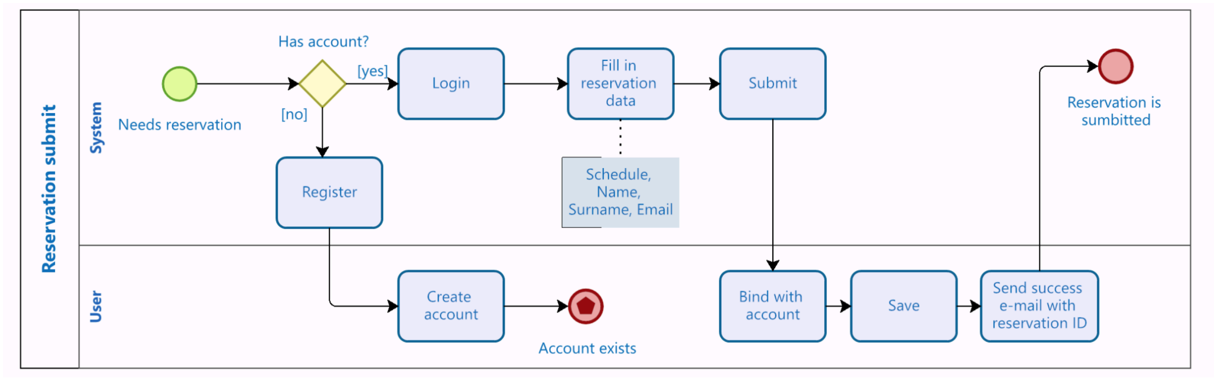


Figure 8. Opt.2 Submit activity diagram

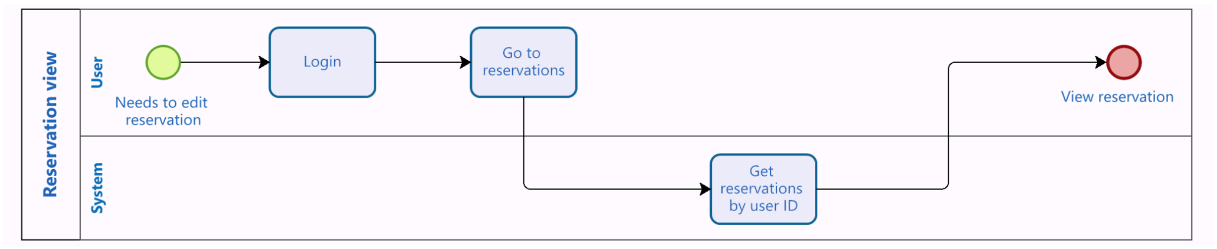


Figure 9. Opt. 2 view activity diagram

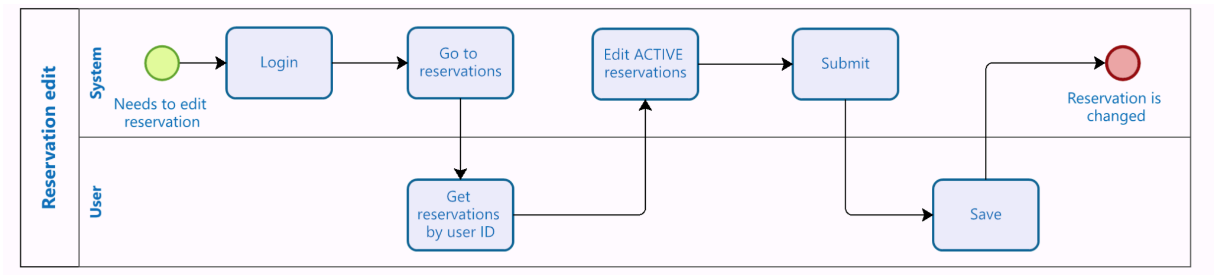


Figure 10. Opt. 2 edit activity diagram

- **Pros**: simple implementation and easy-to-understand flow, this option creates the basis for additional functionality like bulk reservations views, reservation history, statistics…
- **Cons**: unsatisfying user experience - the standard in the industry is not bothering a user with a required account creation.

Option 3: the system allows creating “anonymous” reservations accessible only by a link, as well as creating a reservation bound to an account.

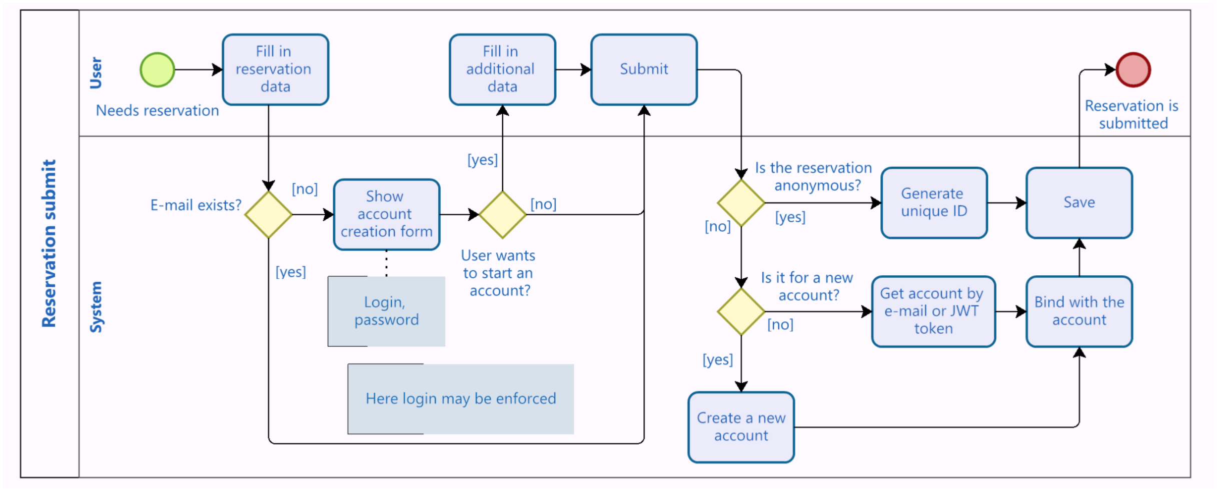


Figure 11. Opt.3 Submit activity diagram

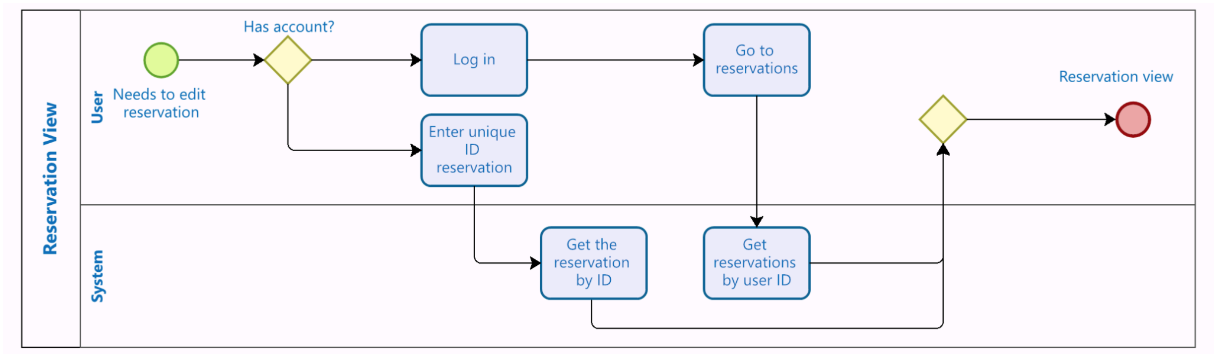

Reservation View
User
System
Needs to edit reservation
Has account?
Log in
Go to reservations
Enter unique ID reservation
Reservation view
Get the reservation by ID
Get reservations by user ID

Figure 12. Opt. 3 view activity diagram

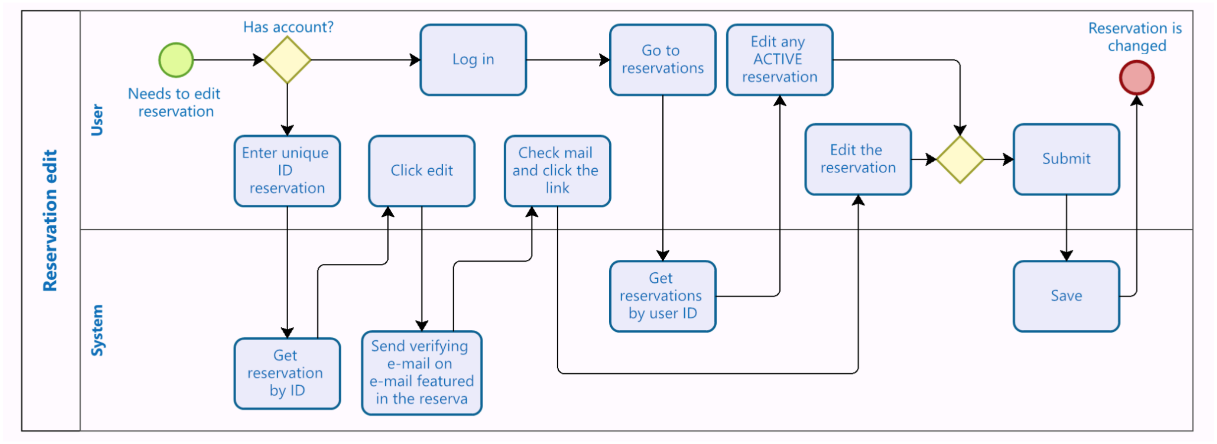

Reservation edit
User
System
Has account?
Needs to edit reservation
Log in
Go to reservations
Edit any ACTIVE reservation
Reservation is changed
Enter unique ID reservation
Click edit
Check mail and click the link
Edit the reservation
Submit
Get reservation by ID
Send verifying e-mail on e-mail featured in the reserva
Get reservations by user ID
Save

Figure 13. Opt. 3 Edit activity diagram

- **Pros**: better user experience, compared to the second option, flexible privacy choice for a user, a lot of opportunities to show off the Keycloak integration.
- **Cons**: sophisticated implementation on UI and backend, may be confusing for a user.

Option 4: the user account is created silently during reservation based on email. The history of reservations is accessible after the account setup.

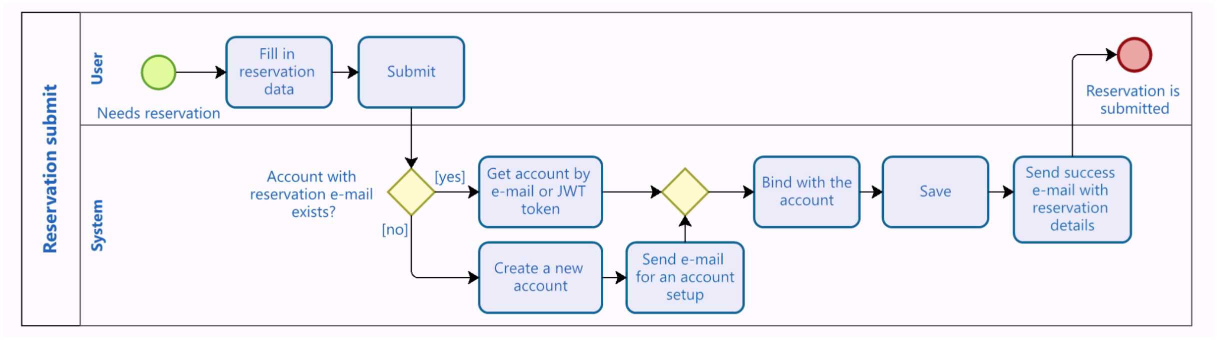


Figure 14. Opt. 4 Submit activity diagram

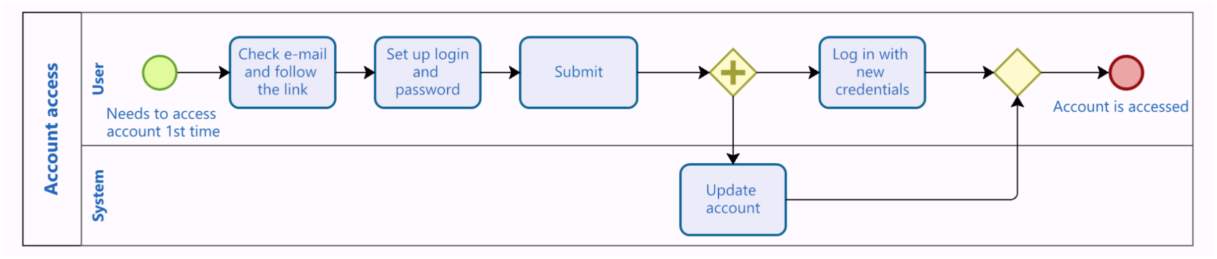

Account access
User
System
Needs to access account 1st time
Check e-mail and follow the link
Set up login and password
Submit
Update account
Log in with new credentials
Account is accessed

Figure 15. Opt. 4 account access activity diagram

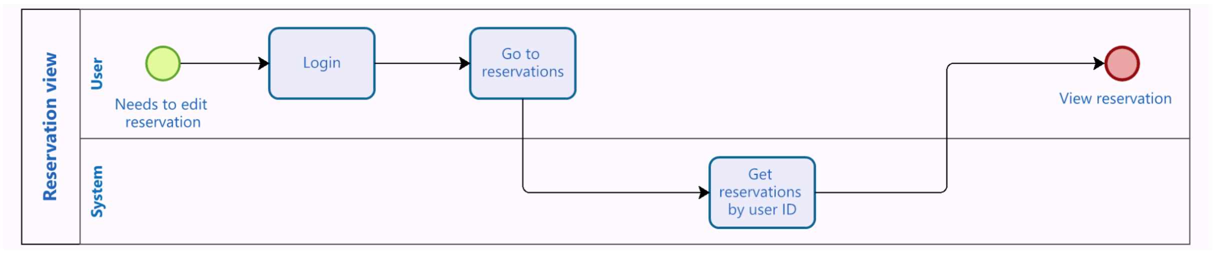


Figure 16. Opt. 4 view activity diagram

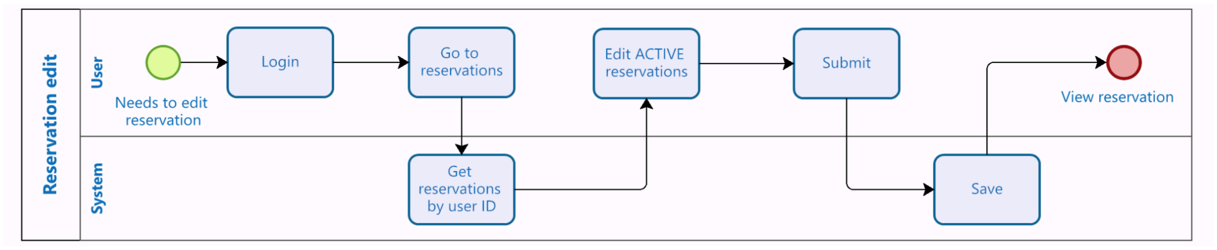


Figure 17. Opt. 4 Edit activity diagram

The flow can be modified later to ask about account creation, which would lead us to a variation of option 3.

- **Pros**: follows a wide-spread account creating practices, allows to show off Keycloak abilities, more straightforward implementation, acceptable user experience level.
- **Cons**: the user data is stored in the system (as an account) without user consent.

The last option (4) has an optimal ratio of implementation complexity and UX, so further implementation is oriented on this scenario.

3.2.3.3 To-Be Architecture

To reach the goals described in the *3.2.3.1 Main and Side Goals*, the component composition and responsibility evolved in several steps during the development process, changing back and forth. The guidelines for the application structure were obtained from the “Microservice Patterns” book by Chris Richardson[37].
The main high-level change mainly consists of adding the Keycloak as a 3rd party securing component (see *Figure 18. To-Be abstract architecture*).

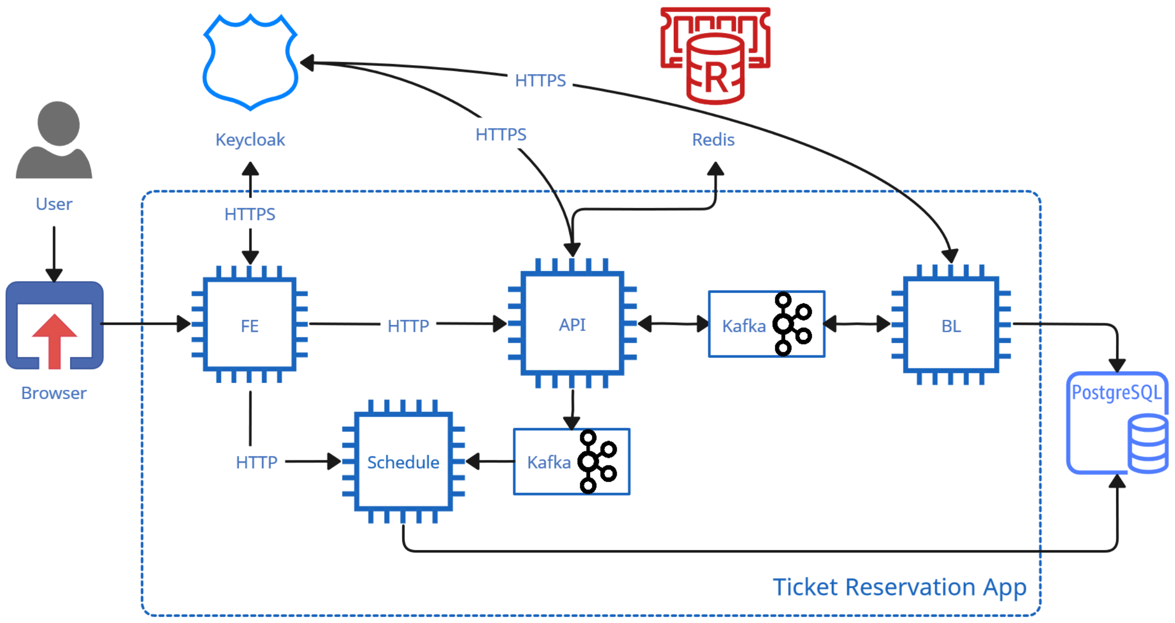


Figure 18. To-Be abstract architecture

In the microservice architecture, it is recommended to enforce security policies on a single point - API Gateway[38]. So if any security policy has to be enforced, it should happen on the API component, specifically on the REST Controller.

3.3 Project Setup

This chapter describes the steps necessary to start developing the demo app.

3.3.1 Source Code

The first step is to clone all relevant repositories from CodeNOW on the local machine.

Prerequisites:

- Be registered on the platform with all necessary permissions or the ADMIN role
- Set up SSH keys for the CodeNOW profile.
  - The minor, but time-consuming trouble for Windows users could be setting up the config file from CodeNOW. On the system disk (usually C:/) go to: *Users\{Your profile}*. Create a folder named "*.ssh*". Create a *.txt* file named "config" and then remove the extension. If you do not see the file extension, go to the View tab in File Explorer and check "File name extensions". Then, copy the config file and modify it according to the instructions on CodeNOW.
- Set up *.m2* folder following the CodeNOW local development tutorial[39].
- For Windows users, have Git SCM to Windows.

Steps:

On CodeNOW go to:

Application ➝ *{Your app}* ➝ *{Your component}* ➝ *Choose Action* ➝ *Clone Repository*.

There the git copy command will be provided, e.g.:

git clone git@gitlab.cloud.codenow.com:codenow/stxplayground/ticket-reservation-dev-api.git

Then open the Git Bash in the project folder and simply paste the command. Repeat this step for each component.

3.3.2 Project Setup in the IDE

Prerequisites

- IntelliJ IDEA (version 2020.3.2+)
- Java 11

3.3.2.1 Project Structure

The reference IDE used to develop this project is **IntelliJ IDEA**, which has recommended itself as an extremely user-friendly IDE. Nonetheless, even a project setup had its challenges. The goal is to set up several modules in one IDE instance. Such a structure saves machine resources, gives easy access to logs, comfortable navigation and deployment management (see *Figure 19. Project structure*).

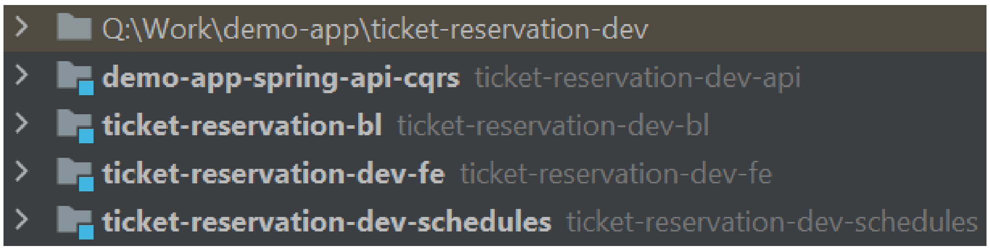

Q:\Work\demo-app\ticket-reservation-dev
demo-app-spring-api-cqrs ticket-reservation-dev-api
ticket-reservation-bl ticket-reservation-dev-bl
ticket-reservation-dev-fe ticket-reservation-dev-fe
ticket-reservation-dev-schedules ticket-reservation-dev-schedules

Figure 19. Project structure

There are several ways to accomplish that. The standard flow is marking directories as source root. This way is **not always reliable**, because autodetection may behave unpredictably, especially if a module is not a java-project.
The best way to create a module structure is from the **Project structure** menu (*ctrl+alt+shift+S*). Alternatively, create a Module from Existing source from the File menu, as shown in *Figure 20. Module creation path*.

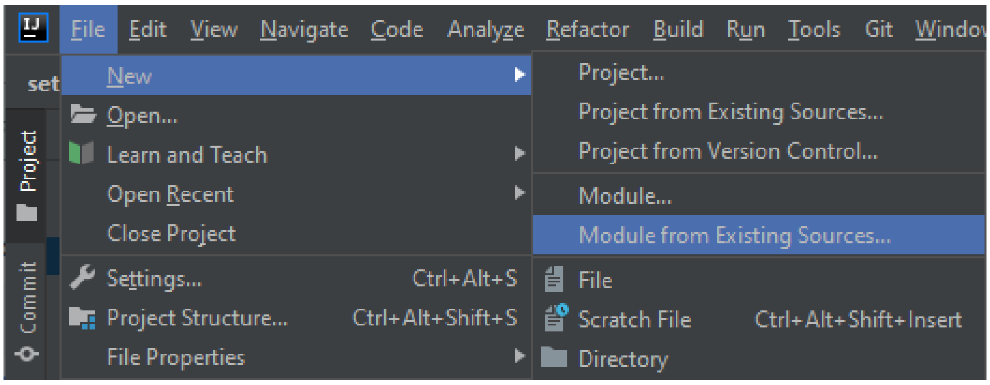


Figure 20. Module creation path

Adding new modules from the Project structure menu is a safer path though. Go to:
File → Project structure → Project Settings → Modules.
There click the + icon (or alt+insert) and choose the Import module option.
For a Maven project, choose the **Import module from external model** option. For React or other non-standard projects choose **Create module from existing source** option.

Warning
DO NOT use the “Create module from an existing source” and “Import separate source files” options for a Maven project! It will create a mess in the module's structure in our case.

3.3.2.2 Spring Boot for CodeNOW Setup

The CodeNOW project has a non-standard path for the Spring configuration file. It is located in the *./codenow/config* directory. For Spring applications to boot, it is necessary to provide a correct Run configuration and set this line as a VM argument in the JVM options[39]:

```
-Dspring.config.location=file:./codenow/config/application.yaml
```

Our case is a bit non-standard though.
Because there are several modules in one project, the provided configuration string does not work and causes a pool of errors, such as:

```
Config data resource 'file[file.\codenow\config\application.yaml]' via location
'file./codenow/config/application.yaml' does not exist.
```

Error 1

After a long investigation and trying out different variations of paths, I found the solution.
To fix this problem, **provide an absolute path** for each module in the VM argument. Below there are examples of such paths for each Spring Boot module.

- API module:

```
-Dspring.config.additional-location=file:C:\demo-app\ticket-reservation-dev\ticket-
reservation-dev-api\codenow\config\application.yaml
```

- BL module:

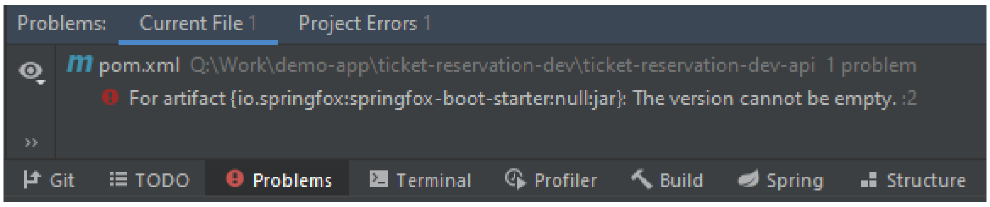
```
-Dspring.config.additional-location=file:C:\demo-app\ticket-reservation-dev\ticket-
reservation-dev-bl\codenow\config\application.yaml
```

- Schedule module:

```
-Dspring.config.additional-location=file:C:\demo-app\ticket-reservation-dev\ticket-
reservation-dev-schedules\codenow\config\application.yaml
```

3.3.2.3 Dependencies bug shooting

Sometimes IDEA doesn't see dependencies from the *pom* file. The problem may be a silent error in the pom file, that will not cause problems during the build, but prevent IDE from indexing dependencies correctly. A typical example is a missing version of a dependency is shown in *Figure 21. Error example - missing version*.



```
pom.xml  Q:\Work\demo-app\ticket-reservation-dev\ticket-reservation-dev-api  1 problem
  For artifact {io.springfox:springfox-boot-starter:null:jar}: The version cannot be empty. :2
```

Figure 21. Error example - missing version

If there are no such errors, but the problem persists, go to

File ➝ Settings ➝ Build, Execution, Deployment ➝ Build Tools ➝ Maven ➝ Repositories

And try to update those repositories. Warning, it takes a long time.

3.3.3 3rd-party Applications Setup

For the demo app to launch and work correctly, certain services must be set up beforehand.

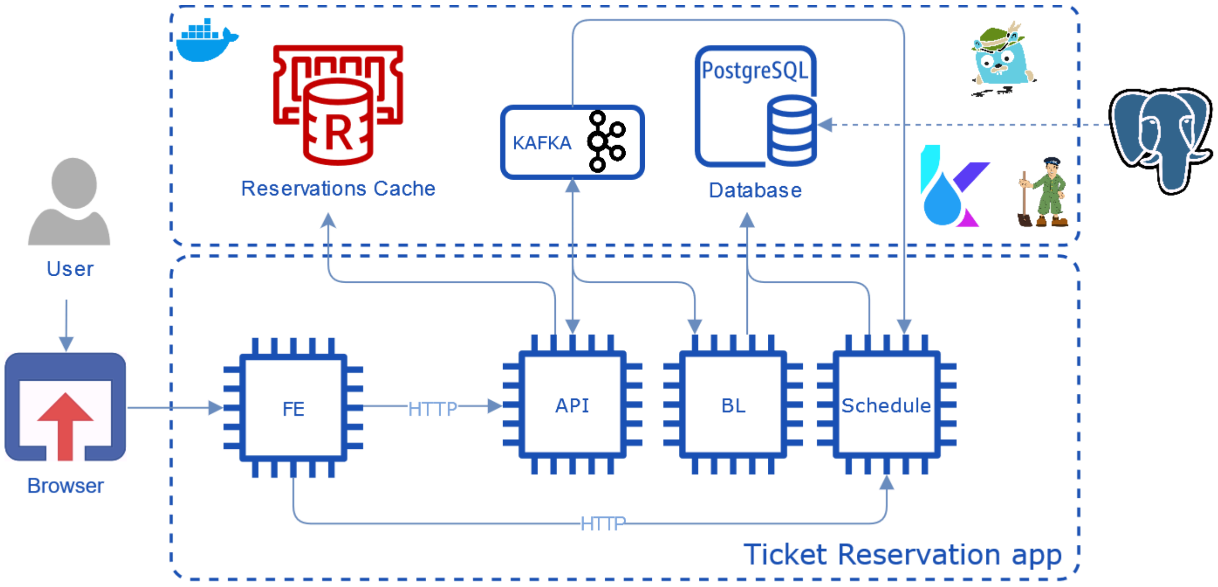


Figure 22. Initial local architecture

To simplify the local development, all side services will be deployed in Docker. As a cloud-native application, Ticket Reservation already has a configuration file (*./codenow/config/application.yaml*) to connect to the required services (see *Figure 22. Initial local architecture*).

Prerequisites

- For Windows users, Docker Desktop for Windows[40]

Initially, I deployed a Keycloak server (version 13) independently on my machine. It has certain positive sides, like easier access to inner files (for example for theme customization), but the overall development process is easier having everything containerized in one place.

Instead of starting the server manually by typing the command:

`./kc.sh start-dev` or `kc.sh start-dev`

It is enough to use the integration of Keycloak and IntelliJ IDEA and start everything from there with one click.

Docker-compose

To automate the creation and build of the containers, the *docker-compose.yaml* file[41] is used. Update the existing docker-compose by adding the Keycloak service image.

Docker-compose Common Structure

The docker-compose file in the project follows the next structure:

```
version:[compose file format version]
services:
[definition of services, e.g. Redis, Keycloak…]
volumes:  [shared volumes[42]]
```

Keycloak Setup

In the "service" section of the docker-compose, add a new Keyloak service. Here is the first version of the Keycloak service:

```
keycloak:
  image: jboss/keycloak:13.0.0
  container_name: keycloak
  environment:
    KEYCLOAK_USER: admin
    KEYCLOAK_PASSWORD: admin
  ports:
    - "28443:8443"
    - "28080:8080"
```

- Keycloak images by JBoss are well-documented and easy to use out of the box. Keycloak images were supported up to the 16.1.1. Although version 13.0.0 is considered outdated, it remains widely used within the Keycloak community.
- To successfully start the server and subsequently access the Admin Console we have to provide the credentials for a master realm ADMIN user in the environment variables.
- Keycloak listens to HTTP traffic on the *8080* port and to HTTPS traffic on the *8443* port. Here, the container's inner *8080* port is mapped to the host machine's *28080* port, providing access to the Admin Console from the host machine. Similarly, the HTTPS port is mapped to the host's *28443* port.
- The image by JBoss doesn't require a command configuration, which is not always the case (see *3.11.1 Keycloak version update guide*)
- The early version of the Keycloak service did not have any volumes, but in the future, it will be added due to theme customization (see *3.7 Fourth Iteration: Custom Theme*).

Note, that It is recommended to provide ports in the string format (e.g. *"28080:8080"*). Otherwise, there may be unpredicted behavior for ports lower than 60, as YAML parses numbers in the xx:yy format as a base-60 value[8].

3.3.4 Keycloak Setup

This section discusses the initial setups integrating Keycloak to the app components locally and on CodeNOW.

3.3.4.1 Keycloak Versions

A couple of words about Keycloak versions used in the project. CodeNOW Keycloak managed service uses Keycloak version 10.0.1. Historically I used **Keycloak 13.0.0** in my local development, which is one of the most popular versions amongst outdated versions, also it is close enough to 10.0.0 and, therefore didn't have groundbreaking changes yet.
Later during development, I had to change the version to a newer one, because I needed support for search by attributes, which starts from the 15.0.0 version. I intended to choose a new version that is as modern as possible but still **compatible with the rest of the project**. Keycloak version **22.0.0 removed support for Java 11 and transferred from Java EE to Jakarta EE namespace**, so I had to settle for the latest 21 version - **Keycloak 21.1.2.** The changes that it took will be described in further chapters.

3.3.4.2 Keycloak Roles

This section describes the roles required for Keycloak to work with the app components.

Admin Role

admin - technical requirement, necessary for creating a Keycloak session from the Spring application.

Warning
This admin role and admin account should not be confused with the admin account of the Master realm! The master realm and the Admin account for logging into the Admin Console are to be used purely for the Keycloak server management.

There are 2 approaches for creating such a role:

1. **Enable service account roles and use a service account for each client separately.** This approach uses the CLIENT CREDENTIALS grant type (see *2.2.3.2 Grant Types*).
2. **Create a realm role and use one user globally.** This approach uses the PASSWORD grant type. This approach is used both for historical reasons and simplicity reasons. In future development, it will be exchanged for the user service and CLIENT CREDENTIALS grant type.

Admin Role Configuration

The **admin** role is assigned to a user to enable various operations (e.g. user creation, mail sending…), therefore the admin role should be composed with most of the predefined Keycloak roles to function properly. For instance, it should have most of the roles defined in the **realm-management client**. To simplify and avoid unnecessary time losses while adding new functionality, simply add all available roles to the admin role.

Administrator Role

administrator - a business role for a user managing the application. This role is used for policy enforcement in REST controllers. For dev purposes, the **admin** role might be used, but it must be mentioned that it creates serious security bridges. It might be created for each client or on the realm level.

Customer Role

customer - a business role for a user making the reservation.

3.3.4.3 Local Keycloak Configuration Guide

Prerequisites

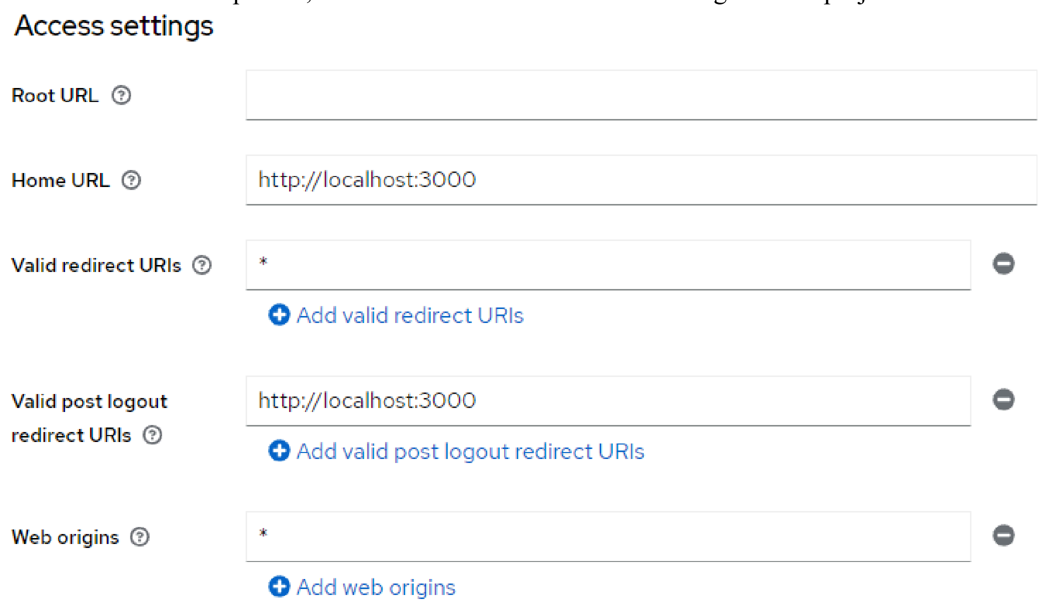
- Running Keycloak server of version 13 and newer.

Note: The screenshots illustrating the Admin Console are of the 21.1.2 version

1. Go to Admin Console. It runs on localhost (or 127.0.0.1 for Windows) on the port defined in the docker-compose file.
2. Log in to the Admin Console with credentials set in the *docker-compose.yaml* file.
3. Create a new realm called *Demo.*
4. Go to: *Realm settings* → *Login tab* and toggle the next options:
    - User registration - enable user self-registration
    - Forgot password - for a case user didn't set up their account in time
    - Login with email - better experience and ensures there is only one mail for each account
    - Edit username - will be important during account setup, as an initial username is autogenerated

    All other options should be off. Other settings will be tackled in the next sections.
5. Go to the Clients tab in the left menu and create clients for the next app components:
    - API - *demo-app-api*
    - BL - *demo-app-bl*
    - FE - *demo-app-fe*

6. API and BL are going to be private clients. Choose the confidential access (old Admin Console) or toggle Client authentication (new Admin Console). There is no need to toggle Authorisation, cause fine-grained authorization will be implemented with Spring Security.
7. Set the access settings as shown in *Figure 23. Access settings example*. The localhost:3000 is the address of the FE component, which will be relevant in the later stages of the project.



Access settings

Root URL	
Home URL	http://localhost:3000
Valid redirect URIs	*
Valid post logout redirect URIs	http://localhost:3000
Web origins	*

Figure 23. Access settings example

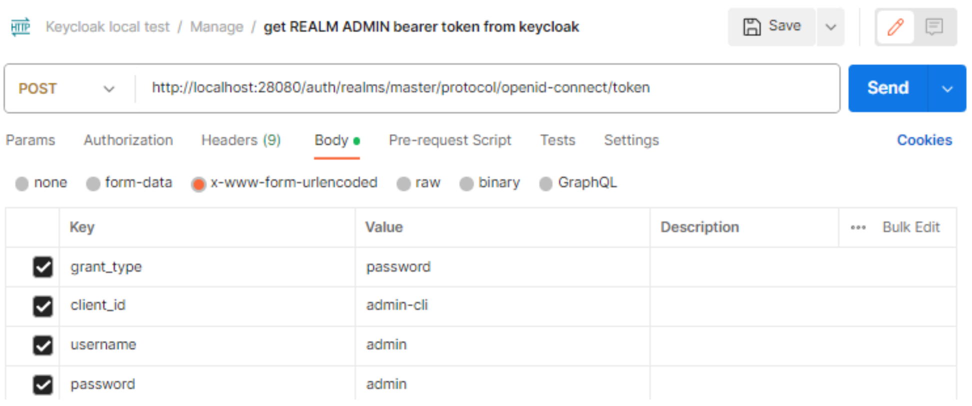
8. Create the necessary roles described in *Keycloak Roles*.
9. Go to: *Users* → *Add* user. Create an admin, a user, and an administrator.
   Demo-admin - necessary for creating a Keycloak session from the Spring application.
   Demo-administrator - this account is for testing purposes.
   Demo-user - this account is for testing purposes.
   Toggle *email verified* and set a password (toggle off the Temporary option) for each user.

After all the steps above, the initial setup of Keycloak is finished.

3.3.4.4 Manage Keycloak with Postman Guide

Sometimes the Admin Console does not provide the functionality needed. For example, you misspelled a realm name, and even after a correction, some autogenerated roles stayed with incorrect names.
In this specific use case, you need to:

1. Obtain a Master realm Admin authorization, as shown in *Figure 24. Admin token endpoint*.



Keycloak local test / Manage / get REALM ADMIN bearer token from keycloak

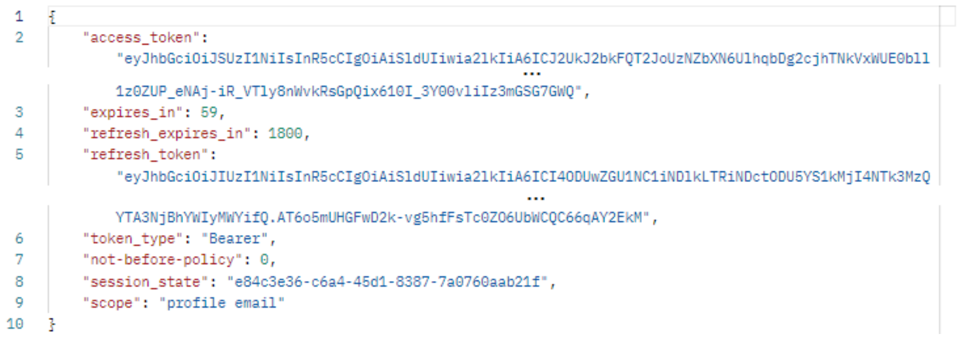
POST http://localhost:28080/auth/realms/master/protocol/openid-connect/token

Body: x-www-form-urlencoded

	Key	Value	Description
✓	grant_type	password	
✓	client_id	admin-cli	
✓	username	admin	
✓	password	admin	

Figure 24. Admin token endpoint

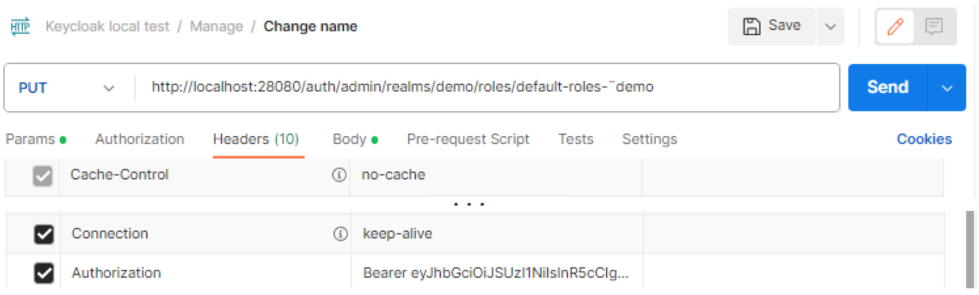
The response JSON will look like this in *Figure 25. Token endpoint response*.



```
{
    "access_token":
        "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJ2UkJ2bkFQT2JoUzNZbXN6UlhqbDg2cjhTNkVxWUE0bll
                                                            ...
        1z0ZUP_eNAj-iR_VTly8nWvkRsGpQix610I_3Y00vliIz3mGSG7GWQ",
    "expires_in": 59,
    "refresh_expires_in": 1800,
    "refresh_token":
        "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI4ODUwZGU1NC1iNDlkLTRiNDctODU5YS1kMjI4NTk3MzQ
                                                            ...
        YTA3NjBhYWIyMWYifQ.AT6o5mUHGFwD2k-vg5hfFsTc0ZO6UbWCQC66qAY2EkM",
    "token_type": "Bearer",
    "not-before-policy": 0,
    "session_state": "e84c3e36-c6a4-45d1-8387-7a0760aab21f",
    "scope": "profile email"
}
```

Figure 25. Token endpoint response

2. Correct the role name. In this example the misspelled name is "*¨demo*", so one of the autogenerated roles is *default-roles-¨demo*. Firstly, put the obtained Realm Admin token into the Authorization header as "Bearer [Access Token]" (see *Figure 26. Role endpoint 1*)



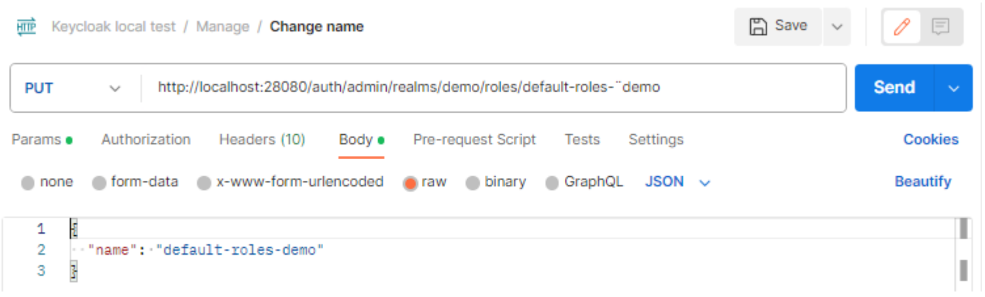
Keycloak local test / Manage / **Change name**

PUT http://localhost:28080/auth/admin/realms/demo/roles/default-roles-¨demo

✓	Cache-Control	no-cache
	...	
✓	Connection	keep-alive
✓	Authorization	Bearer eyJhbGciOiJSUzI1NiIsInR5cCIg...

Figure 26. Role endpoint 1

Then, provide the body of the request with the correct name (see *Figure 27. Role endpoint 2*).



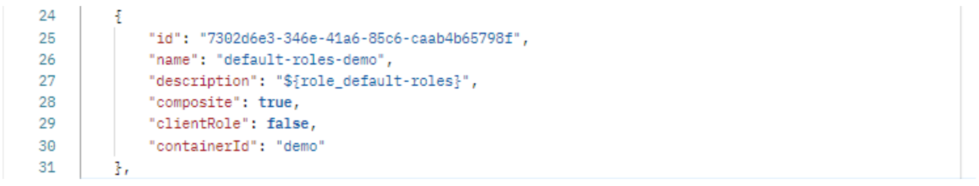
Keycloak local test / Manage / **Change name**

PUT http://localhost:28080/auth/admin/realms/demo/roles/default-roles-¨demo

```
{
  "name": "default-roles-demo"
}
```

Figure 27. Role endpoint 2

3. Check, if the changes were applied correctly by checking the endpoint http://localhost:28080/admin/realms/demo/roles
The name now should be spelled correctly, as shown in *Figure 28. Role info example*.



```
    {
        "id": "7302d6e3-346e-41a6-85c6-caab4b65798f",
        "name": "default-roles-demo",
        "description": "${role_default-roles}",
        "composite": true,
        "clientRole": false,
        "containerId": "demo"
    },
```

Figure 28. Role info example

There are many other useful endpoints, to read about them check the official Keycloak documentation[44].

3.3.4.5 Keycloak as Managed Service on CodeNOW

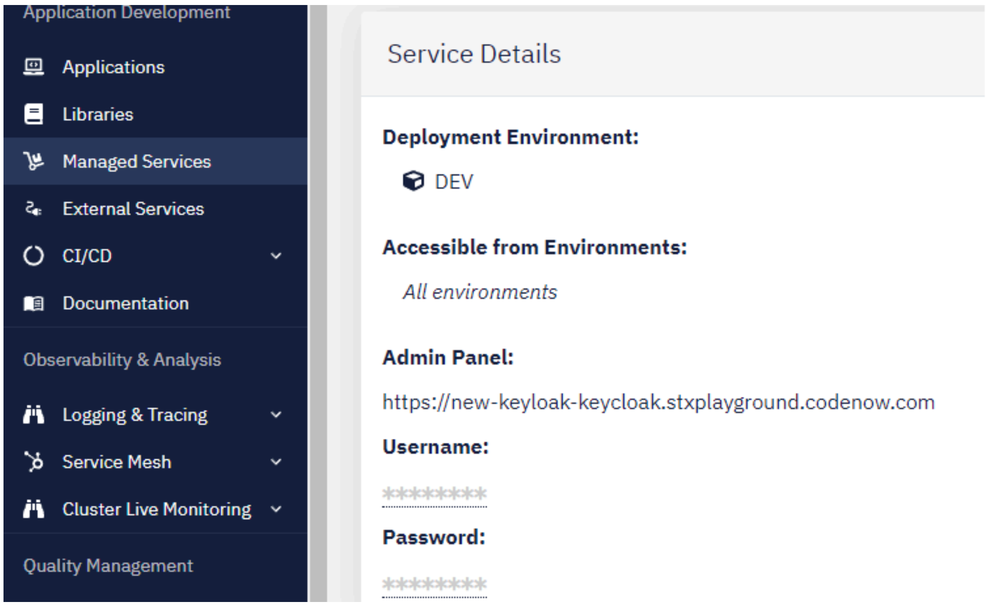
The setup in the admin console locally and on CodeNOW is the same. The challenge is connecting Keycloak managed service to the components within CodeNOW.

Service Setup Guide

The Keycloak instance already runs on CodeNOW. To get to the admin panel go to:

Managed services → Keycloak instance → Service details

There is a link to the admin panel and credentials strings that are accessible from an external network (see *Figure 29. Keycloak Service Details on CodeNOW*). Set up the Keycloak realm as was explained in the previous section *3.3.4.3 Local Keycloak Configuration Guide*.



Service Details

Deployment Environment:
DEV

Accessible from Environments:
All environments

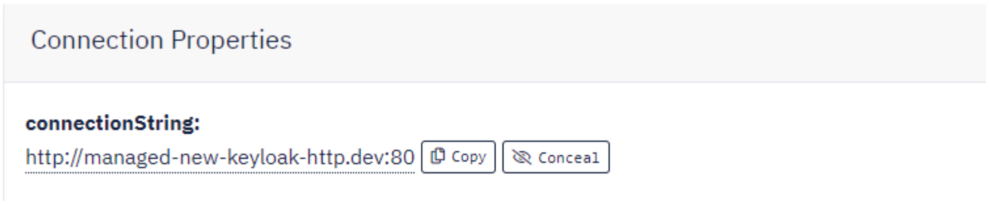
Admin Panel:
https://new-keyloak-keycloak.stxplayground.codenow.com

Username:
****

Password:
****

Figure 29. Keycloak Service Details on CodeNOW

Below the *Service details,* there is the *Connection Properties* section. The *connectionString* is used for connecting the managed service to a component internally on CodeNOW (see *Figure 30. Connection properties*).


Connection Properties

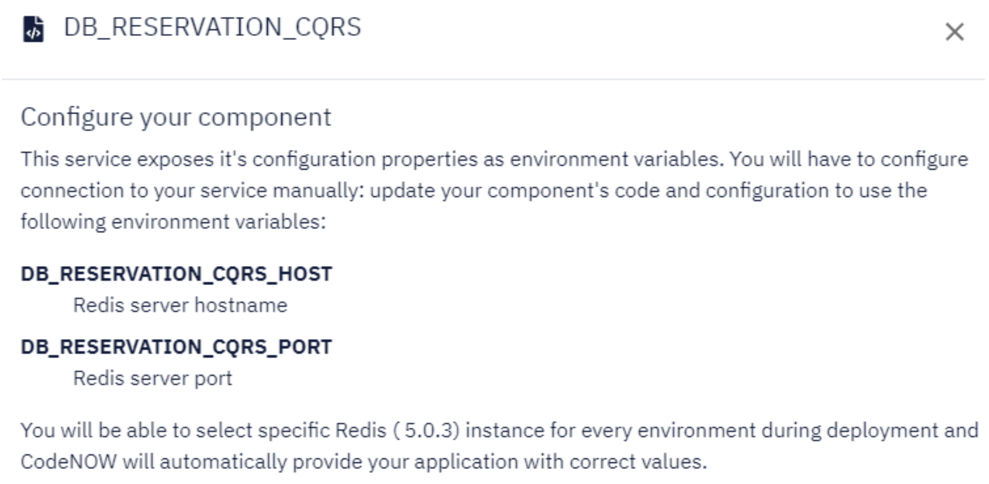
connectionString:
http://managed-new-keyloak-http.dev:80

Figure 30. Connection properties

The steps to connect a managed service to application components are available in the Connect Service to Component guide[45], so the standard instructions are omitted here.

Connected Service and Deployment Configuration Guide

Usually, CodeNOW provides a list of variables that a developer is supposed to add to the configuration file, for CodeNOW to auto-complete during the build. For instance, *Figure 31. Redis service connection properties* shows the list for a Redis service connection.


DB_RESERVATION_CQRS

Configure your component

This service exposes it's configuration properties as environment variables. You will have to configure connection to your service manually: update your component's code and configuration to use the following environment variables:

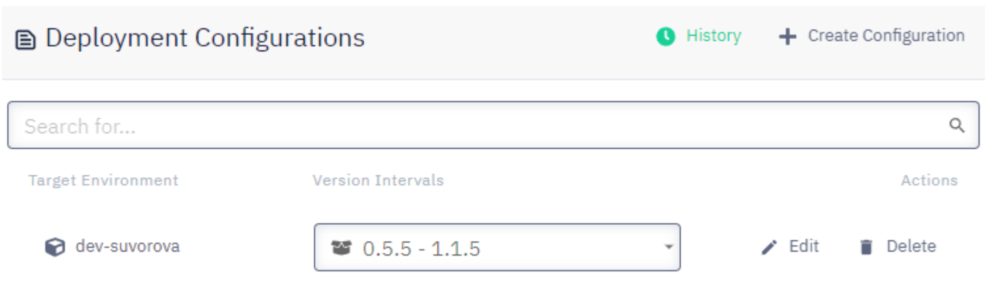
DB_RESERVATION_CQRS_HOST
Redis server hostname

DB_RESERVATION_CQRS_PORT
Redis server port

You will be able to select specific Redis (5.0.3) instance for every environment during deployment and CodeNOW will automatically provide your application with correct values.

Figure 31. Redis service connection properties

Details about Deployment Configuration are in the *2.4.3 CodeNOW - The Cloud Software Delivery Platform*.


Deployment Configurations

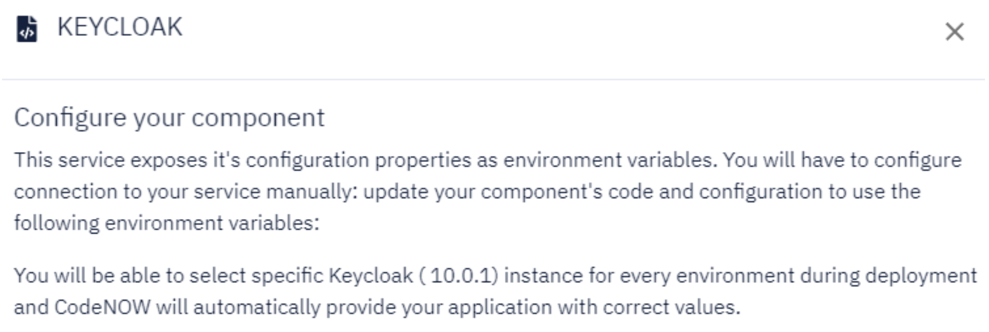
History | Create Configuration

Search for...

Target Environment	Version Intervals	Actions
dev-suvorova	0.5.5 - 1.1.5	Edit Delete

Figure 32. Deployment Configuration interface

The configuration of Keycloak, though, can vary significantly based on the project stack and required functionality. As shown in *Figure 33. Keycloak service connection properties*, **there is no predefined interface** for Keycloak on CodeNOW.


KEYCLOAK

Configure your component

This service exposes it's configuration properties as environment variables. You will have to configure connection to your service manually: update your component's code and configuration to use the following environment variables:

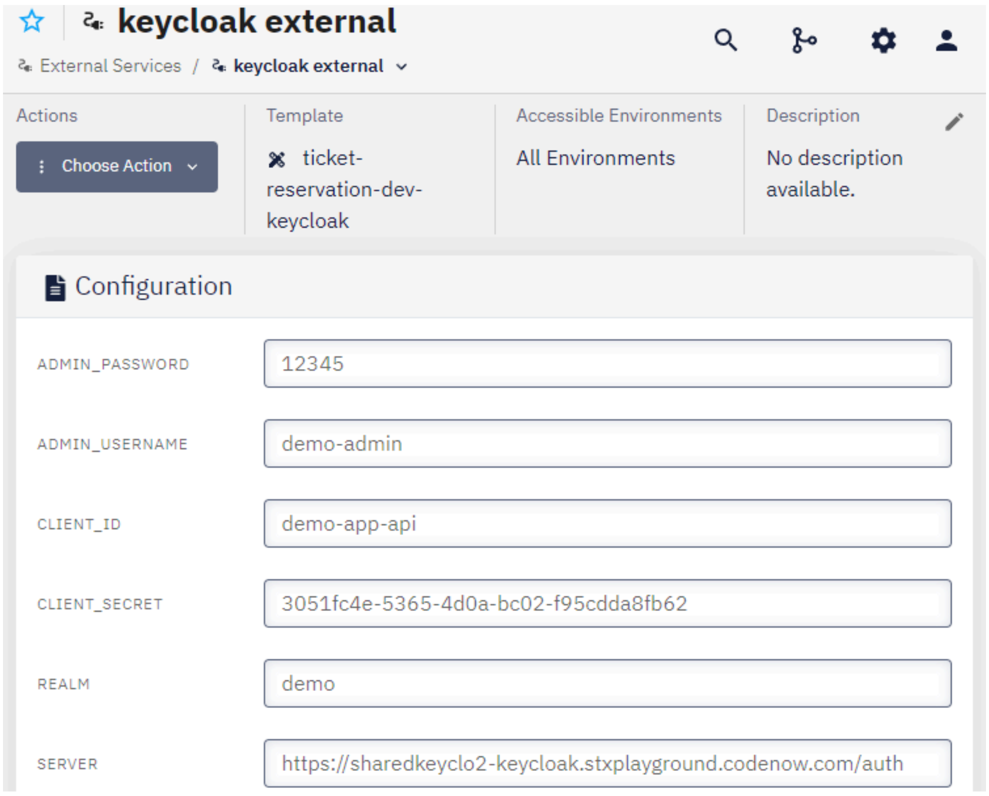
You will be able to select specific Keycloak (10.0.1) instance for every environment during deployment and CodeNOW will automatically provide your application with correct values.

Figure 33. Keycloak service connection properties

Therefore there are two approaches to configuring the Keycloak connection:

1. **Hardcode the required variables into the configuration file**:
```
port: ${SERVER_PORT:8080}
```
   This method was used during the early stages of the project. However, it was insufficient and time-consuming, prone to typos, and most importantly, this method does not meet Cloud Native requirements. To address it, I found a workaround using CodeNOW tools (outlined in the next point).
2. **Utilize CodeNOW External services**. External Services are originally designed for connecting third-party applications into the CodeNOW stack. **It can store any custom variables** safely and in a centralized manner**.** Detailed instructions for creating and connecting it to components are available in the External Services guide[47].

To implement this approach, define a refined variable interface and then set them correctly. The result is shown in *Figure 34. Keycloak external service interface*.



ticket-reservation-dev-keycloak

All Environments

No description available.

ADMIN_PASSWORD	12345
ADMIN_USERNAME	demo-admin
CLIENT_ID	demo-app-api
CLIENT_SECRET	3051fc4e-5365-4d0a-bc02-f95cdda8fb62
REALM	demo
SERVER	https://sharedkeyclo2-keycloak.stxplayground.codenow.com/auth

Figure 34. Keycloak external service interface

CodeNOW requires creating a template for variables, that can be reused in the external services. The choice of variables used in this template will be discussed in detail in the next chapter (*3.3.4.6 Variables for Keycloak*). Here I provide some insights about the variable's values.

- Admin credentials (**ADMIN_PASSWORD**, **ADMIN_USERNAME**) refer to a user serving for connections to Keycloak API from a component (see *Admin Role*). The credentials are defined during the creation of an admin user through the Admin Console.
- **CLIENT_ID** is set during a client creation in the Admin Console. In our cases *demo-app-api*, *demo-app-bl*, and *demo-app-fe*.
- **CLIENT_SECRET** should correspond to a client the external service is created for. It is set only for a private (confidential) client.
- **REALM** references the realm name that contains the clients for the components. It must not be the Master realm.
- **SERVER** is a particularly interesting variable value. It links to the Admin Console via an external link (see *Service Setup Guide*). Nonetheless, it is possible to connect the CodeNOW components via the internal connection link:

  `http://managed-new-keyloak-http.dev:80`

  The external link was chosen as more dev-friendly and will be changed in the future.
  Then there is ***/auth*** element in the link. Old versions of Keycloak had ***/auth*** relative path added to the server base URL. It is very important to check if the running server uses endpoints with or without ***/auth***. To do so, in the Keycloak Admin Console, go to

  Realm settings → Endpoints → OpenID Endpoint Configuration

3.3.4.6 Variables for Keycloak

Spring Boot uses *.application or .yaml* files to instantiate certain classes. We may define custom variables there and then reuse them inside the configuration and throughout the whole application via *@Value* annotation.

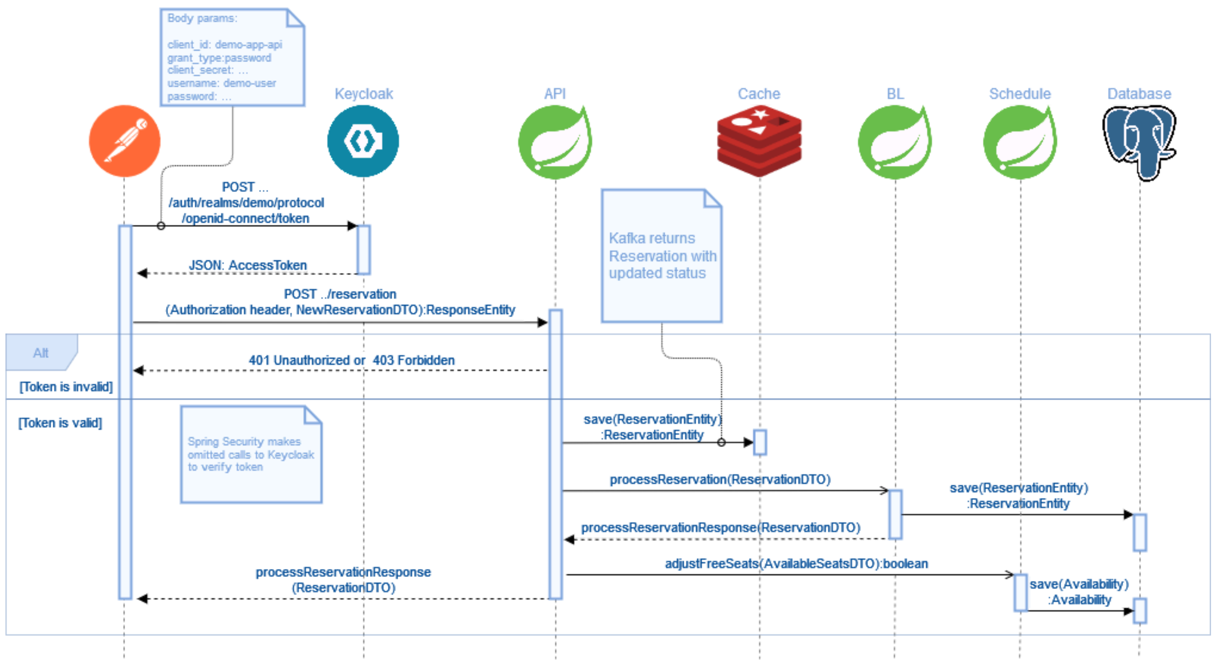
The integration with Keycloak may demand different data depending on the use case and the libraries used. For example, when a Keycloak client is private, it is necessary to define a client secret; otherwise, it can be omitted. To structurize the variables injection across the application and define the interface of External Service (see *Connected Service and Deployment Configuration Guide*), I separated Keycloak-related data into a standalone block in the *./codenow/config/application.yaml* documents and the CodeNOW Deployment Configuration.

- **server-url** - it is the main entry point for various libraries and their class manufacturers, e.g. Spring Security and its *ClientRegistrationRepository.* It looks like
  [application layer protocol - http or https]://[server address]:[port]/auth
- It is a highly reused string worthy of separating into the reusable variable.
- **realm** - also a highly reused string, particularly for the Keycloak Admin client.
- clientId - an important string for the OAuth2 setup.
- **client-secret** - the API and BL components are private, so this string is necessary for making requests to them.
- **admin-username**, **admin-password** - the current version of the application uses the PASSWORD grant type (see *2.2.3.2 Grant Types*) for the connection to the Admin Console, so it is necessary to provide the credentials of the user with sufficient rights to operate on various tasks. In the future the PASSWORD grant type will be changed for the CLIENT CREDENTIALS grant type and those variables will be omitted.
- **fe-url** - the URL used for redirection of the user after login or logout. Also, it is used by Keycloak to build a UI with links to the front-end components. It will be relevant in the later stages of the project.

3.4 First Iteration: Basic Endpoint Security

3.4.1 Securing API component

I started to secure the application from an API component, as its REST Controller is the interaction focus point between the application and the outer world. The first goal to achieve is to be able to filter the requests depending on the authentication status and roles, as shown in *Figure 35. Submit sequence diagram*.



Body params:
client_id: demo-app-api
grant_type:password
client_secret: ...
username: demo-user
password: ...
Keycloak
API
Cache
BL
Schedule
Database
POST ... /auth/realms/demo/protocol /openid-connect/token
JSON: AccessToken
POST ../reservation (Authorization header, NewReservationDTO):ResponseEntity
Kafka returns Reservation with updated status
Alt
401 Unauthorized or 403 Forbidden
[Token is invalid]
[Token is valid]
Spring Security makes omitted calls to Keycloak to verify token
save(ReservationEntity) :ReservationEntity
processReservation(ReservationDTO)
save(ReservationEntity) :ReservationEntity
processReservationResponse(ReservationDTO)
adjustFreeSeats(AvailableSeatsDTO):boolean
save(Availability) :Availability
processReservationResponse (ReservationDTO)

Figure 35. Submit sequence diagram

Prerequesits

- Account on CodeNOW
- IntelliJ IDEA (version 2020.3.2+)
- Set up the local development environment
    - Downloaded and configured app components (API, BL, optionally Schedules)
    - Configured Keycloak server
    - Configured Kafka
    - Configured Redis

3.4.1.1 Dependencies Guide

Initially Keycloak used Keycloak adapters for the integration with various application stacks. For Java applications, there was a Java Adapter. Now, all the adapters are deprecated and replaced with libraries specialized in OAuth2 and OpenID protocols[48].
The stage of the project, where the adapter was used is omitted.

Add the next dependencies to the *pom.xml* file:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
 <version>2.4.4</version>
</dependency>
```

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-oauth2-client</artifactId>
 <version>2.4.4</version>
</dependency>

<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
 <version>2.4.4</version>
</dependency>
```

The details about Spring Security are available in the *2.4.2.1 Spring Security* and in the Official Spring documentation[49]. From the implementation point of view, the *spring-boot-starter-security* dependency allows for securing REST endpoints. The most important interfaces it adds are *@EnableGlobalMethodSecurity* and *@EnableWebSecurity*. It will be discussed in more detail in the *3.4.1.3 SecurityConfig Class Guide*.

The *spring-boot-starter-oauth2-client* and the *pring-boot-starter-oauth2-resource-server* dependencies are related to OAuth2 and serve for establishing the communication with a security server (Keycloak in our case), as well as the exchange of access tokens (JWT tokens in our case). It is responsible for token validation support and checking its roles and scopes (see *2.2.1 Important Terminology*).
The *spring-boot-starter-oauth2-resource-server* will help in the validation of JWT tokens that come in requests from the front-end clients.
The *spring-boot-starter-oauth2-client* is used for obtaining access tokens. It enables the redirection to a login page. Generally, it allows the component to act as an OAuth2 client. It is relevant in the early stages of the project. Later this functionality will be delegated to the FE component.

Warning
Since this project uses Spring Boot, use the artifacts within the *spring-boot-starter* namespace. It helps to avoid dependency mess and allows smooth configuration.

Then, to use Admin CLI and connect to the Admin REST API, add the next dependency:

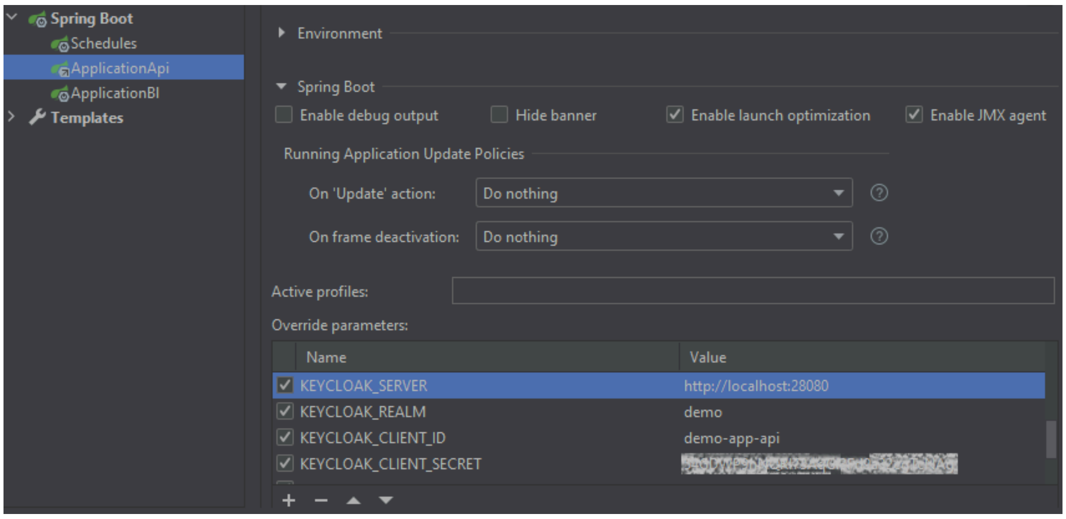
```
<dependency>
 <groupId>org.keycloak</groupId>
 <artifactId>keycloak-admin-client</artifactId>
 <version>13.0.0</version>
</dependency>
```

3.4.1.2 Configuration File Update Guide

To the *application.yaml* file in the *.codenow/config* repository, add the block of custom variables:

```
app:
 config:
   keycloak:
     server-url: ${KEYCLOAK_SERVER}
     realm: ${KEYCLOAK_REALM}
     clientId: ${KEYCLOAK_CLIENT_ID}
     client-secret: ${KEYCLOAK_CLIENT_SECRET}
     admin-username: ${KEYCLOAK_ADMIN_USERNAME}
     admin-password: ${KEYCLOAK_ADMIN_PASSWORD}
     fe-url: ${FE_URL}
```

Then, set the values for them. It is possible to hardcode them or to set them in the *Run/Debug Configuration* in IntelliJ IDEA as shown in *Figure 36. Run/Debug Configuration interface*.



Enable debug output ☐ Hide banner ☐ Enable launch optimization ✓ Enable JMX agent ✓

Running Application Update Policies

On 'Update' action: Do nothing

On frame deactivation: Do nothing

Active profiles:

Override parameters:

Name	Value
✓ KEYCLOAK_SERVER	http://localhost:28080
✓ KEYCLOAK_REALM	demo
✓ KEYCLOAK_CLIENT_ID	demo-app-api
✓ KEYCLOAK_CLIENT_SECRET	[redacted]

Figure 36. Run/Debug Configuration interface

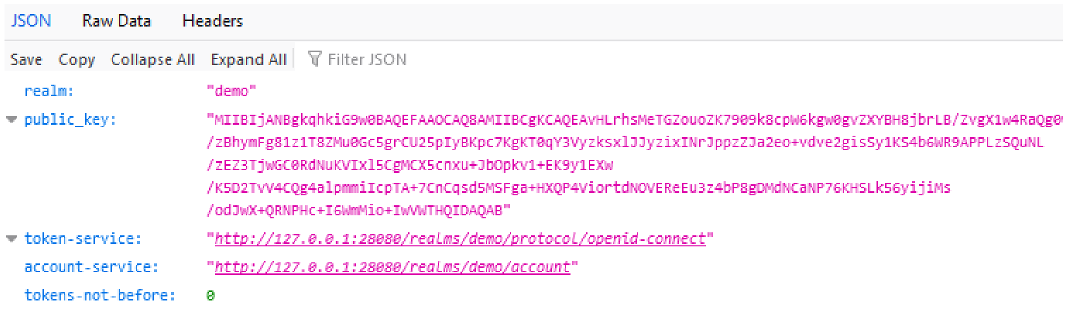
The values for the local host are:

- KEYCLOAK_SERVER: **http://localhost:28080/auth**
- KEYCLOAK_REALM: **demo**
- KEYCLOAK_CLIENT_ID: **demo-app-api**
- KEYCLOAK_CLIENT_SECRET: **[the secret generated by your Keycloak instance]**
- KEYCLOAK_ADMIN_USERNAME: **demo-admin**
- KEYCLOAK_ADMIN_PASSWORD:**12345**
- FE_URL: **http://localhost:3000**

Then it is time to configure OAuth2 providers. We must configure *resourceserver* and *client* under the *spring.security.oauth2*.

For the resource server, we need to specify:

- **issuer-uri** - the URL to the Keycloak realm, from which the parameters are fetched and autoconfigured by Spring Boot. It typically contains the metadata about the identity provider, like in *Figure 37. Issuer endpoint*.



```
realm:               "demo"
public_key:          "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvHLrhsMeTGZouoZK7909k8cpW6kgw0gvZXYBH8jbrLB/ZvgX1w4RaQg0
                     /zBhymFg81z1T8ZMu0Gc5grCU25pIyBKpc7KgKT0qY3VyzksxlJJyzixINrJppzZJa2eo+vdve2gisSy1KS4b6WR9APPLzSQuNL
                     /zEZ3TjwGC0RdNuKVIxl5CgMCX5cnxu+JbOpkv1+EK9y1EXw
                     /K5D2TvV4CQg4alpmmiIcpTA+7CnCqsd5MSFga+HXQP4ViortdNOVEReEu3z4bP8gDMdNCaNP76KHSLk56yijiMs
                     /odJwX+QRNPHc+I6WmMio+IwVWTHQIDAQAB"
token-service:       "http://127.0.0.1:28080/realms/demo/protocol/openid-connect"
account-service:     "http://127.0.0.1:28080/realms/demo/account"
tokens-not-before:   0
```

Figure 37. Issuer endpoint

- **jwk-set-uri** - the URL from which the OpenID certificates are fetched to retrieve the JSON Web Key Set for JWT verification[50].

For the client, we need to specify the data for its registration by the security provider (Keycloak in this case) and the specification of the security provider itself.

Fields under the *client.registration.[provider name]*:

- **client-id** - an identifier provided by the security provider server (e.g. *demo-app-api*)
- **client-secret** - a secret, typically generated by the security provider server for a specific client.
- **authorization-grant-type** - to retrieve both an access and an identity token, the *authorisation_code* grant (Authorization Code Grant flow, see *2.2.3.2 Grant Types*) type is typically used.

- **scope** - a scope requested for an access token. In this case, it is closely interwoven with the authorization grant type. To identify, that the client requested both authentication and an identity token, the OpenID Connect (see *2.2.4 OpenID Connect*) scope - *openid*, must be used.
- **provider** - a name of the identity provider to communicate with, specified under the *client.provider*, `the` "keycloak" in this case.

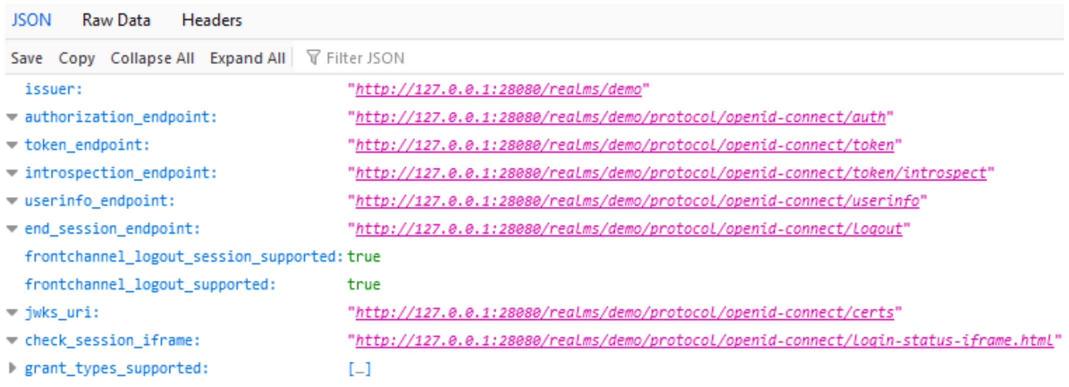
Fields under the *client.provider.[provider name]*:

- **issuer-uri** - same as for the resource server. The client checks the `iss` (issuer) claim in the token payload matches the expected issuer URI.
- **authorization-uri** - the endpoint where the initial step of the OAuth 2.0 authorization process takes place. It's the URI to which the client redirects the user for authentication and authorization.
- **user-name-attribute** - indicates the claim, that represents the user's preferred user identifier.

To specify the parameter values (such as specific URI and available scopes), visit the well-known endpoints, provided in the Keycloak Admin Console:

http://localhost/realms/demo/.well-known/openid-configuration

The illustration of this configuration in the Firefox browser is in the *Figure 38. Well-known endpoints.*



```
issuer:                               "http://127.0.0.1:28080/realms/demo"
authorization_endpoint:               "http://127.0.0.1:28080/realms/demo/protocol/openid-connect/auth"
token_endpoint:                       "http://127.0.0.1:28080/realms/demo/protocol/openid-connect/token"
introspection_endpoint:               "http://127.0.0.1:28080/realms/demo/protocol/openid-connect/token/introspect"
userinfo_endpoint:                    "http://127.0.0.1:28080/realms/demo/protocol/openid-connect/userinfo"
end_session_endpoint:                 "http://127.0.0.1:28080/realms/demo/protocol/openid-connect/logout"
frontchannel_logout_session_supported: true
frontchannel_logout_supported:        true
jwks_uri:                             "http://127.0.0.1:28080/realms/demo/protocol/openid-connect/certs"
check_session_iframe:                 "http://127.0.0.1:28080/realms/demo/protocol/openid-connect/login-status-iframe.html"
grant_types_supported:                […]
```

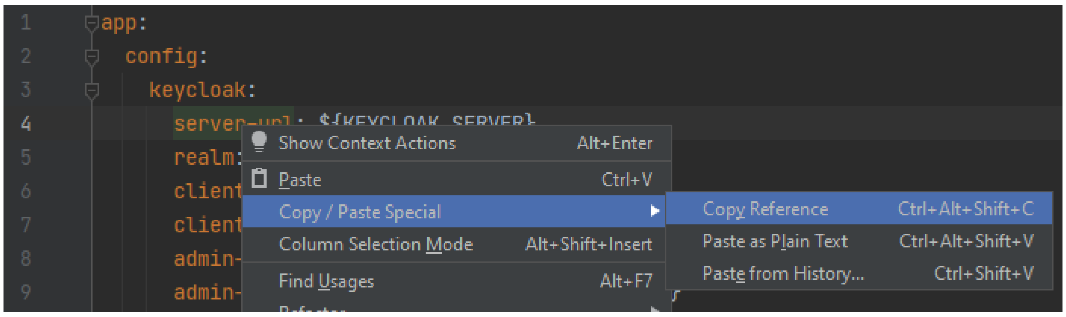
Figure 38. Well-known endpoints

The snipped of the configuration result:

```
spring:
...
 security:
   oauth2:
     resourceserver:
       jwt:
         issuer-uri:
${app.config.keycloak.server-url}/realms/${app.config.keycloak.realm}
         jwk-set-uri:
${spring.security.oauth2.resourceserver.jwt.issuer-uri}/protocol/openid-connect/cer
ts
     client:
       registration:
         keycloak:
           client-id: ${app.config.keycloak.clientId}
           client-secret: ${app.config.keycloak.client-secret}
           authorization-grant-type: authorization_code
           scope: openid
           provider: keycloak
#          redirect-uri: http://localhost:8082/reservation/authenticated
       provider:
         keycloak:
           issuer-uri: ${spring.security.oauth2.resourceserver.jwt.issuer-uri}
           authorization-uri:
${spring.security.oauth2.resourceserver.jwt.issuer-uri}/protocol/openid-connect/auth
          user-name-attribute: preferred_username
```

The commented-out *redirect-uri* parameter is used as a Callback to redirect an end user after successful authentication[51]. It should be registered inside the identity provider service (if the wildcard "*" is not used). It may require additional troubleshooting, as it may cause an infinite redirect loop.

In the configuration snippet, the pre-prepared variables (*see 3.3.4.6 Variables for Keycloakare)* are reused. To easily insert those variables with IDEA, click on one with the right mouse button and choose the *Copy Reference* option as shown in *Figure 39. Copy reference instruction* (or *Ctrl + Alt + Shift + C*).



```
app:
  config:
    keycloak:
      server-url: ${KEYCLOAK_SERVER}
      realm:
      client
      client
      admin-
      admin-
```

Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Copy Reference Ctrl+Alt+Shift+C
Paste as Plain Text Ctrl+Alt+Shift+V
Paste from History... Ctrl+Shift+V

Figure 39. Copy reference instruction

Note, that in the snipped, the ...***resourceserver.jwt.issuer-uri*** variable is recycled in *...resourceserver.jwt.jwk-set-uri, ...client.registration.provider.keycloak.issuer-uri* and *...client.registration.provider.keycloak.authorization-uri.*

CodeNOW Deployment Configuration values

As was discussed earlier, instead of the */codenow/config/application.yaml* file, CodeNOW uses separate configuration files. To set up variables in a centralized manner, the External Service is used (see *Connected Service and Deployment Configuration Guide*).

For CodeNOW to autocomplete External Service's variables, the placeholders must follow the pattern:

[connection name]_[variable name]

Thus, given that the connection name for the Keycloak External Service is *KEYCLOAK1*, the custom variables block in the CodeNOW configuration is:

```
app:
 config:
  keycloak:
   server-url: ${KEYCLOAK1_SERVER}
   realm: ${KEYCLOAK1_REALM}
   clientId: ${KEYCLOAK1_CLIENT_ID}
   client-secret: ${KEYCLOAK1_CLIENT_SECRET}
   admin-username: ${KEYCLOAK1_ADMIN_USERNAME}
   admin-password: ${KEYCLOAK1_ADMIN_PASSWORD}
```

application.yaml Troubleshooting

After the yaml configuration is set up, during the launch of a component an error (2) may occur:

```
...UnsatisfiedDependencyException: Error creating bean with name
'documentationPluginsBootstrapper' defined in URL
...Error creating bean with name 'clientRegistrationRepository'
...Factory method 'clientRegistrationRepository'...IllegalArgumentException:
Unable to resolve Configuration with the provided Issuer of
"http://localhost:28080/auth/realms/demo"
```

Error 2

It happens because the Keycloak server is not available. Make sure the Docker container is running, before launching the component. Also, such an error may occur because of a typo in some of the variable's values (e.g. "lcalhost" or "deno").

Make sure, that URI to the Keycloak server has the “http(s)” part. Otherwise, the “URI is not absolute” error (3) occurs:

```
...UnsatisfiedDependencyException: Error creating bean with name
'org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration$En
ableWebMvcConfiguration'
... Factory method 'clientRegistrationRepository' threw exception...: URI is
not absolute
```

Error 3

3.4.1.3 SecurityConfig Class Guide

The SecurityConfig class defines the general-level security config.

Definitions

To avoid the terminology mess in the future, I must set several definitions:

- General-level security - the security policies defined globally for the whole application. In this project, it is implemented by the *SecurityFilterChain* class.
- Method-level security - the security policies defined above a method individually. In this project implemented by *@PreAuthorize* annotation.
- (Enable Global) Method Security - refers to the *@EnableGlobalMethodSecurity* interface.

On the Internet, the general-level security is called global-level security, but I find it potentially confusing because of the *@EnableGlobalMethodSecurity* interface name.

Creating SecurityConfig Guide

Firstly, create a new folder named “security” In the *./src/main/java/com.codenow.ticket*.
Then create the *SecurityConfig* class. Then, add the following annotations:

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true,
jsr250Enabled = true, proxyTargetClass = true, mode = AdviceMode.PROXY)
@EnableWebSecurity
@RequiredArgsConstructor //optional
```

@Configuration annotation is used to signal to Spring Boot that the current class has configurations.
@EnableGlobalMethodSecurity turns on the support for various method-level annotations[52].

- prePostEnabled - the most important for this project. It allows *@PreAuthorize* and *@PostAuthorize* annotations.
- securedEnabled - it is used to specify a list of roles on a method.
- jsr250Enabled - it allows us to use the *@RoleAllowed* annotation
- proxyTargetClass - “*If true, class based proxying will be used instead of interface-based proxying. Defaults to "false"*” by Spring[53]. It was enabled in order to fix the Error 4 and Error 5 issues (see *SecurityConfig Class Troubleshooting*).
- mode - “*This attribute can be set to "aspectj" to specify that AspectJ should be used instead of the default Spring AOP. Secured methods must be woven with the AnnotationSecurityAspect from the spring-security-aspects module*.” by Spring[53].

@EnableWebSecurity annotation activates default global web security filters for the application and helps Spring to locate the *@Configuration* class responsible for the global web security configuration[57] - our *SecurityConfig* class in this case.

We can override the default security configuration enabled by *@EnableWebSecurity.* To do so, we will configure the *HttpSecurity* class[58].

```
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http){
        http.authorizeRequests(authorize -> authorize
                        .antMatchers(HttpMethod.POST,  "/reservation",
"/reservation/")
                            .permitAll()
                        .antMatchers("/actuator/*").permitAll()
                        .anyRequest().authenticated())
                .httpBasic();
        http.cors();
        http.csrf().disable();
               http.sessionManagement()
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        return http.build();
    }
```

The are two variants of configuration methods on the *HttpSecurity* class - *authorizeRequests* and *authorizeHttpRequests*. The *authorizeHttpRequests* is the improved version of *authorizeRequests*, and it is available for newer versions of Spring Boot and Spring Security. I had to stick with the 2 *2.4.4 Spring Boot version* given by existing architecture (*see 3.2.2 As-Is State: 3.2.2.1 As-Is Architecture*). Using the old version may result in troubles with security filter chains - see the *SecurityConfig Class Troubleshooting*.

Warning
The authorizeRequests method is outdated, use authorizeHttpRequests if possible.

In this method, we allow all kinds of requests for the *Create reservation endpoint* and actuator health-check endpoints. Requests to any other endpoint are required to be authenticated by default. The *httpBasic()* turns on the default HTTP Basic authentication, like the *Authorization* header with an access token[59].
The SCRF protection is deactivated for now for simplicity.
The *http.cors()* method adds CorsFilter, which will be crucial for making requests from the FE component in the future and fixing the CORS-related errors.
The app uses JWT tokens for authentication, to keep the application cloud-native, so the session policy is set to STATELESS (see *2.3 Cloud Native Applications*).

SecurityConfig Class Troubleshooting

During the Security methods configuration, a wide range of unexpected errors may occur. From Null Pointer exceptions, caused by our class not being instantiated (Error 4), to missing beans as in Error 5.

```
NullPointerException: null
    at
com.codenow.ticket.controller.rest.ReservationController.createReservation(Reservat
ionController.java:45) ~[classes/:na]
...
    at
org.springframework.security.web.context.SecurityContextPersistenceFilter.doFilter(
SecurityContextPersistenceFilter.java:80) ~[spring-security-web-5.4.5.jar:5.4.5]
```

Error 4

```
org.springframework.beans.factory.CannotLoadBeanClassException: Cannot find class
[org.springframework.security.access.intercept.aspectj.aspect.AnnotationSecurityAsp
ect] for bean with name 'annotationSecurityAspect$0'...
```

Error 5

In case such problems occur, the troubleshooting may look like this:

1. Add this dependency:

```
<dependency>
 <groupId>org.springframework.security</groupId>
 <artifactId>spring-security-aspects</artifactId>
</dependency>
```

2. Add this annotation to the Application class:

```
@EnableAspectJAutoProxy
```

In my case, those errors appeared, when I tried to use the *antMatchers()* method in my *SecurityConfig* class AND *@PreAuthorize* annotation at the same time.
If the methods above were not sufficient, there are more options for troubleshooting I came across:

3. Make sure that the classes annotated as *@Service* do implement an interface.
4. The trouble may hide in the order in which the security filters are applied. When using *antMatchers()*, it may apply the security rules early in the filter chain, before the filter that handles `@PreAuthorize` annotations, which causes unexpected behavior. It may be resolved by updating the Spring Boot and Spring Security versions. **Use *authorizeHttpRequests* instead of *authorizeRequests*.**

On CodeNOW, the next error may occur:

```
...BeanCreationException: Error creating bean with name
'healthEndpointGroupsBeanPostProcessor' defined in class path resource
[org/springframework/boot/actuate/autoconfigure/health/HealthEndpointConfiguration.
class]:
...
IllegalStateException: In the composition of all global method configuration, no
annotation support was actually activated
```

Error 6

It is caused by Spring Security forbidding any unauthorized request by default. Make sure to open an endpoint for health checks in the securityFilterChain method (actuator in this case).

3.4.1.4 Custom JWT Converter Guide

To convert roles from a JWT token issued by Keycloak into an Authority entity recognizable by Spring Security, the creation of a custom JWT converter is necessary. It also allows to define the prefix for the roles. By default, Spring Security requires the "ROLE_" prefix, which is then used in annotations like *@PreAuthorize* and *@PostAuthorize*. Thanks to this customization, the roles in those annotations may be simply “administrator” and “customer” instead of “*ROLE_administrator*” and “*ROLE_customer*”. The converter's implementation depends on the JWT token structure, so it is worth checking it and adapting it accordingly.

1. Create a new *JwtAuthConverter* implementing the Converter interface in the security folder and add pre-defined strings:

```
@Component
public class JwtAuthConverter implements Converter<Jwt,
AbstractAuthenticationToken> {
   private final JwtGrantedAuthoritiesConverter jwtGrantedAuthoritiesConverter =
                                            new JwtGrantedAuthoritiesConverter();

   @Value("${app.config.keycloak.clientId}")
   private String clientId;

   @Value("${spring.security.oauth2.client.provider.keycloak.user-name-attribute}")
   private String principalAttribute;
```

2. Add methods that will extract the realm-level roles and the client-level roles from a JWT token. The implementation in the snippets does not differentiate those levels and join them into one list.

```
private Collection<? extends GrantedAuthority> extractUserClientRoles(Jwt jwt){
       Map<String, Object> resourceAccess;
       Map<String, Object> resource;
       Collection<String> resourceRoles;

       Map<String, Object> realmAccess;
       Map<String, Object> realm;
       Collection<String> realmRoles;

       if((resourceAccess = jwt.getClaim("resource_access")) == null){
           return Set.of();}
       if((resource = (Map<String, Object>)resourceAccess.get(clientId)) == null){
           return Set.of();}
       resourceRoles = (Collection<String>) resource.get("roles");
       return resourceRoles.stream().map(role -> new SimpleGrantedAuthority
                                  ("ROLE_" + role)).collect(Collectors.toSet());
}

private Collection<? extends GrantedAuthority> extractUserRealmRoles(Jwt jwt){
       Map<String, Object> realmAccess;
       Collection<String> realmRoles;
       if((realmAccess = jwt.getClaim("realm_access")) == null){return Set.of();}
       if((realmRoles = (Collection<String>) realmAccess.get("roles")) == null){
           return Set.of();}
       return realmRoles.stream().map(role -> new SimpleGrantedAuthority("ROLE_" +
role)).collect(Collectors.toSet());
}
```

3. Add the method that combines client and realm levels extractors:

```
private Collection<? extends GrantedAuthority> extractAllRoles(Jwt jwt){
       Collection<? extends GrantedAuthority> clientRoles =
extractUserClientRoles(jwt);
       Collection<? extends GrantedAuthority> realmRoles =
extractUserRealmRoles(jwt);
       return Stream.concat(clientRoles.stream(),
realmRoles.stream()).collect(Collectors.toList());
}
```

4. Finally, add the implementation of the *convert()* method of the Converter interface:

```
@Override
public AbstractAuthenticationToken convert(@NonNull Jwt jwt) {
   Collection<GrantedAuthority> authorities = Stream.concat(
      jwtGrantedAuthoritiesConverter.convert(jwt).stream(),
      extractAllRoles(jwt).stream())
      .collect(Collectors.toSet());
   return new JwtAuthenticationToken
(jwt, authorities, getPrincipalClaimName(jwt));
}

private String getPrincipalClaimName(Jwt jwt){
String claimName = JwtClaimNames.SUB;
if(principalAttribute != null){claimName = principalAttribute;}
return jwt.getClaim(claimName);
}
```

The custom converter is ready. Then update the *SecurityConfig*, created in *3.4.1.3 SecurityConfig Class Guide*.

1. Add the bean to auto-wire:

```
private final JwtAuthConverter jwtAuthConverter;
```

2. Update the *securityFilterChain* method:

```
...
http
     .oauth2ResourceServer()
     .jwt()
     .jwtAuthenticationConverter(jwtAuthConverter);
...
```

3.4.1.5 REST Controller Secure Guide

Now we can secure endpoints on the method level with *@PreAuthorize*. For example, add the policy, that only a user roles with the administrator role can view and cancel reservations. Update the *Get all reservations endpoint and Cancel reservation endpoint:*

```
@PreAuthorize("hasRole('administrator')")
public ResponseEntity<?> getViewOfReservations(
@RequestHeader(required = false, value = "If-None-Match") String etag) {
   try {
       log.info("Get view of all reservations request was received.");
...
```

Or make sure, that only new users or users with the customer role can create a reservation:

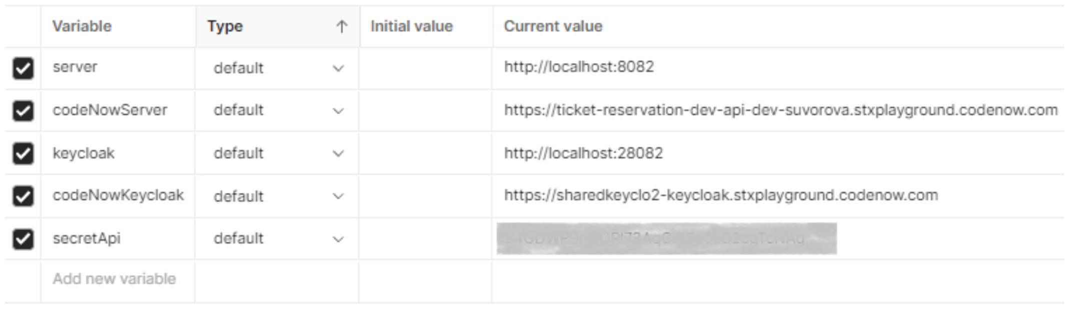
```
@PreAuthorize("(hasRole('customer') || anonymous)")
public ResponseEntity<ReservationDTO> createReservation(
@RequestBody NewReservationDTO newReservationDTO) {...}
```

3.4.2 Tests with Postman

This section illustrates the end point tests via Postman.

3.4.2.1 Postman Environment

To optimize the workflow with Postman, create Environment variables, as shown in *Figure 40. Postman variables example*. They will be utilized in request paths and POST bodies. From the illustration, it can be seen, that the tests can be applied on the locally deployed, as well as on CodeNOW deployed app.

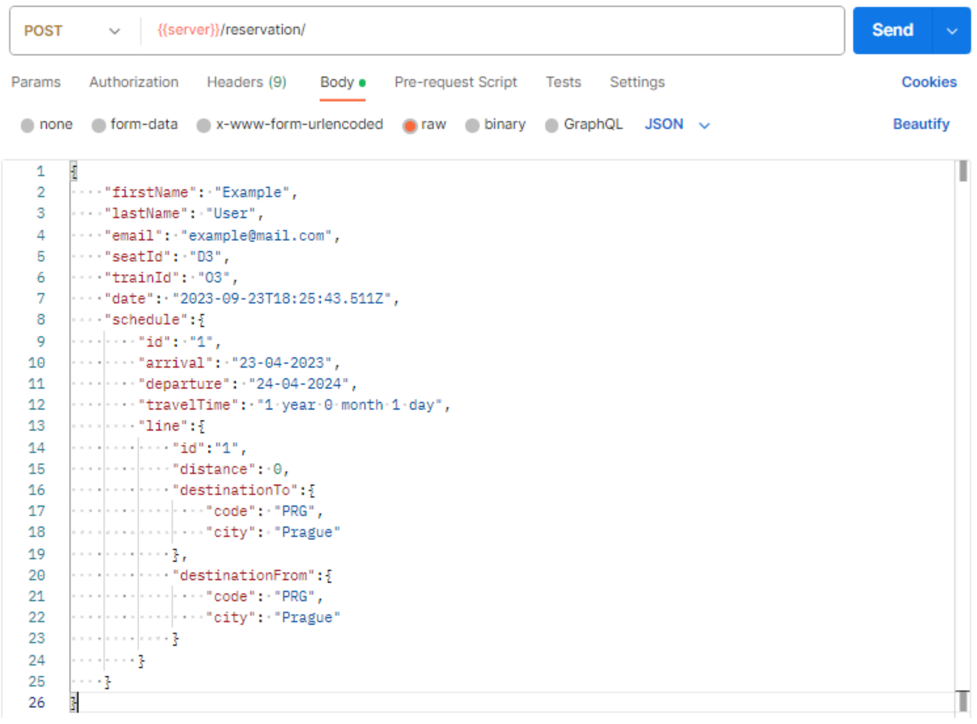


	Variable	Type	Initial value	Current value
✓	server	default		http://localhost:8082
✓	codeNowServer	default		https://ticket-reservation-dev-api-dev-suvorova.stxplayground.codenow.com
✓	keycloak	default		http://localhost:28082
✓	codeNowKeycloak	default		https://sharedkeyclo2-keycloak.stxplayground.codenow.com
✓	secretApi	default		[redacted]
	Add new variable			

Figure 40. Postman variables example

3.4.2.2 Create Reservation Endpoint Test

Send a POST request to the endpoint for reservation creation as shown in *Figure 41. Reservation endpoint request*. It should return a JSON object with the reservation that has the status ACTIVE. The reservation should appear in the PostgreSQL and the Cache. Attention, the path for the old version of Keycloak should have "*/auth*" in the path.



```
{
    "firstName": "Example",
    "lastName": "User",
    "email": "example@mail.com",
    "seatId": "D3",
    "trainId": "03",
    "date": "2023-09-23T18:25:43.511Z",
    "schedule":{
        "id": "1",
        "arrival": "23-04-2023",
        "departure": "24-04-2024",
        "travelTime": "1 year 0 month 1 day",
        "line":{
            "id":"1",
            "distance": 0,
            "destinationTo":{
                "code": "PRG",
                "city": "Prague"
            },
            "destinationFrom":{
                "code": "PRG",
                "city": "Prague"
            }
        }
    }
}
```

Figure 41. Reservation endpoint request

In the future, the respond JSON object will have a "*customer_id*" field.

3.4.2.3 UserController Tests

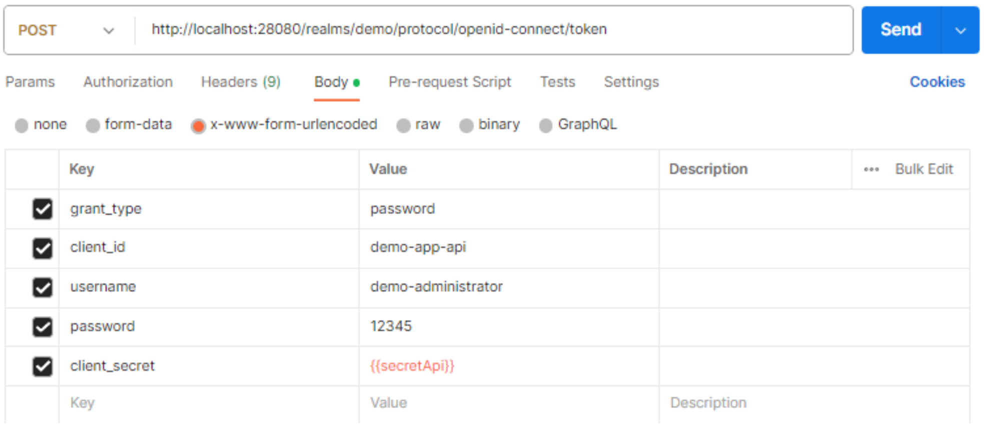
To test if role-based policies work correctly, create a UserController class.

```
@RestController
@RequestMapping("/users")
public class UserControllerImpl {
...
   @Autowired
   public UserControllerImpl(UserService userService){this.userService =
userService;}

   @GetMapping("/admin")
   @PreAuthorize("hasRole('administrator')")
   public String adminInfo(@AuthenticationPrincipal Jwt jwt) {
       return String.format("Hello, administrator %s! Your id is %s.",
                                       jwt.getClaimAsString("preferred_username"),
                                        jwt.getSubject());
   }

   @PostMapping("/customer")
   @PreAuthorize("hasRole('customer')")
   public String customerInfo(){return "Hello, customer!";
   }
}
```

Retrieve the administrator user token as shown in *Figure 42. Administrator token endpoint*.



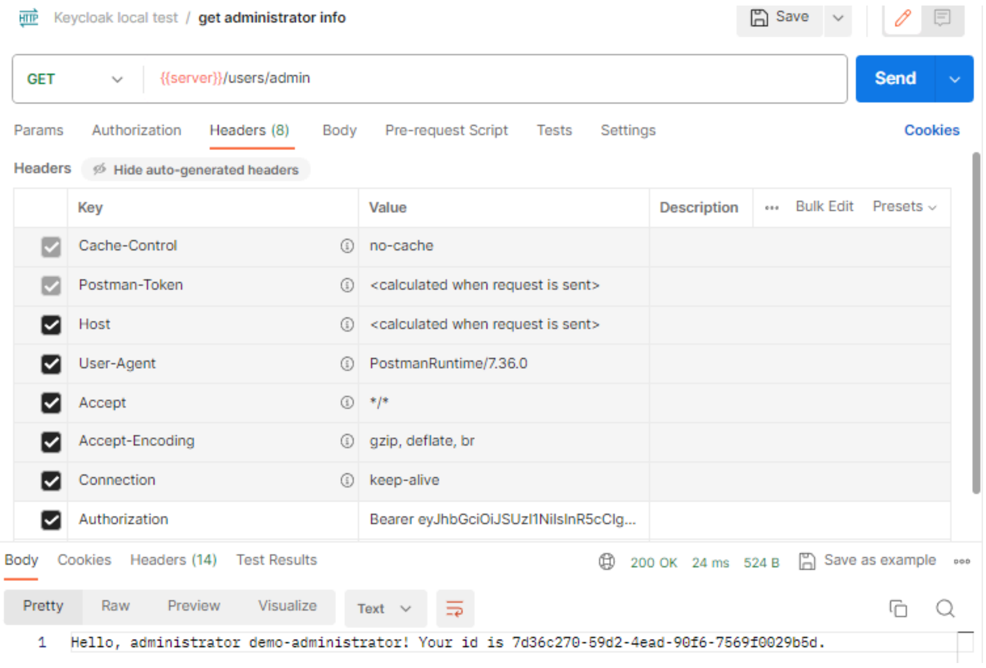
POST http://localhost:28080/realms/demo/protocol/openid-connect/token

Body: x-www-form-urlencoded

	Key	Value	Description
✓	grant_type	password	
✓	client_id	demo-app-api	
✓	username	demo-administrator	
✓	password	12345	
✓	client_secret	{{secretApi}}	

Figure 42. Administrator token endpoint

A similar request for an access token would be for a customer user just with different credentials.
Put the access token in the Authorization header. An example of a Postman request for an administrator endpoint is in *Figure 43. Administrator endpoint test result*.



Keycloak local test / get administrator info

GET {{server}}/users/admin

Headers (8)

	Key	Value	Description
✓	Cache-Control	no-cache	
✓	Postman-Token	<calculated when request is sent>	
✓	Host	<calculated when request is sent>	
✓	User-Agent	PostmanRuntime/7.36.0	
✓	Accept	*/*	
✓	Accept-Encoding	gzip, deflate, br	
✓	Connection	keep-alive	
✓	Authorization	Bearer eyJhbGciOiJSUzI1NiIsInR5cClg...	

200 OK 24 ms 524 B

```
Hello, administrator demo-administrator! Your id is 7d36c270-59d2-4ead-90f6-7569f0029b5d.
```

Figure 43. Administrator endpoint test result

3.5 Second Iteration: Creating Customers

3.5.1 Admin Console API

To create new users inside the Keycloak from within the demo app, we need to use Admin Console API. The necessary dependency for it is described in the *3.4.1.1 Dependencies Guide*.

3.5.1.1 Keycloak Client Guide

Update the *SecurityConfig* class (see *3.4.1.3 SecurityConfig Class Guide*) by adding a *@Bean* that instantiates the Keycloak object.

1. To structurize the Keycloak instance creation, use pre-defined strings (see *3.4.1.2 Configuration File Update: application.yaml Update Guide*)

```
@Value("${app.config.keycloak.server-url}")
private String authServerUrl;

@Value("${app.config.keycloak.realm}")
private String authRealm;

@Value("${app.config.keycloak.clientId}")
private String authClientId;

@Value("${app.config.keycloak.client-secret}")
private String clientSecret;

@Value("${app.config.keycloak.admin-username}")
private String adminUsername;

@Value("${app.config.keycloak.admin-password}")
private String adminPassword;
```

2. Add Keycloak builder. This bean will allow to creation of a Keycloak instance to connect to the Admin Console.

```
@Bean
Keycloak keycloak() {
    return KeycloakBuilder.builder()
            .serverUrl(authServerUrl)
            .realm(authRealm)
            .clientId(authClientId)
            .clientSecret(clientSecret)
            .grantType(OAuth2Constants.PASSWORD)
            .username(adminUsername)
            .password(adminPassword)
            .resteasyClient(new ResteasyClientBuilder()
                .connectionPoolSize(10)
                .build())
            .build();
}
```

The grant type can be changed to the CLIENT CREDENTIALS. It implies the usage of service accounts[60] on Keycloak clients.

3.5.2 Customer and Reservation Bind

Before implementation of the user creation, it is necessary to define a binding mechanism between the user account and its reservations. It is a crucial point, as a user account is primarily created during a reservation creation by an unauthorized user.

There were two approaches that I have tried: the conservative approach when changes to the existing service interfaces and data models are kept as minimal as possible. For example, instead of creating a new field in the existing table in PostgreSQL, using a new linking table that maps customer ID and reservation ID.

bind(<u>customer_id, reservation_id</u>)

FK: reservation_id ⊆ ticket_reservation(id)

After a consultation, I received a confirmation, that it is acceptable to change the data model and interfaces. So the *ReservationEntity, NewReservationDTO*, and *ReservationDTO* classes on API and BL components were given a new field - *String customerId*. For now, the *customerId* is an auto-generated ID by Keycloak. The linking table capturing *reservationId* and *customerId* relation became obsolete and the existing "*ticket_reservations*" table has a new field - *customer_id* of varchar type.

3.5.2.1 Database Table Update

Prerequisites

- running PostgreSQL container
- PgAdmin

To get a view of the PostgreSQL container connect to it through PgAdmin. Register a new server as shown in *Figure 44. PgAdmin interface 1*.

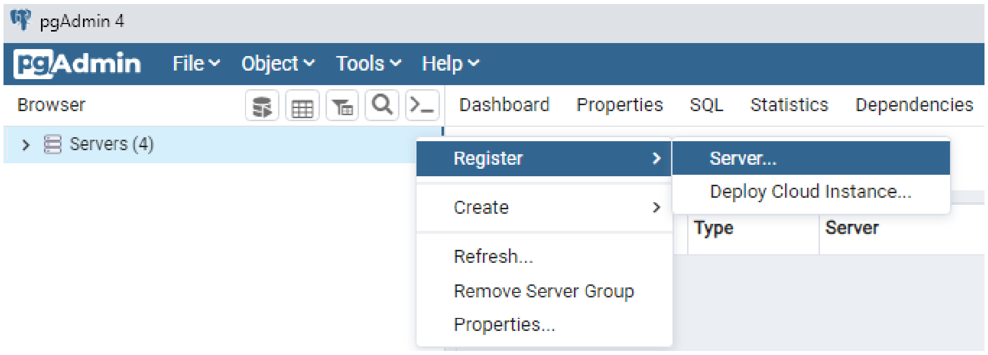
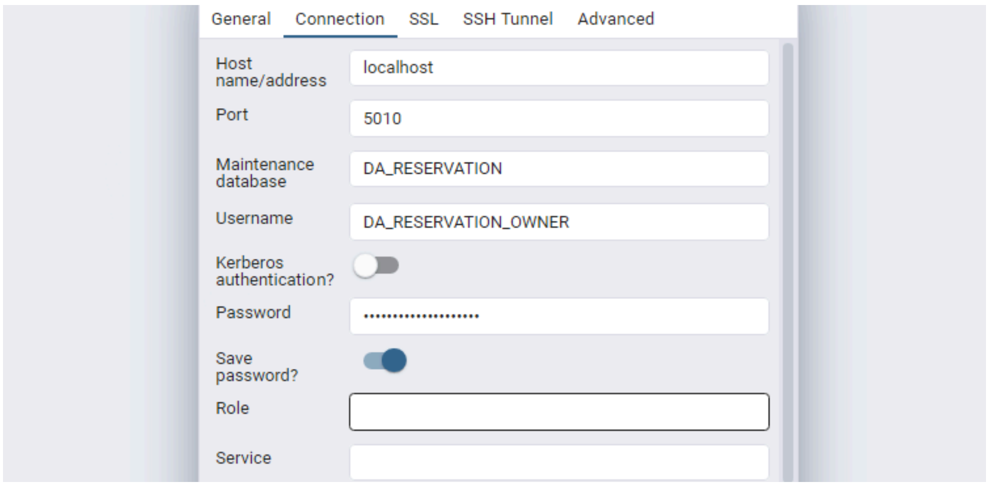


Figure 44. PgAdmin interface 1

The values for the server registration are available in the docker-compose file (see *3.3.3 3rd-party applications setup: Docker-compose*). The example is in *Figure 45. PgAdmin interface 2*.



Host name/address	localhost
Port	5010
Maintenance database	DA_RESERVATION
Username	DA_RESERVATION_OWNER
Kerberos authentication?	
Password	••••••••••••••••••••
Save password?	
Role	
Service	

Figure 45. PgAdmin interface 2

From the BpAdmin you can access the reservations table and add the new field - *customerId* and track the created reservations.
Alternatively, to do it from within the Idea IDE, create a new Database Connection from the menu on the bar on the right of the interface (see *Figure 46. Idea database connections location*)

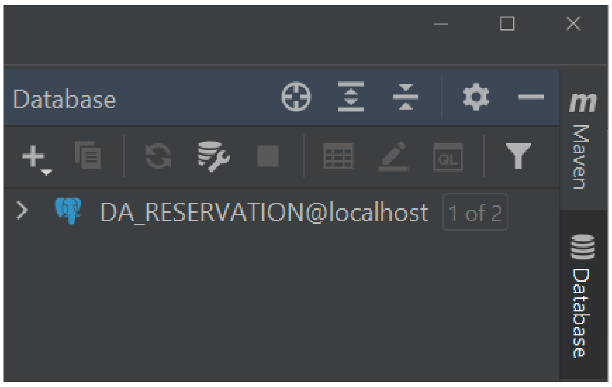
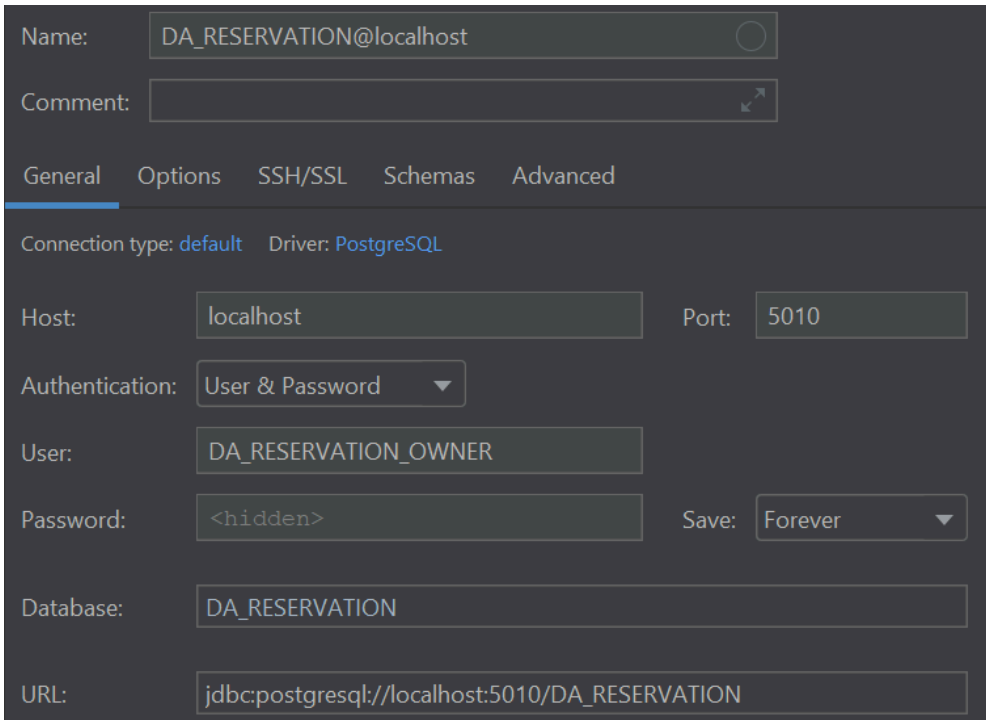


Figure 46. Idea database connections location

The example of the connection configuration is in *Figure 47. Idea database connection example*. The important part is the URL of the database. It depends on the database and the driver for it. In this case, the connection string is

```
jdbc:postgresql://localhost:5010/DA_RESERVATION
```

The setup of a Database connection additionally helps to identify errors in Hibernate and other persistence-related annotations.



Name: DA_RESERVATION@localhost
Comment:
Connection type: default   Driver: PostgreSQL
Host: localhost   Port: 5010
Authentication: User & Password
User: DA_RESERVATION_OWNER
Password: <hidden>   Save: Forever
Database: DA_RESERVATION
URL: jdbc:postgresql://localhost:5010/DA_RESERVATION

Figure 47. Idea database connection example

3.5.3 User Creation

In this project, the responsibility for keeping the users' data is assigned to the Keycloak. In the early stages, the implementation uses a user ID generated by Keycloak. In the later stages, the app will operate with a custom ID set in the user attributes (the reasons are in the *3.10 Seventh Iteration: The Custome ID Assignment Change*). The ID can be obtained from a Keycloak during the user creation from the HTTP response and a JWT token issued after a user's authentication. The Keycloak Client uses the ID to obtain *UserResource* - the interface that provides user management-related methods[61]. The *UserResource* also instantiates *UserRepresentation* - a user DTO[62]. Attention, *UserResorce*, and *Users**s**Resource* are different interfaces.

3.5.3.1 User Service Guide

The functionality for user management is going to be in the Service layer of the application in the *UserService* class.

1. Create the interface for the *UserService*:

```
public interface UserService{
   UserResource createUserFromReservation(NewReservationDTO reservation);
   UserResource getUserResourceById(String id
   List<UserRepresentation> getUsersByUsername(String username);
   List<UserRepresentation> getUsersByEmail(String email);
   UserResource getUserResourceByEmail(String email)
   String getIdByUsername(String id);
   String getIdByEmail(String email);
```

2. Create the class implementing the interface. Annotate it with *@Service*.
3. Add the pre-defined strings (see *3.3.4.6 Variables for Keycloak*):

```
@Value("${app.config.keycloak.realm}")
private String REALM;

@Value("${app.config.keycloak.clientId}")
private String CLIENT_ID;

@Value("${app.config.keycloak.fe-url}")
private String FE_URL;
```

   The client ID may be useful for creating a client-specific changes for account, like settings the client-related roles. The *FE_URL* may be useful in the future to set a redirect URI in various places.

4. Autovire the Keycloak instance:

```
private final Keycloak keycloak;
```

5. Implement the supporting methods.

```
public UserResource getUserResourceById(String id){
   return keycloak.realm(REALM).users().get(id);}

public List<UserRepresentation> getUsersByUsername(String username) {
   List<UserRepresentation> users = keycloak.realm(REALM)
                                    .users().search(username, true);
   return users;}

public List<UserRepresentation> getUsersByEmail(String email) {
   List<UserRepresentation> users = keycloak.realm(REALM)
           .users().search(null, null, null, email.toLowerCase(Locale.ROOT),
null, null);
   return users;}
```

```
public UserResource getUserResourceByEmail(String email){
   List<UserRepresentation> urs = getUsersByEmail(email);
   if(urs.isEmpty()){return null;}
   return getUserResourceById(urs.get(0).getId());}

public String getIdByUsername(String username){
   List<UserRepresentation> users = getUsersByUsername(username);
   if (users.isEmpty()) return null;
   return users.get(0).getId();}

public String getIdByEmail(String email){
   List<UserRepresentation> users = getUsersByByEmail(email);
   if (users.isEmpty()) return null;
   return users.get(0).getId();}
```

This implementation is tested with the Admin client of the 13.0.0 version. In the future versions of the Keycloak Client, the interface will be improved and enriched by searhByAttributes, *searchByEmail*, *searhByFirstName*, *searchByLastName*, and *searchByUsername* methods.

Warning
In case a user does not exist, Keycloak Client (version ≤ 21.1.2) get(@PathParam("id") String var1) does **not** return null or throws an error.

6. Add the implementation of the *createUserFromReservation()* method. The accounts are going to be created based on the email address from a reservation. I have discovered, that the roles can be added only after the user exists in the system. Also, there is no way to change or set a user ID through the Keycloak client (or from the Admin Console), which will greatly affect the development of the application in the future.

```
@Override
public UserResource createUserFromReservation(
                                     NewReservationDTO reservation){

   UserRepresentation ur = new UserRepresentation();
   String role = KeycloakRole.customer.name();
//Setting general data
   ur.setId(reservation.getCustomerId());
   ur.setUsername(generateUsername(role));
   ur.setEmail(reservation.getEmail());
   ur.setFirstName(reservation.getFirstName());
   ur.setLastName(reservation.getLastName());
   ur.setEnabled(true);

   Response response = keycloak.realm(REALM).users().create(ur);
   if(response.getStatus() == 201){
       String userId = getUserIdFromLocationHeader(
                            response.getHeaderString("Location"));
       UserResource user = getUserResourceById(userId);
       addCustomerRoleToUser(user);
       return user;
    }
    else{
       log.warn("ERROR CREATING USER {}. RESPONSE STATUS {}",
                                          ur.getEmail(),
                                          response.getStatus());
    return null;
    }
 }
```

Note, that the user's password is not set. The user's account setup will be implemented in future iterations (see *3.5.6 User account setup.5.6 User account setup*).

The private methods used in the *createUserFromReservation()* :

```
private String generateUsername(String role){
   UUID randomUUID = UUID.randomUUID();
   return role.substring(0, 3)
             .toUpperCase(Locale.ROOT)+"_"+randomUUID.toString()
                                         .replaceAll("_", "");}

private void addCustomerRoleToUser(UserResource resource){
   String role = KeycloakRole.customer.name();
   RoleRepresentation customerRole = keycloak.realm(REALM)
                                    .roles()
                                    .get(role)
                                    .toRepresentation();
resource.roles().realmLevel().add(Collections.singletonList(customerRole));
   resource.update(resource.toRepresentation());
}

private String getUserIdFromLocationHeader(String location){
   return location.substring(location.length() - 36);}
```

3.5.3.2 Reservation Controller Update Guide

Now, when we have the backbone for the user creation, implemented in the *UserService* class, we may implement the trigger for the user creation in the API Controller. The goal is to create a user account only if a given email does not exist in the Keycloak database.
The final version of the *createReservation* endpoint at this stage uses *@AuthenticationPrincipal* annotation to automatically retrieve a Principal from an Authorization header of a request and interact with the *SecurityContextHolder*.

```
@PostMapping
@PreAuthorize("(hasRole('customer') || anonymous)")
public ResponseEntity<ReservationDTO> createReservation(
                                 @RequestBody NewReservationDTO newReservationDTO,
                                 @AuthenticationPrincipal Jwt jwt)
                                 throws InterruptedException, ExecutionException {
   if(jwt != null){ newReservationDTO.setCustomerId(jwt.getSubject());}
   else{
       List<UserRepresentation> users = userService.getUsersByEmail
                                         (newReservationDTO.getEmail())
                                    ;
       if(users.isEmpty()){
           UserResource user = userService
                               .createUserFromReservation(newReservationDTO);
           newReservationDTO.setCustomerId(user.toRepresentation().getId());
       }
       else{newReservationDTO.setCustomerId(users.get(0).getId());}
   }
   ReservationDTO reservationDTO = reservationService
                                 .processReservation(newReservationDTO);
   return new ResponseEntity<>(reservationDTO, HttpStatus.OK);}
```

3.5.4 Reservations Views

This section describes the addition of new functionality - retrieving reservations belonging to a useron BL component and the accompanying changesthroughout on the back end.

3.5.4.1 Obtain Customer Reservations with @Post Filter Guide

There are several methods to make the new endpoint for getting customers' reservations on the API REST Controller to work. This implementation uses the existing method to get all reservations *reservationService.getLastNReservations(50)*first and then filters them with the *@PostFilter* annotation.

```
@PreAuthorize("hasRole('customer')")
@PostFilter("filterObject.customerId == authentication.principal.getId")
@GetMapping("/my-reservations")
public List<ReservationDTO> getViewOfCustomerReservations() throws
ExecutionException, InterruptedException {
   List<ReservationDTO> reservationDTOList =
reservationService.getLastNReservations(50);
   return  reservationDTOList;}
```

This implementation does not make changes to the existing repository interface, but it is not the most optimal, as database SELECT queries are more suitable for this use case.

3.5.4.2 Obtain Customer Reservations with Repository Interface Guide

Spring provides a *CrudRepository* interface to automatically generate database queries[63].
It is enough to add the next method definition to the ReservationRepository interface:

```
List<ReservationEntity> findByCustomerId(String customerId);
```

This work uses this approach.

3.5.4.3 Views on Service Layer Guide

After a background for retrieving the customer's reservations is ready, it is time to implement the service method for obtaining the customer's reservations.

```
public List<ReservationDTO> getCustomerReservations(String customerId){
   List<ReservationDTO> reservationDTOS = new ArrayList<>();
   List<ReservationEntity> reservationEntities = reservationRepository
                                                  .findByCustomerId(customerId);
   for (ReservationEntity entity : reservationEntities){
       reservationDTOS.add(reservationEntityToReservationDTO(entity));}
   reservationDTOS.sort(Comparator.comparing(ReservationDTO::getDate).reversed());
   return reservationDTOS;}
```

3.5.4.4 Views on Repository Layer Guide

Method getViewOfReservations

The old implementation for getting reservations looks like this:

```
@GetMapping
public ResponseEntity<?> getViewOfReservations(@RequestHeader(required = false,
value = "If-None-Match") String etag) {
   try {
       if(etag != null){
          if (reservationService.checkSameEtag(etag)){
              return ResponseEntity
              .status(HttpStatus.NOT_MODIFIED).eTag(etag).build();}
           List<ReservationDTO> reservationDTOList = reservationService
                                                   .getLastNReservations(50);
            return ResponseEntity.ok()
                                .eTag(reservationService.getDBEtag())
                                .body(reservationDTOList);
       }else {
          List<ReservationDTO> reservationDTOList = reservationService
                                                 .getLastNReservations(50);
          String dbEtag = reservationService.getDBEtag();
          if (dbEtag.isEmpty()) reservationViewProducer.sendReservationViewReq(50);
              return ResponseEntity.ok().eTag(dbEtag).body(reservationDTOList);
          }
   }catch (Exception e){return new
ResponseEntity<>(HttpStatus.EXPECTATION_FAILED);}
}
```

There is a little bug in the “else” branch. When the method discovers, that the API cache does not have any reservation entries, it makes a request to the BL component for the last 50 reservation views. Then it, however, returns a response with an empty reservations list.
There is the updated and fixed version of the method:

```
@GetMapping
@PreAuthorize("hasRole('administrator')")
public ResponseEntity<?> getViewOfReservations(
              @RequestHeader(required = false, value = "If-None-Match") String etag)
       {
   try {
      ...
       }
       else {
          String dbEtag = reservationService.getDBEtag();
          if (dbEtag.isEmpty()){
              log.warn("Empty cache, sending view request to BL...");
              reservationAllViewProducer.sendReservationViewReq();
          }
          List<ReservationDTO> reservationDTOList = reservationService
                                             .getAllReservations();
          return ResponseEntity.ok()
                                .eTag(reservationService.getDBEtag())
                                .body(reservationDTOList);
       }
   }
}
```

In the highlighted if statement, the request is sent and then the method waits for the Kafka response. Therefore in this case the method is supposed to return a list with reservations, even though it would take a bit longer.
Also, it is important to notice, that the initial method retrieves **only the last 50 reservations submitted to the system**. It can cause serious problems for the customer's reservations use case, so Kafka topics have been changed - those changes will be covered in the *3.5.5 Kafka Update Guide* section.

Method getViewOfCustomerReservations

The method for retrieving a customer's reservations follows the same structure as the *getViewOfReservations* method. However, it uses a new Kafka Producer - *reservationAllViewProducer*, the new *getCustomerReservations* method, and utilizes the *@AuthenticationPrinciple* annotation.

```
@PreAuthorize("hasRole('customer')")
@GetMapping("/customer-reservations")
public ResponseEntity<?> getViewOfCustomerReservations(
                                        @AuthenticationPrincipal Jwt jwt,
                                        @RequestHeader(required = false, value =
                                        "If-None-Match") String etag){
   if(etag != null) {
       if (reservationService.checkSameEtag(etag)) {
           log.info("Data was not modified.");
           return
ResponseEntity.status(HttpStatus.NOT_MODIFIED).eTag(etag).build();
       }
       log.info("Etag exists, but data were modified. The request etag is {}",
etag);
       List<ReservationDTO> reservationDTOList = reservationService
                                .getCustomerReservations(jwt.getSubject());
       return ResponseEntity.ok()
                         .eTag(reservationService.getDBEtag())
                         .body(reservationDTOList);
   }
   else {
       String dbEtag = reservationService.getDBEtag();
       if (dbEtag.isEmpty()) {
           log.warn("Empty cache, sending customer view request to BL...");
           reservationAllViewProducer.sendReservationViewReq();
       }
       List<ReservationDTO> reservationDTOList = reservationService
                                .getCustomerReservations(jwt.getSubject());
       return ResponseEntity.ok()
                         .eTag(reservationService.getDBEtag())
                         .body(reservationDTOList);
   }
}
```

3.5.5 Kafka Update Guide

Now, when getting reservations corresponding to a user account, it is not enough to load last 50 random reservations from the BL to API cache, as it was implemented in *getViewOfReservations()* method. It would lead to situations when a user does not obtain all their reservations because it was displaced by new reservations from other users.

To avoid it, the API cache will be filled with all disposable reservations from BL. It would require new Kafka Consumers and Producers both on API and BL components.

3.5.5.1 New Kafka Topic

First of all it is necessary to register a new Kafka topic for retrieving all reservations.
In the *docker-compose* file update, the Kafka service to the Kafka service adds *reservation-view-all* topic:

```
...
KAFKA_CREATE_TOPICS:
new-reservation:1:1,...,reservation-view-all:1:1,reservation-view-response:1:1
...
```

And in the *./codenow/config/application.yaml* file update the Kafka topics (both on API and BL):

```
spring.kafka.topic:
reservation-view-all: reservation-view-all
...
```

Notice, that there is no new response topic, as it is not necessary and the existing response topic and the listener can be reused by both producers - *reservation-view* and *reservation-view-all*.

3.5.5.2 Kafka Producer

On the API component, there are already implemented *producerFactory* and *kafkaTemplate* beans, configuring the producer's serialization and connections.

Also, there is already a Kafka *ReservationViewProducer*, to get the last X amount of reservations (in initial implementation used for the last **50** reservations):

```
...
private final KafkaTemplate<String, Integer> newReservationViewSender;
private final String newReservationViewTopic;

public ReservationViewProducer(
                    KafkaTemplate<String, Integer> newReservationViewSender,
                    @Value("${spring.kafka.topic.reservation-view}") String
                    newReservationViewTopic) {...}

public boolean sendReservationViewReq(Integer numberOfReservations){
   SendResult<String, Integer> sendResult = newReservationViewSender
                                  .send(newReservationViewTopic,
                           numberOfReservations)
                                  .get();
   log.info("Reservation view of {} entries request sent to
BL",numberOfReservations);
   log.info(sendResult.toString());
   return true;
}
```

To get all available reservations, add the new *@Component* Kafka producer - *ReservationAllViewProducer*. It is going to utilize the new *reservation-view-all* topic.

```
@Component
public class ReservationAllViewProducer {
   private static final Logger log = LoggerFactory
                                    .getLogger(ReservationAllViewProducer.class);
   private final KafkaTemplate<String, Integer> reservationAllViewSender;
   private final String reservationAllViewTopic;

   public ReservationAllViewProducer(
                           KafkaTemplate<String, Integer> ReservationViewSender,
                           @Value("${spring.kafka.topic.reservation-view-all}")
                           String reservationAllViewTopic) {...}

   public boolean sendReservationViewReq(){
       ListenableFuture<SendResult<String, Integer>> future =
reservationAllViewSender
                                                .send(reservationAllViewTopic,
                                        null);
       future.addCallback(new ListenableFutureCallback<>() {
          @Override
          public void onFailure(Throwable e) {log.error(...);}
          @Override
          public void onSuccess(SendResult<String, Integer> result)
{log.info(...);}
       });
       try {
          future.get(3, TimeUnit.SECONDS);// Wait for result 3 seconds
          return true;
       } catch (InterruptedException | ExecutionException | TimeoutException e) {
          log.error(...);
          return false;}
   }}
```

This implementation uses *ListenableFuture<T> extends Future<T>* interface to return usable boolean value. It lays the foundation for future app development in better handling non-standard scenarios.

3.5.5.3 Kafka Consumer

We need to create the corresponding consumer class on the BL component. It will trigger the new function on the *ReservationService* that will return all available reservations instead of the last X. In the future, this method will be restricted to a certain period.

```
@Service
public class ReservationAllViewConsumer {
   private static final Logger log =
LoggerFactory.getLogger(ReservationAllViewConsumer.class);
   private final ReservationService reservationService;

   public ReservationAllViewConsumer(ReservationService reservationService) {
       this.reservationService = reservationService;
   }
   @KafkaListener(topics =
"${spring.kafka.reservation.topic.reservation-view-all}", containerFactory =
"reservationViewKafkaListenerContainerFactory")
   public void reservationListener(Acknowledgment ack) {
       reservationService.getAllReservations();
       ack.acknowledge();}
}
```

3.5.5.4 ReservationService Update on BL component

User Service will use the Reservation Repository to obtain the reservations. The mechanism is the same as described in the *3.5.4.2 Obtain Customer Reservations with Repository Interface Guide*.

In the *ReservationRepository* interface, add

```
List<ReservationEntity> findAll();
```

Spring will automatically generate the database query.

The new service method to get all reservations, used in the Kafka listener:

```
@Override
public void getAllReservations(){
   List<ReservationEntity> reservationEntities = reservationRepository.findAll();
...
   reservationViewProducer.reservationViewResponse
                                            (new ReservationViewDTO(reservationDTOS));
}
```

This method uses the same view producer, as the original *getLastNReservations* method.

3.5.6 User account setup

Following the creation of a reservation and a new account, a user would naturally desire to log in to the Reservation Application. To access their account, the user must provide credentials, and obtaining these credentials requires the prior setup of an account.
The details about the chosen flow are in the *3.2.3.2 User Workflow Study* section.
The function for the user creation (see *3.5.3.1 User Service Guide*) autogenerates a username but does not provide any credentials. It allows a user to set up an account using the “Forgot Password” link, which is fairly easy to add to a standard Keycloak Login page. For it to appear, it is enough to toggle the *Forgot Password* option in the realm settings in the Login tab.

It is not suitable for the official account setup flow. The standard account setup flow would have the following steps:

1. Obtain an email with a link to set up the account
2. Follow the link within a certain period of time before it expires (24 hours for example)
3. Follow the profile setup steps, defined by the Keycloak required actions:
    a. Update profile (username, first name, etc.)
    b. Set password
4. Login with own credentials

Fortunately, Keycloak Admin Client provides some interface for such functionality, namely sending mails and required actions. The required actions are: “... one-time actions that a user must perform before they are logged in”[65]

3.5.6.1 UserService Update for Email Sending

It is possible to set up required actions during the account creation, but it does not fit our use case. We will make users perform account setup through email.

There are 4 standard required actions that Keycloak provides[65]:

- **Update Password** - The user must change their password.
- **Configure OTP** - The user must configure a one-time password generator on their mobile device using either the Free OTP or Google Authenticator application.

- **Verify Email** - The user must verify their email account. An email will be sent to the user with a validation link that they must click. Once this workflow is successfully completed, the user will be allowed to log in.
- **Update Profile -** The user must update profile information, such as name, address, email, and phone number.

We are interested in the **Update Password** action and **Update Profile** actions. The **Verify Email** action is implicitly built into our flow.

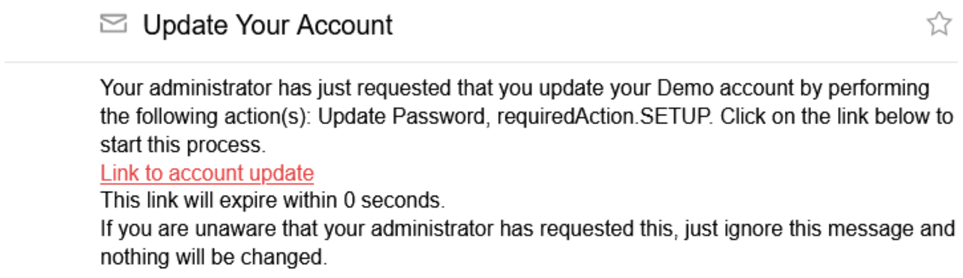
The Admin client has the following email-related API in the *UserRecource* class[*61*]:

- *void executeActionsEmail(*
  @QueryParam("client_id") String var1,
  @QueryParam("redirect_uri") String var2,
  @QueryParam("lifespan") Integer var3,
  List<String> var4)
- *void sendVerifyEmail()*
- *void sendVerifyEmail(@QueryParam("client_id") String var1)*

The *executeActionEmail* provides the most flexible interface and allows to make a user to perform several actions at once. Add the following method to the *UserService* class:

```
public void sendCredentialsSetupMail(UserResource ur){
    log.info("SENDING SETUP MAIL REQUEST TO KEYCLOAK...");
    ur.executeActionsEmail(CLIENT_ID,
                           FE_URL,
                           86400,
                           Arrays.asList("UPDATE_PROFILE",
                          "UPDATE_PASSWORD"));
    log.info("REQUEST FOR SENDING SETUP MAIL WAS SENT");
}
```

This method utilezes pre-defined variables (see *3.3.4.6 Variables for Keycloak, 3.5.3.1 User Service Guide*). The *FE_URL* - the address of the front-end component, is needed for Keycloak to incorporate the “Back to application” link to the UI.
The *86400* refers to the lifespan of the link - 24 hours in seconds.
There is a list of default required actions, but it can be expanded using custom SPI. A couple of words about Keycloak SPI are in the *3.8.2 SPI* section.
Nonetheless, providing an action unknown to Keycloak will not cause any critical error, but Email templates and UI will have problems parsing and rendering it. You can see the example of an email with an unknown action in *Figure 48. Email with custom action*.


Update Your Account

Your administrator has just requested that you update your Demo account by performing the following action(s): Update Password, requiredAction.SETUP. Click on the link below to start this process.
Link to account update
This link will expire within 0 seconds.
If you are unaware that your administrator has requested this, just ignore this message and nothing will be changed.

Figure 48. Email with custom action

Now it is time to update the *createUserFromReservation* method. After the user is successfully created, the email sender should be triggered:

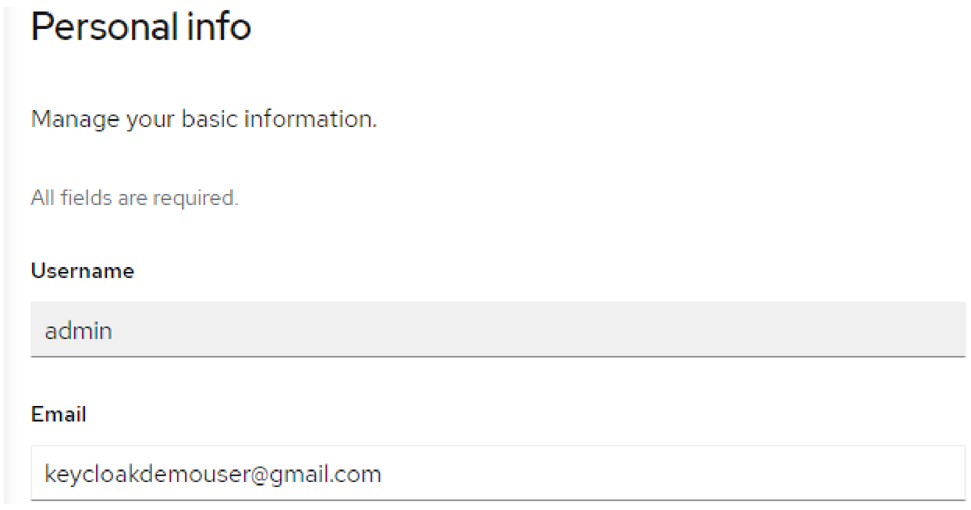
```
...
Response response = keycloak.realm(REALM).users().create(ur);
if(response.getStatus() == 201){
...
   UserResource user = getUserResourceById(userId);
   addCustomerRoleToUser(user);
   sendCredentialsSetupMail(user);
   return user;
}
...
```

3.5.6.2 Keycloak Email Setup

For emails to be actually sent it is necessary to set up SMTP server. Google is a good starting point for an email sender. To set up a Google account to provide SMTP functionality, follow the tutorial about creating an app password[64]. Attention, The Google flow has been changed recently (2022-2023), so the majority of tutorials describing the Keycloak email setup with the Google email server are invalid.

After a Google account is set up and an **app password** is obtained, it is possible to proceed to the Keycloak Email Setup.

1. Setup mail for the Master Realm Admin account. In the upper-right corner click on the admin's username and click on Manage account. It will take you to the Keycloak account management console. Go to Personal Info and set the mail that will be used for sending mail as shown in *Figure 49. Account management interface*.


Personal info

Manage your basic information.

All fields are required.

Username

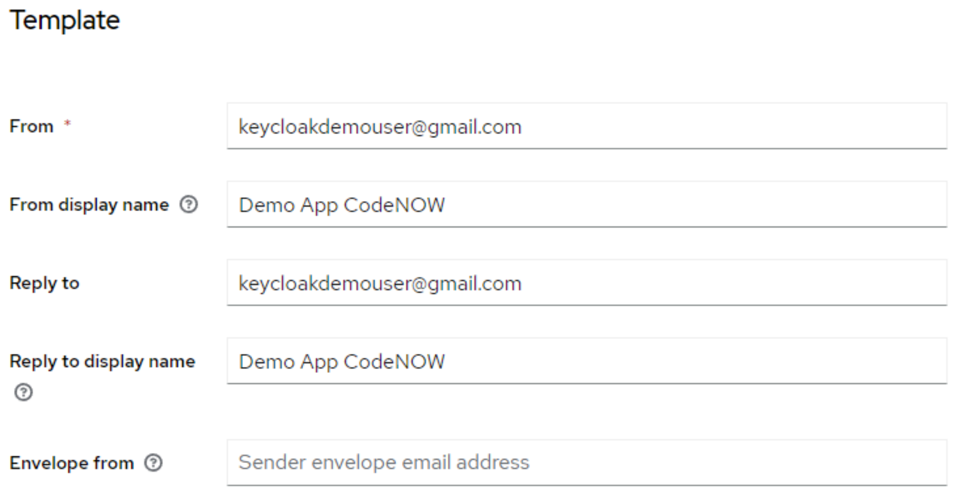
admin

Email

keycloakdemouser@gmail.com

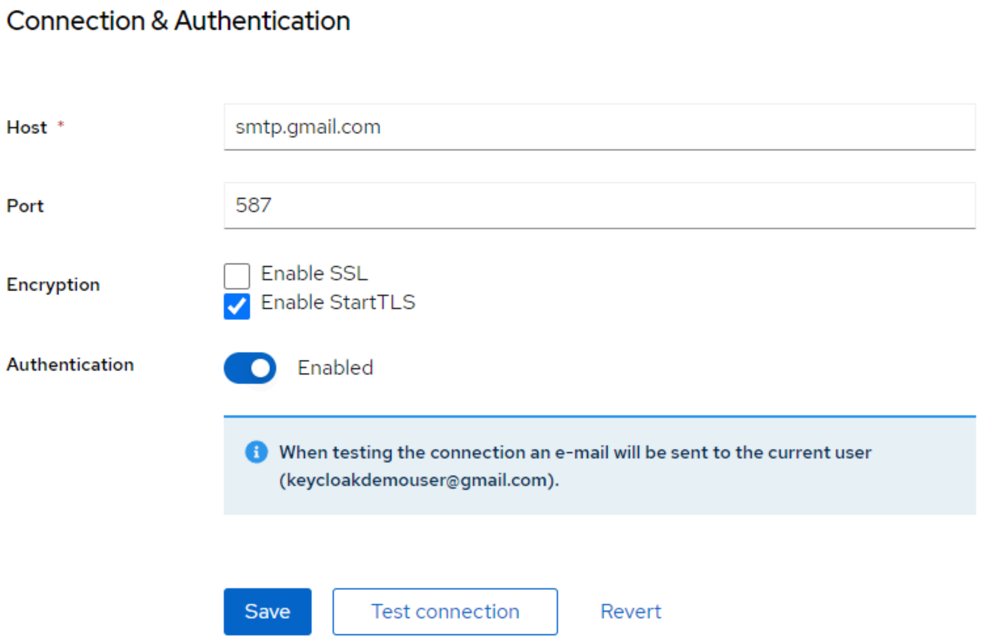
Figure 49. Account management interface

2. Fill up the Email form tab in the Realm Settings as shown in *Figure 50. Email settings 1 Template* and *Figure 51. Email settings 2 Connection&Authentication*.


Template

From *	keycloakdemouser@gmail.com
From display name	Demo App CodeNOW
Reply to	keycloakdemouser@gmail.com
Reply to display name	Demo App CodeNOW
Envelope from	Sender envelope email address

Figure 50. Email settings 1 Template


Connection & Authentication

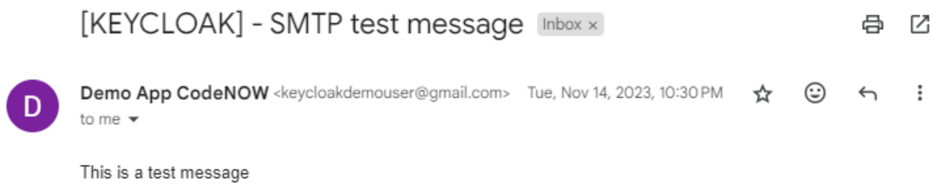
Host *	smtp.gmail.com
Port	587
Encryption	☐ Enable SSL ☑ Enable StartTLS
Authentication	Enabled

When testing the connection an e-mail will be sent to the current user (keycloakdemouser@gmail.com).

Figure 51. Email settings 2 Connection&Authentication

When you toggle the Authentication, it shows the form for the Google account username and the app password that was retrieved in the first step.

3. Test the connection. On the provided email address should arrive an email with configurated values like in *Figure 52. Email test message*.

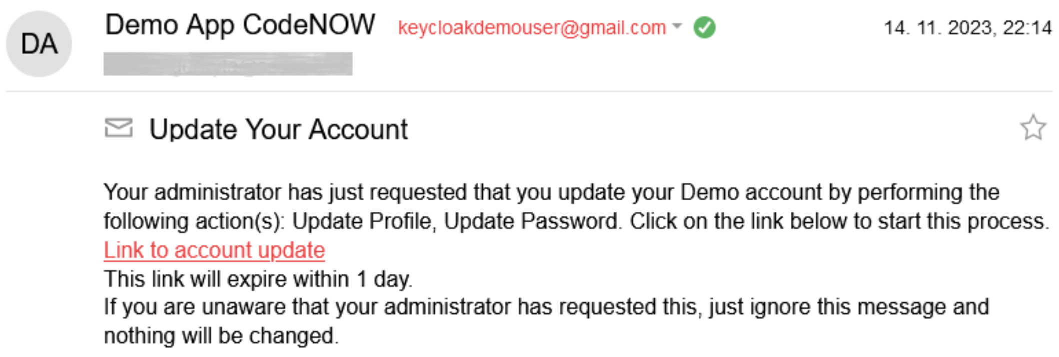

[KEYCLOAK] - SMTP test message Inbox ×

Demo App CodeNOW <keycloakdemouser@gmail.com> Tue, Nov 14, 2023, 10:30 PM
to me

This is a test message

Figure 52. Email test message

After everything is set, a reservation with a new email will trigger an account creation. During the account creation an email to set up the new account will be sent to the mail address in the reservation, the example is in *Figure 53. Example of a default executeAction email*.


DA

Demo App CodeNOW keycloakdemouser@gmail.com 14. 11. 2023, 22:14

[redacted]

Update Your Account

Your administrator has just requested that you update your Demo account by performing the following action(s): Update Profile, Update Password. Click on the link below to start this process.
Link to account update
This link will expire within 1 day.
If you are unaware that your administrator has requested this, just ignore this message and nothing will be changed.

Figure 53. Example of a default executeAction email

The email text and look customization is described in *3.8 Fifth Iteration: Custom Email Template* section.

3.5.7 Email Tests

3.5.7.1 Kafka Update for Tests

If the emails are sent from the BL component (see *3.6 Third Iteration: Architecture change)* it is necessary to send a request to it through the API endpoint.
To test the mail sending via Postman, I created a new endpoint on the API gateway. It uses a new Kafka topic *setup-mail* to send a mail request to BL. BL sends the request to Keycloak via the Admin Client. To do that, the corresponding topics on API and BL were registered, and Consumer and Producer were created with the corresponding *mailDTO*.

3.5.7.2 UserService and UserController Email Test

This section provides the simple basis for sending test emails with Postman.
In the *UserService* class add the next method:

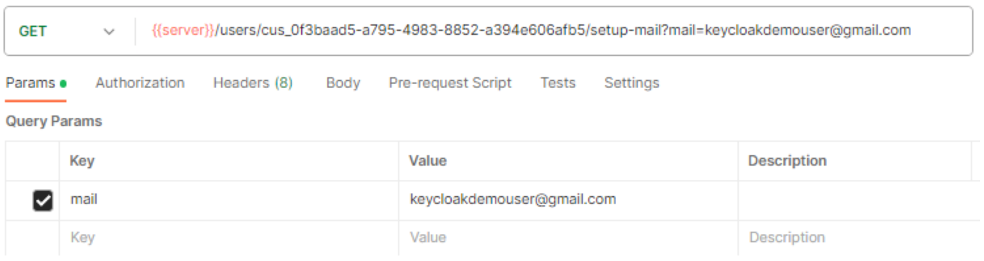
```
@Override
public void sendSetupMail(NewMailDTO mail){
    newMailProducer.sendNewMail(mail);}
```

Then for the *UserController* class add endpoint:

```
@GetMapping("/{username}/setup-mail")
public String sendSetupMail(@PathVariable String username, String mail){
   userService.sendSetupMail(new NewMailDTO(mail, username));
   return "Sent an email to BL (rest)";
}
```

3.5.7.3 Postman Email Test

After the basis for the testing is done, retrieve an access token for any existing user and send an email.


GET {{server}}/users/cus_0f3baad5-a795-4983-8852-a394e606afb5/setup-mail?mail=keycloakdemouser@gmail.com

Query Params

	Key	Value	Description
✓	mail	keycloakdemouser@gmail.com	
	Key	Value	Description

Figure 54. Postman email test request

3.6 Third Iteration: Architecture change

Progressively, the current “draft” of the application started to look odd, particularly the responsibility of components in the overall architecture. The API component gains too many functionalities - besides being a Security Gateway it is now also a “user service” component (see *Figure 55. Current component responsibility diagram*).

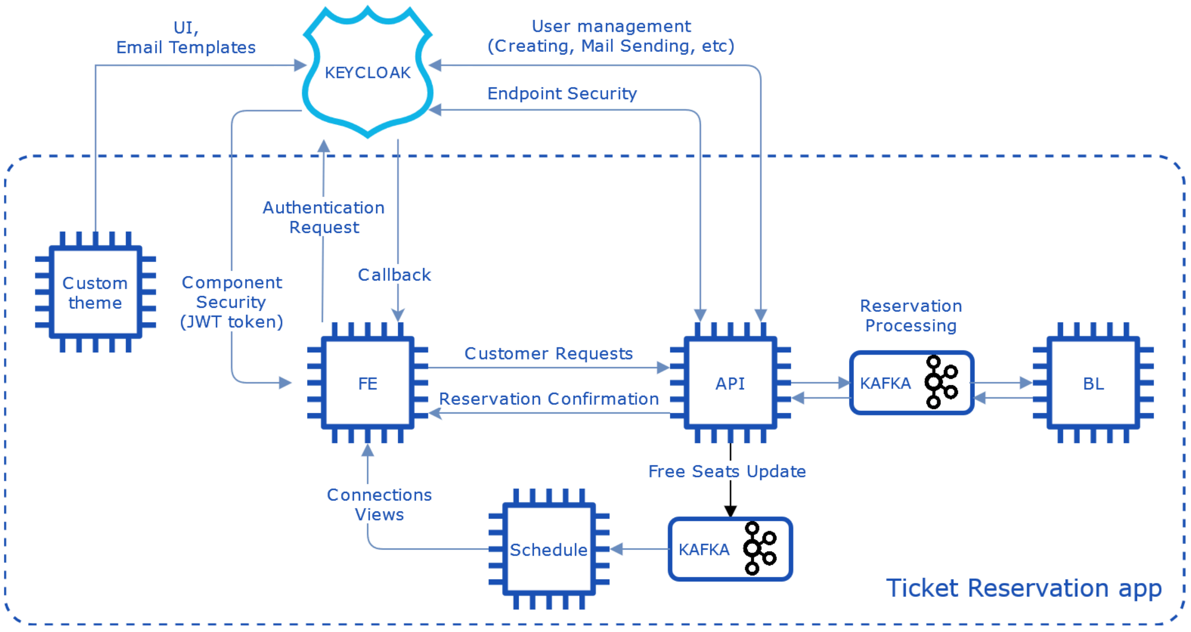
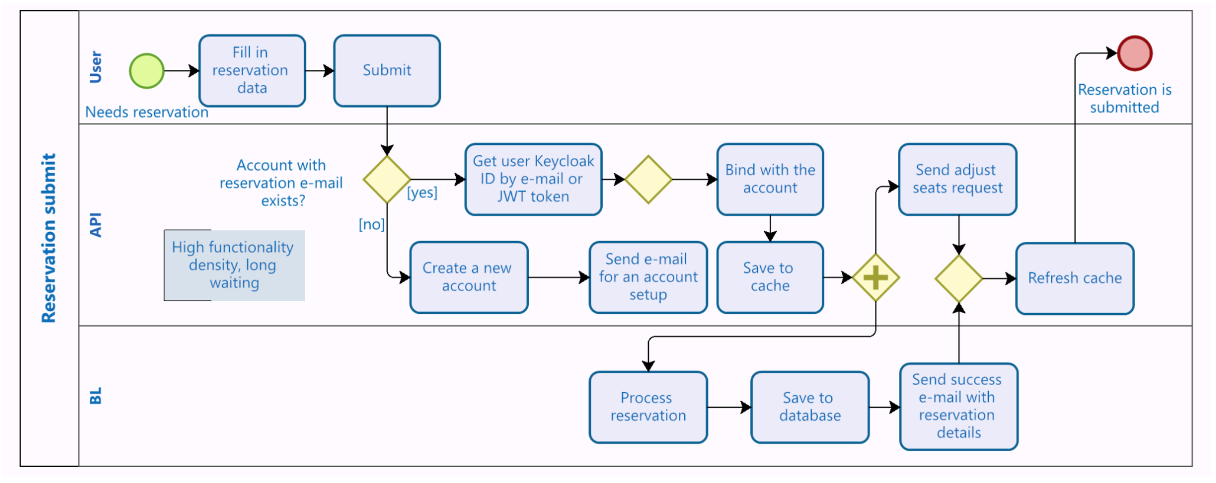


Figure 55. Current component responsibility diagram

The reservation processing and account creation look like in *Figure 56. Current reservation submit, detailed activity diagram*.



Reservation submit
User
API
BL
Needs reservation
Fill in reservation data
Submit
Reservation is submitted
Account with reservation e-mail exists?
[yes]
[no]
High functionality density, long waiting
Get user Keycloak ID by e-mail or JWT token
Bind with the account
Send adjust seats request
Create a new account
Send e-mail for an account setup
Save to cache
Refresh cache
Process reservation
Save to database
Send success e-mail with reservation details

Figure 56. Current reservation submit, detailed activity diagram

It seemed to me, that the API gateway functionality should remain as pure as possible - restricted to the endpoint access and the security policies reinforcement (as shown in *Figure 57. New component responsibility diagram*).

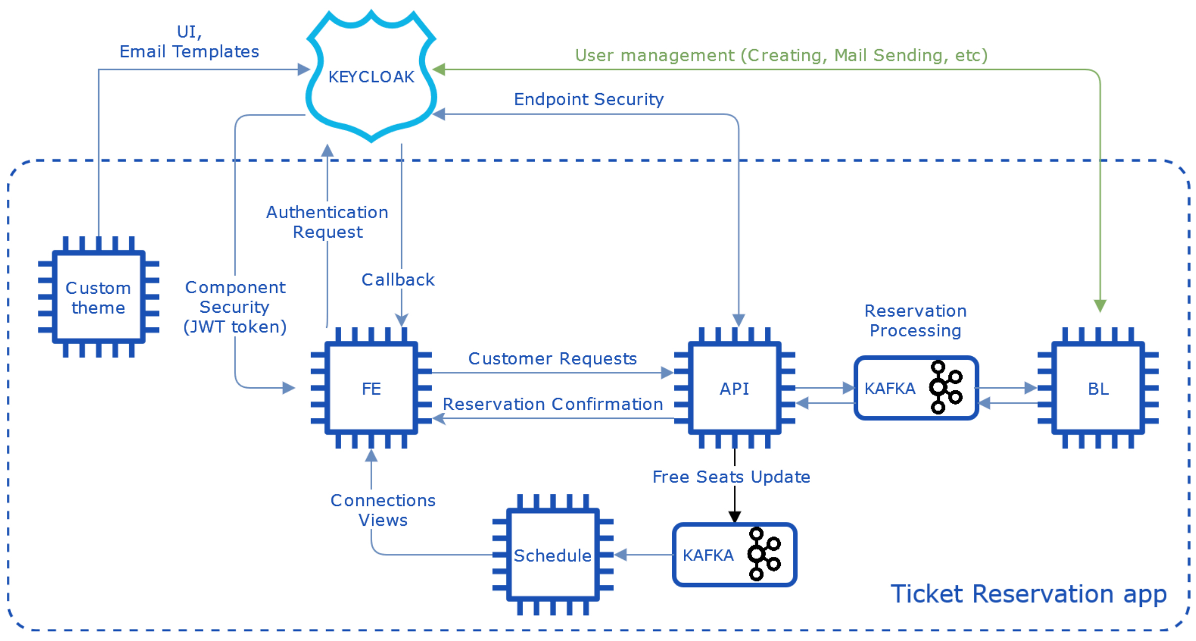


Figure 57. New component responsibility diagram

The functionality of a Use Service is encapsulated in the BL component now and API is left to deal only with the REST security. The reservation flow in the new architecture, therefore, would look like in *Figure 58. Reservation submin flow in the new architecture*.

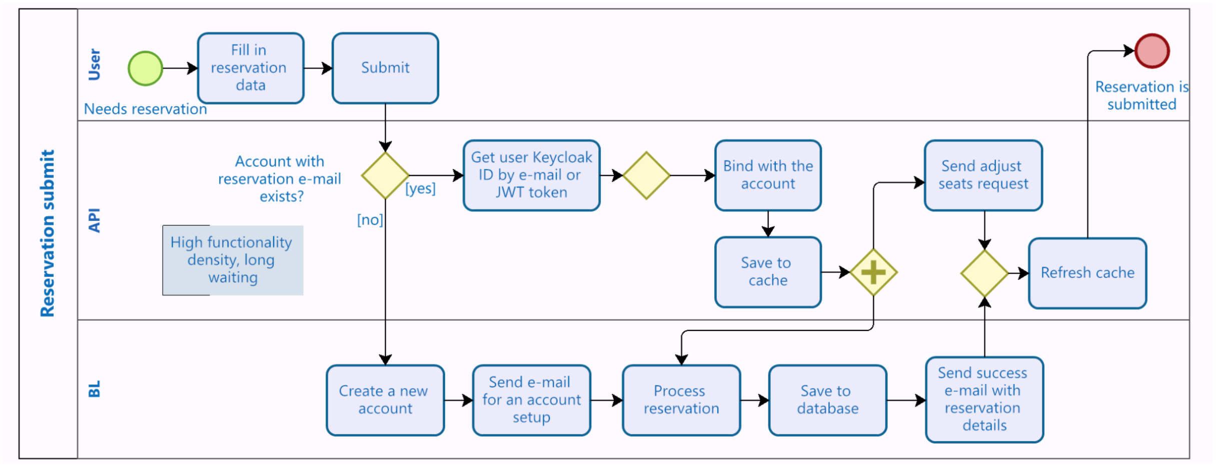

Reservation submit
User
Needs reservation
Fill in reservation data
Submit
Reservation is submitted
API
Account with reservation e-mail exists?
[yes]
[no]
Get user Keycloak ID by e-mail or JWT token
Bind with the account
Save to cache
Send adjust seats request
Refresh cache
High functionality density, long waiting
BL
Create a new account
Send e-mail for an account setup
Process reservation
Save to database
Send success e-mail with reservation details

Figure 58. Reservation submin flow in the new architecture

Therefore, to move user management-related functionality, the Keycloak Admin Client should be integrated with the BL component. At the moment this change looks like a logical architecture improvement step, however, in the future, it will create unexpected challenges with user ID assignments (see *3.9.8 Long Reservation View Response Bug*).

3.6.1 BL Component Update

Some steps from the previous iterations should be reapplied for the API component, mainly related to the Keycloak Admin Client (see *3.5.1.1 Keycloak Client Guide*).

1. Add the Keycloak Admin Client dependency

```
<dependency>
 <groupId>org.keycloak</groupId>
 <artifactId>keycloak-admin-client</artifactId>
 <version>21.1.2</version>
</dependency>
```

2. Update the *./codenow/config/application.yaml* file by adding the variables to reuse:

```
app:
 config:
   keycloak:
     server-url: ${KEYCLOAK_SERVER}
     realm: ${KEYCLOAK_REALM}
     clientId: ${KEYCLOAK_CLIENT_ID}
     client-secret: ${KEYCLOAK_CLIENT_SECRET}
     admin-username: ${KEYCLOAK_ADMIN_USERNAME}
     admin-password: ${KEYCLOAK_ADMIN_PASSWORD}
     fe-url: ${FE_URL}
```

3. Create the SecurityConfig class and copy the *keycloak()* bean and the pre-defined strings there.
4. Copy the *UserService* interface and implementation class into the BL component.
5. Update the ReservationService, so the user is created before the reservation is saved.

The kafka listener calls *processReservation* method. Here we should check, if the email exists in the system, then, in case it does not, trigger the creation of a new user and set the user ID to the reservation entity to be saved.

```
@Override
public boolean processReservation(ReservationDTO newReservationDTO) throws
ExecutionException, InterruptedException {
...
if(newReservationDTO.getCustomerId() == null){
   newReservationDTO.setCustomerId(getUserIdByMail(newReservationDTO));
}
ReservationEntity entity = new ReservationEntity(
       newReservationDTO.getId(),
       newReservationDTO.getFirstName(),
       ...
       newReservationDTO.getSchedule().getLine().getDistance(),
       ReservationStatus.ACTIVE,
       newReservationDTO.getCustomerId()
);
reservationRepository.save(entity);
...
}
```

The private function *getUserIdByMail* is supposed to check, if the reservation is made by an unauthorized user with an account or by a new user (respectively by a mail that is not presented in the system).

```
private String getUserIdByMail(ReservationDTO reservationDTO){
   return userService.getUserIdByMailOrCreate(reservationDTO);}
```

Update the user service with the *getUserIdByMailOrCreate* method:

```
public String getUserIdByMailOrCreate(ReservationDTO reservationDTO){
   List<UserRepresentation> users =
getUsersByEmail(reservationDTO.getEmail());
   UserRepresentation ur;
   if(users.isEmpty()){
       ur = createUserFromReservation(reservationDTO).toRepresentation();
   }
   else{
       ur = users.get(0);
   }
   return ur.getId();
}
```

The BL component update on this stage is finished. It can be noticed, that this implementation leads to a bug - the cached version of the reservation with the status PENDING does not get the *customerId* in case the reservation was made by an unauthorized user. The detailed description and the solution will be presented in the *3.9.8 Long Reservation View Response Bug*, *3.10 Seventh Iteration: The Custome ID Assignment Change*, and *3.11.2 The Completion of Customer ID Assignment Change chapters*.

3.7 Fourth Iteration: Custom Theme

To provide a better user experience, the Keycloak look and feel can be customized. This chapter provides guides for applying a custom theme locally and on CodeNOW.

It is possible to change the look of the UI, as well as the email content. Email customization has its own intricacies, so the details about it are described in the *3.8 Fifth Iteration: Custom Email Template* chapter.

3.7.1 Create Custom Theme Locally Guide

Prerequisites

- Running Keycloak Docker container
- IntelliJ IDEA (version 2020.3.2+)

The easiest way to create a custom Keycloak theme, is to modify the existing default themes.

1. Download a Keycloak standalone server of your docker image version
2. Find the *theme* folder in the archive (the location may differ from version to version). There are usually 3 subfolders:
    - base
    - keycloak
    - keycloak.v2

   The *base* folder contains the basis for the whole Keycloak UI (e.g. HTML templates, messages texts, etc.)[69] For deep customization of this folder, the custom Keycloak providers (SPI) can be used, which is out of the scope of this work, because CodeNOW does not provide an interface for building SPI into the Keycloak managed service. **We are interested in the content of the *keycloak* folder**. It extends the *base* folder and mostly worlds with CSS files, JS scripts, and images.
3. Create a folder named *custom* in the same location as your *docker-compose* file and copy the content of the *keycloak* folder.
4. Make changes to the individual themes.

Example of changes for the login page:
In the *./custom/login/resources/img* added the CodeNow logo. It should be edited the way so it resembles the size of the original Keycloak logo - approximately 300x63 px.

In the *./custom/login/resources/css/login.css*
Change the background to solid blue color:

```
.login-pf body {
   background: #3498ff;
   background-size: cover;
   height: 100%;
}
...
@media (max-width: 767px) {
   .login-pf body {
         background: #3498ff;
   }
```

Changed the logo:

```
div.kc-logo-text {
      background-image: url(../img/codenow.png);
      background-repeat: no-repeat;
...
}
```

To mount the theme folder into the Keycloak docker container, update the *docker-compose* file:

```
keycloak:
   image: jboss/keycloak:13.0.0
...
     - "28080:8080"
volumes:
- /custom:opt/jboss/keycloak/themes/
```

The */custom* path refers to the folder to copy into the docker container and *opt/jboss/keycloak/themes/* is the path to where to copy. Attention, this path is image-dependent!
Most of the time, the container is updated with the new folders without problem, but it can happen, that the container will be rebuilt and all the data will be lost.

Warning
It is strongly recommended to make a backup copy of the the existing Keycloak realm before launching the container after the docker-compose modification.

Also, to make the development process more convenient, it is possible to add the next commands into the docker-compose file to turn off the theme caching:

```
--spi-theme-static-max-age=-1 --spi-theme-cache-themes=false
--spi-theme-cache-templates=false
```

The IntelliJ idea has smooth integration with the Docker. You can make sure that the changes were applied by navigating inside the Docker container via the console or using the IDE interface (see *Figure 59. Docker integration within IDE*)

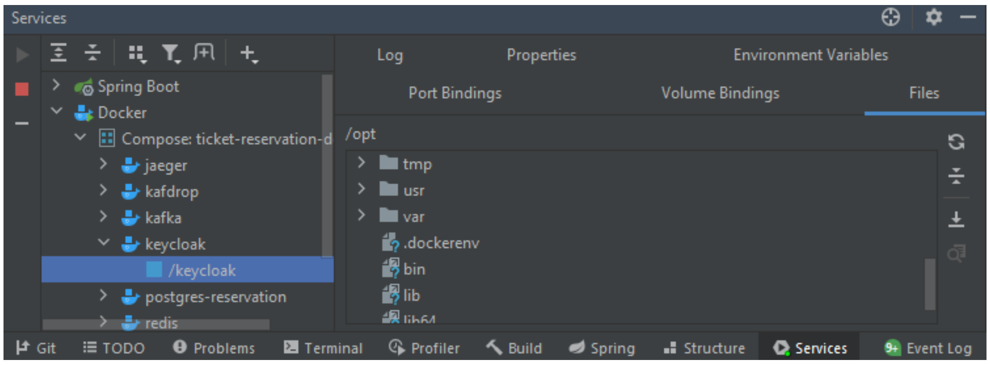


Figure 59. Docker integration within IDE

3.7.2 Custom Theme on CodeNOW Guide

CodeNOW provides an interface for the theme customization of the managed Keycloak. The configuration steps are described in the Custome theme for the Keycloak tutorial[67]

3.7.2.1 Official Tutorial Quick Summary

1. It is necessary to create a new docker-generic component in the application and clone it (see *3.3.1 Source Code*)
2. Change the autogenerated Dockerfile inside the component according to the tutorial.
```
FROM busybox
COPY theme /custom
```
3. Set up and toggle on the Custom Theme in the Keycloak managed service interface.

Be sure, the committed Dockerfile does not contain any mistakes. It can lead to the managed service Keycloak instance stopping to work (see *3.7.2.2 Deadlock on CodeNOW).*

3.7.2.2 Deadlock on CodeNOW

The first Dockerfile that was deployed in the docker-generic component had a mistake. It turned out to be a crucial mistake, as the toggling of the *Custom Theme* created a deadlock on CodeNOW. CodeNOW was trying to rebuild the container, but the build failed and it stuck in a loop. There was no way to cancel the process, or fall back to default settings. The existing instance of the Keycloak managed service was lost forever and the bug was reported to the CodeNOW team.
To avoid further misunderstandings, I have suggested corrections for the "Custom theme for Keycloak" tutorial.

3.7.3 Custom Theme Test

To make Postman redirect to the login page, use the inbuilt support of the OAuth2 protocols.
Create a new POST request and navigate to the *Authorization* tab.

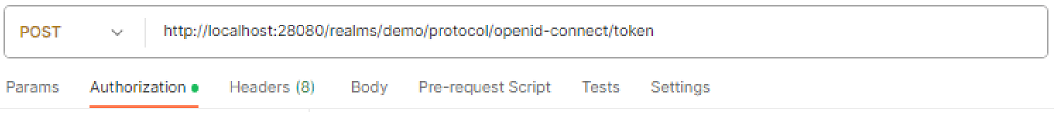
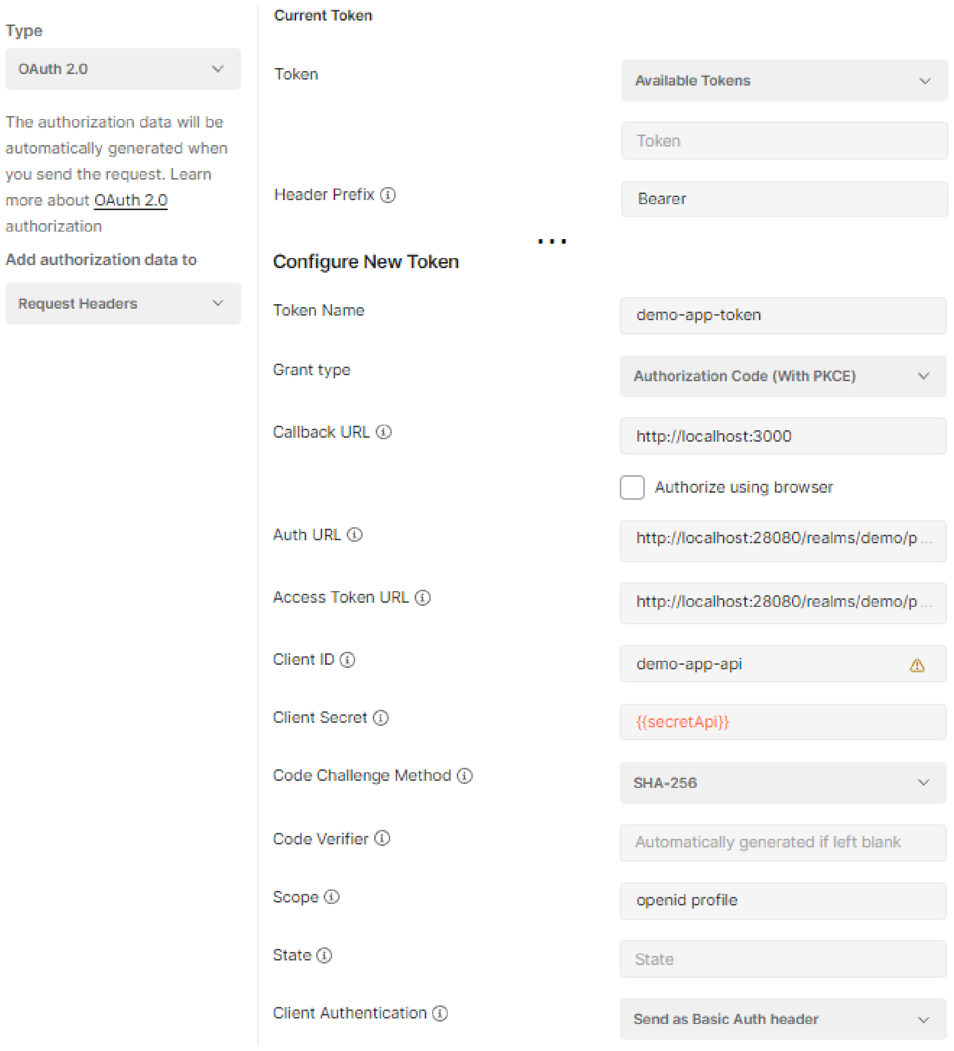

POST http://localhost:28080/realms/demo/protocol/openid-connect/token

Figure 60. Postman request to get an access token

Choose the OAuth 2.0 Type option and fill the form as shown in *Figure 61. Postman OAuth authorization example*. (Attention to the */auth* path). It uses the AUTHORIZATION CODE grant type, which flow requires the redirection to the login page.


Type: OAuth 2.0

The authorization data will be automatically generated when you send the request. Learn more about OAuth 2.0 authorization

Add authorization data to: Request Headers

Current Token

Token: Available Tokens
Token
Header Prefix: Bearer

Configure New Token

Token Name: demo-app-token
Grant type: Authorization Code (With PKCE)
Callback URL: http://localhost:3000
Authorize using browser
Auth URL: http://localhost:28080/realms/demo/p ...
Access Token URL: http://localhost:28080/realms/demo/p ...
Client ID: demo-app-api
Client Secret: {{secretApi}}
Code Challenge Method: SHA-256
Code Verifier: Automatically generated if left blank
Scope: openid profile
State: State
Client Authentication: Send as Basic Auth header

Figure 61. Postman OAuth authorization example

Click the *Get New Access Token*. The Postman will redirect you to the login page, where you should be able to see the applied theme changes.
An example of a new login page look is in *Figure 62. Login page with custom theme*.

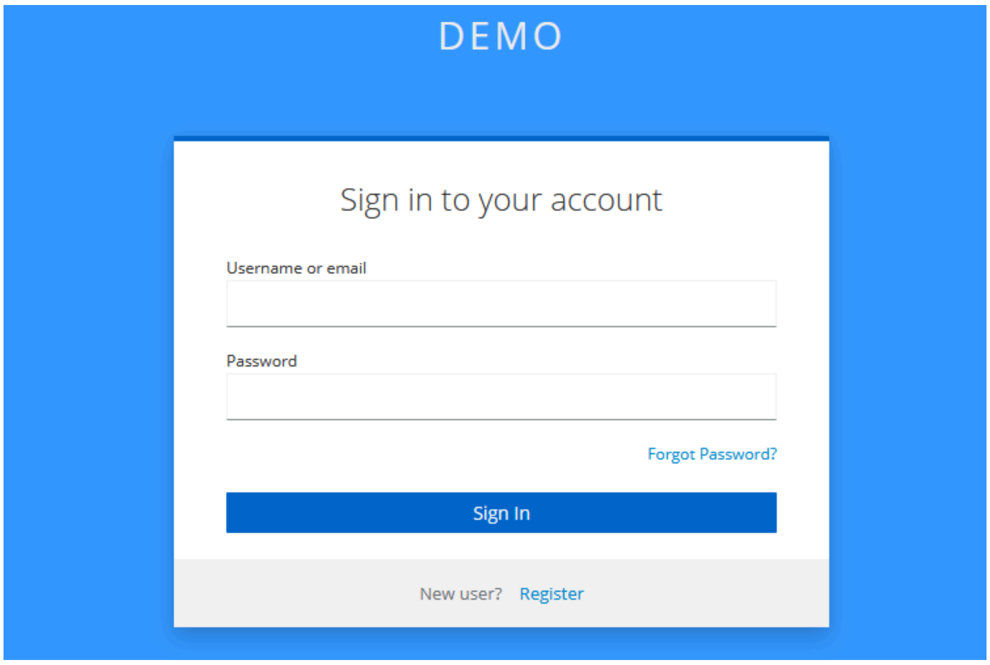
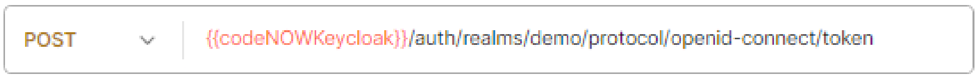


Figure 62. Login page with custom theme

To test the changes on CodeNOW, provide the corresponding address in a Postman request like in *Figure 63. Postman request for access token for CodeNOW*.



POST {{codeNOWKeycloak}}/auth/realms/demo/protocol/openid-connect/token

Figure 63. Postman request for access token for CodeNOW

3.8 Fifth Iteration: Custom Email Template

Emails are a crucial part of the user's account setup. It differs from the normal work with Keycloak theme customization, as it requires a comprehension of the Keycloak email build workflow,

3.8.1 Email Templates

To get to the email templates, find the theme folder and go to the email subfolder. There you will find *html*, *messages,* and *text* folders. HTML and text folders contain the templates for mail in the HTML formatting and the plain text format respectively. The messages folder contains localisations of the text for various languages. This project works with the English localization - the *messages_en.properties* file.

Keycloak builds emails in 2 steps:

1. *EmailTemplateProvider* finds a template from the *html* or *text* folder, that is chosen based on the used method and actions list provided in it.
2. Fill in the template with the text provided in the messages folder from a *message_x.ftl* file, where *x* stands for a localization language, as "en", "cz", etc.

The templates use *.ftl* format for the FreeMarker Template format[68].

The main files we will be working with *executeActions.ftl* in both HTML and text folders, as well as the *messages_en.properties* file.
The **execute action endpoint** is the most flexible and convenient for customization. The standard approach is to rewrite the text body of the *messages_x.properties* file. It, however, ruins the template for the whole standard action and would change this email for other realms in the Keycloak. I would like to come up with a solution, that customizes the mail behavior and preserves the standard behavior at the same time.
The solution is shown in the *3.8.1.1 ExecuteActions.ftl File in html Folder* in the *3.8.1.2 ExecuteActions.ftl File in text Folder* allows adding custom text and stylization while **saving the standard Keycloak email template behavior for every realm**.

The custom behavior will be triggered, only if the *ExecuteAction endpoint* receives a specific combination of actions - "UPDATE_PROFILE" and "UPDATE_PASSWORD". In case the combination of actions does not occur, the template falls into the default template. Ideally, we could send a unique action, like "SETUP" and provide a custom SPI for it (see *3.8.2 SPI* section). This is not in the scope of this work, mainly because mounting of a custom SPI in a Keycloak-managed service on CodeNOW is not supported now. Nevertheless, a custom action can be sent to an *ExecuteAction endpoint* even without SPI and any major problems, but then the artificial action renders poorly during the account setup workflow which negatively affects the user experience.

In the templates, the *${msg(...)}* block allows accessing variables from the Client request and setting a custom text that will be parsed from a *messages_x.properties* file. As will be shown below in the *3.8.1.1 ExecuteActions.ftl File in html Folder* and *3.8.1.2 ExecuteActions.ftl File in text Folder* sections, we can access the user's real name and username to build a message body. The *${msg(...)}* acquires a custom text body through the "*executeSetupBodyHtml*" variable, which refers to a pre-prepared text block in the messages file (see *3.8.1.3 Message_en.properties File* section).

3.8.1.1 ExecuteActions.ftl File in html Folder

```
<#outputformat "plainText">
<#assign requiredActionsText><#if requiredActions??><#list requiredActions><#items
as reqActionItem>${msg("requiredAction.${reqActionItem}")}<#sep>,
</#sep></#items></#list></#if></#assign>
</#outputformat>
<#assign profileAction = "UPDATE_PROFILE">
<#assign passwordAction = "UPDATE_PASSWORD">
<#assign found1 = false>
<#assign found2 = false>
<#list requiredActions as reqActionItem>
	<#if reqActionItem == profileAction><#assign found1 = true></#if>
	<#if reqActionItem == passwordAction><#assign found2 = true></#if>
</#list>
<#if found1 && found2>
	<html>
		<body>
			${kcSanitize(msg("executeSetupBodyHtml",
					link,
					linkExpiration,
					realmName,
					 requiredActionsText,
					linkExpirationFormatter(linkExpiration),
					 user.username))?no_esc}
		</body>
	</html>
<#else>
	<html>
		<body>
	 ${kcSanitize(msg("executeActionsBodyHtml",link, linkExpiration, realmName,
requiredActionsText, linkExpirationFormatter(linkExpiration)))?no_esc}
		</body>
	</html>
</#if>
```

3.8.1.2 ExecuteActions.ftl File in text Folder

```
<#ftl output_format="plainText">
<#assign requiredActionsText><#if requiredActions??><#list requiredActions><#items
as reqActionItem>${msg("requiredAction.${reqActionItem}")}<#sep>,
</#items></#list><#else></#if></#assign>
<#assign profileAction = "UPDATE_PROFILE">
<#assign passwordAction = "UPDATE_PASSWORD">
<#assign found1 = false>
<#assign found2 = false>
<#list requiredActions as reqActionItem>
	<#if reqActionItem == profileAction><#assign found1 = true></#if>
	<#if reqActionItem == passwordAction><#assign found2 = true></#if>
</#list>
<#if found1 && found2>
	${msg("executeSetupBody", link,
				linkExpiration,
				realmName,
				requiredActionsText,
				linkExpirationFormatter(linkExpiration),
				user.username)}
<#else>
	${msg("executeActionsBody",link, linkExpiration, realmName,
requiredActionsText, linkExpirationFormatter(linkExpiration))}
</#if>
```

3.8.1.3 Message_en.properties File

This file is formatting-sensitive. Every new line is a new topic. The numbers in the curly braces (e.g. *{0}*) refer to a corresponding variable placed from a template *${msg(...)}* function.

For the text template, add the next lines:

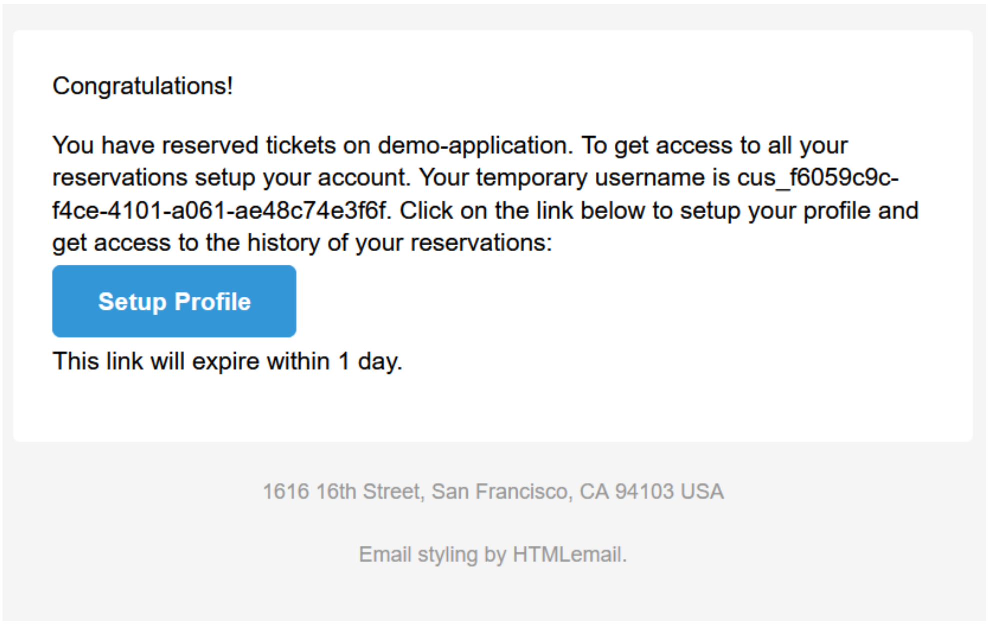
```
executeSetupSubject=Setup your account
executeSetupBody=Congratulations! You have reserved tickets on demo-application. To
get access to all your reservations setup your account. Click on the link below to
start this process.\n\n{0}\n\nThis link will expire within {4}.\n\nIf do not want
to setup your account, just ignore this message and nothing will be changed. Your
temporary username is {5}.
```

For the html template, add next lines:

```
executeSetupBodyHtml=<p >Congratulations!</p><p >You have reserved tickets on
demo-application. To get access to all your reservations setup your account. Your
temporary username is {5}. Click on the link below to setup your profile and get
access to the history of your reservations:</p><a href="{0}">Setup
profile</a><p>This link will expire within {4}.</p>
```

3.8.1.4 Email Styling

It is possible to stylize messages with HTML for a better user experience. On the internet, there are plenty of free templates. I used an email template by Lee Munroe[70].
The result is in the *Figure 64. Styled email example*.



Congratulations!

You have reserved tickets on demo-application. To get access to all your reservations setup your account. Your temporary username is cus_f6059c9c-f4ce-4101-a061-ae48c74e3f6f. Click on the link below to setup your profile and get access to the history of your reservations:

This link will expire within 1 day.

1616 16th Street, San Francisco, CA 94103 USA

Email styling by HTMLemail.

Figure 64. Styled email example

3.8.2 SPI

It is possible to create your own implementation of the email provider service for Keycloak, so there is no need to implement workarounds in the email templates. To do that, it is necessary to implement and mount in the Keycloak container implementation of a custom Service Provider Interface SPI[71].
Keycloak uses *EmailTemplateProvider* and *EmailTemplateProviderFactory* for creating a mail. This approach was not used in this project, as there is no interface on CodeNOW for Keycloak service customization with its own SPI.

3.9 Sixth Iteration: FE Component

Finally, it is time to link the updated back end with the front end and make requests using UI instead of Postman. FE component uses React and TypeScript. For styling, the project uses the MUI library.

3.9.1 Keycloak Integration

Keycloak uses keycloak-js library for integration with Javascript. The instructions for the installation and configuration can be found in the official documentation[54].
The main inspiration for the front-end security structure comes from Niko Köbler's tutorials[55]

3.9.1.1 User service

In order to use Keycloak, we need to implement code utilizing keyloak-js. It is not a cloud native-friendly solution, so to negate the negative effects of implementation-specific usage of a service, the Keycloak functionality is gathered in one place - *UserService*.
The code presented here is mostly an adaptation and extension of the open-source demo by Niko Köbler[56].

Keycloak Initialisation

To initialize Keycloak correctly, the *index.tsx* file should be updated as follows:

```
const renderApp = () => root.render(
   <React.StrictMode>
       <App />
   </React.StrictMode>);
UserService.initKeycloak(renderApp);
```

We must not omit providing the Callback to the *UserService.initKeycloak* function, otherwise the page will be built before the Keycloak initialization, which can lead to silent errors and undefined behaviour.

Keycloak instantiation:

```
const _kc = new Keycloak({
   url: 'http://localhost:28080/',
   realm: 'demo',
   clientId: 'demo-app-fe'
});
```

The implementation of the *initKeycloak* function:

```
const initKeycloak = (onAuthenticatedCallback) => {
   try {
       const authenticated = _kc.init({
           redirectUri: "http://localhost:3000",
           pkceMethod: "S256",
           onLoad:"check-sso",
           silentCheckSsoRedirectUri: window.location.origin +
'/silent-check-sso.html'
       }).then((authenticated) =>{
           console.log(`User is ${authenticated ? 'authenticated' : 'not
authenticated'}`);
           onAuthenticatedCallback();
       });
        } catch (error) {
            console.error('Failed to initialize adapter:', error);
        }
     };
```

The *check-sso* options defines, that is the user is not logged in, they will be redirected to the main page of the application (provided in the *redirecUri*). The alternative value is *login-required*, which would redirect the user to the login page automatically.

Silent SSO

The implementation of the *initKeycloak* function enables the silent SSO check.
The official Keycloak documentation provides the following description of the silent SSO option:
"*You can configure a silent check-sso option. With this feature enabled, your browser will not perform a full redirect to the Keycloak server and back to your application, but this action will be performed in a hidden iframe. Therefore, your application resources are only loaded and parsed once by the browser, namely when the application is initialized and not again after the redirect back from Keycloak to your application. This approach is particularly useful in case of SPAs (Single Page Applications).*"[72]

The *silent-check-sso.html* file is located in the *./public* folder:

```
<html>
   <body>
       <script>parent.postMessage(location.href, location.origin)
       </script>
   </body>
</html>
```

Other functions implementation

```
const doLogin = _kc.login;
const doLogout = _kc.logout;
const getToken = () => _kc.token;
const getTokenParsed = () => _kc.tokenParsed;
const isLoggedIn = () => !!_kc.token;
const updateToken = (successCallback) =>
   _kc.updateToken(5)
       .then(successCallback)
       .catch(doLogin);
const isExpired = () => _kc.isTokenExpired(5);
const getUsername = () => _kc.tokenParsed?.preferred_username;
const getFullName = () => _kc.tokenParsed?.name;
const getEmail = () => _kc.tokenParsed.email;
const hasRole = (roles) => roles.some((role) =>
_kc.hasRealmRole(role));
```

3.9.1.2 HTTP Service

We want to automize the work with the Authorization and Etag headers during requests. Using Axios, It is possible to configure *axios.interceptors* for these purposes.
The configured Axios instance will be wrapped in the *HttpService*.

The interceptor also refreshes an access token, when a user performs an action that uses *HttpService.* In the future, the token refreshment will be handled in a side loop and will be independent of the user's activity, as the system does not require a high level of protection.

If a user performs an action that requires authorization, then the message about it pops up and the action is not performed.

The work on the cache system is still in progress. The code snipped presented here is not the final version.

```
export const HttpService = () => {
   const cacheAxios = axios;
   axios.interceptors.request.use((config) => {
       if (UserService.isLoggedIn()) {
           const cb = () => {
               config.headers.Authorization = `Bearer ${UserService.getToken()}`;
               return Promise.resolve(config);
           };
           return UserService.updateToken(cb);
       }
       return config;
   });

   axios.interceptors.response.use(
       (response) => {
           if (response.headers && response.headers.etag) {
               console.log('Got an ETag header from response, including
If-None-Match: {}', response.headers.get('ETag'));
               axios.defaults.headers.get["If-None-Match"] = response
                                                             .headers
                                                             .get('ETag');}
           return response;
       },
       (error) => {
           if(!UserService.isLoggedIn()){
               enqueueSnackbar(`To access this page you need to log in.`,
                                               { variant: "error" });
               setTimeout(() => {
                   window.location.href = '/';
               }, 3000)
               return Promise.reject(error);
           }
           if(error.response && error.response.status === 401){
               enqueueSnackbar(`Your session has expired, you will be redirected to
the login page.`, { variant: "error" });
               setTimeout(() => {
                   UserService.doLogin();
               }, 3000);
               return Promise.reject(error);
           }
           if(error.response.status === 304 && error.response.config.cachedData){
               console.log('There was 304 response!');
               error.response.data = error.response.config.cachedData;
               return Promise.resolve({
                   ...error.response,
                   data: error.response.config.cachedData,
               });
           }
           return Promise.reject(error);
       }
   )

   function setupCommonHeader( header, value ){
       HttpService.defaults.headers.common[header] = value;
   }

   return {cacheAxios, setupCommonHeader};
}
```

3.9.2 Dynamic React Components

The dynamic elements will react on the authentication status and roles of a user.

The elements, that are supposed to be shown only to authenticated users will be wrapped in the special component - *RenderOnAuthenticated*.

```
// @ts-ignore
const RenderOnAuthenticated = ({children}) =>
                                        (UserService.isLoggedIn()) ? children :
                                 null;
```

Components that are supposed to be shown only to unauthenticated users will be wrapped in the *RenderOnAnonymous* component.

```
// @ts-ignore
const RenderOnAnonymous = ({ children }) =>
                                 (!UserService.isLoggedIn()) ? children : null;
```

The role-specific elements can be wrapped in the *RenderOnRole* element:

```
interface props {
   roles : string[],
   showNotAllowed: boolean;
   children : ReactElement<any, any> | null;
}
const RenderOnRole : FC<props> = ({ roles, showNotAllowed, children }) =>
(UserService.hasRole(roles))? children : showNotAllowed
                                 ? <NotAllowed/> : null;
```

The *showNotAllowed* option defines, if instead of a component the *NotAllowed* component will be rendered:

```
          const NotAllowed = () => (
             <h1 className="text-info">Access is not allowed! ⛔</h1>)
```

3.9.3 Interface Update for Login Guide

The first task in the FE interface update is providing an opportunity for the login flows. For that, the Login button will be added to the *Navbar* component. The button is supposed to change depending on the authorization status of a user - Login for an unauthorized user and Logout for an authorized user.

3.9.3.1 Login Button Component

The Login button component utilises the *UserService*, to redirect a user to the Keycloak login page.

```
         export const LoginButton = () => {
            const handleLoginClick = async () => {
                try {await UserService.doLogin();}
                catch (error) {console.error('Error during login:', error);}};
                return(
                    <Stack {/*styling*/}>
                        <IconButton
                            onClick={handleLoginClick}
                            sx={{ color: "white", backgroundColor: "#3498ff"}}>
                            <Typography variant="h6" color="white"
         paddingRight={1}>
                                 Login
                            </Typography>
                            <Login /> //SvgIconComponent
                        </IconButton>
                    </Stack>);}
```

3.9.3.2 Logout Button/Profile Component

Profile button shows the name of a user and also serves as a Logout button, utilising *UserService*.

```
export const ProfileButton = () => {
   const handleLogoutClick = async () => {
       try {await UserService.doLogout();}
catch (error) {console.error('Error during logout:', error);}};
   return(
       <Stack {/*styling*/}>
       <Box alignItems={"center"} alignSelf={"center"}>
           <Typography variant="h6" color="white">
                   {UserService.getFullName()}
            </Typography>
       </Box>
       <Tooltip
           title={"Logout"}>
           <IconButton
               onClick={handleLogoutClick}
               sx={{ color: "white", backgroundColor: "#3498ff"}}>
               <Logout /> //SvgIconComponent
           </IconButton>
       </Tooltip>
   </Stack>);
}
```

3.9.3.3 View Button Component

The view buttons were not extracted into standalone components and currently are in-line in the Navbar component. The text and navigation behavior changes depending on the user's role.

```
<RenderOnRole roles={["customer"]} showNotAllowed={false}>
   <Button
       variant="text"
       onClick={() => navigate("/customer-reservations")}
       disableRipple
       sx={{ textTransform: "none" }}>
       <Typography variant="h6" color="white">My reservations</Typography>
   </Button>
</RenderOnRole>
<RenderOnRole roles={["administrator"]} showNotAllowed={false}>
   <Button
       variant="text"
       onClick={() => navigate("/reservations")}
       disableRipple
       sx={{ textTransform: "none" }}>
       <Typography variant="h6" color="white">Reservations</Typography>
   </Button>
</RenderOnRole>
<ProfileButton/>
```

For the View Buttons to work properly, update the *Router* in the *App.tsx:*

```
...
<Route path="/customer-reservations" element={<CustomerReservationsView />} />
<Route path="/reservations" element={<ReservationsView />} />
...
```

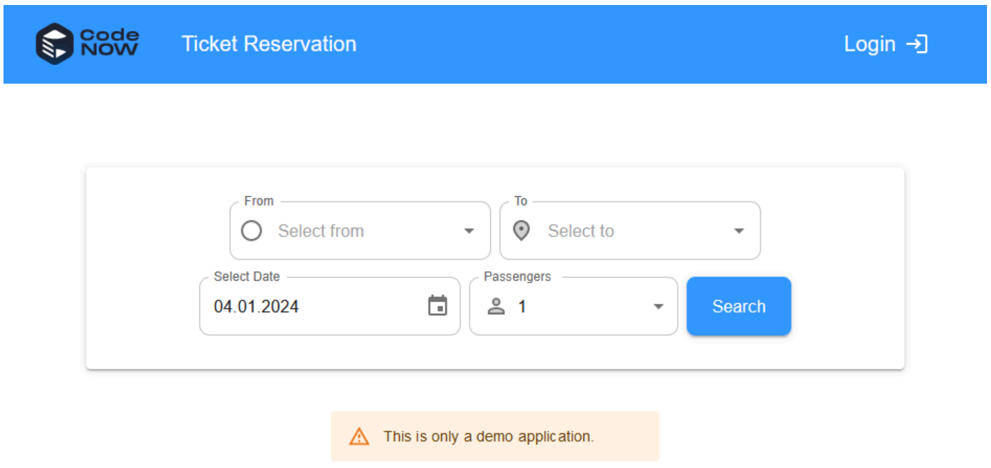
It redirects the user to the pages created in the *3.9.5 Reservation Views Pages* section.

3.9.3.4 Navbar Component Update

There is the modified structure of the Navbar component after the incorporation of the new elements. Thanks to the responsive components (see *3.9.2 Dynamic React Components*), the Login/Logout interface is changed depending on the user authentication status.

```
...//the logo, web name is omitted
          <Stack direction={"row"} spacing={4} alignItems={"center"} >
              <RenderOnAuthenticated>
             //Customer view button
             //Administrator view button
              <ProfileButton/>
              </RenderOnAuthenticated>
              <RenderOnAnonymous>
                  <LoginButton/>
              </RenderOnAnonymous>
          </Stack>
...
```
In the screenshots below there is the Navbar look after the changes were finished.

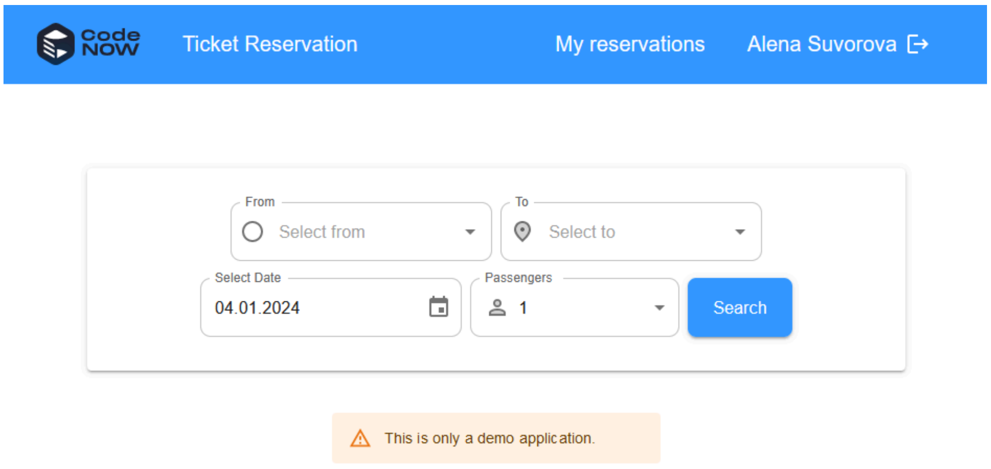
- The interface of unauthorized users - *Figure 65. FE unauthorized interface*.



This is only a demo application.

Figure 65. FE unauthorized interface

- The interface of authorized users with the customer role - *Figure 66. FE authorized interface*.



This is only a demo application.

Figure 66. FE authorized interface

3.9.4 FE API update

The existing hooks were modified to use the newly implemented HTTP service instead of plain Axios (see *3.9.1.2 HTTP Service*). New hooks were implemented for obtaining the customers's reservations (customer view) and all reservations (administrator view) from the API. For hooks to work, the data model was updated with new data structures. The *Reservation* type was renamed to *ReservationRequest* for better integrity, as it is used as a *NewReservationDTO* during the reservation process itself.

3.9.4.1 New Hooks

Create the hooks in the *./src/api/hooks* folder for connection to the new API endpoints to get reservation views..

useCustomerReservation Hook

This hook is the *useQuery* type. It calls the customer's reservations endpoint of the *getViewOfCustomerReservations* API method.

```
// @ts-ignore
const apiBaseUrl = window.env.API_BACKEND_URL;

type CustomerReservationOptions =
Partial<UseQueryOptions<CustomerReservationListResponse, unknown,
CustomerReservationListResponse>>;
export const useCustomerReservation = (options?: CustomerReservationOptions) => {
                    console
                    .log("Sending query to API to get customer reservations.");

   return useQuery<CustomerReservationListResponse,
                  unknown,
                  CustomerReservationListResponse>
(
       ["customerReservation"],
       () => HttpService().cacheAxios.get
                              (apiBaseUrl + "/reservation/customer-reservations"
        ).then(({ data }) => data),
                           options,
        );
};
```

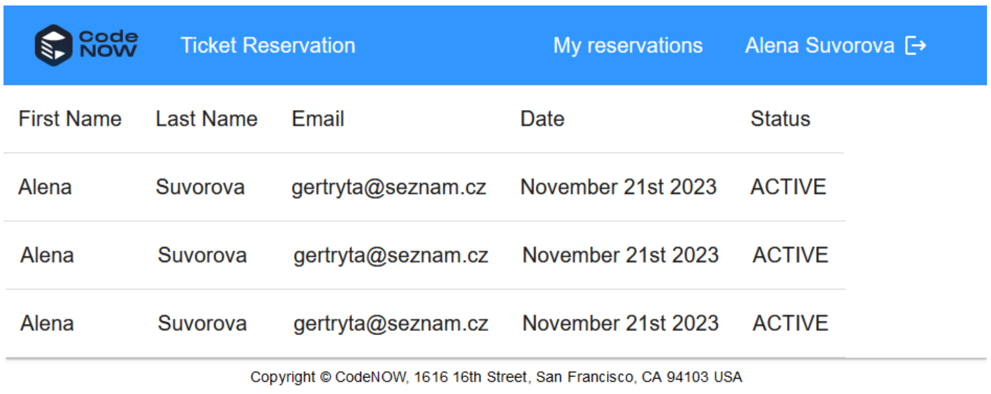
useAllReservations Hook

This hook is the *useQuery* type. It calls the endpoint of the *getViewOfReservations* API method.

```
// @ts-ignore
const apiBaseUrl = window.env.API_BACKEND_URL;
type ReservationOptions = Partial<UseQueryOptions<ReservationListResponse,
                                                 unknown,
                                                 ReservationListResponse>>;
export const useAllReservations = (options?: ReservationOptions) => {
                           console
                           .log("Sending query to API to get all reservations.");
   return useQuery<ReservationListResponse, unknown, ReservationListResponse>(
       ["customerReservation"],
       () => HttpService().cacheAxios.get(apiBaseUrl + "/reservation"
       ).then(({ data }) => data),
             {...options,
             refetchIntervalInBackground: false,
             refetchOnWindowFocus: true},
       );
};
```

3.9.5 Reservation Views Pages

To display reservations obtained with the new hooks, create the corresponding pages. They will incorporate the dynamic components, described in the *3.9.2 Dynamic React Components* section.
To render the date correctly, those pages use the Moment library[73].
An example of a reservations view page is in *Figure 67. FE reservations view*.



First Name	Last Name	Email	Date	Status
Alena	Suvorova	gertryta@seznam.cz	November 21st 2023	ACTIVE
Alena	Suvorova	gertryta@seznam.cz	November 21st 2023	ACTIVE
Alena	Suvorova	gertryta@seznam.cz	November 21st 2023	ACTIVE

Copyright © CodeNOW, 1616 16th Street, San Francisco, CA 94103 USA

Figure 67. FE reservations view

3.9.5.1 All Reservations View Page

The reservation page is protected by dynamic components (*RenderOnAuthenticated*) and on the back-end API level. Only authorized users with the right role will see the content of the page.
The page utilizes the new *useAllReservations* hook. While the results are loading it shows a placeholder of data.

```
export const ReservationsView = () => {
   const reservationsQuery = useAllReservations({
       onSuccess:(data => res = data),
       onError: (()=>console.log("ERROR GETTING RESERVATIONS"))});
   return (
       <><Navbar/>
          <RenderOnAuthenticated>
              <TableContainer component={Paper}>
                 <TableHead>
       ...//a TableRow withFirst name, Last name, Username, Email, Date, Status
       cells
                 </TableHead>
                 <TableBody>
                     {reservationsQuery.isLoading && (//shows placeholder of data
                     <Stack spacing={2} alignItems="center">
                        {Array.of(1, 2, 3, 4).map((i) => (
                            <Skeleton key={i} variant="rounded" width="100vw"
                     height={50} />))}
                     </Stack>)}
                     {reservationsQuery.isSuccess && (//data after finished loading
                        reservationsQuery.data.map(
                            ({id,firstName,lastName,email,date,status,customerId})
                     =>(
                               <TableRow key={id}>
                                   <TableCell>
                                          <Typography variant="h6">
                                                 {firstName}
                                          </Typography>
                                   </TableCell>
                            ... //Rest of TableCells with variables
                               </TableRow>)))}
```

```
                </TableBody>
...// closing tags
```

3.9.5.2 Customer Reservation View Page

The customer's reservation page follows the same structure as the all reservations view page. It is protected on front-end and back-end levels as well. Unlike the all reservations page, it shows a bit less data and utilizes the request to the *getViewOfCustomerReservations* API method.

```
export const CustomerReservationsView = () =>{
   const [reservations, setReservations] =
useState<Array<CustomerReservation>>([]);
   const reservationsQuery = useCustomerReservation({
       onSuccess:(data => setReservations(data)),
       onError: (()=>console.log("ERROR GETTING RESERVATIONS"))
   });
   return (
       <>
           <Navbar/>
           <RenderOnAuthenticated>
               <TableContainer component={Paper} >
                   <TableHead>
                       <TableRow>
                           <TableCell>
<Typography variant="h6" >First Name</Typography>
                           </TableCell>
                          ...// Last name,Email, Date, Status cells
                           </TableRow>
                   </TableHead>
                   <TableBody>
                       {reservationsQuery.isLoading && (
                           <Stack spacing={2} alignItems="center">
                               {Array.of(1, 2, 3, 4).map((i) => (
                                   <Skeleton key={i} variant="rounded"
width="100vw" height={50} />))}
                           </Stack>)}
                       {reservationsQuery.isSuccess && (
                           reservationsQuery.data.map(
                               ({id, firstName, lastName, email, date, status}) =>(
                               <TableRow key={id}>
                                   <TableCell  >
                                        <Typography variant="h6" >
                                              {firstName}
                                        </Typography>
                                   </TableCell>
                           ... //Rest of TableCells with variables
                               </TableRow>)))}
                   </TableBody>
               </TableContainer>
           </RenderOnAuthenticated>
           <Footer/>
       </>
   )
}
```

3.9.6 Email Field Security Guide

On of the business rules defines, that a reservation can be done only for the email of the authenticated user. To enforce it, let's update the reservation form, so the user's email is pre-filled and unchangeable. The changes are applied to the *ConnectionDetail.tsx* page.

```
...
<Formik
   initialValues={{
       firstName: "",
       lastName: "",
       email: UserService.isLoggedIn() ? UserService.getEmail() : ""
   }}
...
<Grid container rowSpacing={2}>
   <Grid item xs={12} lg={8}>
       <Field
           as={TextField}
           name="email"
           type="email"
           label="Email"
           disabled={UserService.isLoggedIn()}
           fullWidth
           required/>
...
```

The result of the changes above is seen in *Figure 68. Secured email input*.

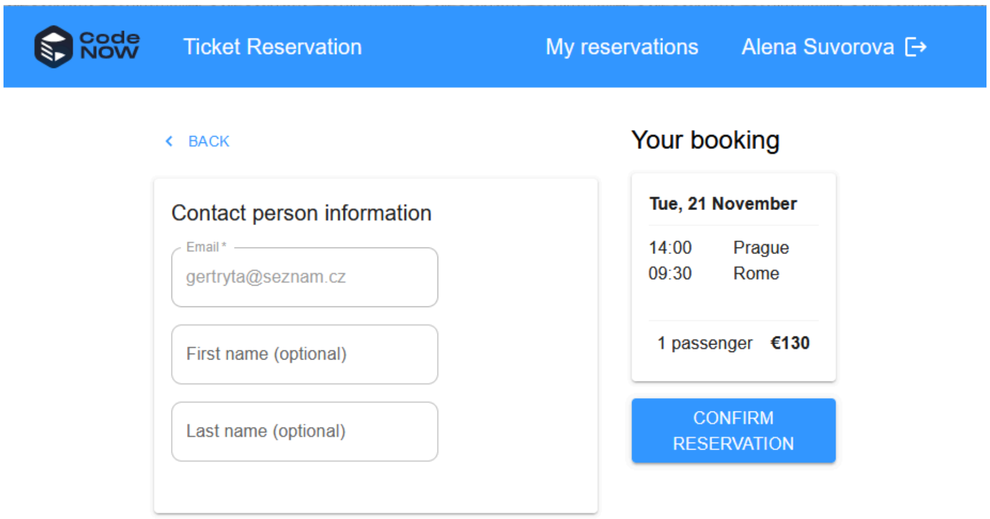


Figure 68. Secured email input

Of course, an experienced user can change this value using the browser console. To finilize the protection of this field, update the *createReservation* REST API method.

```
...
   if(jwt != null){
       if(!userService.isSameEmail(newReservationDTO.getEmail(),
                                   jwt.getSubject())){
           log.error("The request contains mismatched emails);
           return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
       }
...
```

The *userService.isSameEmail* method checks if a mail from the reservation corresponds to the mail of an account and returns a boolean value.

3.9.7 Front End Errors

3.9.7.1 CORS

"CORS Preflight Did Not Succeed" error indicates that the front-end application is trying to make a cross-origin request (from FE to API component in our case), and the API component server a is not allowing the request due to CORS (Cross-Origin Resource Sharing) policy.

To resolve this problem, make sure to have CORS enabled in the *SecurityFilterChain* method in the *SecurityConfig* class.

For development purposes, it is possible to use a browser plugin, that turns off the CORS policies. An example of such a plugin for Google Chrome and Firefox browsers is Allow CORS: Access-Control-Allow-origin[74].

3.9.7.2 Error Code

The 304 NOT MODIFIED error may occur when the FE component makes a request for a reservation view. If data were not changed, then the API Controller *getViewOfReservations* or *getViewOfCustomerReservations* returns this error code. The snippet of the methods:

```
...
if (reservationService.checkSameEtag(etag)){
   log.warn("Data was not modified.");
   return ResponseEntity.status(HttpStatus.NOT_MODIFIED).eTag(etag).build();
}
...
```

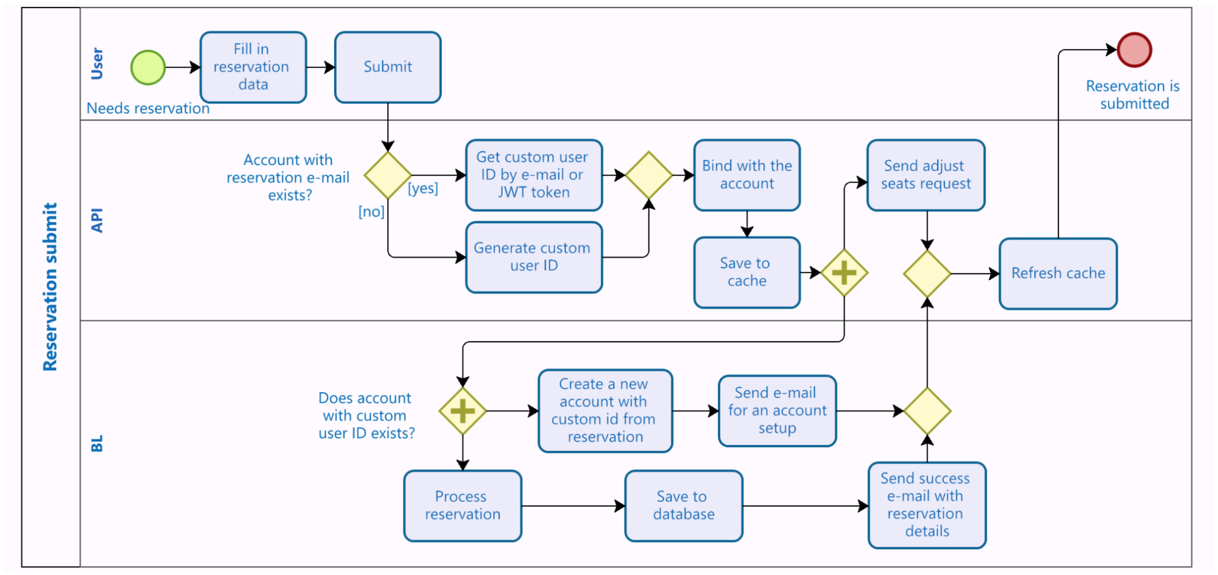
To resolve it, the error should be handled and the cached version of the data should be applied.

3.9.8 Long Reservation View Response Bug

Because of the precedent change in the application architecture, described in *3.6 Third Iteration: Architecture change,* a certain bug has occurred conflicting with the CQRS pattern of the application. The reservation view could not be updated instantly from the API cache with a newly created PENDING reservation in case the new reservation was made by an unauthorized user. It happens because the user creation and existence check happens later in the BL component. The situation is fixed only after the reservation is refreshed with the status ACTIVE. It creates an unpleasant pause during a reservations view render. Because of this, the necessity to change the user ID assignment flow has occurred. It will be described in the next section of this work - *3.10 Seventh Iteration: The Custome ID Assignment Change.*

3.10 Seventh Iteration: The Custome ID Assignment Change

To resolve a problem with a customer ID not being set to a reservation by an anonymous user (see), an alternative ID assignment flow has been made. The illustration of the new concept is in *Figure 69. New Reservation submit detailed activity diagram*.



Reservation submit
User
API
BL
Needs reservation
Fill in reservation data
Submit
Reservation is submitted
Account with reservation e-mail exists?
[yes]
[no]
Get custom user ID by e-mail or JWT token
Generate custom user ID
Bind with the account
Save to cache
Send adjust seats request
Refresh cache
Does account with custom user ID exists?
Create a new account with custom id from reservation
Send e-mail for an account setup
Process reservation
Save to database
Send success e-mail with reservation details

Figure 69. New Reservation submit detailed activity diagram

The point is to manually generate an ID instead of using an ID, generated by Keyclock during user creation. It has several benefits: it can be created at any moment, even before the user registers in the system, it can have any format and follow a business rule if such a requirement appears.

It would be optimal to generate an ID beforehand and set it as the Keycloak ID afterward. Unfortunately, the trials have shown, that there is no way to set a Keycloak ID before or after a used has been created (even though the Keycloak Admin Client displays such an interface. One of the reasonable solutions is using Keycloak attributes. The details of the implementation of the new ID assignment flow will be in the *3.11 Eighth Iteration: Keycloak Version Update from 13 to 21*.

3.10.1 Spring Boot Asynchronous Configuration Guide

The new assignment flow gives an opportunity for parallelization of a user reaction and a reservation process. To make them run independently on the BL component, we need to enable support asynchronous methods in Spring Boot.

1. In the Application class with the main function, add the *@EnableAsync* annotation.
2. Create an *AsynchronousConfiguration* class and annotate it with *@EnableAutoConfiguration*.
3. Instantiate threads pool task executors. The detailed instructions can be found in the tutorial[75].
4. In the *UserService* interface, annotate the function that triggers a user existence check and creation with *@Async*
5. Put this function early in the *processReservation* method of the *ReservationService*.

This way we optimized the flow and shortened the period of the back-end response.

3.11 Eighth Iteration: Keycloak Version Update from 13 to 21

The study of possible solutions of the custom customer ID assignments has shown, that it would be suitable to use Keycloak attributes to set a custom customer ID. The problem is, that Keycloak Admin Client does not fully support this functionality in 13.x.x versions of Keycloak. The interface for searching by user's attributes starts with the 15 versions of Keycloak. Thus, I asked for permission to use a newer version of the Keycloak server, even though it could bring breaking changes to the code, as CodeNOW supports only 10.0.1 version of the Keycloak server. The choice process of the Keycloak version is described in the *3.3.4.1 Keycloak Versions*.

3.11.1 Keycloak version update guide

Prerequisites

- Docker Desktop for Windows

First of all, export the existing realm in Keycloak, it will save time in the future, as all the data from the container will be lost during the version upgrade. Make sure, that you have the theme folder and email templates saved out of the Docker container

Then, update the keycloak service in the *docker-compose* file:

```
keycloak:
 image: quay.io/keycloak/keycloak:21.1.2
 command: start-dev
 container_name: keycloak
...
 volumes:
   - ./custom:/opt/keycloak/themes/custom
```

The Keycloak image by quay.io requires the command field for the server to launch. Also, the path for the custom theme to mount is different.
Then launch the container and Docker Desktop will create the container from the new image.

After the update is finished, multiple tweaks throughout the whole app must be done! For example:

- Update the *pom.xml* file on API and BL components
- Update the Keycloak Adaptor version on the FE component
- The *resteasyClient* in the *keycloak()* bean uses different *ResteasyClientBuilder* - *ResteasyClientBuilderImpl()*.

3.11.2 The Completion of Customer ID Assignment Change

Now, to update the demo app for it to support the new concept of the ID assignment a lot of minor and medium changes have to be done across the application.
First of all, it is necessary to provide functionality for searching users by custom ID from the attribute, casting a Keycloak-generated ID into the custom ID and vice versa. Then rewrite methods across the app to use the custom ID instead of the Keycloak one. The API component will set a random ID for reservations with unknown emails, and then BL will check if a custom ID is new or already exists in the system. In case the custom ID is new, the BL component creates a user account post-factum.

3.11.2.1 API Reservation Controller Update

Starting with the entry point of the reservation request, let's look at the API REST *createReservation* method. It has to check if an email of an anonymous user exists in the keycloak, and if not, set a custom user ID to *newReservationDTO*.

```
...
if(jwt != null){
       String customId = userService.getCustomIdByKeycloakId(jwt.getSubject());
       if(!userService.isSameEmail(newReservationDTO.getEmail(), customId)){...}
       newReservationDTO.setCustomerId(customId);
   }
   else{newReservationDTO
                    .setCustomerId(userService
                           .setUserId(newReservationDTO.getEmail()));
   }
   ReservationDTO reservationDTO = reservationService
                                   .processReservation(newReservationDTO);
...
```

As shown in the snippet above, the *createReservation* now uses Keycloak ID from the token to obtain a custom ID to put in the *newReservationDTO* as customerId. The isSameEmail method, which enforces the reservation to have the same mail as the user account then uses the customId instead of the Keycloak one. If the user is anonymous, then the responsibility for determining what ID to set is passed to the UserService. The implementation of the used methods will be in the *3.11.2.2 API User Service Update*.

Then customers' reservations view endpoint also should be updated to work with a custom ID.

```
public ResponseEntity<?> getViewOfCustomerReservations(...){
...
   if(etag != null) {
       if (reservationService.checkSameEtag(etag)) {...}
       List<ReservationDTO> reservationDTOList = reservationService
                                   .getCustomerReservations(userService
                                          .getCustomIdByKeycloakId(jwt.getSubject()))
                                   ;
       ...}
   else {...
       List<ReservationDTO> reservationDTOList = reservationService
                                   .getCustomerReservations(userService
                                          .getCustomIdByKeycloakId(jwt.getSubject()))
                                   ;
...
```

3.11.2.2 API User Service Update

This section provides the implementation of new methods in the API *UserService* interface.

```
public String getCustomIdByKeycloakId(String id){
   UserResource resource = keycloak.realm(REALM).users().get(id);
   return resource.toRepresentation()
                  .getAttributes()
                  .get("custom_id").get(0);}

public UserRepresentation getUserByCustomId(String customId) {
   List<UserRepresentation> users =
keycloak.realm(REALM).users().searchByAttributes("custom_id:"+customId
);
   if(users.isEmpty()) return null;
   return users.get(0);}

public boolean isSameEmail(String email, String customId){
   UserRepresentation ur = getUserByCustomId(customId);
   if(ur != null){return ur.getEmail().equals(email);}
   return false;}
```

The entry point for the custom ID generation is the *setUserId* function, used in the REST Controller.

```
public String setUserId(String email) {
   List<UserRepresentation> users;
   if((users = getUsersByEmail(email)).isEmpty()){
       return UUID.randomUUID().toString();
   }
   return users.get(0).getAttributes().get("custom_id").get(0);}
```

3.11.2.3 BL Reservation Service Update

The only thing that will remain in the processReservation method of the *ReservationService* is the *userService.checkIfUserExists(newReservationDTO)* method, which will delegate the decision of a new user account creation to the *UserService*. The *checkIfUserExists* function is asynchronous, therefore it should be called early in the processReservation method. The reservation at this point is completed with a valid *customerId*.

3.11.2.4 BL User Service Update

The entry point for the user creation flow is the *checkIfUserExists* function. It should be annotated *@Async* in the interface.

```
public void checkIfUserExists(ReservationDTO reservationDTO){
   if(getUserByAttributeId(reservationDTO.getCustomerId()) != null) return;
   createUserFromReservation(reservationDTO);
}
```

After the Admin Client version upgrade, it is very easy to search users by attributes:

```
public UserRepresentation getUserByAttributeId(String id){
   String query = "custom_id:"+id;
   List<UserRepresentation> users = keycloak.realm(REALM)
                                         .users()
                                         .searchByAttributes(query);
   if(users.isEmpty()) return null;
   return users.get(0);
}
```

The *createUserFromReservation* now has to set the custom ID to attributes correctly:

```
public UserResource createUserFromReservation(ReservationDTO reservation){
       UserRepresentation ur = new UserRepresentation();
       String role = KeycloakRole.customer.name();
       ur.setAttributes(Map.of("custom_id", List.of(reservation.getCustomerId())));
       ur.setUsername(generateUsername(role));
       ...//the rest of standard data set
       Response response = keycloak.realm(REALM).users().create(ur);
       if(response.getStatus() == 201){
          UserRepresentation newUser = getUserByAttributeId(reservation
                                                    .getCustomerId());
          UserResource resource = getUserResourceById(newUser.getId());
          addCustomerRoleToUser(resource);
          sendCredentialsSetupMail(resource);
          return resource;
       }
       else{
           log.warn("ERROR CREATING USER {}. RESPONSE STATUS {}",
                                         ur.getEmail(), response.getStatus());
           return null;}
   }
```

3.11.3 Keycloak on CodeNOW as Custom Container

After the local Keycloak version update, it is necessary to deploy the same new version (21.1.2) of Keycloak on the odeNOW platform.
Even though there is only one version of Keycloak provided by CodeNOW explicitly - 10.0.1, it is possible to deploy any version of a custom Docker image, using a *Docker/Generic* Component scaffolder.
The detailed instructions for the initial setup of a custom container can be found in the Custom Docker image tutorial[76]. The difficulty is that there is no ready, tested Dockerfile for Keycloak to run on CodeNOW, and the creation of it requires a certain level of experience in working with Docker.

As for now, this stage of the project remains in progress and will be completed in the near future.

3.12 Implementation Results

Most of the points, planned for the implementation part are complete or researched. There was more emphasis on the local implementation, outrunning the development on the CodeNOW platform, nonetheless, the pilot versions of the demo app were launched and tested there, which provided insights regarding the demo app development and CodeNOW functionality itself.

3.12.1 Local development

- ✔ The Keycloak is fully integrated with demo app components.
    - ✓ The basis for API REST endpoint security using Spring Security is fully configured.
    - ✓ Keycloak and the demo app use business roles (administrator and customer)
    - ✓ The Keycloak themes and email templates are customized for the demo app.
- ✔ The Back end is secured
    - ✓ The API component has general-level security (*@EnableWebSecurity interface*)
    - ✓ The API REST Controller has method-level security (*@EnableGlobalMethodSecurity* interface)
- ✔ The Front end is secured
    - ✓ React components dynamically change depending on status and user roles
    - ✓ Provided the flow for user login and self-registration
    - ✓ Configured CORS
- ✔ Added functionality to manage users.
    - ✓ The BL component is integrated with Keycloak for user management
    - ✓ User accounts are created based on their reservation
    - ✓ Users obtain custom emails for the account setup
    - ✓ Added back-end functionality for reading customer reservations
    - ✓ Created missing View components on the front end component (customer reservations view and all reservations view)

- ✖ The work with Etag and cache on the front-end component is not finished
- ✖ Some polishing of method-level security on API REST Controller might be done
- ✖ Further work on protection against CSRF attacks required
- ✖ Migration of the Keycloak inbuild database to PostgreSQL to be prod-friendly required.

3.12.2 CodeNOW development

- ✔ The app successfully launched on the CodeNOW platform using Keycloak as a managed service.
- ✔ The custom theme for Keycloak was applied.
- ✔ Found a deadlock on CodeNOW in the Custom Theme interface for Keycloak.

- ✖ The custom theme on the Keycloak managed service requires a fix and aesthetical polishing, due to version incompatibility.
- ✖ The Keycloak version update on CodeNOW was not finished
    - ✕ The docker file for the 21.1.2 Keycloak server in a custom container does not work yet.

Charter IV

Conclusion

In this work, all points of the assignment have been covered.

The Keycloak system was integrated into the Ticker Reservation application for authentication and authorization utilizing role-based access control (RBAC) through JWT tokens.

The back-end REST API was secured with Keycloak and Spring Security integration.

The front end was secured using the Keycloak-js adapter and extended with an interface adapting to a user's authorization status and roles.

The Ticket Reservation application was extended with user management functionality (user creation, account setup, etc.) It was enhanced with additional functionalities (like customers' reservations view, navbar login-related elements, etc.) along with bug fixes.

In the early development iterations, it was deployed on the CodeNOW platform. In the later stages, the Keycloak version upgrade was required, which is not explicitly supported by CodeNOW yet. In order to use a certain Keycloak version, a custom docker component should be created, which was not finished in the scope of this work. The implementation of this task will be completed within the next months during work for CodeNOW.

The new version of the Ticket Application is still on track to becoming production-ready software, requiring work on remaining bugs and front-end aesthetics. The key points to improve are: work with cache (ETag), front-end React components look and functionality, and migrating from the managed Keycloak service to a custom Keycloak container on CodeNOW for future development.

This work described all the steps for the project setup as well as the implementation journey in eight iterations. The guides were intended to be as detailed as possible so that even an inexperienced reader could follow them. Additionally, it aims to act as a foundation for new tutorials and manuals for CodeNOW documentation. As a result, this work exceeded the recommended volume of 60 pages. Nonetheless, it is necessary to achieve the set goals described above.

Personally, at the beginning of this journey, my knowledge of web security was extremely limited. Through the research and implementation phases, I learned a broad spectrum of topics, gaining familiarity and deeper understanding of various technologies and software - security protocols (OAuth, OIDC, CORS…), patterns (CQRS, caching with ETags…), as well as Keycloak, Redis, Docker, Kafka, and many more, mostly mentioned in the *3.2.1 Technologies and Patterns* sections.

Personally, this project has significantly improved my skills and equipped me to face real-life challenges.

5 Bibliography

1 *Introduction: A Little Bit of History*, September 05, 2007. Online. OAuth 2.0. Available at: *oauth.net/about/introduction/*.

2 *What is OAuth 2.0?*, ©2024. Online. OCTA, INC. Auth0 by Okta. Available at: *auth0.com/intro-to-iam/what-is-oauth-2*.

3 *What is OpenID Connect*, 2023. Online. OPENID FOUNDATION. OpenID. Available at: *openid.net/developers/how-connect-works/*.

4 *OAuth 2.1*, 2023. Online. OAuth 2.0. Available at: *oauth.net/2.1/*.

5 *What is Authentication?*, ©2024. Online. Auth0 by Okta. Available at: *auth0.com/intro-to-iam/what-is-authentication*.

6 *What is Authorization?*, ©2024. Online. Auth0 by Okta. Available at: *auth0.com/intro-to-iam/what-is-authorization*.

7 OKTA, INC, ©2024. *Auth0*. Online. Available at: *https://auth0.com/docs/*.

8 *Identity Glossary*, ©2024. Online. Auth0 by Okta. Available at: *auth0.com/docs/glossary*.

9 *What is Role-Based Access Control (RBAC)?: Examples, Benefits, and More*, 2023. Online. Fortra. Available at: *www.digitalguardian.com/blog/what-role-based-access-control-rbac-examples-benefits-and-more*.

10 *RFC 6749 Section 2.1: OAuth 2.0 Client Types*, ©2023. Online. OAuth 2.0. Available at: *oauth.net/2/client-types/*.

11 *OAuth 2.0 Authorization Code Grant*, ©2023. Online. OAuth 2.0. Available at: *https://oauth.net/2/grant-types/authorization-code/*.

12 *RFC 7636: Proof Key for Code Exchange*, ©2023. Online. OAuth 2.0. Available at: *https://oauth.net/2/pkce/*.

13 *OAuth 2.0 Client Credentials Grant*, ©2023. Online. OAuth 2.0. Available at: *oauth.net/2/grant-types/client-credentials/*.

14 *OAuth 2.0 Device Authorization Grant*, ©2023. Online. OAuth 2.0. Available at: *oauth.net/2/grant-types/device-code/*.

15 *OAuth 2.0 Refresh Token*, ©2023. Online. OAuth 2.0. Available at: *oauth.net/2/grant-types/refresh-token/*.

16 *OAuth 2.0 Implicit Grant*, ©2023. Online. OAuth 2.0. Available at: *oauth.net/2/grant-types/implicit/*.

17 *OAuth 2.0 Password Grant*, ©2023. Online. OAuth 2.0. Available at: *oauth.net/2/grant-types/password/*.

18 *OpenID Connect Core 1.0 incorporating errata set 2*, 2023. Online. Available at: *https://openid.net/specs/openid-connect-core-1_0.html*.

19 RICHARDSON, Chris, [2019]. 11.1.2 Implementing security in a microservice architecture: Handling Authorization. In: *Microservices patterns: with examples in Java*. Shelter Island: Manning, p. 356.

20 RICHARDSON, Chris, [2019]. 11.1.2 Implementing security in a microservice architecture: Using JWT to Pass User Identity and Roles. In: *Microservices patterns: with examples in Java*. Shelter Island: Manning, p. 356-357.

21 *The Twelve-Factor App*, 2017. Online. Available at: *https://12factor.net*.

22 *Keycloak*, ©2024. Online. THE LINUX FOUNDATION. Cloud Native Computing Foundation. Available at: *www.cncf.io/projects/keycloak/*.

23 DARIMONT, Thomas, 2005. *How to secure your Microservices with Keycloak: Server Architecture*. Online. In: YouTube. Available at: *www.youtube.com/watch?v=FyVHNJNriUQ*. Timestamp 24:23.

24 *JGroups - A Toolkit for Reliable Messaging*, ©2002-2035. Online. Available at: *www.jgroups.org/*.

25 *Infinispan*, [2020]. Online. Available at: *infinispan.org*.

26 *Spring Framework*, ©2005-2024. Online. Spring. Available at: *spring.io/projects/spring-framework/*.

27 *Spring Boot*, ©2005 - 2024. Online. Spring. Available at: *spring.io/projects/spring-boot/*.

28 *Spring Security Architecture*, ©2005 - 2024. Online. Spring. Available at: *spring.io/guides/topicals/spring-security-architecture/#web-security*.

29 PARASCHIV, Eugen, 2023. *Spring Security Authentication Provider*. Online. In: Baeldung. Available at: *www.baeldung.com/spring-security-authentication-provider*.

30 *Servlet Authentication Architecture*, ©2005 - 2024. Online. Spring. Available at: *docs.spring.io/spring-security/reference/servlet/authentication/architecture.html*.

31 *The CodeNOW Platform*. Online. CODENOW, INC. CodeNOW. Available at: *www.codenow.com/platform*.

32 CODENOW, INC, ©2022. *CodeNOW Documentation*. Online. Available at: *docs.codenow.com*.

33 *CQRS pattern*, ©2023. Online. Cloud Computing Services - Amazon Web Services (AWS). Available at: *docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/cqrs-pattern.html*.

34 *ETag*, ©1998–2024. Online. MDN Web Docs. Available at: *developer.mozilla.org/en-US/docs/Web/HTTP/Headers/ETag*.

35 *Cross-Origin Resource Sharing (CORS)*, ©1998–2024. Online. MDN Web Docs. Available at: *https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS*.

36 OBJECT MANAGEMENT GROUP, INC., ©1997 - 2023. BPMN Specification - Business Process Model and Notation. Online. Available at: www.bpmn.org/.

37 RICHARDSON, Chris, [2019]. Microservices patterns: with examples in Java. Shelter Island: Manning.

38 RICHARDSON, Chris, [2019]. 11.1.2 Implementing security in a microservice architecture: Handling authentication in the API gateway. In: Microservices patterns: with examples in Java. Shelter Island: Manning, p. 354.

39 *Java Spring Boot Local Development*. Online. CODENOW, INC. CodeNOW Documentation. Available at: *docs.codenow.com/local-development/java-spring-boot-local-development*.

40 *Install Docker Desktop on Windows*, ©2013-2024. Online. Docker Docs. Available at: *docs.docker.com/desktop/install/windows-install/*.

41 *Compose file*, ©2013-2024. Online. DOCKER INC. Docker Docs. Available at: *docs.docker.com/compose/compose-file/03-compose-file/*.

42 *Volumes*, ©2013-2024. Online. Docker Docs. Available at: *docs.docker.com/storage/volumes/*.

43 *Compose file version 3 reference: Ports*, ©2013-2023. Online. DOCKER INC. Docker Docs. Available at: *docs.docker.com/compose/compose-file/compose-file-v3/#ports*.

44 *Keycloak Admin REST API*, ©2023. Online. Keycloak. Available at: *www.keycloak.org/docs-api/21.1.2/rest-api/index.html*.

45 *Connect Service to Component*, ©2023. Online. CODENOW, INC. CodeNOW Documentation. Available at: *docs.codenow.com/advanced-features/component-service-connection*.

46 OBSIDIAN DYNAMICS, ©2023. *Kafdrop: README*. Online. GITHUB, INC. GitHub. Available at: *github.com/obsidiandynamics/kafdrop#readme*.

47 *External Services*. Online. CODENOW, INC. CodeNOW Documentation. Available at: *docs.codenow.com/services/external*.

48 THORGERSEN, Stian, ©2023. *Deprecation of Keycloak adapters*. Online. THE LINUX FOUNDATION. Keycloak. Available at: *www.keycloak.org/2022/02/adapter-deprecation.html*.

49 *Spring Security*, ©2005 - 2023. Online. VMWARE, INC. Spring. Available at: *docs.spring.io/spring-security/reference/index.html*.

50 JONES, Michael B., May 2015. *JSON Web Key (JWK)*. Online. OPENID FOUNDATION. OpenID. Available at: *datatracker.ietf.org/doc/html/rfc7517*.

51 *Core Configuration: Setting the Redirect URI*, ©2023. Online. VMWARE, INC. Spring. Available at: *docs.spring.io/spring-security/reference/servlet/oauth2/login/core.html#oauth2login-sample-redirect-uri*.

52 *Introduction to Spring Method Security*, [2020]. Online. TARNUM JAVA SRL. Baeldung. Available at: *www.baeldung.com/spring-security-method-security*.

53 *Method Security*, ©2005 - 2023. Online. VMWARE, INC. Spring. Available at: *docs.spring.io/spring-security/reference/servlet/appendix/namespace/method-security.html*.

54 *Securing Applications and Services Guide: Keycloak JavaScript adapter*, ©2023. Online. KEYCLOAK AUTHORS. Keycloak. Available at: *www.keycloak.org/docs/latest/securing_apps/index.html#_javascript_adapter*.

55 KÖBLER, Niko, ©2012-2023. *Keycloak & React.JS & Router Integration How to*. Online. In: Niko Köbler – Keycloak Experte, Software-Architekt & Trainer. Available at: *www.n-k.de/2021/01/keycloak-react-router-integration-how-to.html*.

56 KÖBLER, Niko, ©2023. *Keycloak-reactjs-demo*. Online. GITHUB, INC. GitHub. Available at: *github.com/dasniko/keycloak-reactjs-demo*.

57 YATES, Roger, [2020]. *Spring @EnableWebSecurity vs. @EnableGlobalMethodSecurity*. Online. In: Baeldung. Available at: *https://www.baeldung.com/spring-enablewebsecurity-vs-enableglobalmethodsecurity*.

58 *Class HttpSecurity*, ©2005 - 2023. Online. VMWARE, INC. Spring. Available at: *docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/builders/HttpSecurity.html*.

59 *Basic Authentication*, ©2005 - 2023. Online. VMWARE, INC. Spring. Available at: *docs.spring.io/spring-security/reference/servlet/authentication/passwords/basic.html*.

60 *Server Administration Guide: Using a service account*, ©2005 - 2023. Online. Keycloak. Available at: *www.keycloak.org/docs/latest/server_admin/#_service_accounts*.

61 *Interface UsersResource*, ©2005 - 2023. Online. OPENID FOUNDATION. Keycloak. Available at: *www.keycloak.org/docs-api/21.0.0/javadocs/org/keycloak/admin/client/resource/UsersResource.html*.

62 *Class UserRepresentation*, ©2005 - 2023. Online. OPENID FOUNDATION. Keycloak. Available at: *www.keycloak.org/docs-api/21.1.1/javadocs/org/keycloak/representations/idm/UserRepresentation.html*.

63 *1. Working with Spring Data Repositories*, ©2005 - 2024. Online. Spring. Available at: *docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html*.

64 *Sign in with app passwords*, ©2024. Online. Google Account Help. Available at: *support.google.com/accounts/answer/185833?hl=en*.

65 *Interface RequiredActionProvider*, ©2023. Online. JBOSS BY RED HAT. Keycloak Docs Distribution 21.0.0 API. Available at: *www.keycloak.org/docs-api/21.0.0/javadocs/org/keycloak/authentication/RequiredActionProvider.html*.

66 *Server Administration Guide: Defining actions required at login*, ©2005 - 2023. Online. Keycloak. Available at: *www.keycloak.org/docs/21.1.2/server_admin/index.html#con-required-actions_server_administration_guide*.

67 *Custom theme for Keycloak*, ©2022. Online. CodeNOW Documentation. Available at: */docs.codenow.com/custom/keycloak-theme*.

68 *FREEMAKER*, ©1999–2023. Online. Available at: *freemarker.apache.org/index.html*.

69 WAGDE, Sampada, [2020]. *Customizing Themes for Keycloak*. Online. Baeldung. Available at: *www.baeldung.com/spring-keycloak-custom-themes*.

70 MUNROE, Lee, ©2023. *Free Responsive HTML Email Template*. Online. GitHub. Available at: *github.com/leemunroe/responsive-html-email-template*.

71 *Server Developer Guide: Service Provider Interfaces (SPI)*, ©2005 - 2023. Online. Keycloak. Available at: *www.keycloak.org/docs/21.1.2/server_development/index.html#_providers*.

72 *Securing Applications and Services Guide: 2023-12-15*, ©2005 - 2023. Online. Keycloak. Available at: *www.keycloak.org/docs/latest/securing_apps/#using-the-adapter*.

73 *Moment.js | Docs*, [2020]. Online. Available at: *https://momentjs.com/*.

74 *Allow CORS: Access-Control-Allow-origin*, 2015. Online. Allow CORS: Access-Control-Allow-origin. Available at: *mybrowseraddon.com/access-control-allow-origin.html*.

75 VIKASH, Divya, 2023. *SpringBoot @Async: The magic and the gotchas*. Online. In: Medium. Available at: *medium.com/@dvikash1001/springboot-async-the-magic-and-the-gotchas-17f9471c6fe4*.

76 *Custom docker image*, ©2022. Online. CODENOW, INC. CodeNOW Documentation. Available at: *docs.codenow.com/custom/docker-image*.

6 List of Images

Appendix A

Original Code Sources

The production versions of the components can be cloned from the following repositories:

Front end - FE component:
https://gitlab.cloud.codenow.com/public-docs/ticket-reservation-demo/spring-boot/microservice-patterns/fe-reservation-spring

Back end - BE component:
https://gitlab.cloud.codenow.com/public-docs/ticket-reservation-demo/spring-boot/microservice-patterns/bl-reservation-spring

API component:
https://gitlab.cloud.codenow.com/public-docs/ticket-reservation-demo/spring-boot/microservice-patterns/api-reservation-spring

Schedules component:
https://gitlab.cloud.codenow.com/public-docs/ticket-reservation-demo/spring-boot/cloud-native/ticket-reservation-schedules