FH University of Applied Sciences TECHNIKUM WIEN

TECHNICAL DOCUMENTATION

Low-Code vs. Traditional Development: Effort and Performance

By: Stefan Bittgen, BSc

Student Number: 2310299003

Supervisor: Priv. Doz. Dipl.-Ing. Dr. Karl M. Göschka

May 6, 2025

Executive Summary

In this thesis, the low-code platform Mendix is examined in comparison to traditional approaches such as Java/Angular using criteria such as development effort, complexity, scope of application and performance. Traditional programming languages are often associated with long development cycles and a lack of qualified specialists, which results in high development and personnel costs. Low-code platforms, on the other hand, promise shorter development times and minimize the need for specialized programmers. According to Gartner, a leading research and consulting company, around 75% of all new software applications will be developed using low-code technologies by 2026 [1]. The driving factors are the increasing pressure to automate and the growing shortage of skilled workers.

Current status

Three standardized use cases were partial created and developed in Mendix and Java/Angular. The documentation of the individual work steps in the respective environment is provided in this document under *Appendices*.

- 1. User registration and login
  First Iteration: Basic Authentication
  Second Iteration: With Oauth

- 2. CRUD operations through a Crypto Dashboard
  First Iteration: Dashboard and Details Page

- Task & Planning (Kanban Board)
  First Iteration: Create Tasks

One of the main difficulties this paper addresses is the comparison of a model-driven platform like Mendix with an object-oriented programming language like Java Spring Boot and a front-end framework like Angular. While Mendix simplifies development through abstraction and speed and enables rapid prototyping, Spring Boot and Angular offer full control and flexibility - but at the cost of greater complexity. With Mendix, it is possible to create applications in a short time, provided the user has the necessary knowledge and experience with the platform. In contrast, development with Java/Angular often requires a longer development time, as projects usually have to be programmed from scratch. However, Java and Angular benefit from a broad developer community and extensive support on platforms such as Stack Overflow, where solutions are available for almost every problem. There are also resources and community support for Mendix, but these are hardly comparable in scope and variety to the support provided by open-source communities. This makes it difficult for less experienced users in particular to solve problems and share knowledge in connection with Mendix.

In summary, while Mendix focuses on speed and abstraction, Java and Angular offer greater flexibility and customizability for more demanding scenarios. However, Mendix also requires specialized experts to use the platform effectively and efficiently - without this knowledge, the benefits of low-code cannot be fully realized.

Project objectives and goals

The research question is: How does the development effort and performance differ between a low-code implementation (Mendix) and a traditional development (Spring Boot/Angular)?

- This project report documents the various steps required to implement the use cases and iterations.
- For Java/Angular, SonarQube is used to read metrics such as cyclomatic complexity.
- With Mendix SDK it is possible to write scripts to evaluate microflows, nanoflows, widgets and pages. This makes it possible to determine how many of the different components are used in Mendix. A cyclomatic complexity can also be calculated with Mendix SDK.
- JMeter is used for performance tests.
- The plan is to write down the measurements in tables.

- **Challenges during data acquisition:**
  The scripts for Mendix SDK still need to be adapted/refined to obtain the desired result.

Development and Implementation:

The use cases under consideration have already been implemented and the associated measurement data from the projects is available. SonarQube, Mendix SDK and JMeter were used to collect the data. While SonarQube can collect the relevant metrics largely automatically, the use of the Mendix SDK and JMeter in the context of this work requires the independent implementation of certain components as well as the manual execution and validation of the measurements.

Lessons Learned

A key result of using Mendix is the realisation that the frequent assumption that low-code development is possible entirely without programming experience is only true to a limited extent. Although the author of this work has several years of experience with traditional high-level languages and Mendix, the realisation of various projects has shown that basic technical knowledge is essential.

For example, working with Mendix requires an understanding of entities, their relationships and the basic principles of data modelling and processing. The platform supports both an embedded database and the connection of external data sources, for example via APIs. This requires appropriate expertise - particularly with regard to modelling and querying data. In addition, questions arise in practice, such as the transfer of objects between pages or the selection of suitable integration strategies. These challenges make it clear that programmatic thinking is also required when working with low-code platforms - the relevant expertise cannot be completely abstracted.

In addition, Mendix offers the option of integrating your own Java or JavaScript actions in order to implement functionalities that go beyond the standard scope of the platform. Java actions are typically used for complex business logic or connecting external systems, while JavaScript actions are primarily used for client-side customisations or UI-specific functions. However, such extensions require in-depth knowledge of the respective programming languages. This underlines the fact that the effective use of Mendix does not replace technical expertise, but rather builds a bridge between model-based and classic software development.

Another critical aspect concerns the long-term benefits of low-code platforms such as Mendix. The platform follows a model-driven development (MDD) approach and translates the models created directly into machine code without allowing access to the generated source code. This raises questions about sustainability: What happens if Mendix is no longer supported in the future? How can

existing applications be migrated? In any case, specialised personnel are needed to regularly maintain and further develop applications, which requires a corresponding build-up of knowledge.

Appendices

Use Case 1 - Mendix (v 10.12.0)

The aim of the implementation is to enable users to register and log in, as well as to implement secure password management. Users should be able to enter their full name and e-mail address. After successful registration, the user receives an e-mail with a confirmation link. By clicking on the confirmation link, the user is activated in the database and can access the application. The passwords are stored securely in the database as BCrypt hashes. Mendix uses its own **Hybrid Server-Side Query Language database (HSSQL)** by default, but alternative database types such as PostgreSQL and Microsoft SQL Server are also supported [1].

Prerequisites**:**

- Mendix Studio Pro (v 10.12.0)

Modules:

- Forgot Password module

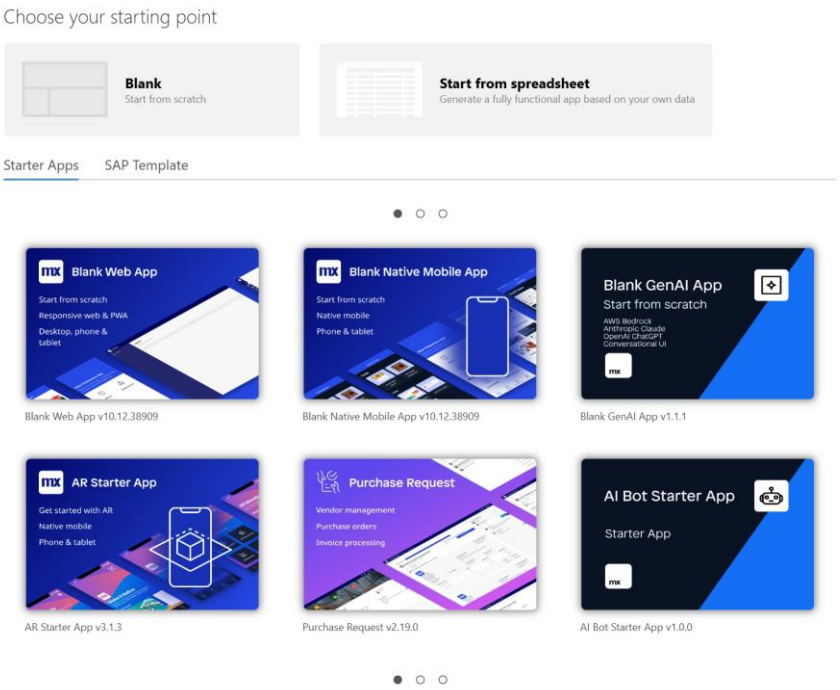
Dependencies:

- Email Connector
- Encryption
- Mx Model Reflection

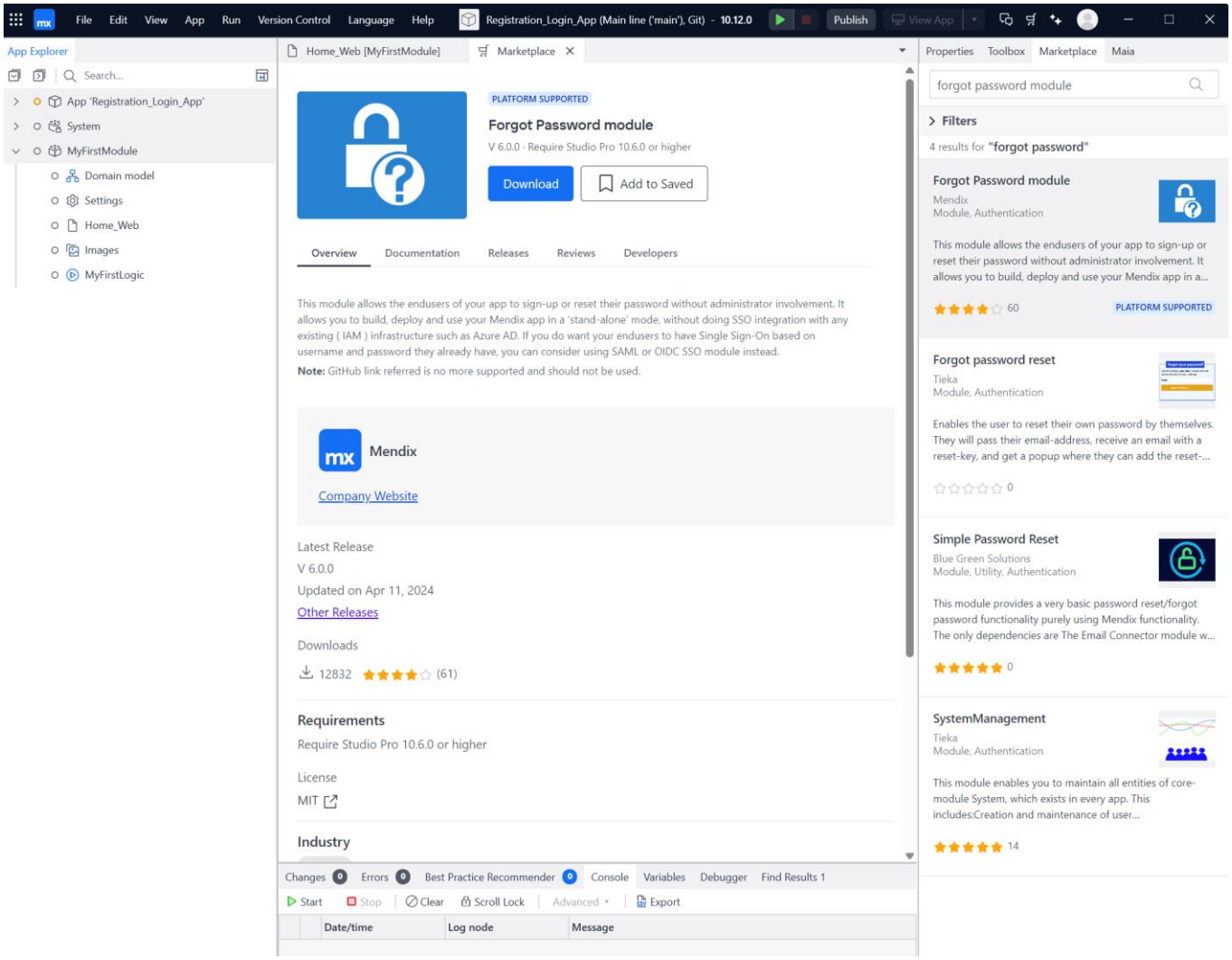
Implementation:

An already finished and official implementation of Mendix could be found and used for this use case [2]. Finished implementations and dependencies are called modules in Mendix, in this case it is the 'Forgot Password module'.

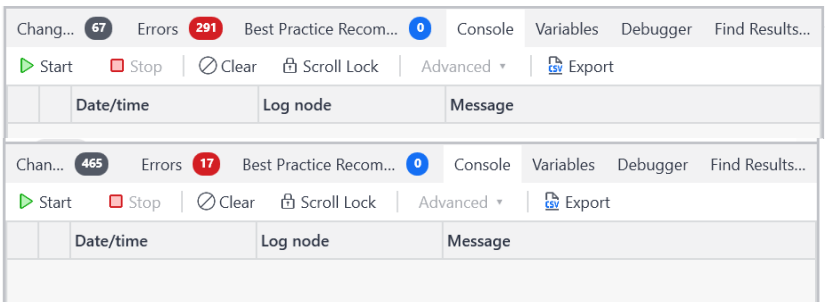
1. The first step is to check whether the current installed Mendix version is compatible with the module used. This version is continuously updated as described on the already documented page[2] of Mendix. There are some steps in the official documentation from the manufacturer that are not mentioned here, as these only apply to older Mendix versions and modules. After ensuring that this module is compatible with the current Mendix version, a "Blank Web App" is selected, and the project is created.



Download and install the 'Forgot Password module' in the Mendix app via the Marketplace.

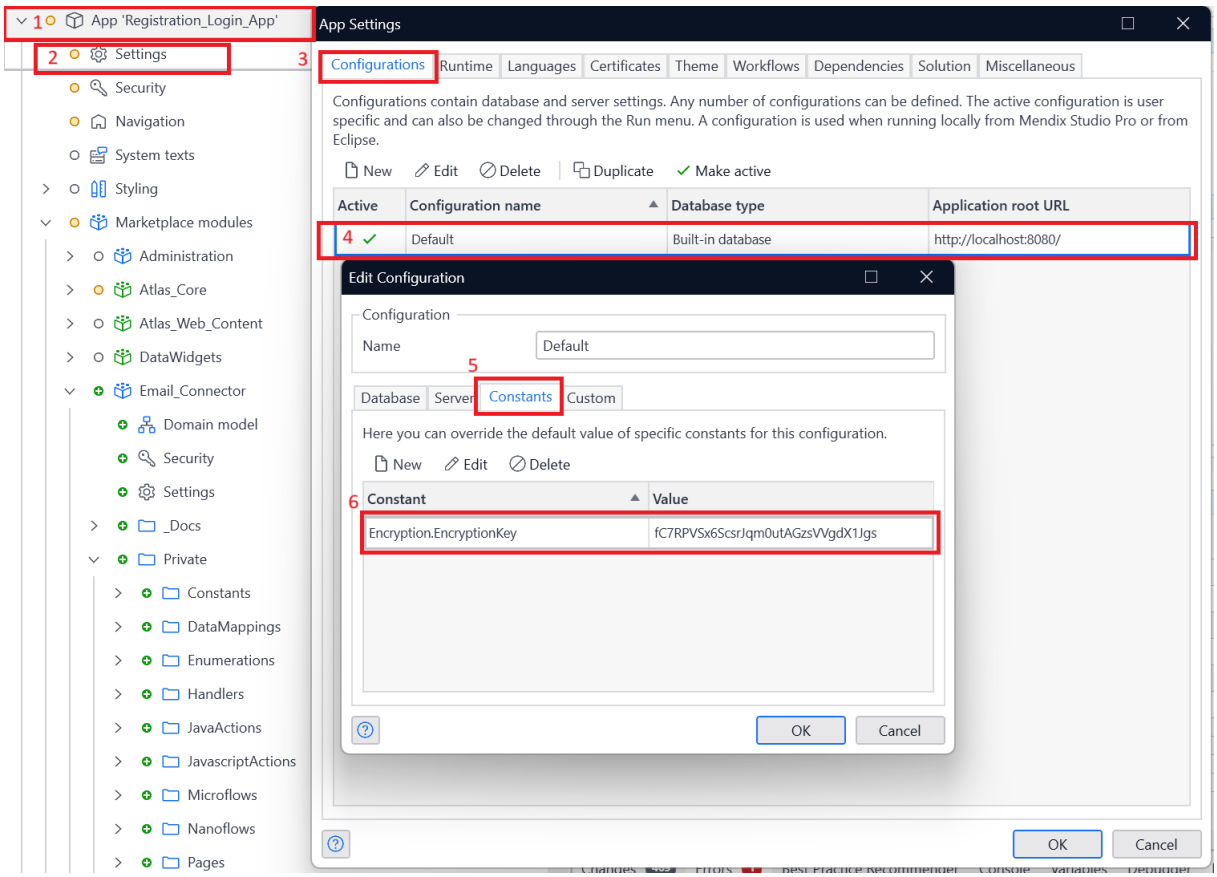


After this has been done, you suddenly receive 291 errors in the IDE, although you receive a successful response from the application. This is because the other required dependencies have not yet been installed. These can also be installed via the Marketplace, and the number of errors drops again quite quickly once these have been integrated. The
remaining 17 errors are resolved in the following steps.

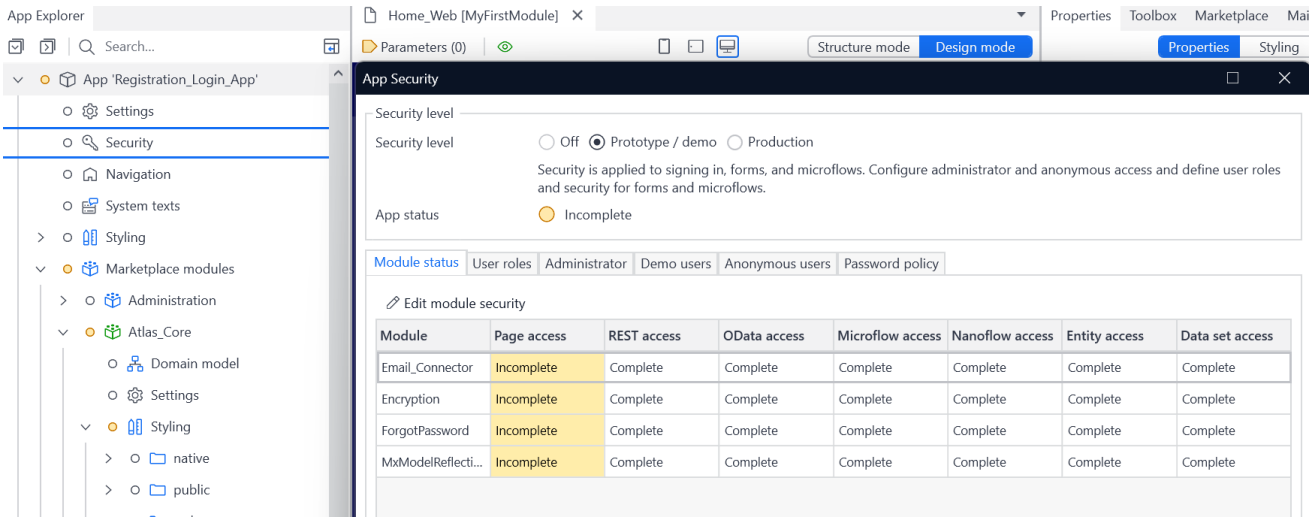


2. In the App Explorer under Configurations, the existing configuration is set with a 32-character string value, in this documentation 'fC7RPVSx6ScsrJqm0utAGzsVVgdX1Jgs' is used. The constant is used to encrypt data securely. This key is loaded when the application is started and used in the encryption methods. This token must be stored under "App -> Settings -> Configurations -> Select Configuration

-> Constants". Please note that this key is strictly confidential and must not be shared under any circumstances, whether in public or insecure environments. This is for demonstration purposes only.

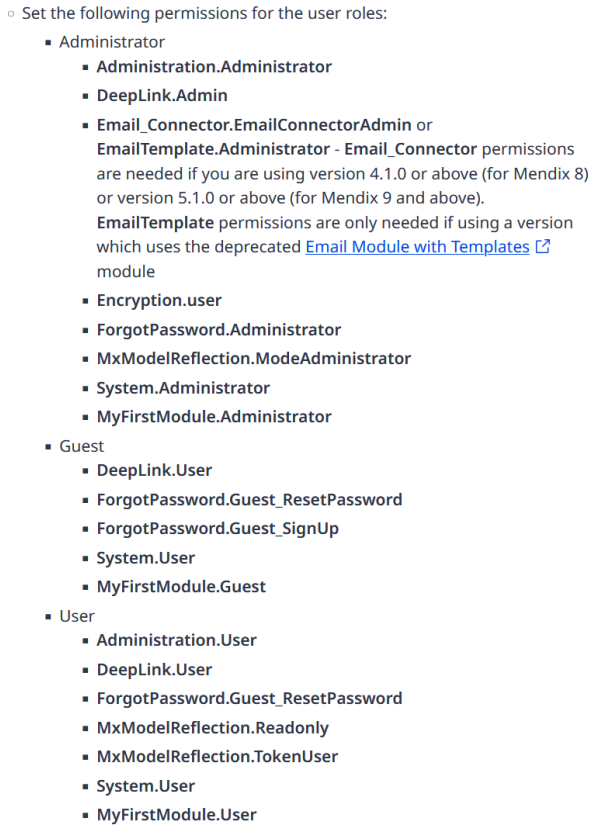


3. Open the App Security Settings and add a new role 'Guest' and switch the security level from 'Off' to 'Prototype / demo'. The authorizations must be adjusted for the users Administrator, User and Guest (see screenshot).

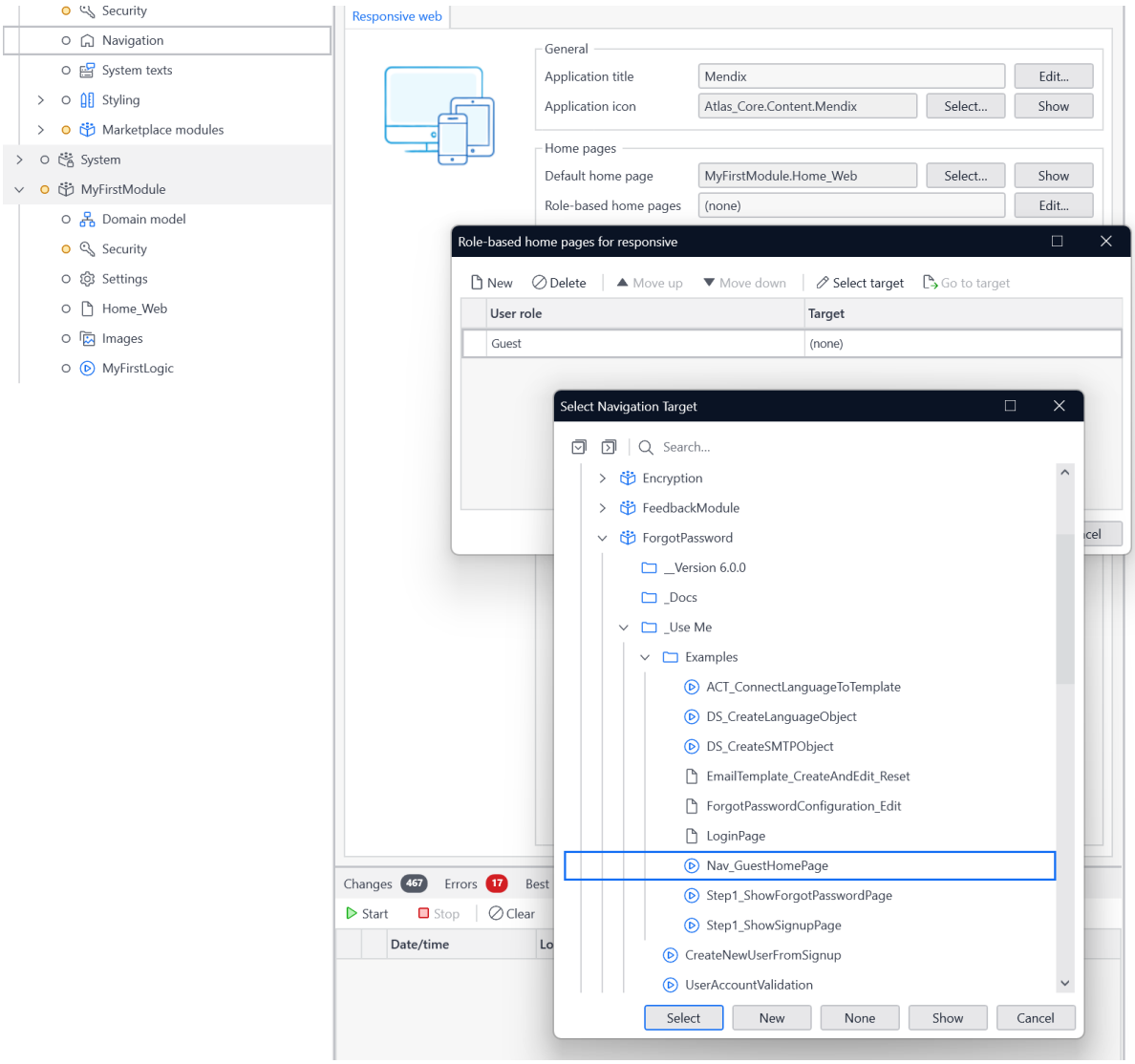


Here is a screenshot from the documentation[2], these must be added to the corresponding user roles. The DeepLink authorizations are excluded from this, as this module is not in use. In the

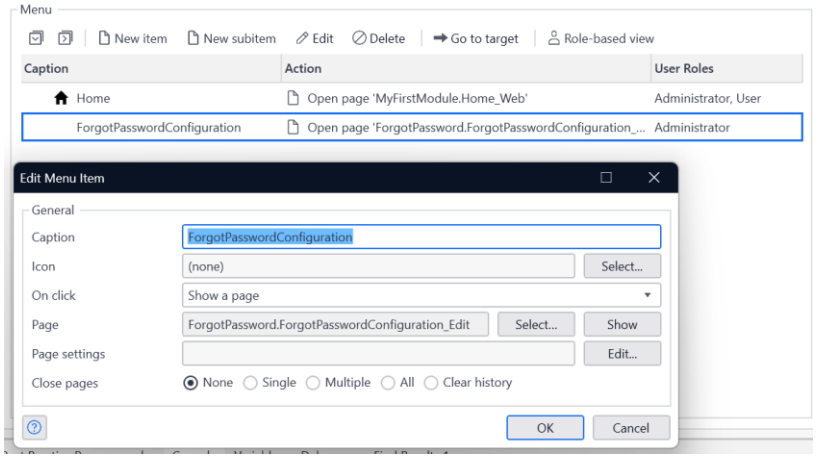
'Anonymous users' tab, the setting 'Allow anonymous user' must be selected with 'yes'.



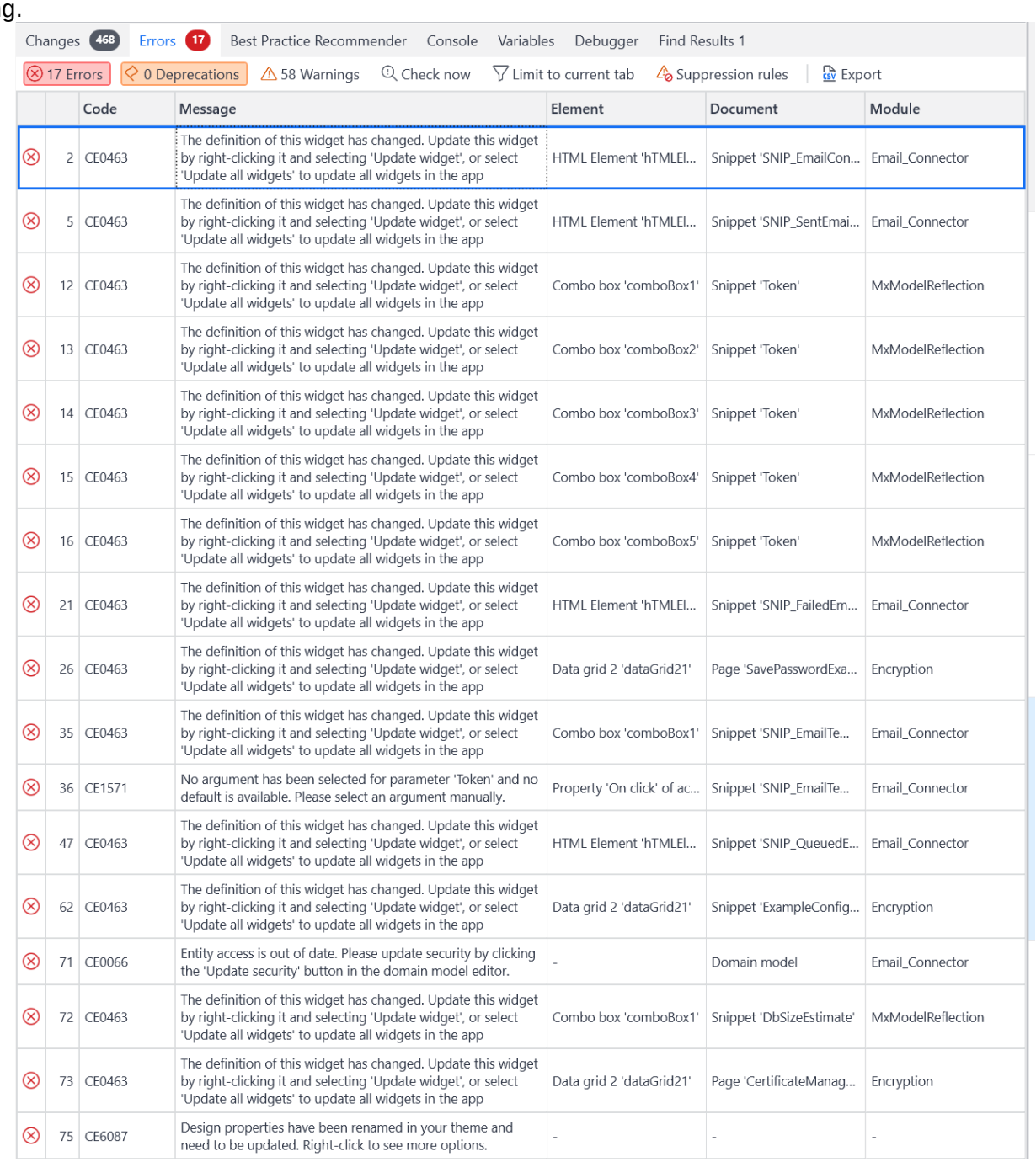
4. Now select 'Navigation' in the App Explorer and select 'Guest' as the user role in the 'Set Role-based home pages' window. A microflow for anonymous users is stored for the home page. This triggers either the login page or the password reset process.



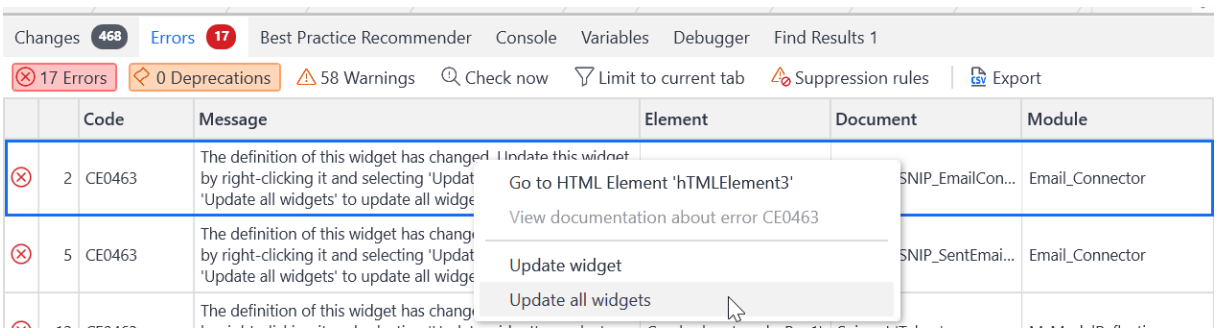
5. Next, another page is added to 'Navigation'. Select "New item" and a new window will open. As 'Caption' the documentation suggests 'ForgotPasswordConfiguration', under 'On click' the 'ForgotPassword.ForgotPasswordConfiguration_Edit' with the Administrator user role is added.

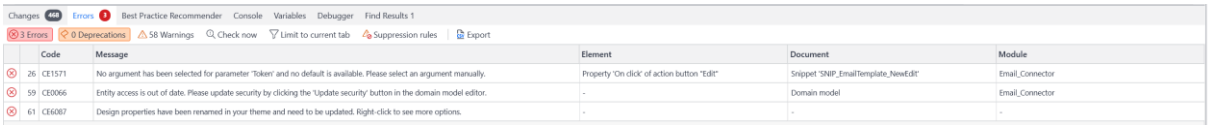


6. In the next step, the official documentation states that the application can be started. However, this is currently not possible as the 17 errors from before are still open and prevent the application from starting.

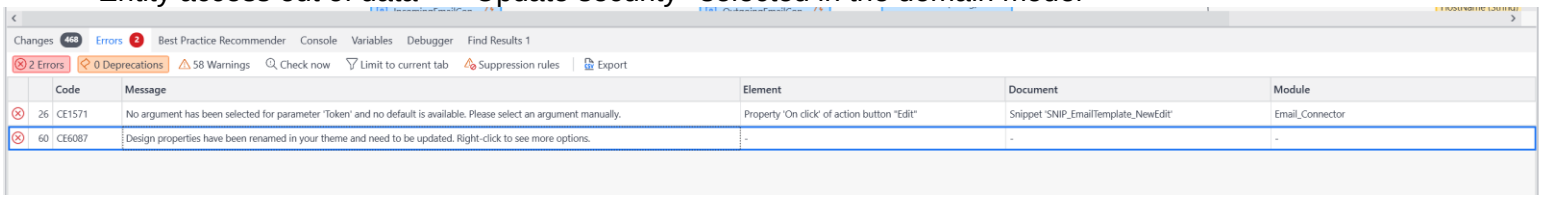


		Code	Message	Element	Document	Module
⊗	2	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	HTML Element 'hTMLEl...	Snippet 'SNIP_EmailCon...	Email_Connector
⊗	5	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	HTML Element 'hTMLEl...	Snippet 'SNIP_SentEmai...	Email_Connector
⊗	12	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Combo box 'comboBox1'	Snippet 'Token'	MxModelReflection
⊗	13	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Combo box 'comboBox2'	Snippet 'Token'	MxModelReflection
⊗	14	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Combo box 'comboBox3'	Snippet 'Token'	MxModelReflection
⊗	15	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Combo box 'comboBox4'	Snippet 'Token'	MxModelReflection
⊗	16	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Combo box 'comboBox5'	Snippet 'Token'	MxModelReflection
⊗	21	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	HTML Element 'hTMLEl...	Snippet 'SNIP_FailedEm...	Email_Connector
⊗	26	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Data grid 2 'dataGrid21'	Page 'SavePasswordExa...	Encryption
⊗	35	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Combo box 'comboBox1'	Snippet 'SNIP_EmailTe...	Email_Connector
⊗	36	CE1571	No argument has been selected for parameter 'Token' and no default is available. Please select an argument manually.	Property 'On click' of ac...	Snippet 'SNIP_EmailTe...	Email_Connector
⊗	47	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	HTML Element 'hTMLEl...	Snippet 'SNIP_QueuedE...	Email_Connector
⊗	62	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Data grid 2 'dataGrid21'	Snippet 'ExampleConfig...	Encryption
⊗	71	CE0066	Entity access is out of date. Please update security by clicking the 'Update security' button in the domain model editor.	-	Domain model	Email_Connector
⊗	72	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Combo box 'comboBox1'	Snippet 'DbSizeEstimate'	MxModelReflection
⊗	73	CE0463	The definition of this widget has changed. Update this widget by right-clicking it and selecting 'Update widget', or select 'Update all widgets' to update all widgets in the app	Data grid 2 'dataGrid21'	Page 'CertificateManag...	Encryption
⊗	75	CE6087	Design properties have been renamed in your theme and need to be updated. Right-click to see more options.	-	-	-





Entity access out of data -> ' Update security ' selected in the domain model



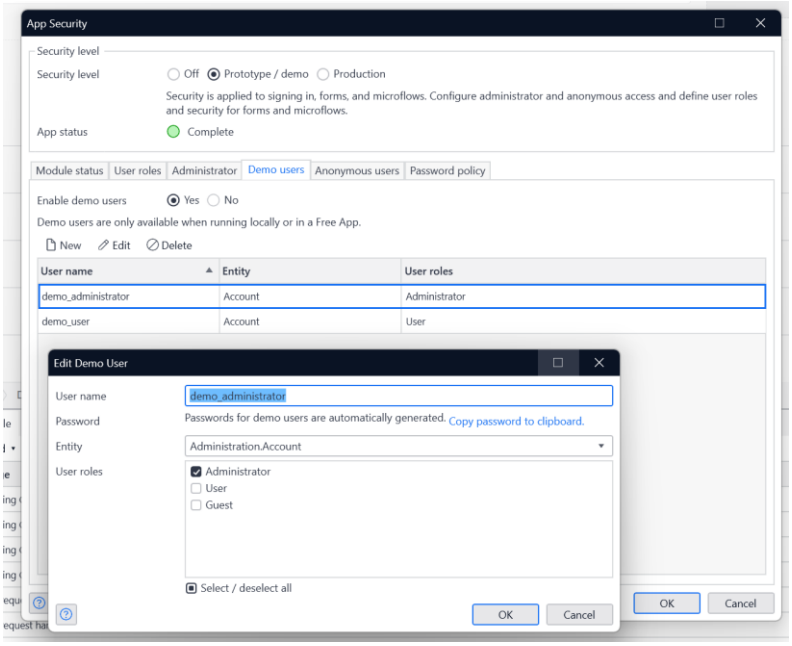
Design properties have been renamed in your theme and need to be updated -> Right click -> update all renamed design properties in project.

7. One error remains in the next step, and that is **CE1571: No argument has been selected for parameter 'Token' and no default is available. Please select an argument manually.**

After extensive testing and multiple checks of the configurations, no explanation for the cause of the error could be found in the official documentation. As a final measure, the error-causing **on-click event** was temporarily deactivated, the changes were saved and then the event was reactivated. The desired page "**MxModelReflection.Token_NewEdit**" was explicitly selected again. After these steps, the error no longer occurred. It is unclear why the error **"No argument has been selected for parameter 'Token'"** occurred, as all configurations were checked several times and implemented correctly. No reference to possible causes or typical triggers was found in the official documentation. The error could have been caused by **caching problems** or an **inconsistency in the internal configuration** of the Mendix development environment. This can occur, for example, if:

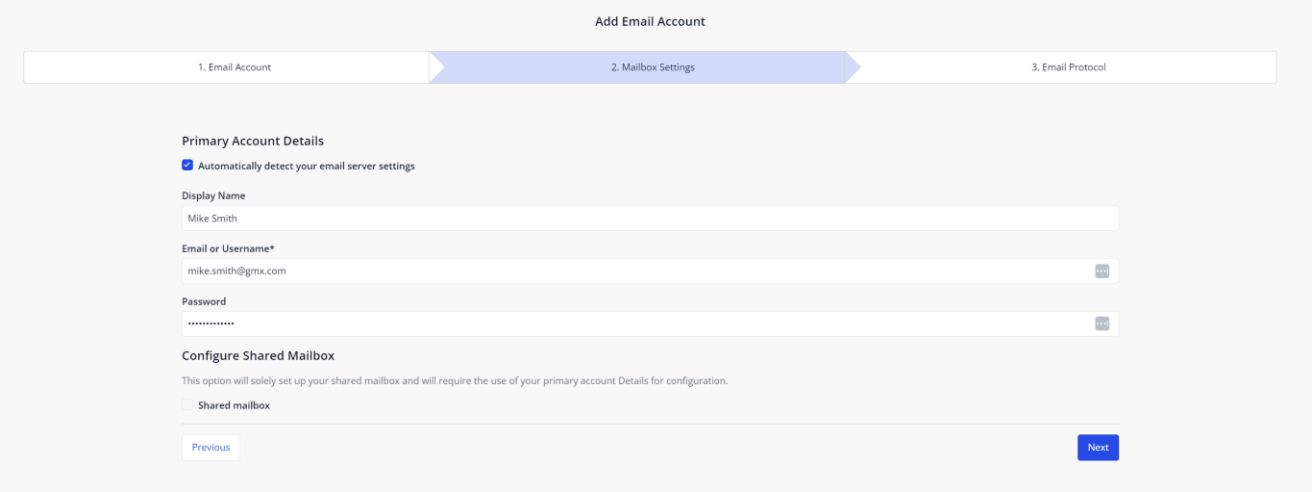
- Changes to a page or button have not been fully applied.
- An outdated assignment of a parameter was cached.
- The Mendix environment was incorrectly synchronized with the project structure.

8. Now the application is started for the first time and further configured. Mendix offers the option of activating demo users in order to work with them temporarily. For further configuration, "demo_administrator" is used and the password can be copied directly from the IDE. This is possible with 'Copy password to clipboard'. The application is started in the IDE using "Run locally" and a web browser window opens.



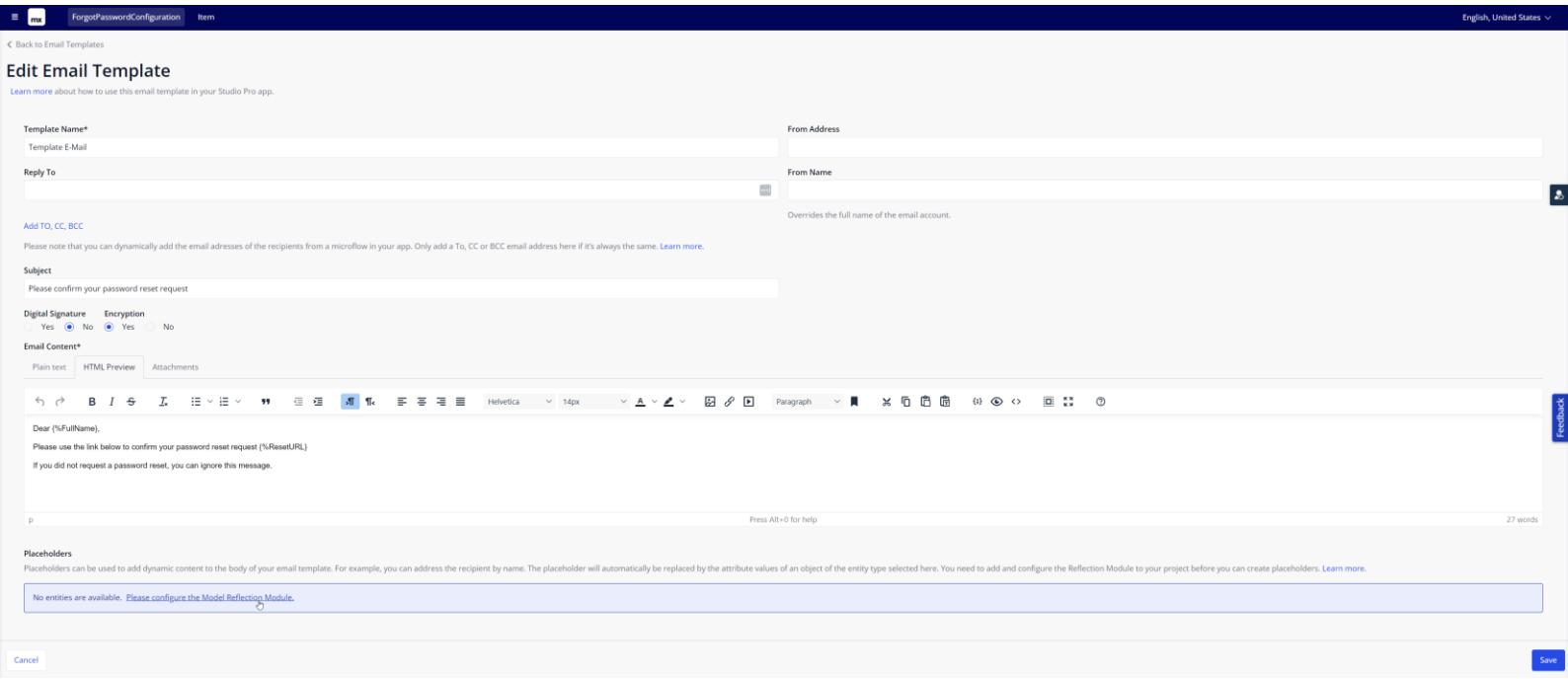
A login window appears, but no users have been configured yet. Therefore, the "demo_administrator" already explained is used and logged in with "Sign in". The system switches to the "ForgotPasswordConfiguration" page, where the SMTP settings can be set.

9. Further steps are carried out in the "Reset Password Email" tab. Select "SMTP settings" and then "Get Started". Here the connection to an email account is established, it would also be possible to select "Microsoft Entra ID" or "Use Basic Credentials". The latter is selected in this documentation. Subsequently, the e-mail settings are set which must be used to send e-mails.

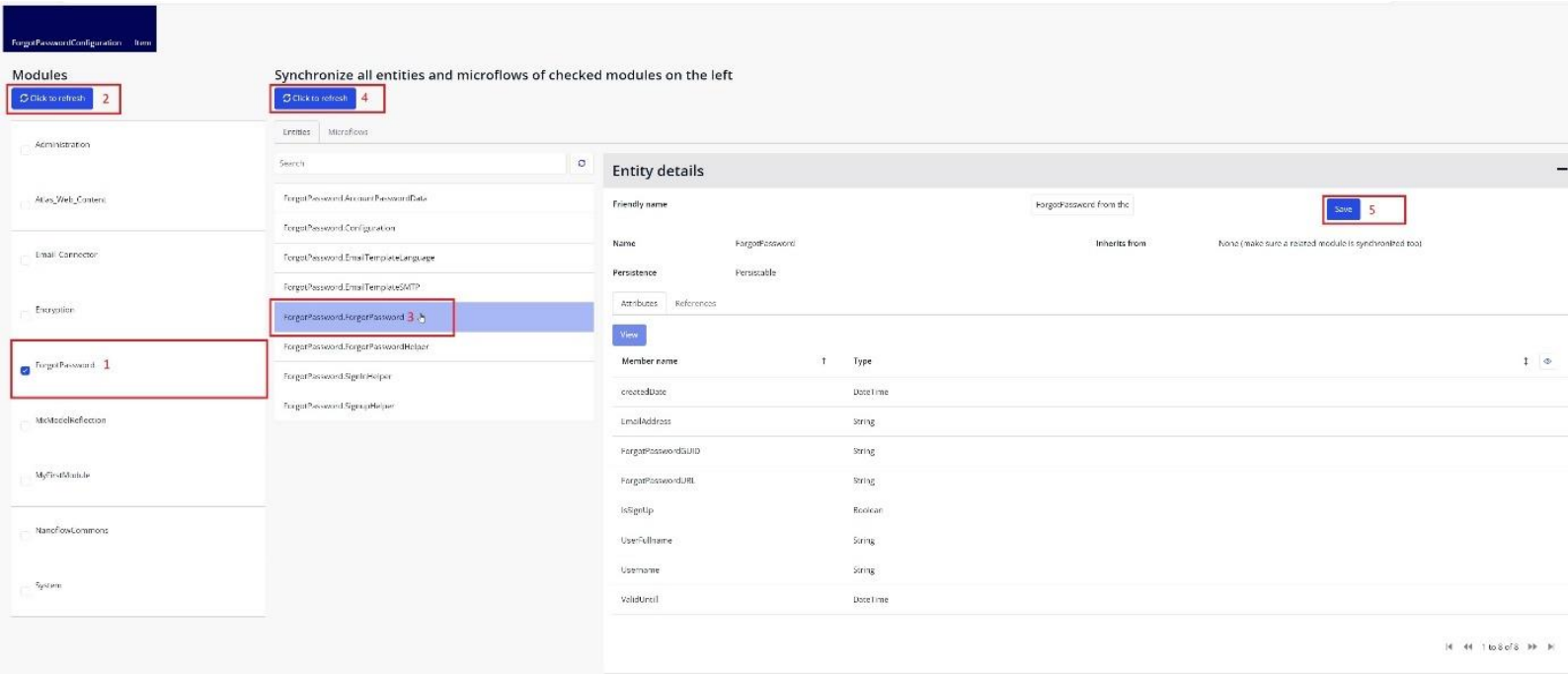


10. In the last step, select that it is only necessary to check "Send Emails". Mendix then immediately returns the standard SMTP configuration data. In this case, GMX was used, and an internal or external database should be accessed to get this information. It is therefore probably not necessary to enter the SMTP servers in most cases and it is sufficient to enter the e-mail address and password. In this prototype, the SSL connection is selected, which establishes an encrypted connection between the Mendix application and the e-mail server. If the addition of the email account was successful, there is a positive response **"Email account was successfully added."**

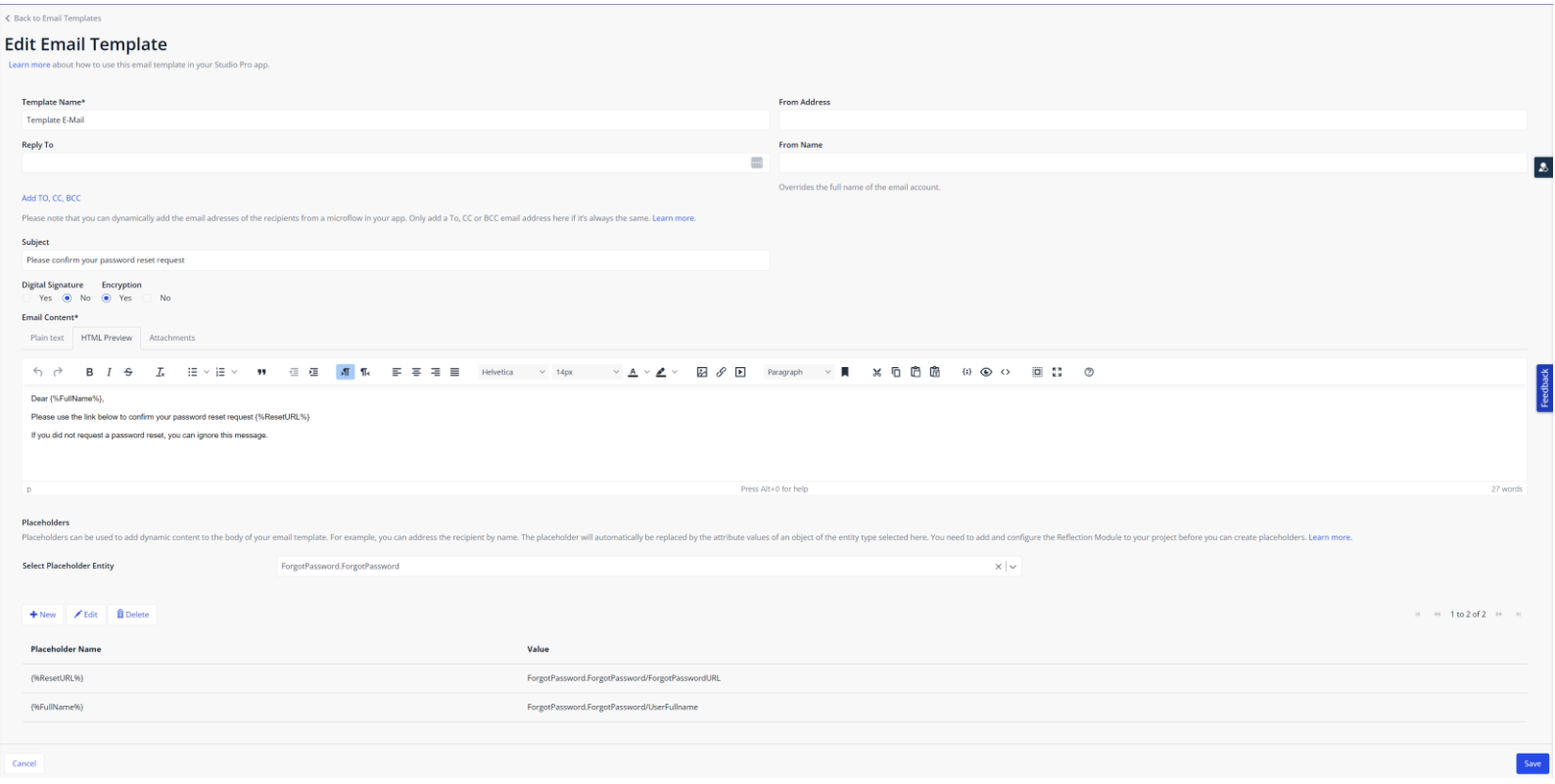
11. In the next step, it is still possible to create an email template. To do this, select the "Email Templates" button and add a new template under "New". The note **"No entities are available. Please configure the Model Reflection Module."** When configuring this module, it is possible to dynamically add text and data to the email to be sent. In this case, we want to do this and click on the link.



12. Select "Modules-> ForgotPassword" and select "Click to refresh". All entities and microflows that are in this module are synchronized. In this particular case, "ForgotPassword.ForgotPassword" is selected. Now the "Entity details" are displayed, this is only for an overview. As the attributes contained are interesting for sending e-mails, this entity is saved with "Save".



13. Return to the "e-mail template" and use these attributes. This is done under "Select Placeholder Entity -> ForgotPassword.ForgotPassword". The attributes of this entity can now be added. Basically, only a name is written for the placeholder and the link to the actual attribute is created. The "ForgotPasswordURL" and the "FullName" are added and "Save" is pressed. The "FullName" is the FullName that the user entered during registration and "ForgotPasswordURL" is the URL that the user receives during registration, with which the account can be activated.



14. Return to the "ForgotPasswordConfiguration" window and find no configuration. This is because it is now created with "Create". The nice thing is that the template just created is opened immediately - with a small but subtle difference. Before saving this configuration again, it is required to select the previously created "SMTP Configuration". After selecting it, click "Save" once more. At this point, all configurations are complete, and a user registration can now be simulated. To perform this simulation, the fields "Fullname" and "E-mail address" must be filled in.

15. Next, the configuration must also be carried out in the "Signup Email" tab. In this case, the same as for "Reset Password Email" can be used. Click on "Create email template" and create a new template. The SMTP settings must also be entered again in the template. It is basically identical to step 14, except that the "Subject" and "Email Content" change.

Now start the app again and try to register a user. To do this, enter a "Fullname" and an "Email address". If the process is successful, a response stating, "We have sent you a confirmation email." will be displayed. Follow the instructions in the confirmation email to setup your account". To do this, log in to the email account entered. In the email there is a confirmation link, which must be clicked to complete the user registration. When the link is opened, a popup appears in which the new password for the user must be entered. The user can then log in successfully with username + password.

Use Case 1 - Java/Angular/MySQL

Backend: Developed with Java Spring Boot for RESTful APIs.

Frontend: Implemented with Angular v18 for a reactive user interface.

Database: MySql v8.4.0 for relational data.

Containerization: Docker v4.35.1

This project is based on a containerized infrastructure with Docker Compose, which orchestrates the frontend, backend and database in separate containers. Additional Dockerfiles exist for the frontend and backend to enable specific configurations for the build and runtime environment. The aim is to implement an application that stores user data securely with BCrypt hashes. MySQL is used as the relational database, as the requirements do not call for more complex database structures.

Dependencies

The following dependencies are used in the **Spring Boot** project, which are defined via **Maven** in pom.xml:

- Spring Boot Starter Web
  Purpose: Provision of RESTful APIs.
- Spring Boot Starter Security
  Purpose: Implementation of authentication and authorization.
- Spring Boot Starter Data JPA
  Purpose: Communication with the database via JPA.
- MySQL Connector
  Purpose: Connection to a MySQL database.
- Lombok
  Purpose: Reduction of boilerplate code.
- Dotenv Java
  Purpose: Processing environment variables from .env files.
- Spring Boot Maven Plugin
  Purpose: Support for the build and deployment of Spring Boot applications.

In addition, a short list of dependencies for Angular:

- @angular/animations
  Purpose: Enables animations in Angular.
- @angular/forms
  Purpose: Supports reactive forms and template-based forms.
- @angular/router
  Purpose: Enables navigation and routing between components.
- Rxjs
  Purpose: A library for reactive programming with observables.

Implementation: No suitable prefabricated implementation could be found for this use case, which is why all functionalities were developed independently. Basically, the backend is divided into 3 functions: User registration, email verification and login authentication. The frontend is divided into two components, registration and login. In Spring Boot, the application.properties is configured accordingly. This is where the connection settings for the MySQL database are defined, such as URL, username, and password. In addition, the Hibernate dialect for MySQL is set, the automatic update of the database schema is activated (**ddl-auto=update**) and the display of executed SQL queries (**show-sql=true**) is activated. These settings ensure that the application connects and interacts with the database.

1. Set up the projects in Angular and Spring Boot with the respective dependencies and docker files. A Docker-Compose file must also be created, which provides the respective infrastructure for each container.

   Start by setting up the backend, as the frontend will then access the necessary APIs. However, it would also be possible to start the other way round.

   The directory structure is as follows:

   ```
   /main/java/app/web/registration
   ├── controller # Contains REST controllers
   ├── enums # Contains enums (e.g., roles)
   ├── model # Data models (e.g., User, VerificationToken)
   ├── repository # Database repository interfaces
   ```

├── security # Spring Security configuration
├── service # Business logic (e.g., email, user management)

2. Create the entities for the database.

   User.java

- Fields: id, fullname, username, email, password, enabled, role
- This entity represents the user of the application. The password is stored hashed (BCrypt) and the **enabled** field shows whether the user has been activated after email verification.

  VerificationToken.java

- Fields: id, token, expiryDate, user
- This entity stores the verification tokens that are used to activate user accounts. It has a 1:1 relationship with User.

3. Now the repository interfaces are integrated and Spring Data JPA is used to manage user and token data. The **UserRepository** is an interface provided by **Spring Data JPA**. It is used to enable CRUD operations (create, read, update and delete) for the **User** and **VerificationToken** entities without having to write your own database access code. The repository extends the **JpaRepository** interface and thus inherits standard methods such as **save**, **findById** and **delete**.

   UserRepository.java

   A specific method has been added to the **UserRepository** to support the business logic.

- **findByUsername (String username):**
  Searches for a user based on their user name. This method is used when logging in to check the existence of a user.

  VerificationTokenRepository.java

- A method has also been added here to extend the functionality. Just like the **UserRepository**, this interface also extends the **JpaRepository**.
- **findByToken(String token):**
  This method makes it possible to search for a verification token based on its value (token). The method is used to check the token when verifying a user account and to validate the validity and expiry date of the token.

4. The next step is to implement basic Spring Security. Spring Security is used in the project to secure access to the API endpoints.

   SecurityConfig.java

   Access restrictions: Endpoints such as /api/register, /api/login, /api/verify and /api/test are publicly accessible (**permitAll()**). All other endpoints require authentication (**authenticated()).**

- CORS (Cross-Origin Resource Sharing): . cors(Customizer.withDefaults()) enables cross-origin requests (e.g. from the Angular frontend). A custom CORS configuration allows requests from specific origins (http://localhost:4200).

- CSRF protection: CSRF (Cross-Site Request Forgery) is disabled in the development environment because it is typically not necessary for API calls (csrf.disable()).
- Password hashing: A PasswordEncoder is configured that hashes passwords securely using **BCrypt** . This protects saved user passwords from misuse.
- HTTP Basic Authentication: Authentication is enabled in the basic configuration via HTTP Basic, which allows for easy API usage during development.

5. Now the main functions for the backend are created. The **UserService** contains the central business logic for managing user data and user registration. The **EmailService** takes care of sending verification emails. It uses the **JavaMailSender** defined in the **MailConfig** to send emails via a configured SMTP setup. The **MailConfig** is a configuration class that provides the **JavaMailSender**. It reads the e-mail settings from environment variables or an .env file and configures the SMTP server for sending e-mails.

   UserService.java

   The **UserService** is the central component for user registration and the management of verification tokens.

   Registration:
   - Creates a new user with the passed data (**fullname**, **username**, **email**, **password**).
   - Hashes the password with **BCrypt** before it is securely stored in the database.
   - Saves the user along with a default role (e.g. USER) as disabled in the database (**enabled = false**).

   Token generation and storage:
   - Creates a verification token (UUID) for the new user.
   - Stores this token in the **VerificationToken** entity associated with the user.

   Email Verification:
   - After successful verification (e.g. via **/api/verify**), the user is activated (**enabled = true**).

   Additional methods:
   - **enableUser(User user):** Activates a user in the database.
   - **findByUsername(String username)**: Searches for a user by their username.
   -

   EmailService.java

   The **EmailService** takes care of sending verification emails. It uses the **JavaMailSender** defined in the **MailConfig** to send emails via a configured SMTP setup. The **EmailService** ensures that new users receive a verification link via email to activate their account. This ensures easy and secure authentication.

   The main function is **sendVerificationEmail(String toEmail, String verificationUrl).** This creates an email with a verification link and sends this email to the address given (**toEmail**) with a subject and the link (**verificationUrl**).

   MailConfig.java

The **MailConfig** is a configuration class that provides the **JavaMailSender**. It reads the email settings from an .env file and configures the SMTP server to send emails. The main function here is **getJavaMailSender().** This feature configures the SMTP host, port, username, password, enables SSL/TLS for a secure connection, and defines other SMTP settings such as authentication and debugging. The configuration reads values such as MAIL_HOST, MAIL_PORT, MAIL_USERNAME and MAIL_PASSWORD. **MailConfig** ensures a central, secure and flexible configuration of e-mail sending.

6. Now the REST controller is created, and there are basically three endpoints:

api/register (POST): Receives the user data (fullname, username, email, password) and creates the user. Uses the **UserRegistrationRequest** class as a DTO (Data Transfer Object) to encapsulate the input data from the JSON body. The data is then forwarded to the **UserService**, where the user is stored in the database and a verification token is generated. Then send a verification link by email.

- api/verify (GET): Activates the user based on the verification token. The token is passed as a query parameter (token) and checked using the **VerificationTokenRepository** . If the token is valid (exists and has not expired), the user account is activated (**enabled = true**). Otherwise, an error message is returned (for example, "Invalid token" or "Token expired").
- /api/login (POST): Authenticates users using username and password if they are enabled. Use the **UserService** to check if the user exists, the password is correct (validated via **BCrypt**), and the account has been activated (enabled = true). If the login is successful, a success message is returned. If the check fails (e.g. incorrect password or inactive account), a corresponding error message is displayed (e.g. "Invalid username or password").

7. Since the frontend was already set up in the very first step, the implementation will now continue. The entry point of the application is "app.component.ts" and "app.route.ts" provides the routing within Angular. There are /login and /register which can be called. Login redirects to the login component and register to the registration component.

   Registration Component (register/)

   - Logic (register.component.ts)
     This file contains the main logic for the registration. A reactive form is created using **FormBuilder** to capture user input such as **fullname**, **username**, **email,** and **password**. The inputs are validated and sent to the backend (**/api/register**) via an HTTP POST request

   - User Interface (register.component.html) The form provides input fields for the above fields, as well as submit buttons. It displays error messages for invalid inputs.

   - Styling (register.component.css)
     Defines the visual appearance, such as spacing, colors, and hover effects, as well as error markers for validation errors.

8. Login component (login/)

   - Logic (login.component.ts):
     This file implements another reactive form for login, where **username** and **password are** checked and sent to the backend **(/api/login**).

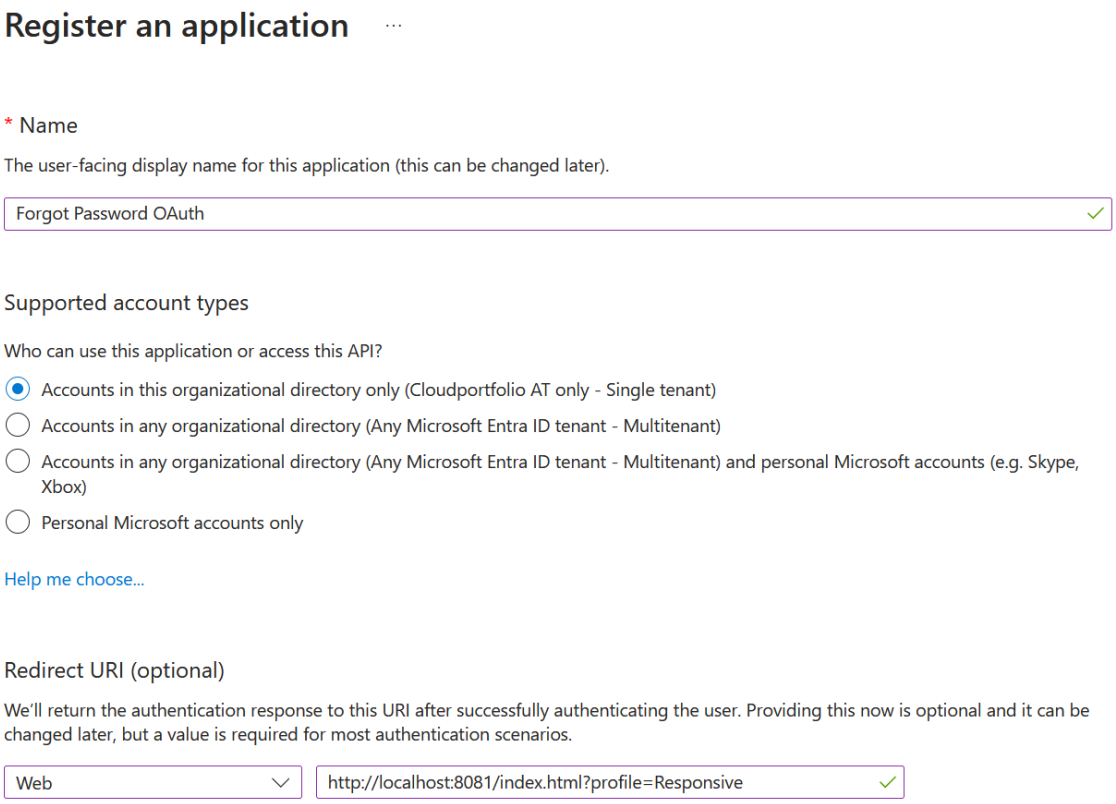
   - User Interface (login.component.html):
     This form provides fields for username and password, as well as buttons to submit and navigate to the registration page.

   - Styling (login.component.css):
     Similar to the registration component, it provides a responsive design and user-friendly error messages.

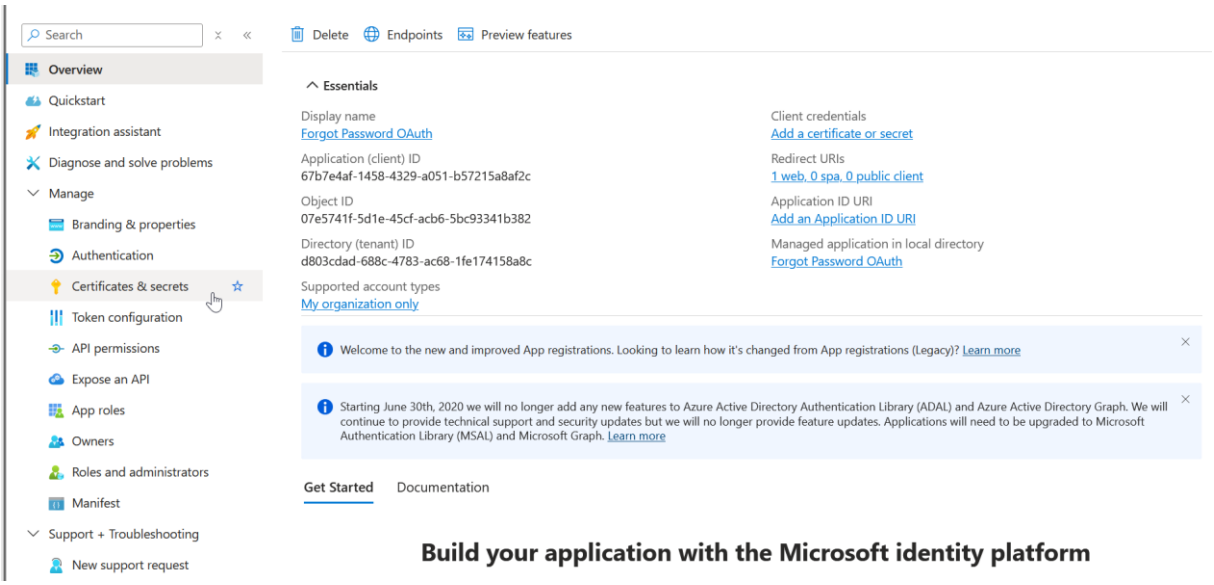
The application is now fully prepared and can be started using the Docker Compose file. The entire environment can be started with the ‘docker-compose up’ command in the root directory of the project, where the docker-compose file should also be located.

Use case 1, Second iteration with Mendix - Enable OAuth2

1. In this demonstration, Azure will be used as an OAuth provider for this purpose. Basically, this is also possible with a free license. This requires registration, and then you can already perform an OIDC configuration in the Azure Cloud under 'Microsoft Entra ID -> App registrations'. Basically, only one name is selected here, the 'supported account type' and a ' redirect URI '. Now the app can already be registered.



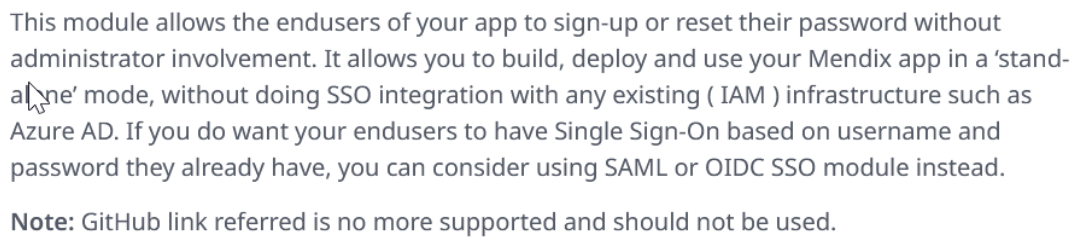
2. After registration, a new secret can be created in Azure in the same mask under 'Certificates & secrets'. The description is ' Forgot Password OAuth Secret ' and the validity period is 24 months. The generated secret is saved immediately because it is only shown once. In the Endpoints section, the ‘Authorization Endpoint’ and the ‘Token Endpoint’ are also listed. These URLs are also copied.



3. To configure this in Mendix, the following information is required:
   - Under the registered application in Azure, find the Application (client) ID.
   - The Client Secret has already been generated before and can therefore be used directly.
   - The 'Callback URL' is http://localhost:8081/index.html?profile=Responsive .

- Under 'Token endpoint URL', the URL **OAuth 2.0 Token Endpoint (v2)** " from Azure is used.

4. An attempt was initially made to integrate OAuth into the current module (Forgot Password Module) in Mendix, but this did not work. The reason could then also be found in the documentation (see screenshot). It states that SSO integration is not supported. However, it also writes which modules can be used for this use instead, such as SAML or OIDC SSO modules.


This module allows the endusers of your app to sign-up or reset their password without administrator involvement. It allows you to build, deploy and use your Mendix app in a 'stand-alone' mode, without doing SSO integration with any existing (IAM) infrastructure such as Azure AD. If you do want your endusers to have Single Sign-On based on username and password they already have, you can consider using SAML or OIDC SSO module instead.

Note: GitHub link referred is no more supported and should not be used.

However, further research was done, and a module could be found directly from Mendix itself: 'Mendix SSO'. Since the author wants to use the native possibilities of each platform, an attempt was made to integrate the in-house SSO login. After trying it out, however, it turned out that the in-house Mendix SSO module is not free to use. Instead, a paid extension must be made to set up environments in Mendix. In principle, however, it is possible to download the Mendix SSO module free of charge, but its use is subject to certain license and environmental conditions. This license is not sold individually for the SSO module but is part of a broader Mendix licensing model that includes the operation of production applications. As things stand today, the cheapest model "Basic" starts at € 52.50/month [3]. Another module, the 'OIDC SSO module', will now be integrated into the existing project.

OIDC SSO modules:

Source: https://marketplace.mendix.com/link/component/120371

Dependencies:

- Encryption
- Community Commons
- Nanoflow commons
- Mx Model reflection
- User Commons

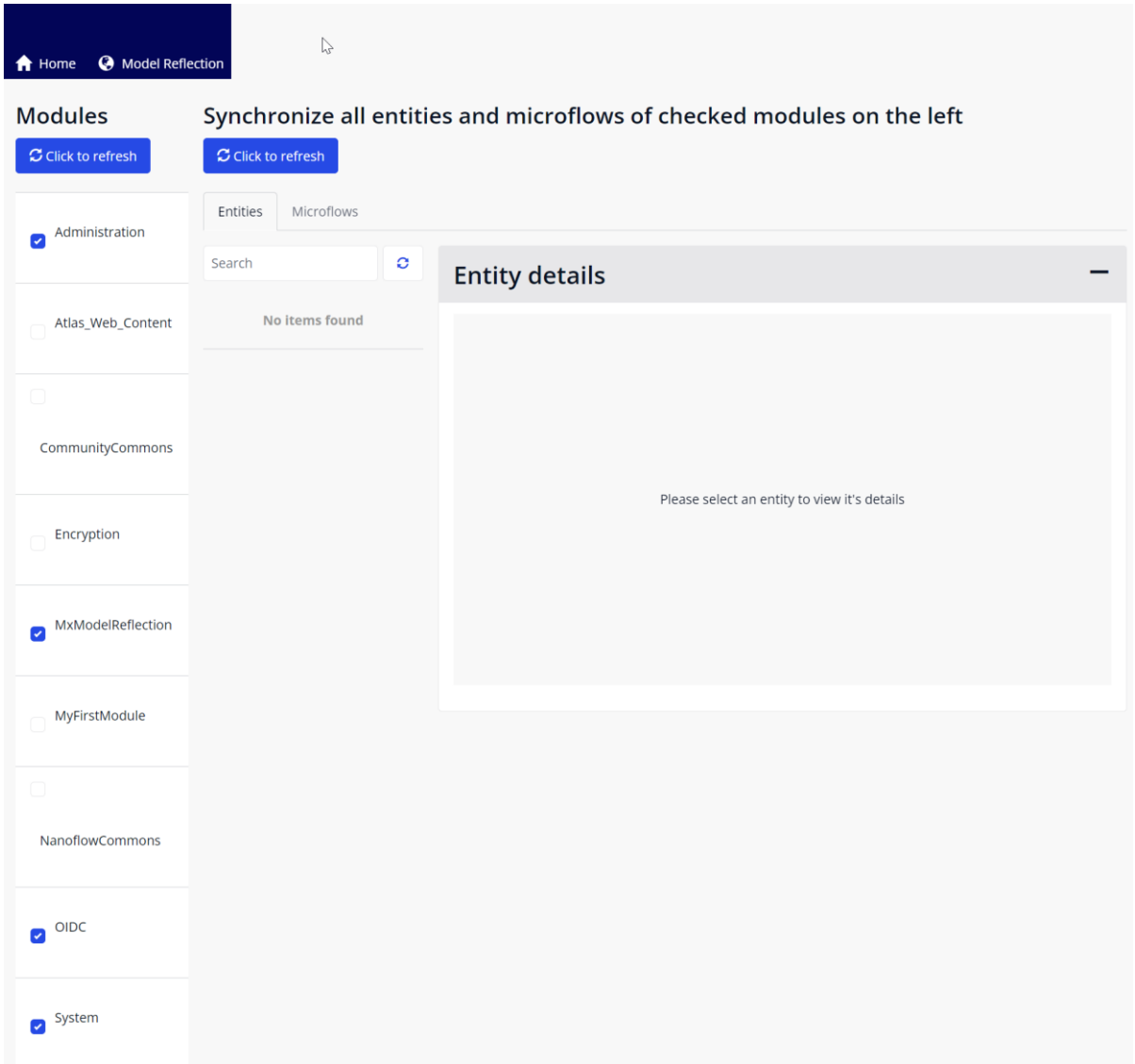
In the first iteration, a Mendix application with the ForgotPassword module was already configured to implement basic authentication. Since it was not possible to estimate in advance to what extent this would behave with an additional module such as the "OIDC SSO", this configuration was first used in isolation in a fresh environment. However, it was then integrated into the existing solution. The documentary will follow now.

1. Add the module 'OIDC SSO' to the application.
   Create a page in Mendix and integrate the snippet 'Snip_Configuration [OIDC] '

2. In app explorer, add a page under Navigation (module Mx Model Reflection -> MxObjects_Overview

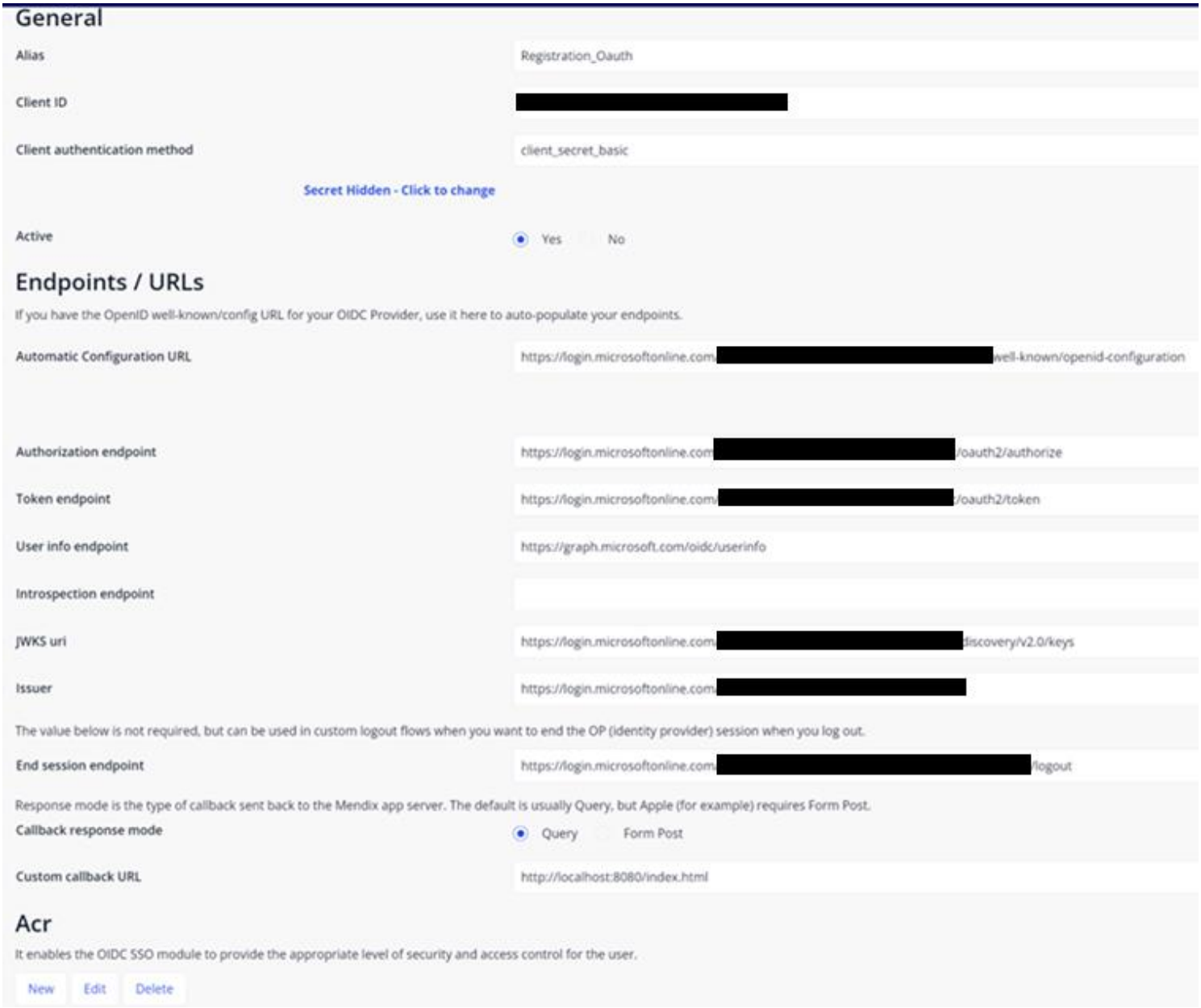
3. Under Settings -> Navigation, select this page

4. Start the app and open the page MxObjects_Overview. The page can be identified by the icon you selected in the Mendix app, which appears on the left side of the navigation bar. After opening the page, a module will be displayed on the left side. Select ‘Administration, MxModelReflection, OIDC, and System’, and then click on ‘Click to refresh’ under Modules and Entities.



5. 'IdPs for SSO and API security' -> New

6. Enter the following information for the OIDC config:

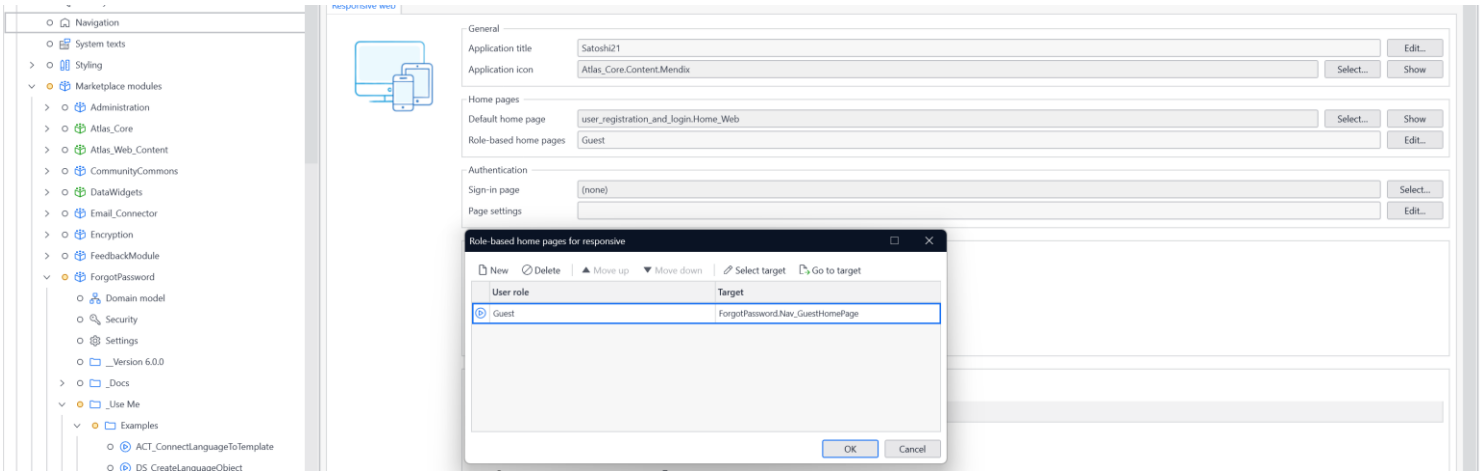


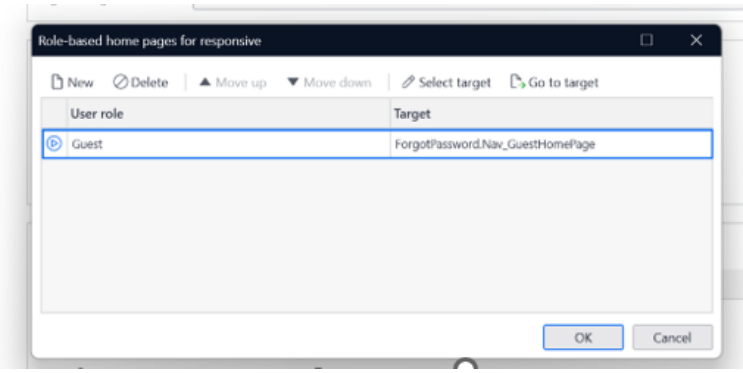
7. Scopes and claims have also been specified and control what information and permissions are requested and provided by the Microsoft Identity Platform. Scopes define the **access** that the application requests. Here, specify the actions and data that the application is authorized to use. For example, Openid is required to enable authentication. Without this scope, no ID token is generated. Claims are the **data** contained in the ID Token and/or Access Token. They provide details about the user or session.

8. Basically, these configurations were sufficient to perform an OIDC login via Microsoft. But integrating this module into the already existing application with the ForgotPassword module turned out to be a bit more difficult. For this purpose, it was necessary to understand how the ForgotPasswordModule works in order to be able to make adjustments.

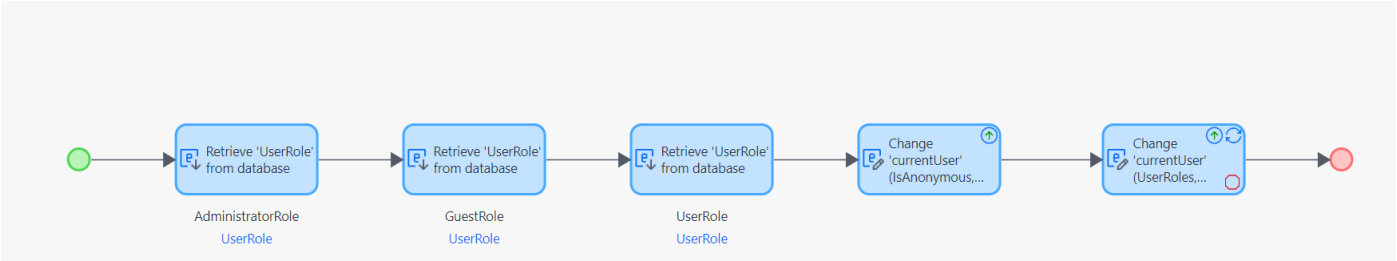
9. After an in-depth analysis of how the application works, it turned out that a login mask is displayed at startup. This is triggered by a microflow that is hidden under *Navigation > Page Settings > Edit*. The microflow runs exclusively for users with the Guest role to ensure that users sign in before they can

continue. With the Basic Auth implementation, this already worked without any problems. However, since the OIDC module is separate from this logic and has its own module roles, an adjustment was necessary. The chosen solution serves to make both modules executable and to enable a comparison of the two implementations.





10. To complete the OIDC integration, a microflow (**ACT_ChangeUserRoleToAdmin**) was created that automatically assigns the administrator role to a user once they have successfully logged in via OIDC.



With this adjustment, it was possible to log in with both authentication methods.

Use case 1, Second iteration with Java/Angular - Enable OAuth

1. Add the dependency ' spring-boot-starter-oauth2-client ' to the existing pom.xml and don't forget maven clean + install.

   In application.properties, add the OAuth2 details:

```
# Base URL for the application (important for redirect URIs)
spring.security.oauth2.client.registration.azure.client-name=Azure
spring.security.oauth2.client.registration.azure.client-id=[redacted]
spring.security.oauth2.client.registration.azure.client-secret=[redacted]
spring.security.oauth2.client.registration.azure.scope=openid, profile, email, offline_access, User.Read
spring.security.oauth2.client.registration.azure.authorization-grant-type=authorization_code
spring.security.oauth2.client.registration.azure.redirect-uri=http://localhost:8080/index.html?profile=Responsive

# Microsoft Azure OAuth2-Provider URLs
spring.security.oauth2.client.provider.azure.authorization-uri=https://login.microsoftonline.com/[redacted]
spring.security.oauth2.client.provider.azure.token-uri=https://login.microsoftonline.com/[redacted]
spring.security.oauth2.client.provider.azure.user-info-uri=https://graph.microsoft.com/oidc/userinfo
spring.security.oauth2.client.provider.azure.jwk-set-uri=https://login.microsoftonline.com/[redacted]
spring.security.oauth2.client.provider.azure.user-name-attribute=sub
```

3. In the SecurityConfig class, the method "securityFilterChain(HttpSecurity http)" is extended with:

.requestMatchers("/oauth2/**").permitAll()"

and:

```
.oauth2Login(oauth2 -> oauth2  configure OAuth2 login
        .defaultSuccessUrl("/home", true))  forward after successful login
```

It is not necessary to create a controller with endpoints for OAuth2 in Spring Boot, because Spring Security automatically creates the Oauth2 flow. Spring Security automatically provides the endpoints /oauth2/authorization/{registrationId} and all OAuth2 authentication logic. The redirect flow to the identity provider (e.g. Azure) and the retrieval of the access token are already handled by Spring Security. This redirects the user to the login page of the OAuth2 provider (e.g. Microsoft Azure) to authenticate. After successful registration, Spring Security takes over the further processing.

4. Now, a successful response is only given if the user has been able to log in or register.

5. In Angular, a auth.service.ts is created that can be used to retrieve an endpoint from Oauth in the backend:

```
loginWithOAuth(): void {
window.location.href = '${this.baseUrl}/oauth2/authorization/Azure';
}
```

6. In the already existing login.component.ts, loginWithOAuth() is also created, which calls the method in auth.service.ts. This moves the logic to auth.service. In the login.html a button has been added which enables a login with OAuth. After a successful login you will be redirected to the /home page. Here it is possible to log out again using the logout button.

Use Case 2 - Mendix (v 10.12.0)

In this use case, a crypto dashboard is created in which further conversions are also possible via an additional details page for the respective cryptocurrency. When the user selects the details of a specific cryptocurrency, a converter in dollars is offered.

On this details page, it is also possible to select other national and cryptocurrencies via a combo box in order to obtain additional conversions. For example, if Bitcoin is selected in the details view, it is possible to change the value to, for example, 0.1 BTC and the conversion to USD dollars is displayed. However, it is also possible to get numerous other currencies via the combo box and display the conversion rate of 0.1 Bitcoin in these currencies as well.

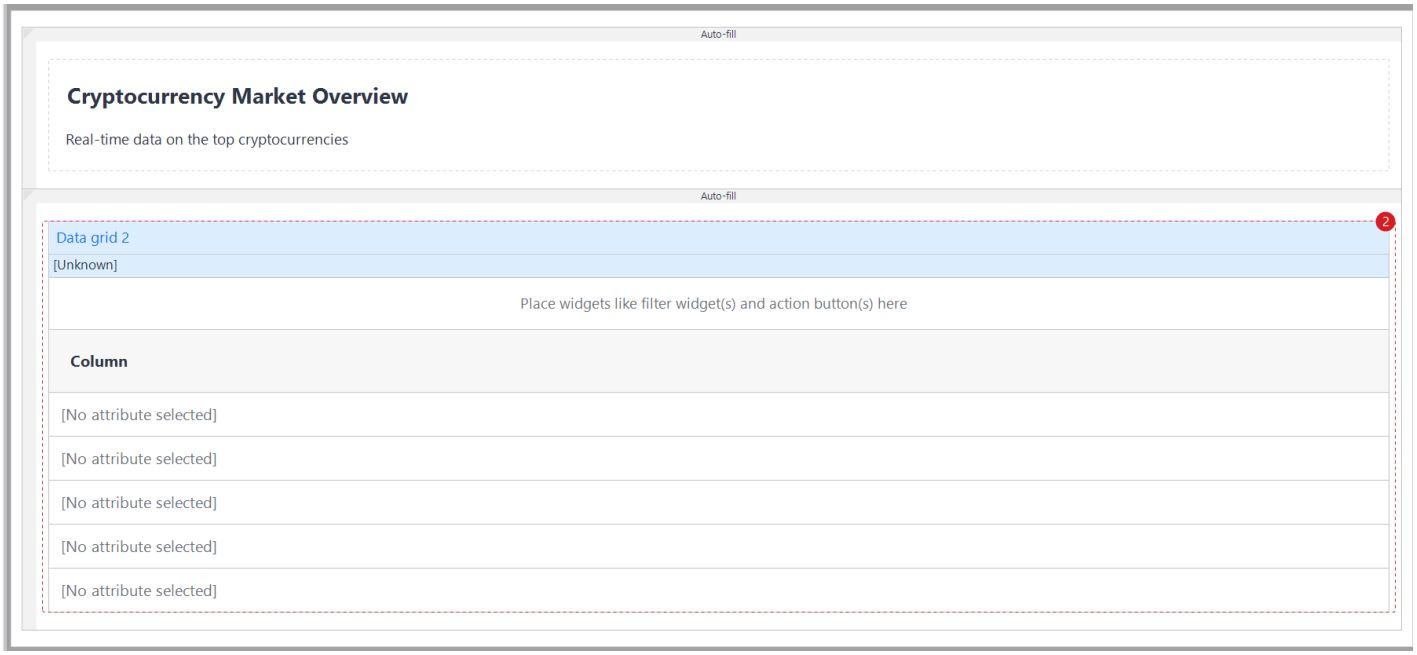
Preconditions:

- Mendix Studio Pro (v 10.12.0)

Implementation

1. Similar to Use Case 1, the project was also initialized as a ‘Blank Web App’ for this use case. In the first step, a page with the name ‘Crypto_Dashboard’ was created and then an element of type ‘Data grid 2’ was added. It should be noted that Mendix also offers a standard ‘data grid’ in addition to ‘Data grid 2’, which is why care must be taken when selecting it.

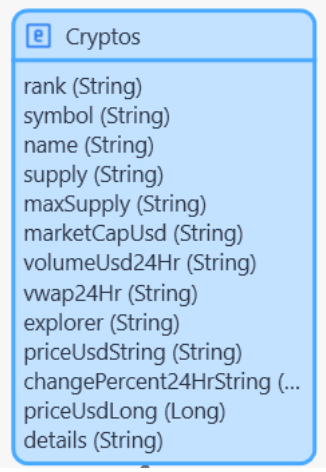
   It is not yet possible to proceed with the actual design of the page, as the necessary functions and the associated entities must first be created. In the further course of this work, the term microflow will be used for the implementation of functions, in line with the terminology commonly used in the Mendix platform.



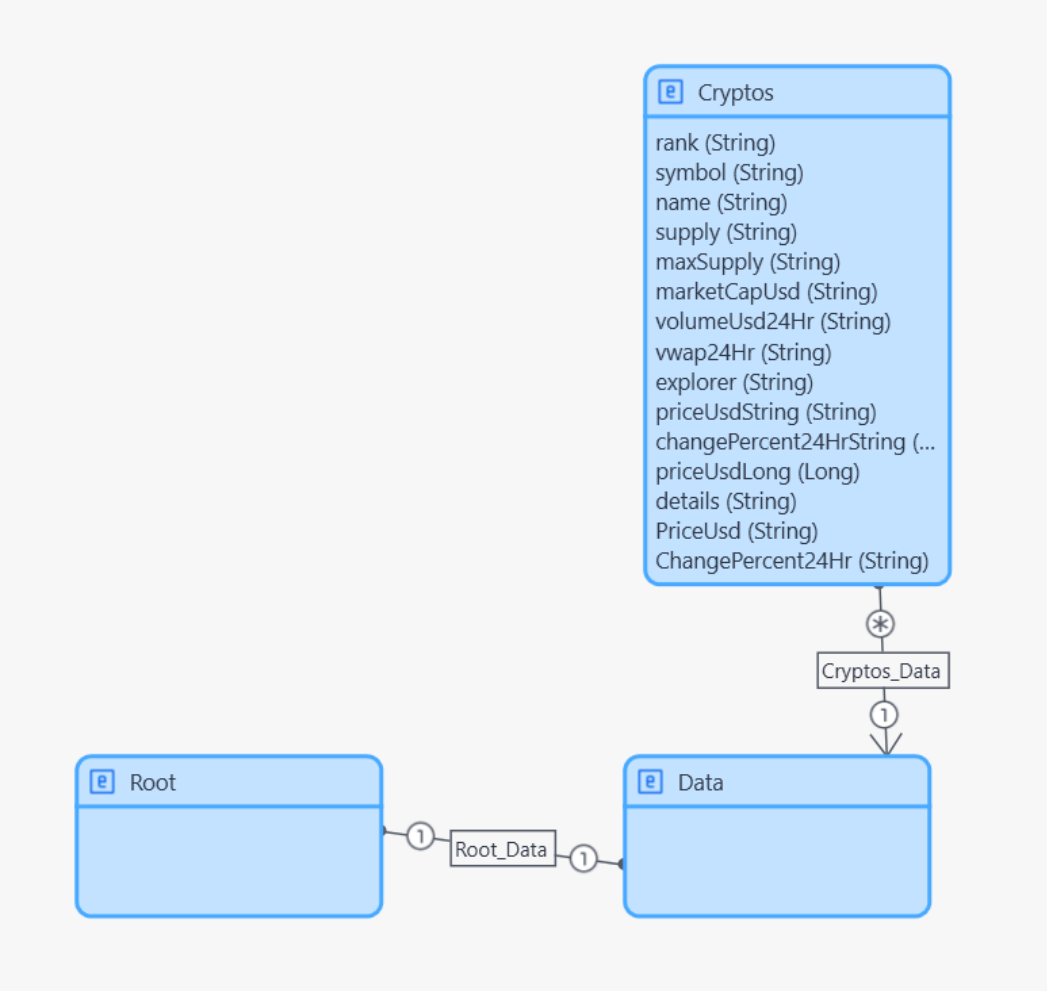
2. Before working on this page, it is necessary to configure a microflow and the domain model.

   In the Domain Model, an entity is created for the object to be used for the API. The "Cryptos" entity is created. The names of the attributes were based on those of the API.

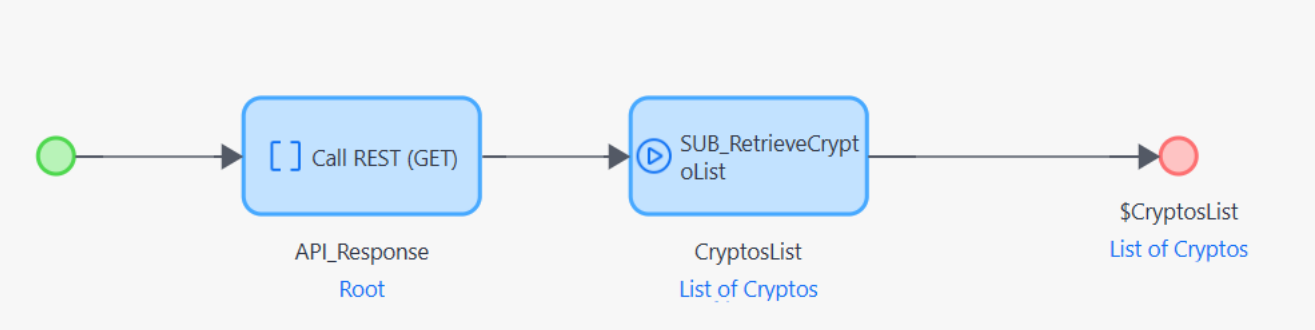
   rank (string): Unique identification number (ID) for each cryptocurrency.
   symbol (string): Abbreviation of the cryptocurrency.
   name (string): Full name of the cryptocurrency.
   supply (string): Current amount of coins/tokens issued.
   maxSupply (String): Maximum number of coins/tokens that can ever exist.
   marketCapUsd (String): Total market value of the coins/tokens based on the current price.
   volumeUsd24hr (String): Trading volume of the last 24 hours in USD.
   priceUsdString (String): Current price of the cryptocurrency in USD.
   changePercent24Hr (String): Percentage price change over the last 24 hours.
   vwap24Hr (String): Volume-weighted average price over the last 24 hours.
   priceUsdLong (Long): Used in the Details page to calculate.
   explorer (string): Link to a blockchain explorer to see details about the cryptocurrency.
   details (String): More details accessible via a button.



The entity has now been set up, but an association still needs to be established. The relationship between the entities is set like this:

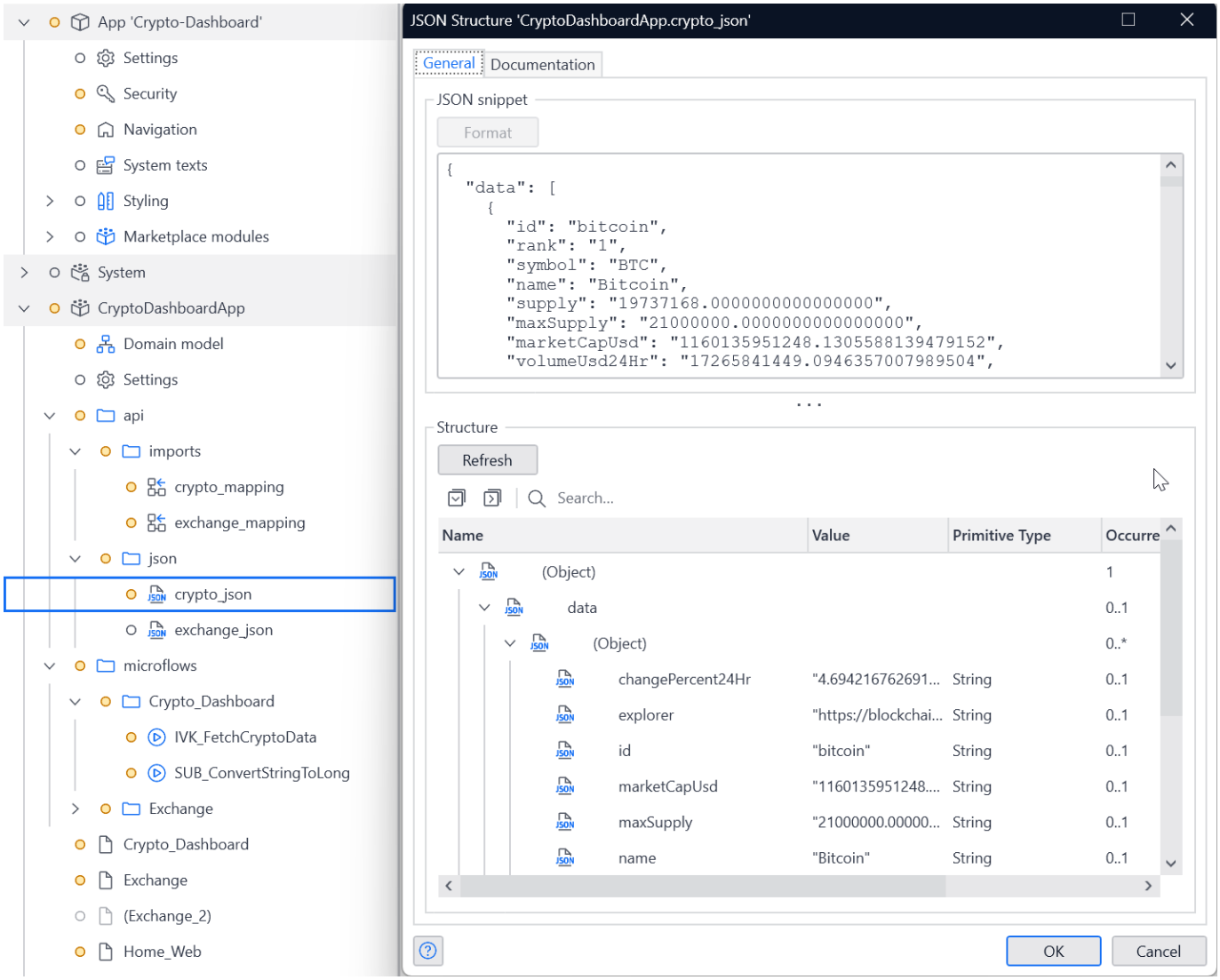


3. The microflow "IKV_FetchCryptoData (Invoke) to retrieve the API data has yet to be created.

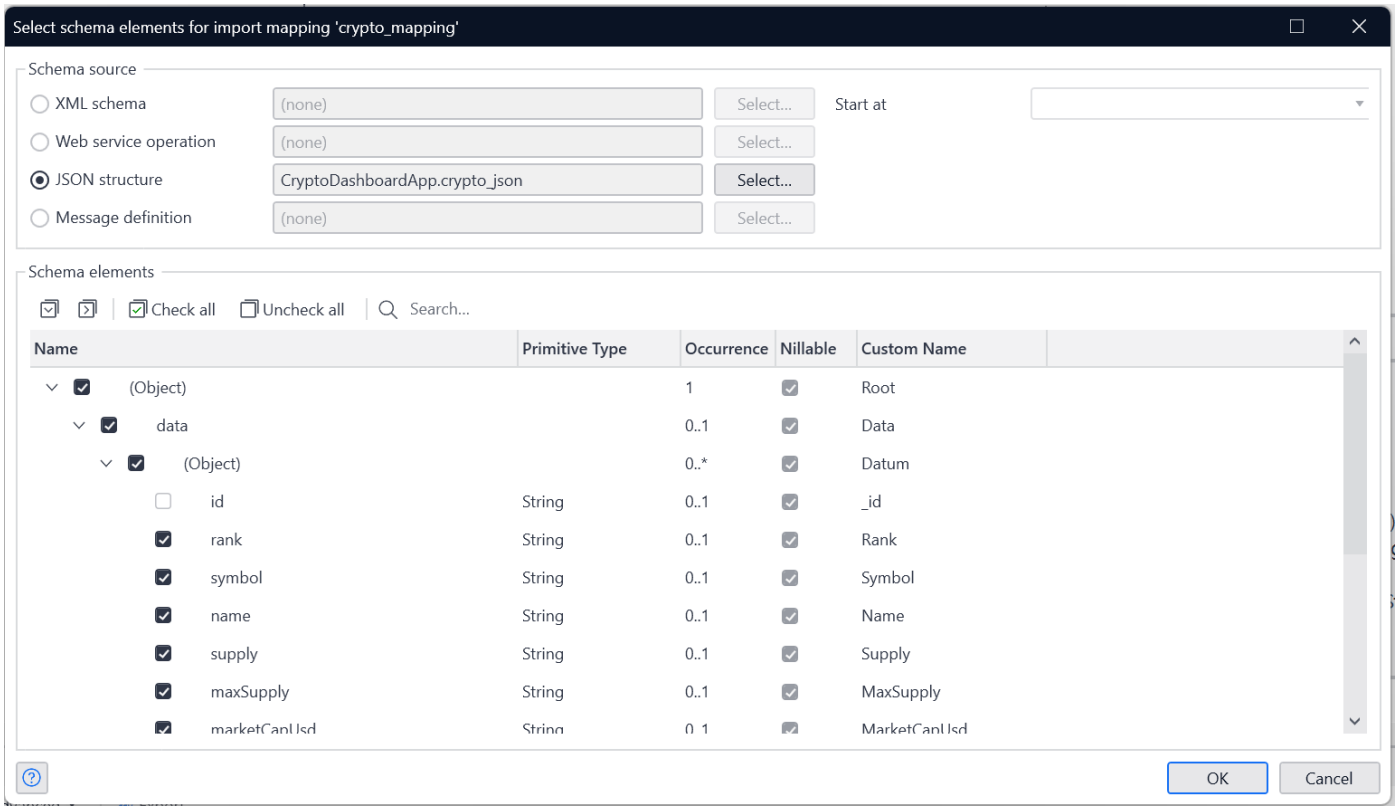


This screenshot only provides an overview of the target. In order to create the microflow in this form and make it functional, some manual steps are still required. These steps are discussed in detail below. Basically, 'Call REST (GET)' loads the data from the API into an already defined data structure in Mendix. This is subsequently created by an entity. The response from the API is saved in the variable ‘API_Response’ of type Root. The ‘SUB_RetrieveCryptoList’ sub-microflow is then called, which further processes the received data structure. This sub-microflow is used to extract the cryptocurrency data and converts it into a list of type CryptoList, which is then returned as the result.

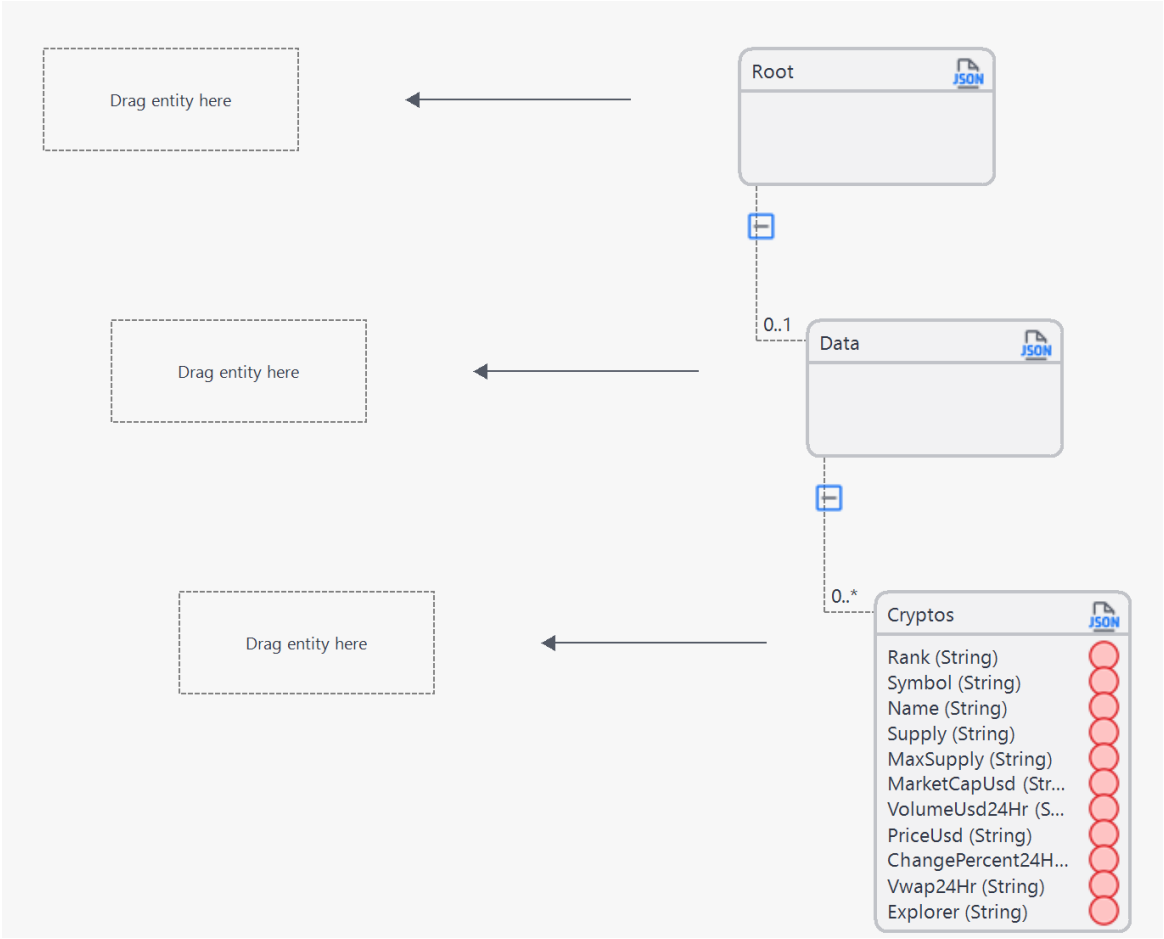
4. The REST call requires associated mapping so that Mendix knows what kind of data to expect. In a GET call, an "import mapping" is necessary. For this mapping, a "JSON structure" is provided in Mendix. In principle, however, other structures such as XML would also be possible. In this context, the "JSON structure" is created first, and the import mapping then uses it. Basically, the structure of the JSON in this case looks like this:



5. The JSON was stored labeled "crypto_json". Now an "Import Mapping" can be created, and this JSON can be selected. When selecting the schema, it is also possible to select which data should be taken from the API and which should not.

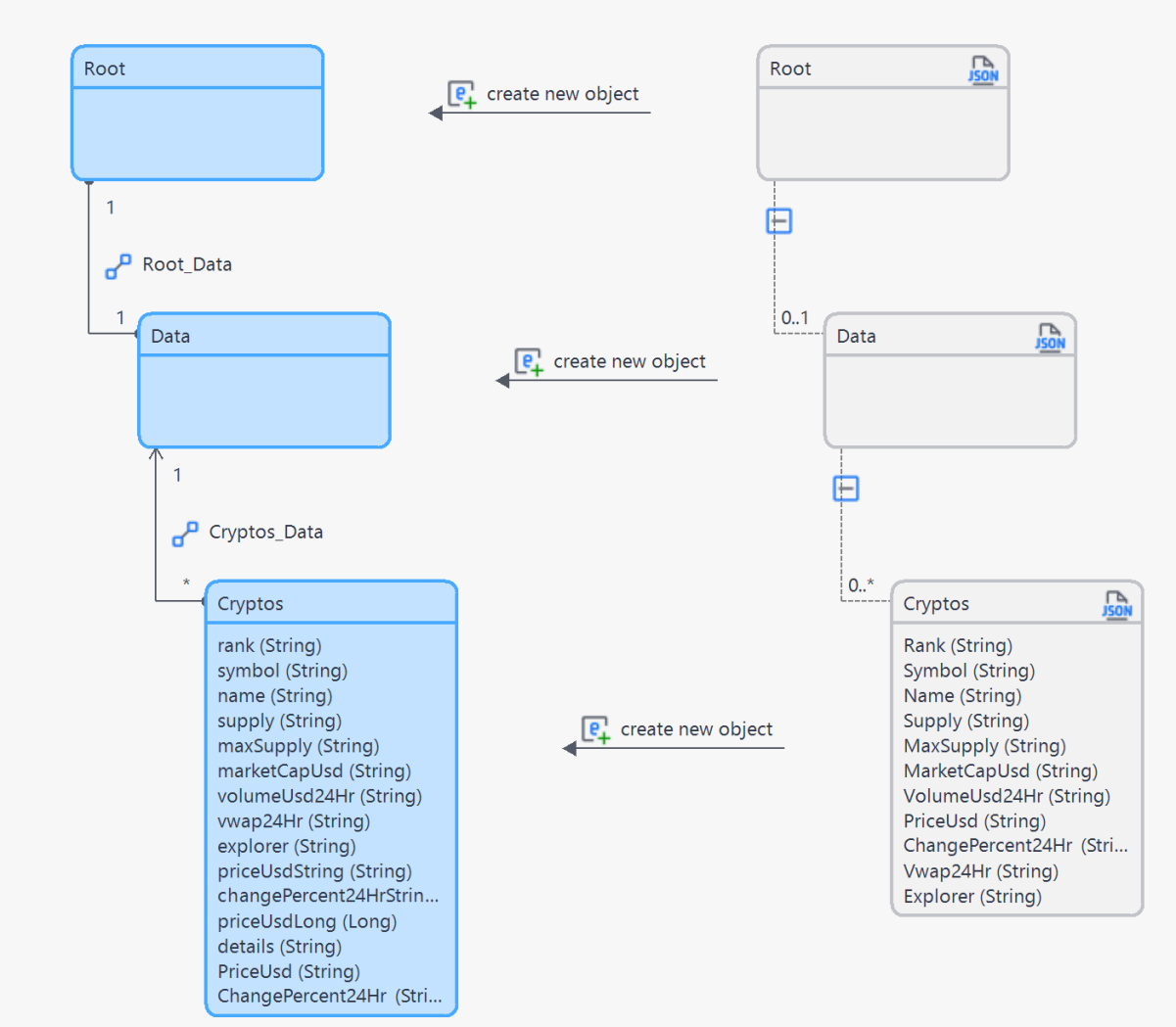


6. Once the mapping is added, Mendix closely follows the structure of the JSON to ensure that all nested objects and arrays can be mapped correctly.



- Root is created from the outer object.
- Data represents the array.
- Cryptos corresponds to the individual objects within the Data array.

7. Now the mapping of the attributes takes place:



8. Once the domain model and mapping have been configured, it is possible to create the microflow "IKV_FetchCryptoData (Invoke)" from step 3 shown at the beginning. Here the activity "**Call REST service**" is selected and added to the microflow. Within this activity, you can select these settings:

   General tab

   Location: https://api.coincap.io/v2/assets/

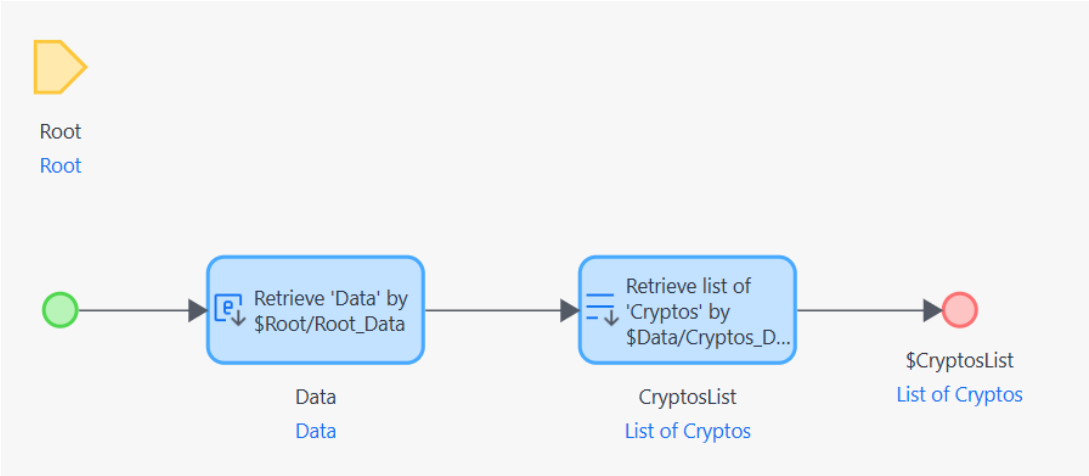
   HTTP method: GET

   Response tab

   Response handling: Apply import mapping

   Mapping: CryptoDashboardApp.crypto_mapping

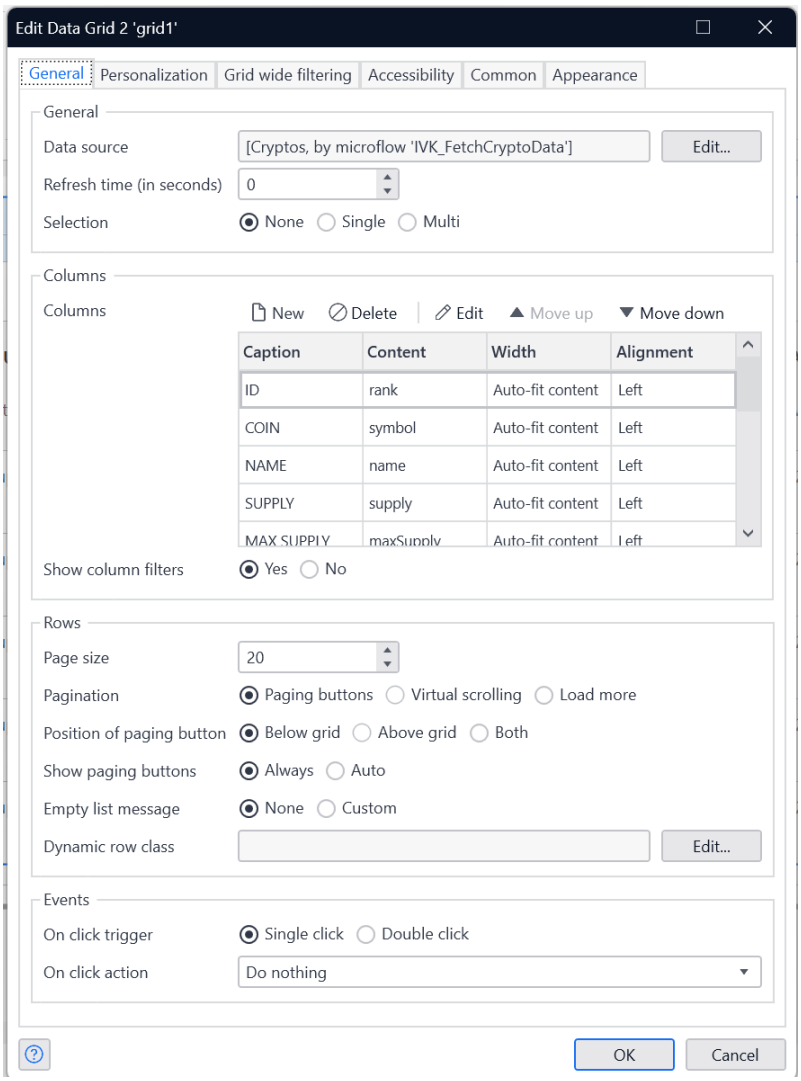
9. Another microflow has to be created, which has already been briefly mentioned, it is "SUB_RetrieveCryptoList. The functionality has already been explained, so only a graphical interpretation is given here:



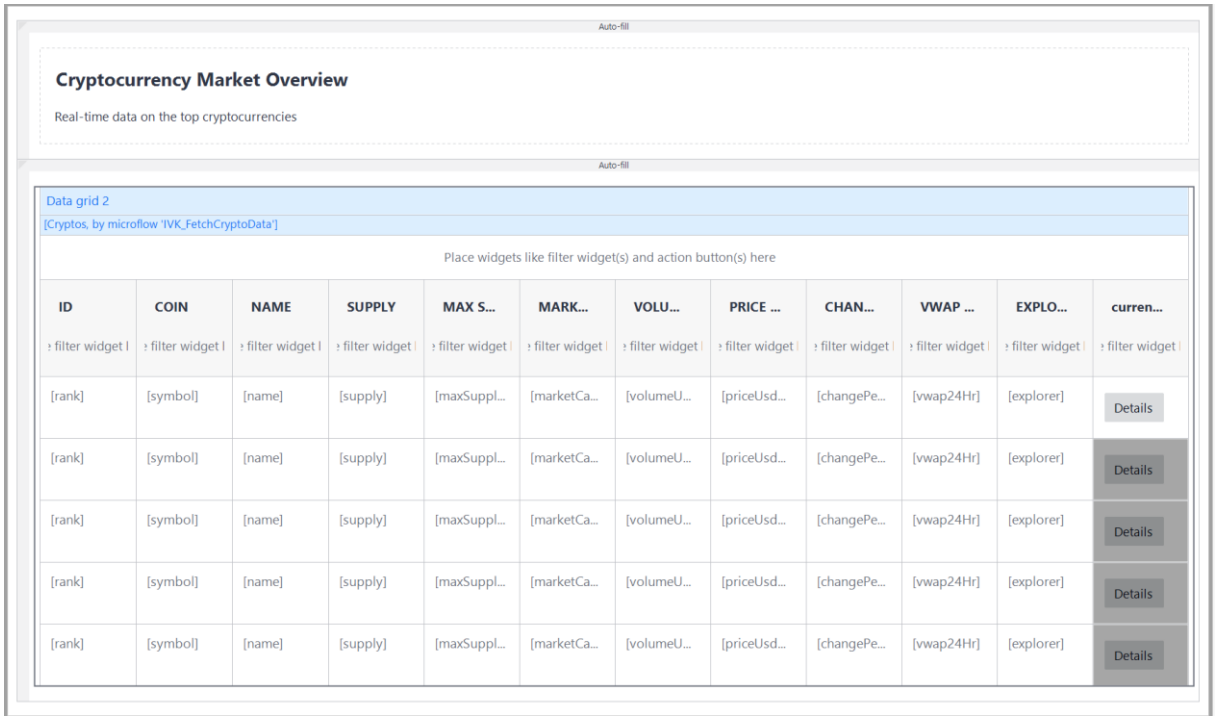
10. Now you can continue working on the "Crypto_Dashboard" page. A "Data grid 2" has already been added here. This is now opened with "double-click" and the following settings are configured:

General tab

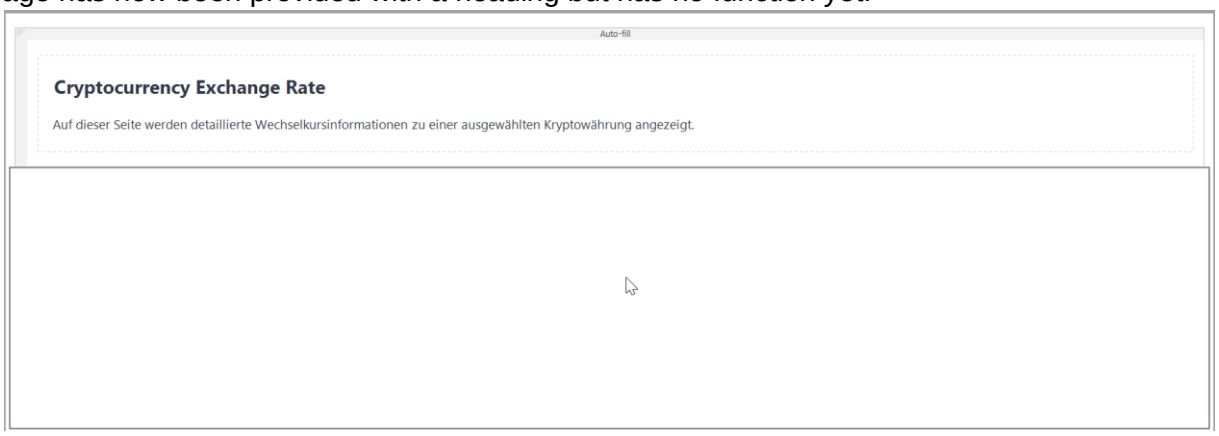
- Datasource: Edit
    - Type: Microflow -> Select -> IVK_FetchCryptoData -> Select all attributes



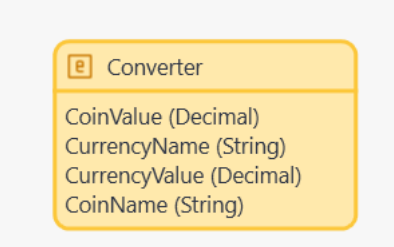
Basically, the dashboard page is now finished, but to get to the Details Page, a button called 'Open page button' is created from the toolbox and used as the label 'Details'. It is sufficient to add it to the first row of the last column. This is automatically added to each additional line.



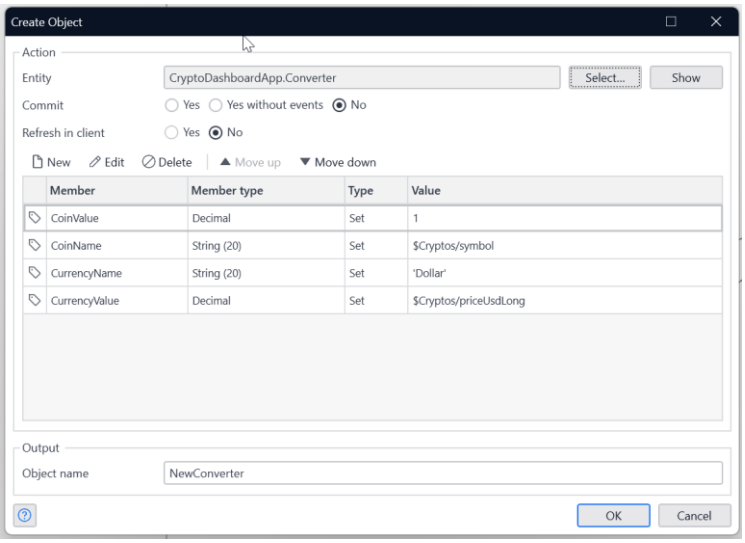
11. A new page is created with the name "Exchange" and the "Details" button is linked to this page. The page has now been provided with a heading but has no function yet.

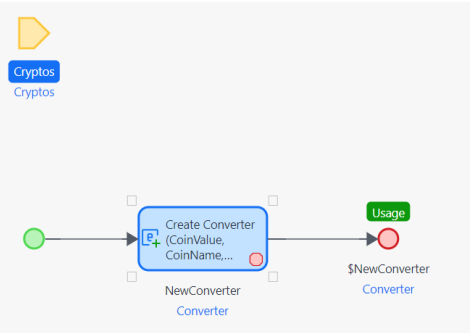


12. In the next step, it is planned that the cryptocurrency, which was selected with "Details", can now be converted into USD dollars. For this purpose, the microflow "ACT_ConvertCurrency" is created (ACT_ stands for user actions). With this microflow, 1 unit in the respective cryptocurrency is initially calculated with USD dollars. In the domain model, another entity is also created for this purpose with the following attributes.

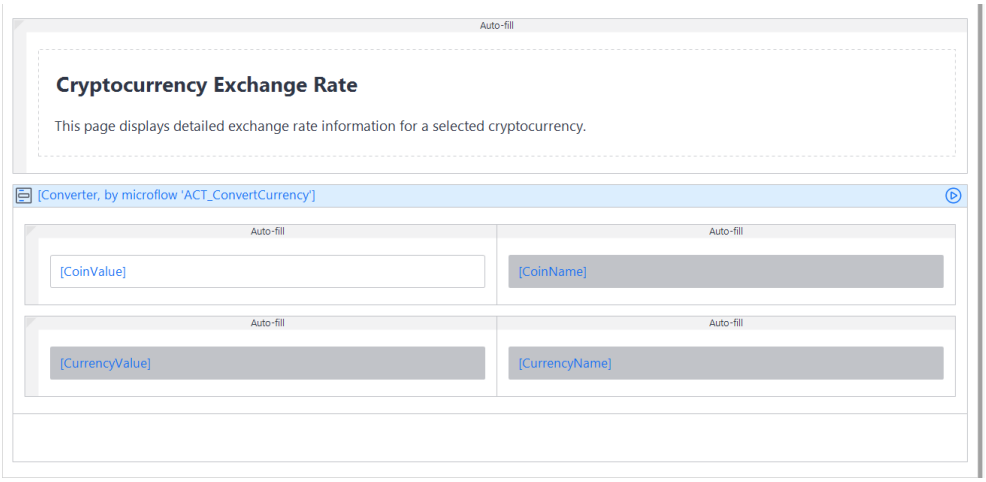


13. In the microflow "ACT_ConvertCurrency", a new converter object is created with the following attributes.



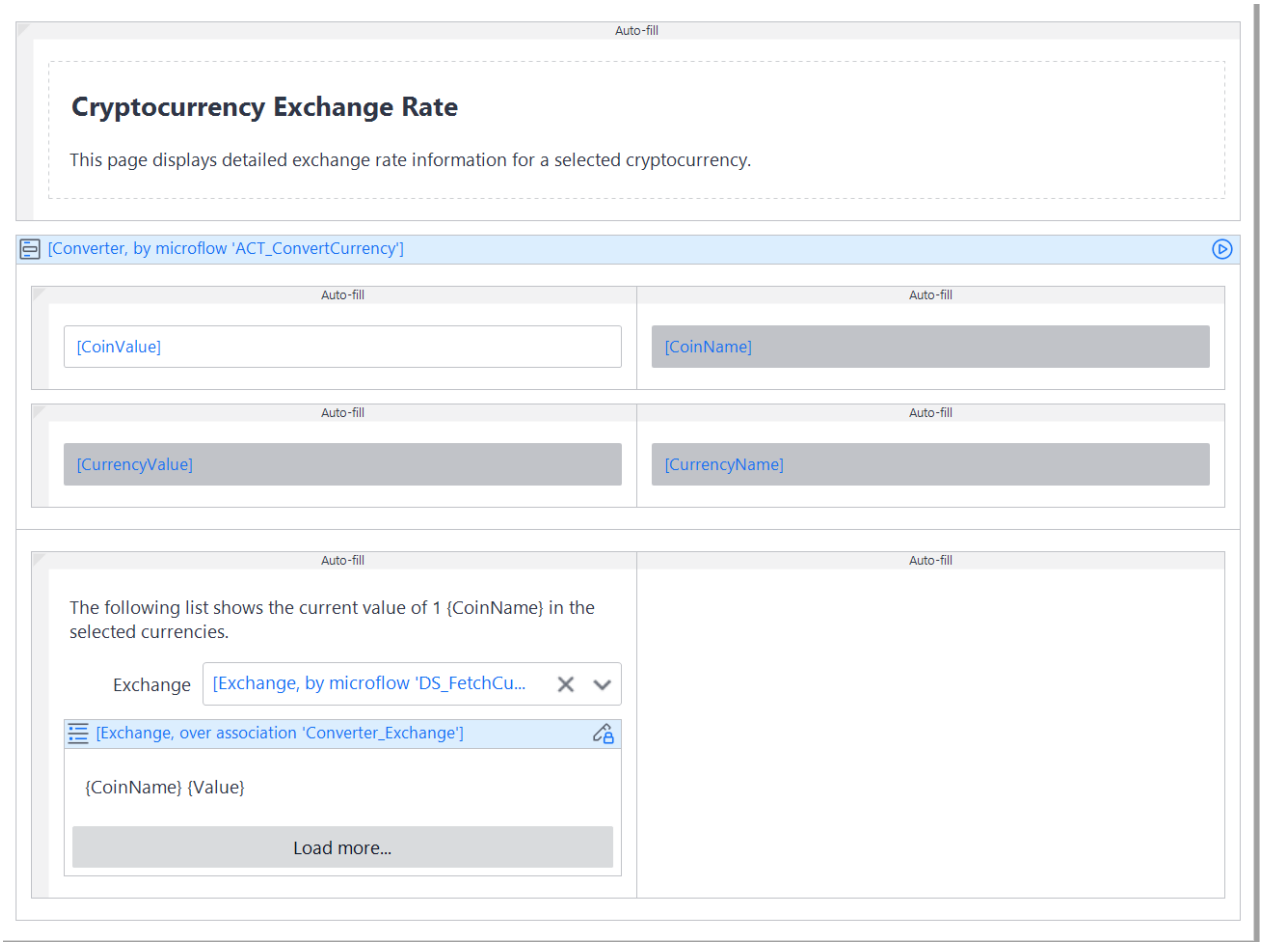


14. In the next step, a dataview is created on the "Exchange Page", which uses the "ACT_ConvertCryptocurrency" microflow that has just been created as a "datasource". In this datagrid, two "layout grids" with the setting "6.6" are inserted. A "Text box" is added to each column. In the first text box, the variable "CoinValue" is configured as "Attributes", in the second "CoinName", then "CurrencyValue" and in the last column "CurrencyName". The "Show Label" setting is set to "No" on all attributes. Furthermore, all attributes except for "CoinValue" are set to "Never".

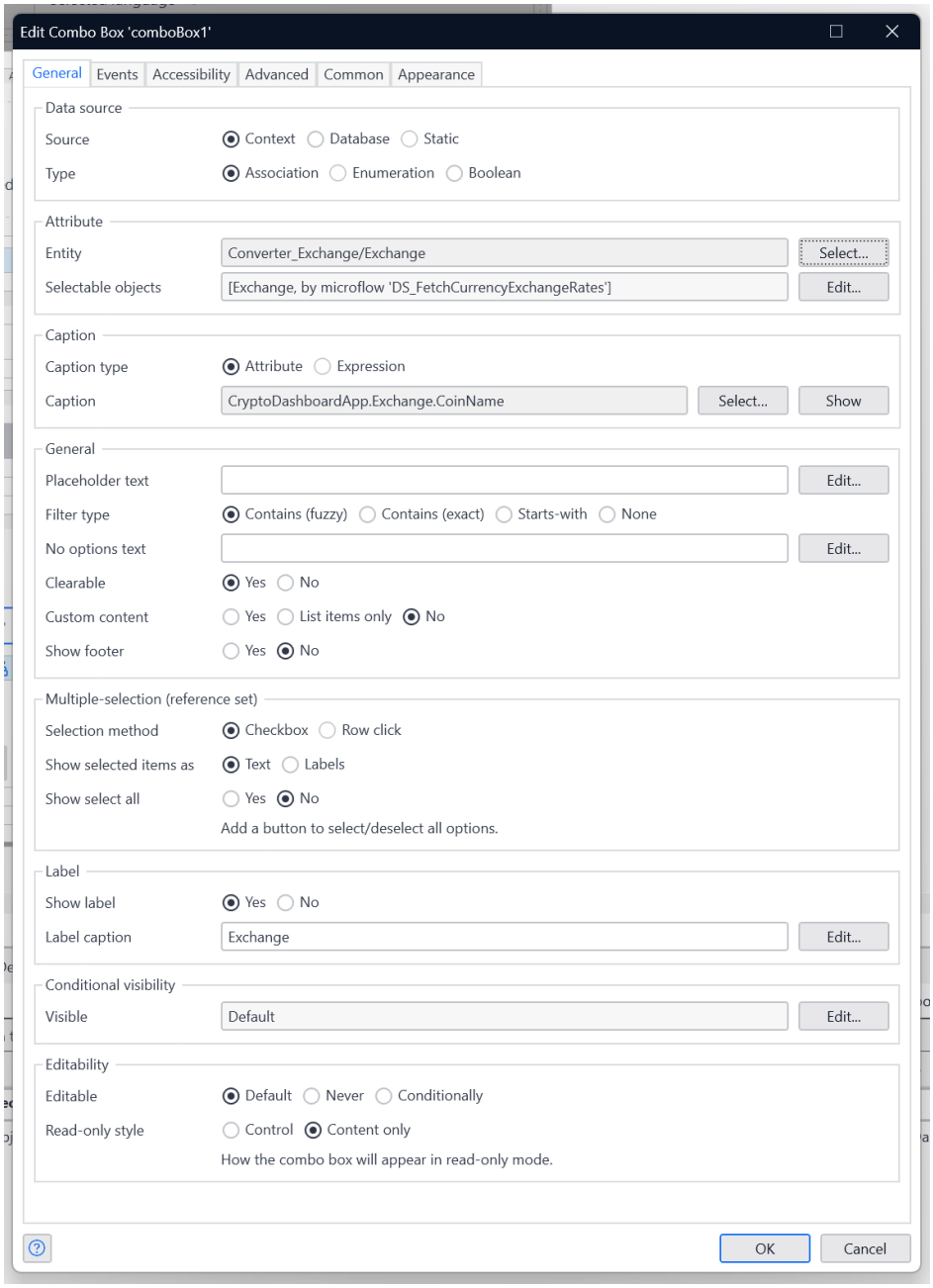


15. Now the original value of the respective cryptocurrency can be displayed in USD dollars when calling up the "Details" page, but if the initial value is changed to another, no change happens. In order to enable a dynamic price adjustment here, another microflow is required. For this purpose, the microflow "SUB_ConvertCurrencyAndChange" is configured. Basically, the respective price is calculated dynamically according to user input. It could have been integrated into the previous Microflow, but the author wanted to make it more modular in order to be able to make adjustments more easily at appropriate points and thus make the application more maintainable.

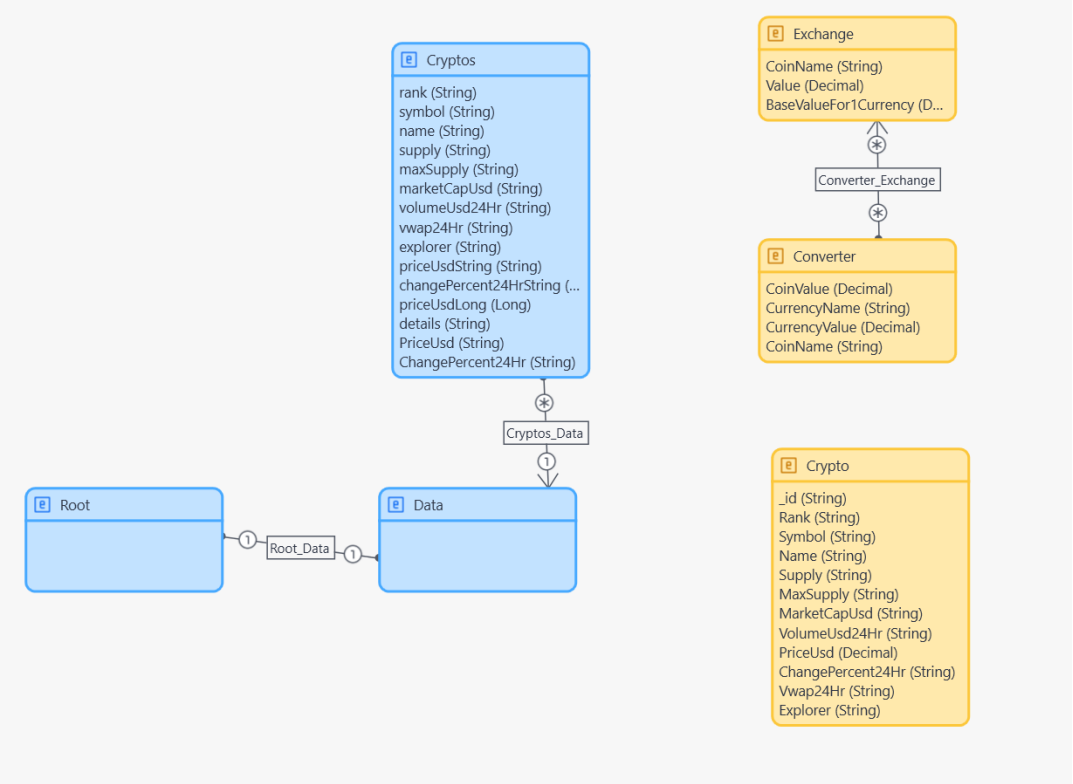
16. The combo box with the selectable national and cryptocurrencies and the dynamic calculation must now be implemented. In this context, the microflow SUB_ConvertCurrencyAndChange is also run through. Firstly, a ‘layout grid’ is created in the page, into which the other widgets are integrated. The combo box is created first, followed by a list showing the selected currencies.



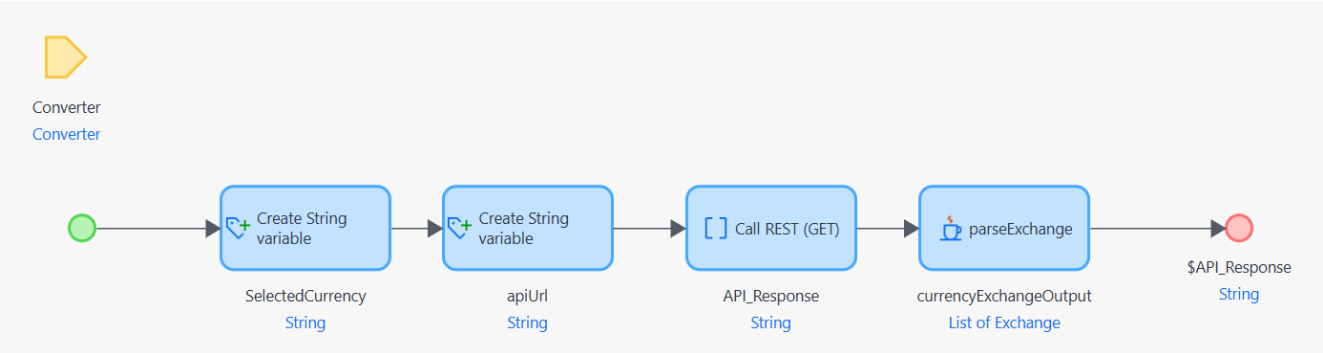
17. The combobox can be added via the Marketplace, the following settings must be configured to obtain the multiselects.



- **Entity:** Here, ‘Converter_Exchange/Exchange’ is selected via an association of Converter. A different API is accessed for the multiselect, which not only has access to cryptocurrencies, but also to national currencies. The entity ‘Exchange’ is a new entity which is basically mapped to each currency that comes back from the API.



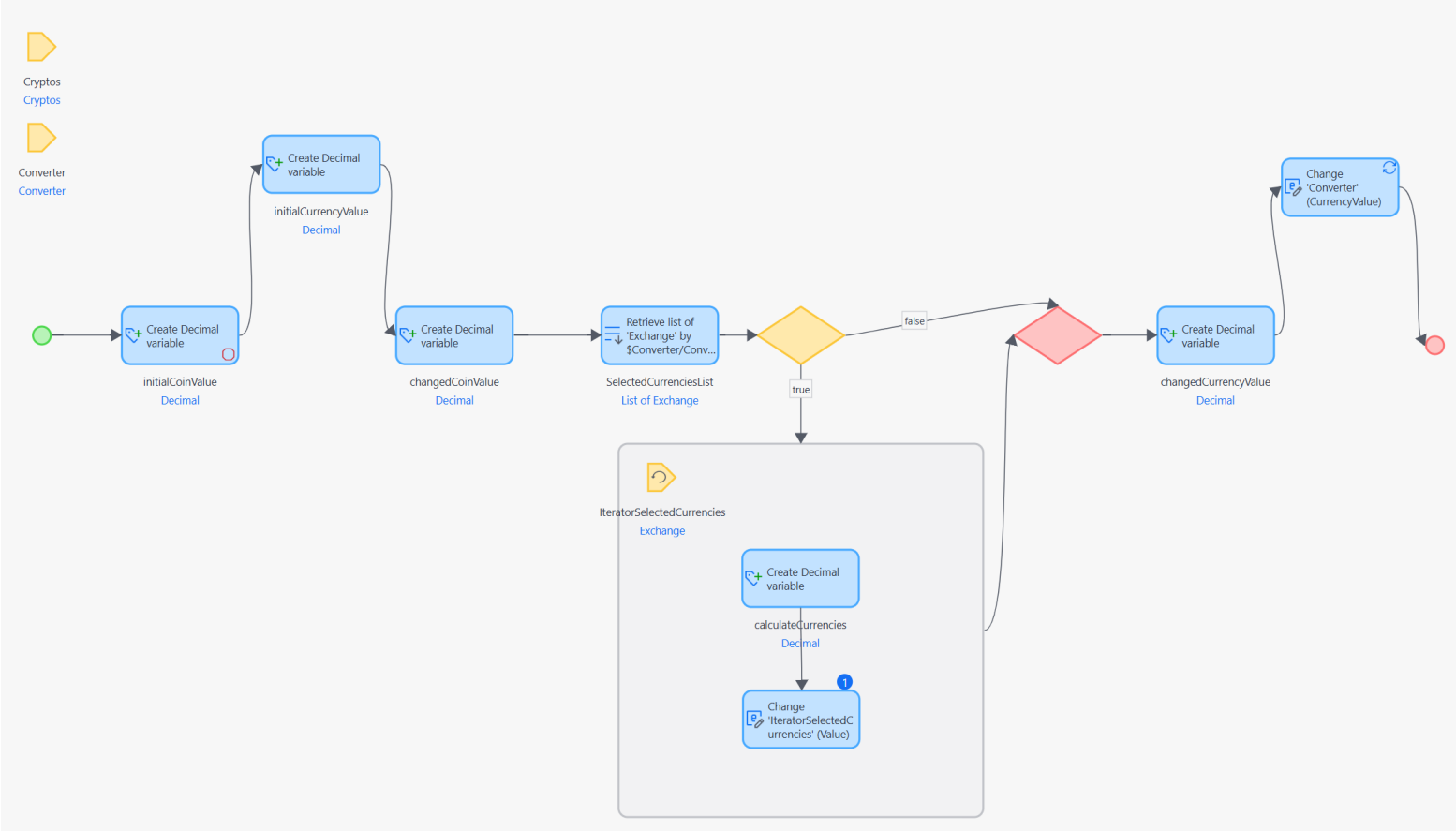
- **Selectable objects:** The microflow ‘DS_FetchCurrencyExchangeRates’ is added here, which basically retrieves the API and maps the data into exchange objects. In this context, the microflow was slightly different to the last one, which was used to retrieve data via an API because in this case a Java action had to be used. In the last API call, it was possible to use an import mapping because it was a fixed structure. This API call is also a JSON response, but this time it is not static but dynamic. A dictionary (key-value pairs) is returned for various currencies, it is not possible to create an entity in Mendix for each currency. That's why the author had to use a Java action that parses the JSON manually, that goes through the individual (dynamic) keys and creates an exchange object for each key.



- **Caption:** Here you basically only select what is to be displayed in the combo box, in this case ‘CoinName’ is used.

18. A ‘List view’ can now be added, the currencies are accessed here as ‘Data source’ via the ‘Converter_Exchange/Exchange’ association. ‘CoinName’ and “Value” are stored in the “List view” so that only these attributes are displayed. Basically, the only thing missing now is the calculation for the dynamic currencies that have been selected.

19. To do this, the microflow ‘SUB_ConvertCurrencyAndChange’ must be edited, which will now be discussed in detail.



- **initalCoinValue**: Creates a variable and sets it to 1.
- **initialCurrencyValue**: Returns the converted USD dollar price that the selected cryptocurrency has at the time the user calls up the details page.
- **changedCoinValue**: Corresponds to the value that was entered in the input. field.
- **SelectedCurrenciesList**: The list of selected currencies in this list is called here
- **calculateCurrencies**: If this list is not empty, the calculation for the dynamic selection takes place in the loop. In the microflow “**DS_FetchCurrencyExchangeRates**” shown above, the initial value is stored in the variable “BaseValueFor1Currency” for each currency that is retrieved from the API and mapped into an exchange object. This means that the initial value of the selected currency is always multiplied by changedCoinValue. This ensures that the initial value is always used, even in the case of multiple calculations.
- With the Change Object Activity within the loop, the “Value” attribute is then updated for each object with the calculation of “calculateCurrencies”.
- **changedCurrencyValue**: Regardless of whether this list is empty or not, “**changedCoinValue”** is then multiplied by “**initialCurrencyValue”** in this variable. This is then the calculation for the cryptocurrency that was called up in the details page. The changes in the UI are then updated with Change Object.

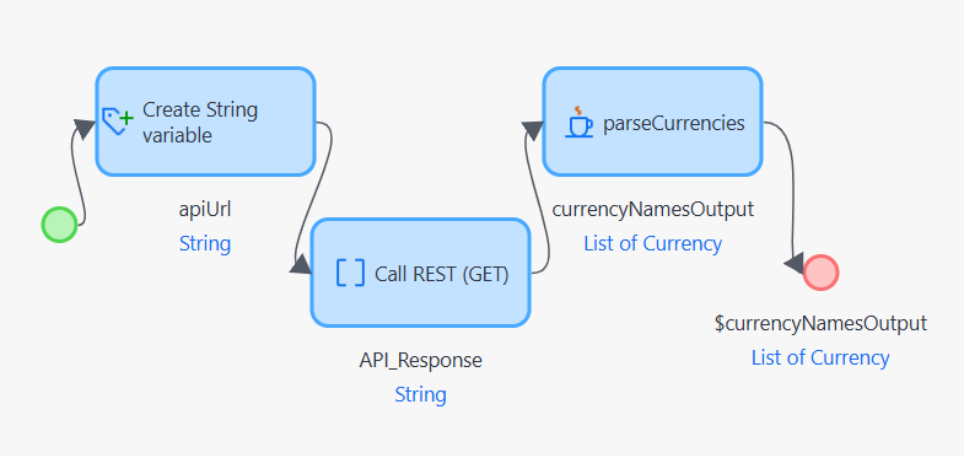
It is now possible to select and dynamically calculate both central bank and cryptocurrencies in the multi-select. Currently, however, only the symbol and the corresponding value are displayed. To additionally display the name, instead of

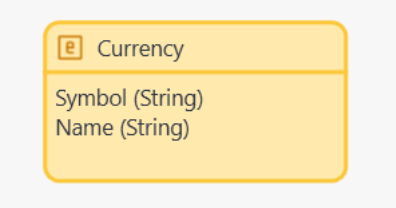
```
ada: 350097.16,
…
```

the following format must be achieved instead:

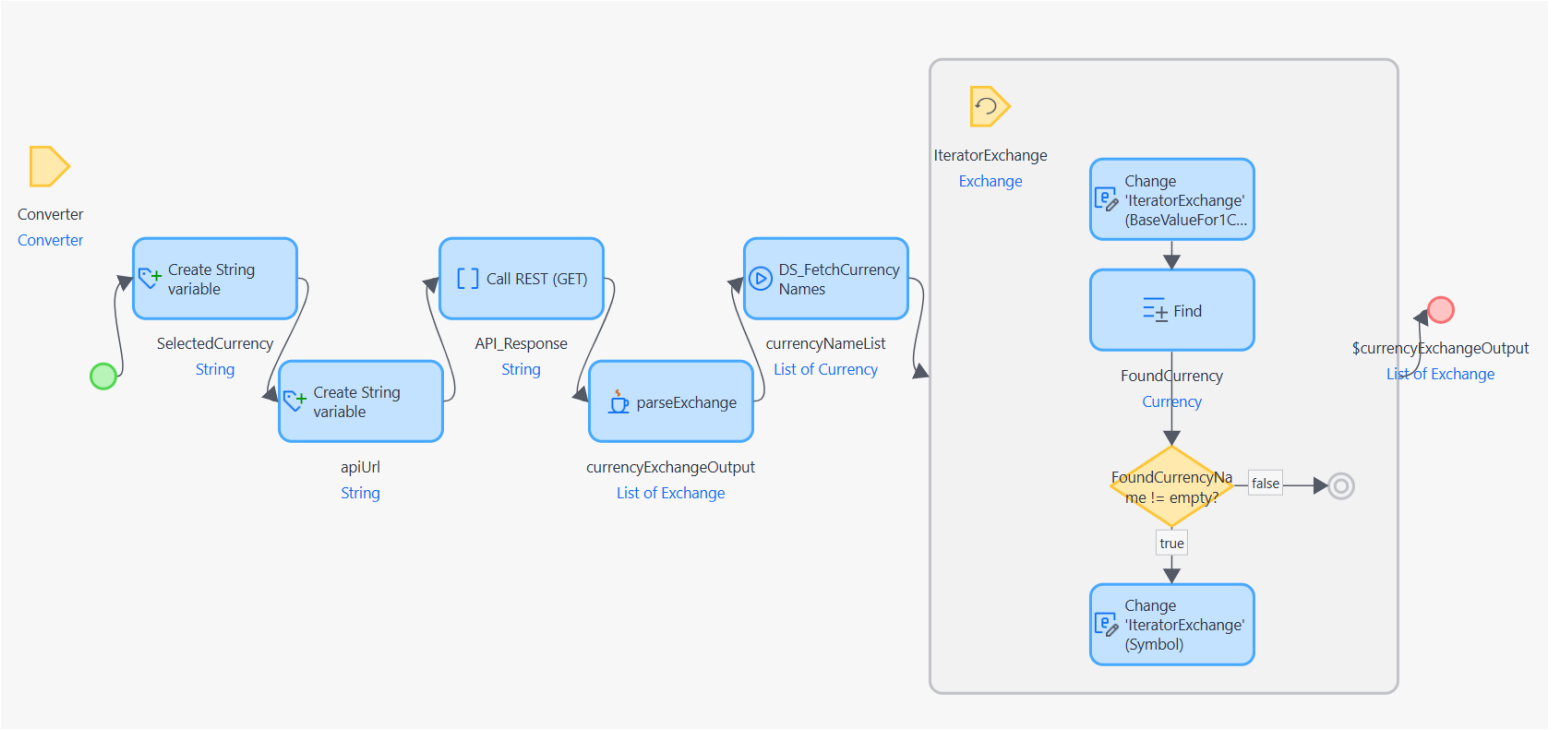
```
Cardano – ada: 350097.16,
…
```

Another microflow called ‘DS_FetchCurrencyNames’ must be created for this purpose. This calls another external API (from the same source) and uses the Java action ‘parseCurrencies’ to save the received key-value pairs in a Mendix entity. A ‘Currency’ entity was defined for this purpose.





20. The newly created microflow is then integrated into the existing microflow ‘DS_FetchCurrencyExchangeRates’. The lists must be adapted so that the desired display format is achieved.



Essentially, the new microflow is called, whereupon a list ‘currencyNamesList’ is returned. This list is run through in a loop, as already described in step 17. In addition to the direct attribute access, this time the list operation ‘Find’ is used to check whether there is a matching symbol in ‘currencyNamesList’ for an element in ‘currencyExchangeOutput’. If this is the case, the attribute is updated to ‘$FoundCurrency/Name + “ - ” + $IteratorExchange/CoinName’.

Now the implementation is completed and it is possible with Mendix and Java/Angular to call up other cryptocurrencies via a combo box in the Details Page, this was not possible in the first attempt with Mendix.

Use Case 2 - Java/Angular/MySQL

Backend, Frontend, Database and Containerization same version as above in first usecase.

Dependencies

The following dependencies are used in the **Spring Boot** project, which are defined via **Maven** in pom.xml:

- Spring Boot Starter Web
  Purpose: Provision of RESTful APIs.
- Spring Boot Starter Security
  Purpose: Implementation of authentication and authorization.
- Spring Boot Starter Data JPA
  Purpose: Communication with the database via JPA.
- MySQL Connector
  Purpose: Connection to a MySQL database.

- spring-boot-starter-validation
  Purpose: Enables validation of request and data objects using annotations like @Valid and @NotNull.
- spring-boot-starter-webflux
  Purpose: Provides support for reactive programming and non-blocking web applications.
- Lombok
  Purpose: Reduction of boilerplate code.
- Spring Boot Maven Plugin
  Purpose: Support for the build and deployment of Spring Boot applications.

  In addition, a short list of dependencies for Angular:

- @angular/animations
  Purpose: Provides support for animations in Angular applications (e.g., transitions, keyframes).
- @angular/common
  Purpose: Contains commonly used Angular directives and services, such as NgIf, NgFor, and HTTP utilities.
- @angular/compiler
  Purpose: Compiles Angular templates and components, enabling dynamic template rendering.
- @angular/core
  Purpose: The core of Angular, including essential building blocks like components, directives, and dependency injection.
- @angular/forms
  Purpose: Adds support for template-driven and reactive forms, making it easier to manage user input and validation.
- @angular/platform-browser
  Purpose: Provides services and utilities for browser-based applications, including DOM manipulation
- angular/platform-browser-dynamic
  Purpose: Supports Just-in-Time (JIT) compilation for dynamically loading and rendering Angular components.
- @angular/router
  Purpose: Enables routing and navigation between different views or components in the application.
- Primeicons
  Purpose: A library of pre-designed icons, typically used with PrimeNG UI components.
- Primeng
  Purpose: A rich library of UI components for Angular, including data tables, dropdowns, and calendars.
- rxjsPurpose: A library for reactive programming, allowing you to handle asynchronous data streams using observables.
- Tslib
  Purpose: A runtime library for TypeScript that provides helper functions to reduce boilerplate in the compiled JavaScript.
- zone.js
  Purpose: Enables Angular's change detection by managing execution contexts, ensuring UI updates are reflected correctly.

  Implementation: For this use case, a prefabricated implementation was deliberately omitted so that all functionalities were implemented independently. The backend can essentially be divided into three central task areas: Firstly, the storage and retrieval of data, secondly, the retrieval of all entries from the database and thirdly, the retrieval of a single cryptocurrency from the database. In Spring Boot, the application.properties is configured accordingly. The connection settings for the MySQL database are defined here, such as URL, username and password. In addition, the Hibernate dialect

for MySQL is defined, the automatic update of the database schema is activated (ddl-auto=update) and the display of the executed SQL queries (show-sql=true) is switched on. These settings ensure the connection and interaction of the application with the database.

1. Set up the projects in Angular and Spring Boot with the respective dependencies, settings and docker files. A Docker-Compose file must also be created, which provides the respective infrastructure for each container.

   Implementation begins with setting up the backend, as the frontend accesses the provided interfaces (APIs) later on. In principle, however, it would also be possible to choose the reverse development path and implement the frontend first.

   The directory structure is as follows:

```
/main/java/app/web/cryptodashboard
├── config              # provide HTTP client handling for external API calls
├── controller          # Contains REST controllers
├── model               # Data models (e.g., CryptoCurrencyDTO, CryptoApiResponse)
├── repository          # Database repository interfaces
├── security            # Spring Security configuration
├── service             # Business logic (e.g., Crypto Currency Service)
```

   Create the entity for database.

   Cryptocurrency.java

- Fields: id, symbol, name, supply, maxSupply, marketcapUsd, volumeUsd24hr, priceUsd, changePercent24hr, Vwap24hr, explorer
- This entity represents a cryptocurrency in the application's database. It includes the attributes with mappings to corresponding database columns using JPA annotations.

   Create CryptoCurrencyDTO

   A simple data object that is used for data exchange between the service and controller. Contains only the necessary fields (e.g. symbol, priceUsd) to deliver data to the REST API endpoints and to abstract the internal logic from the API.

   Create CryptoApiResponse

   Models the response of the external API (https://api.coincap.io/v2/assets/). Contains a list of CryptoCurrencyDTO that represents the retrieved cryptocurrency data.

2. Now the repository interfaces are integrated and Spring Data JPA is used to manage cryptocurrencies. The **CryptoCurrencyRepository** is an interface provided by **Spring Data JPA**. It is used to enable CRUD operations (create, read, update and delete) for the **Cryptocurrency** entity without having to write your own database access code. The repository extends the **JpaRepository** interface and thus inherits standard methods such as **save**, **findById** and **delete**.

CryptoCurrencyRepository.java

One specific method has been added to this Repository to support the business logic.

- **findBySymbol(String symbol)**
  This method is used in the fetchAndSaveCryptoData() method to check whether there are already entries with the symbol in the database.

3. Create CurrencyService and CryptoDashboardService(Service Layer)
   Implements the business logic of the project.

CryptoDashboardService

- **fetchAndSaveCryptoData():**
  Retrieves data from the API and saves it in the database. Checks whether an entry already exists and updates it or creates a new one.
- **getAllCryptoCurrencies():**
  Retrieves all cryptocurrencies from the database and returns them as DTOs.
- **getCryptoCurrencyByID(Long id):**
  Retrieves a cryptocurrency based on its ID and returns it as a DTO.

Mapping methods

- **mapDtoToEntity():** Converts a DTO into a database entity.
- **mapEntitytoDto():** Converts an entity into a DTO

CurrencyService

- **fetchCurrencyNames()**
  Retrieves data from an external API to get currency names for the MultiSelect. No saves in database.
- **fetchCurrenciesForBase()**
  This endpoint provides the conversion rates of the selected cryptocurrency (e.g. BTC) into various fiat and cryptocurrencies, which can then be selected via the MultiSelect. No saves in database.

4. Create CryptoDashboardController and CurrencyController (REST Controller)
   Provides the REST endpoints. Processes HTTP requests and delegates the logic to the CryptoDashboardService and CurrencyService.

CryptoDashboardController

- **/api/crypto/fetch-and-save:** Starts the retrieval and storage of cryptocurrency data.
- **/api/crypto/all:** Returns all stored cryptocurrencies
- **/api/crypto/{id}:** Retrieves the data of a specific cryptocurrency based on the ID.

CurrencyController

- **/api/currencies/names:** Retrieves names of the currencies
- **/api/currencies/base/{baseCurrencies}:** Returns conversion rates of one specific cryptocurrency

5. Creates AppConfig (Configuration Class)
   Defines beans that are required for the project. Provides a **RestTemplate** bean that is used for API calls.

6. The next step is to implement basic Spring Security. Spring Security is used in the project to secure access to the API endpoints.

   SecurityConfig.java

   Essentially, this security config is configured in exactly the same way as Use Case 1, with the difference that other enpoints are permitted.

7. Since the frontend was already set up in the very first step, the implementation is now continued. The entry point of the application is ‘app.component.ts’ and ‘app.route.ts’ provides the routing within Angular. There are /crypto-table and /crypto-details/:id which can be called.

8. crypto-table.component.ts connects the API data with the table and controls scrolling sorting and navigation. The loadCryptocurrencies() method retrieves data from the REST API and processes it (e.g. converting strings into numbers). Functions such as next(), prev() and reset() control navigation through the pagination. goToDetails(cryptoId) navigates to a details page for a specific cryptocurrency. crypto-table.component.html displays an interactive table with cryptocurrency data. It uses PrimeNG components such as ‘p-table’ and ‘p-button’ to enable sorting, pagination and dynamic content.

9. crypto-detail.component.ts implements the loading of cryptocurrency details and the calculation of conversion values. Loads details of a cryptocurrency based on the ID from the URL. Implements functions to convert cryptocurrency to USD (calculateUsdFromCrypto) and vice versa (calculateCryptoFromUsd). Provides a dynamic MultiSelect list for comparison with other currencies and updates the corresponding conversion values. crypto-detail.component.html shows the details of a selected cryptocurrency and offers functions for conversion to USD and comparison with other currencies. Displays information such as price, market capitalisation, volume and change (24h). Enables conversions between the cryptocurrency and USD with input fields Offers a drop-down selection (MultiSelect) for comparison with other currencies.

   The application is now fully prepared and can be started using the Docker Compose file. The entire environment can be started with the docker-compose up command in the root directory of the project, where the docker-compose file should also be located.

Use Case 3 – Mendix (v 10.18.1)

An existing project from the official Mendix Marketplace is used for the comparison.

Mendix Task & Planning**:** https://marketplace.mendix.com/link/component/108938

The aim of the implementation is to have a Kanban board with which it is possible to create tasks and set them to To Do, In Progress, To Review or Done. For each task a title, description, due date, priority and status can be set. Files can also be uploaded and assigned.

Prerequisites**:**

- Mendix Studio Pro (v 10.18.1)

Modules:

- Standard

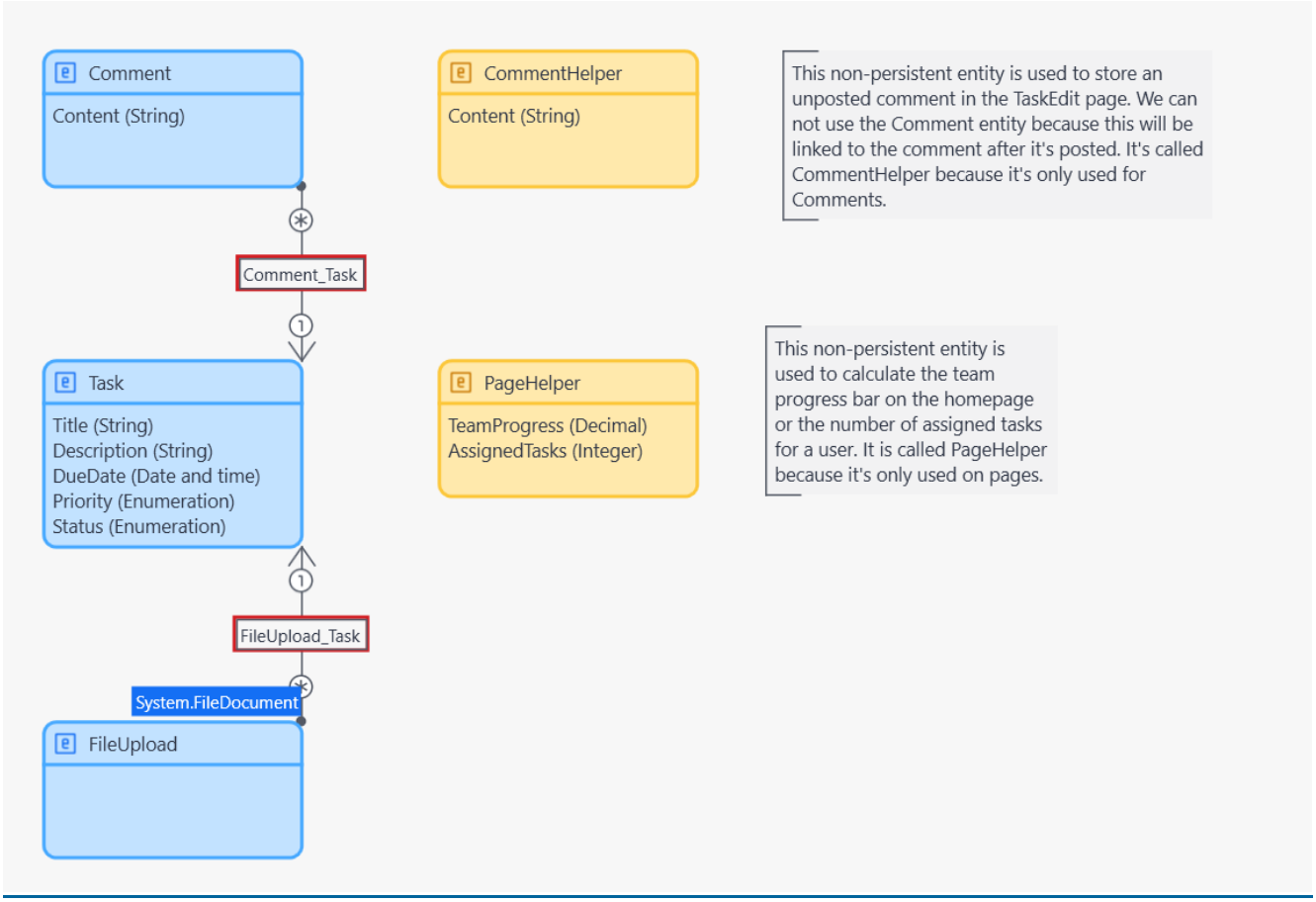
Dependencies:

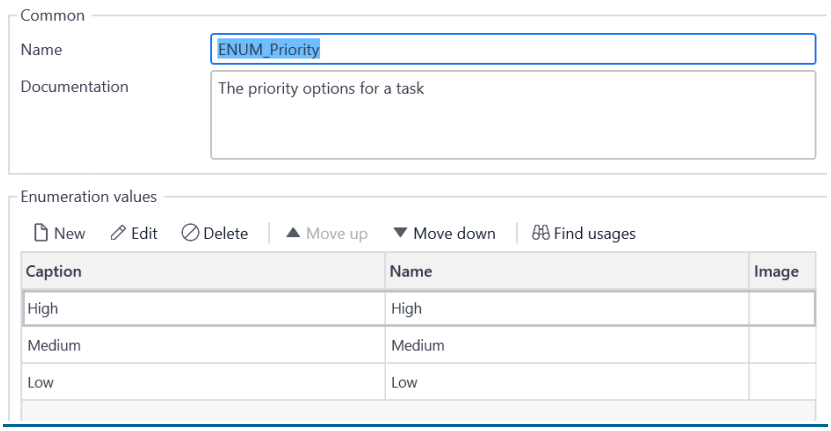
- Standard

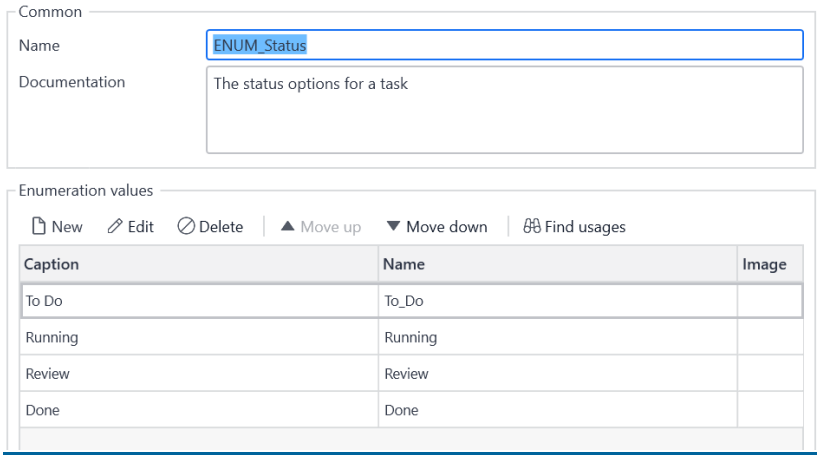
Implementation:

An already finished implementation of Mendix could be found and used for this use case [5]. The development steps are only reproduced here. The Mendix SSO access and everything associated with it was removed as it was not usable in the free version.

1. The following entities are created for the domain model. In addition to the domain model, enumerations for priority and status are also created.

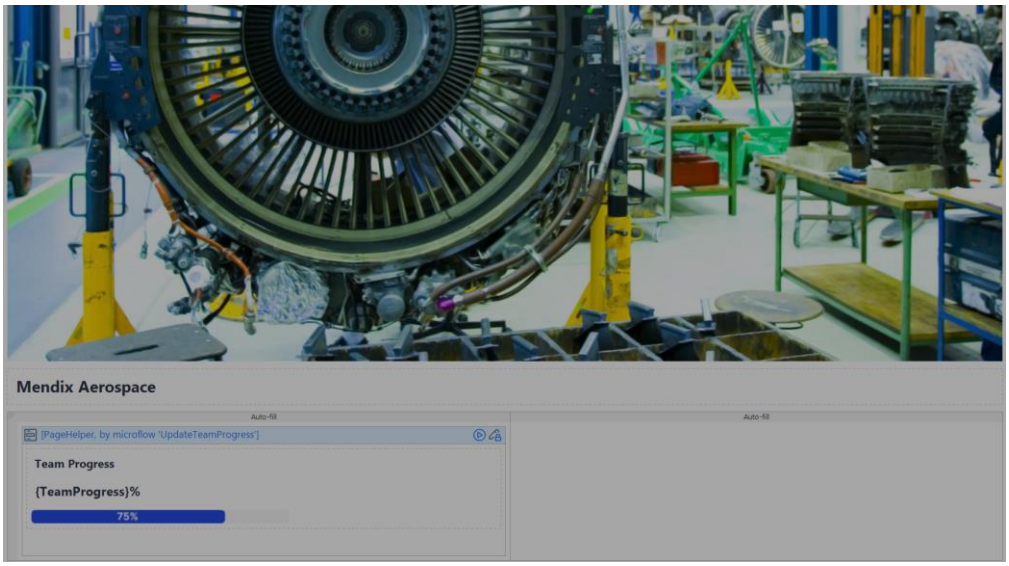




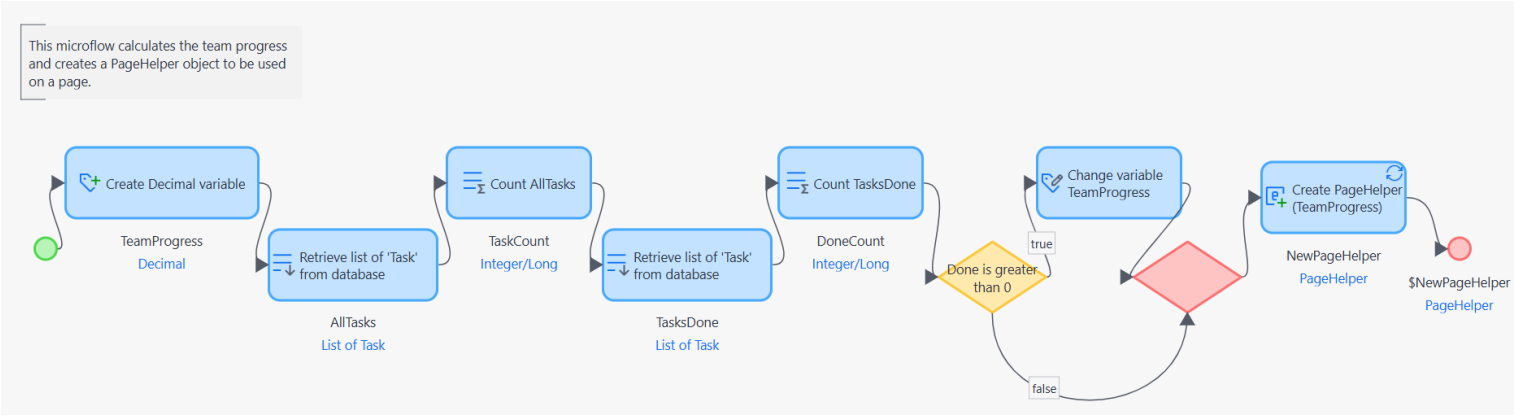


2. The Kanban board with the columns **To Do, In Progress, To Review, Done** is located on the **TaskOverview** page. This is the main page of the application. The page is divided into header, board and individual tasks. Tasks are added via an action button (+ icon). The individual components are now created.

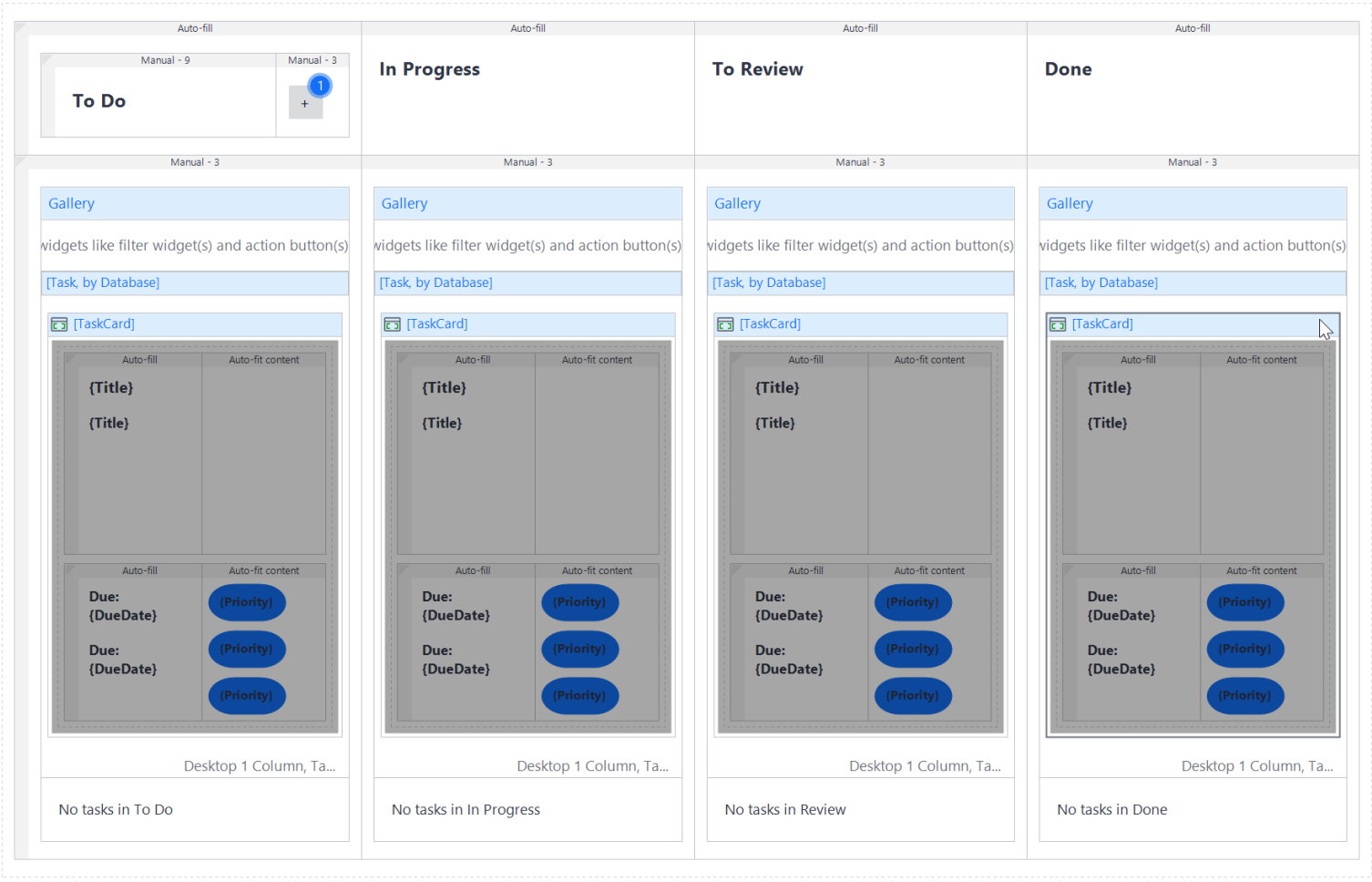
   The header contains an image (resolution 1254x836) and the progress bar. The graphic is simply added. A data view is created and a new microflow, 'UpdateTeamProgress', is added as the data source. The progress bar can be found in the toolbox and is stored in the data view.

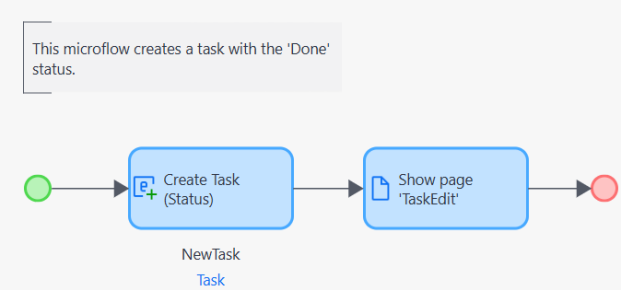


   The microflow 'UpdateTeamProgress' must calculate the progress of how many tasks have already been set to 'Done'.

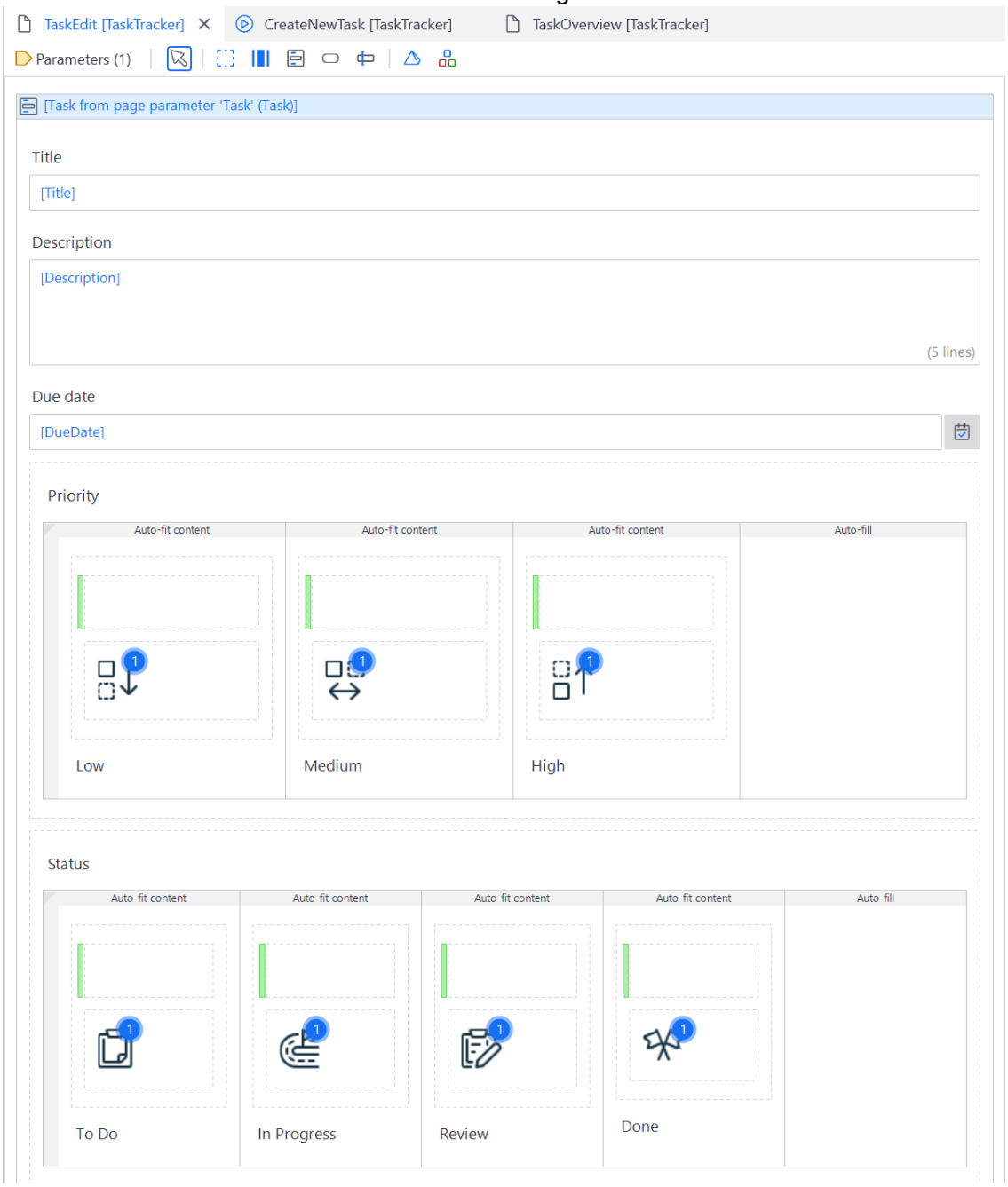


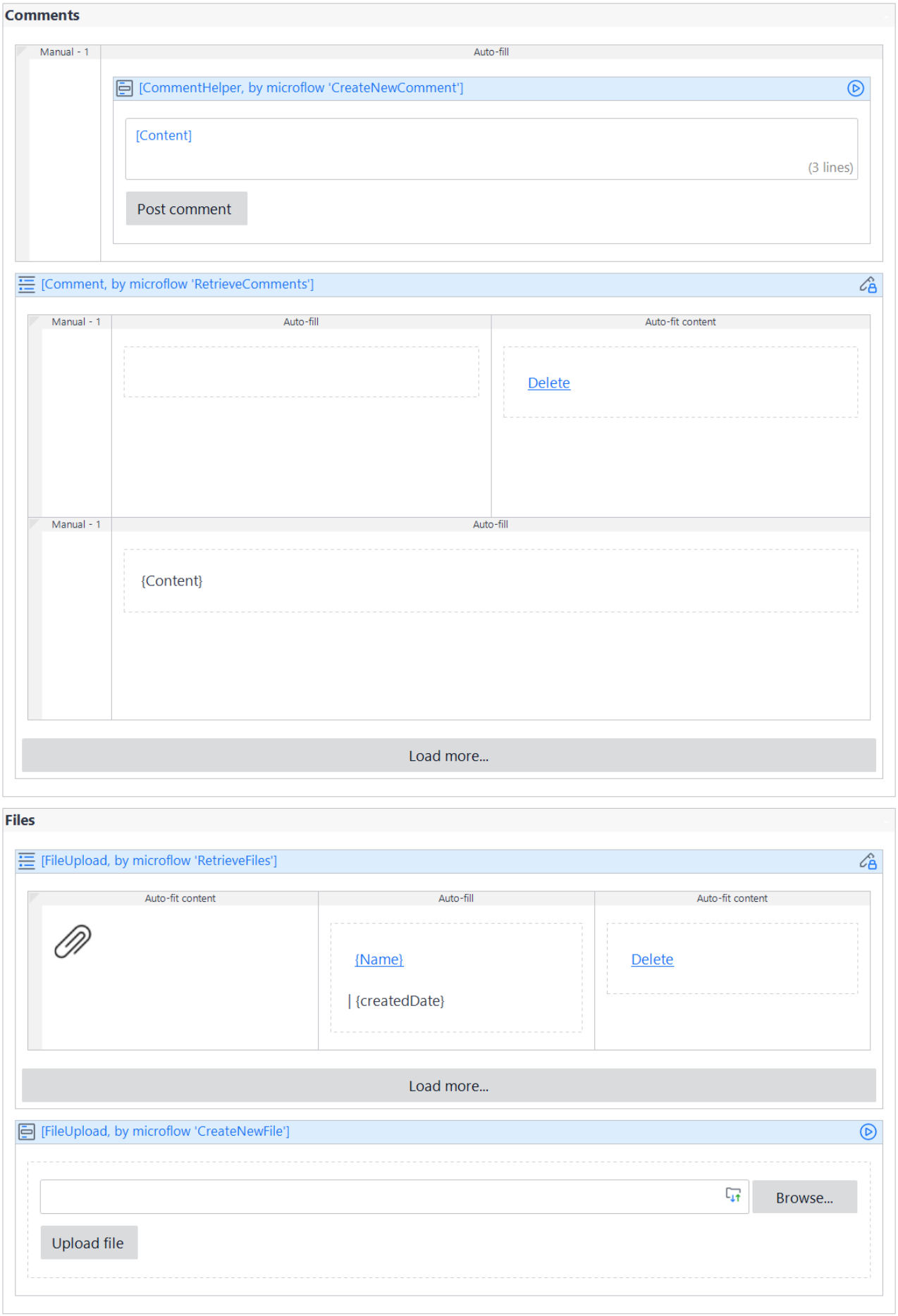
3. Back on the TaskOverview page, start creating the board itself. A div and two layout grids with 4 columns each are created for this purpose. The first 4 columns practically only represent the title, and the tasks can be created via the action button '+'. The action button '+' calls the new microflow 'CreateNewTask'.

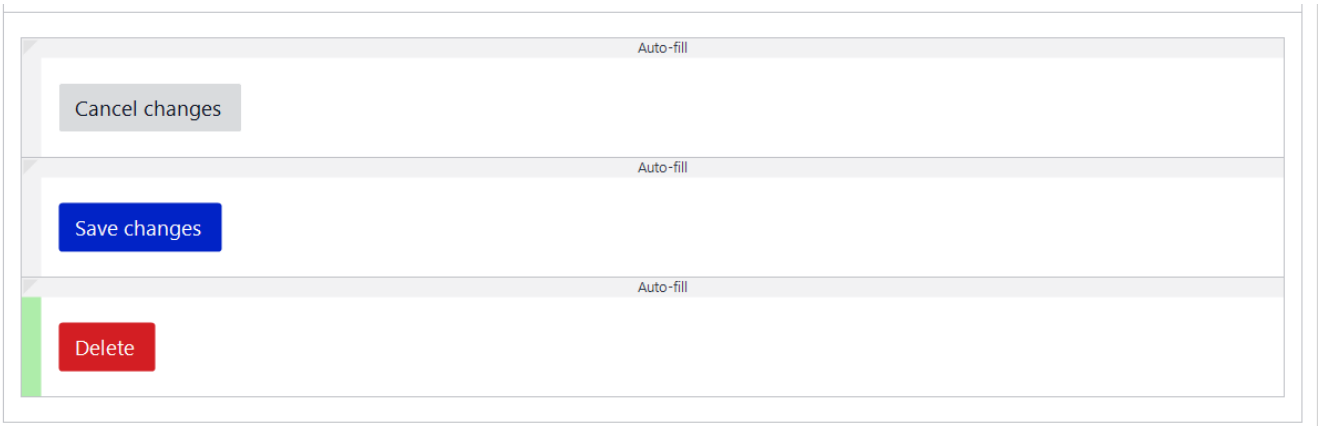




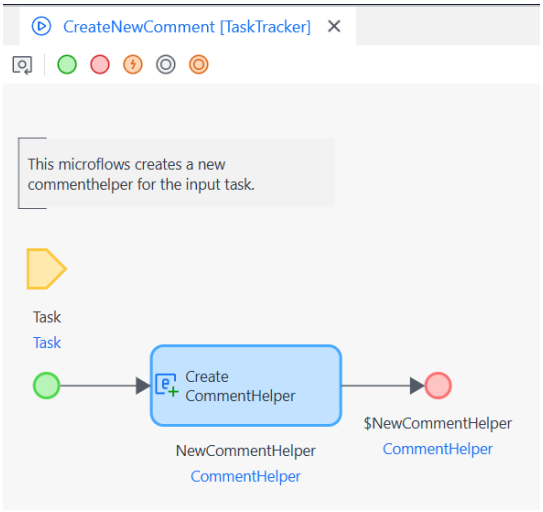
4. The microflow has transferred a task object to the new 'TaskEdit' page. All details for the task are created here. First, a dataview is created and a task is transferred to it as a datasource. A 'Text Box' from the toolbox is used as the 'Title' and 'Description'; the attributes are already provided by the task object. A date picker from the toolbox is also used for the 'Due date'. A layout grid with 4 columns is created for the 'Priority' and the 'Status'. The graphics for Priorities and Status are inserted and the enumeration for both are also assigned.

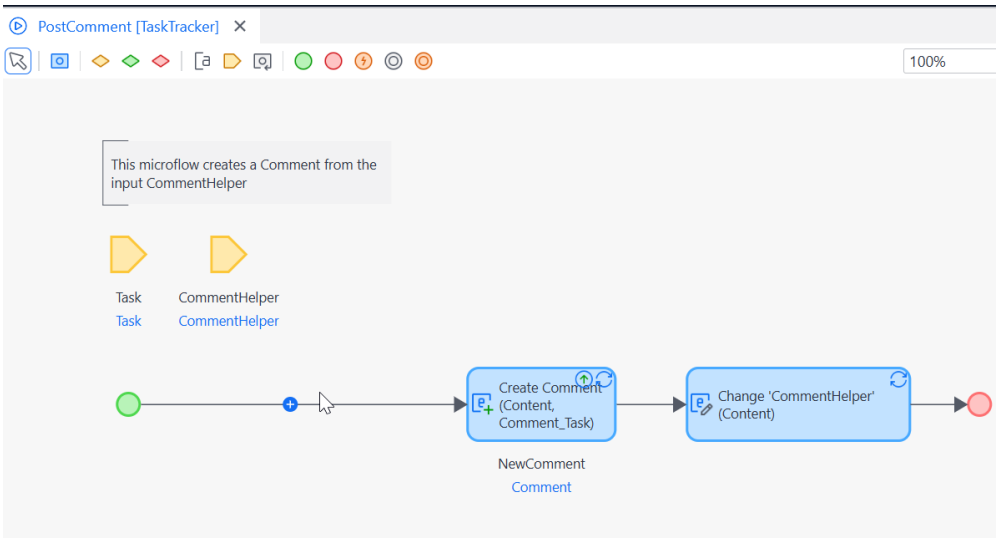



Comments
Manual - 1
Auto-fill
[CommentHelper, by microflow 'CreateNewComment']
[Content]
(3 lines)
Post comment
[Comment, by microflow 'RetrieveComments']
Manual - 1
Auto-fill
Auto-fit content
Delete
Manual - 1
Auto-fill
{Content}
Load more...
Files
[FileUpload, by microflow 'RetrieveFiles']
Auto-fit content
Auto-fill
Auto-fit content
{Name}
{createdDate}
Delete
Load more...
[FileUpload, by microflow 'CreateNewFile']
Browse...
Upload file

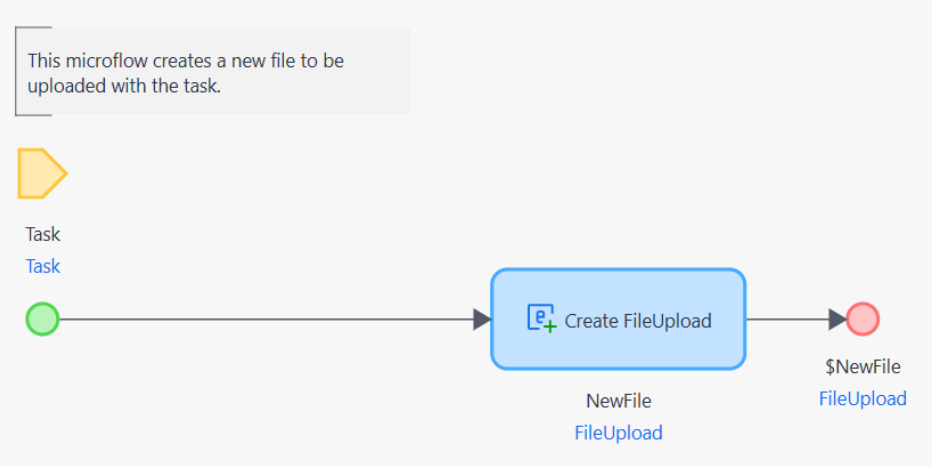


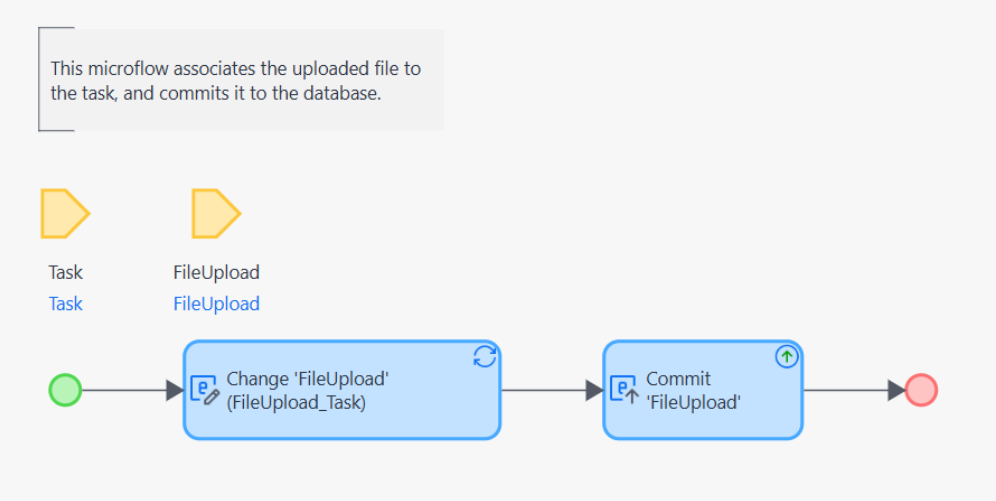
5. A dataview is created for 'Comments' which uses a new microflow 'CreateNewComment' as datasource. This microflow transfers the new comment to the new microflow 'PostComment' via the entity 'CommentHelper' by triggering the action button 'Post comment'. This is then responsible for processing the new comment.



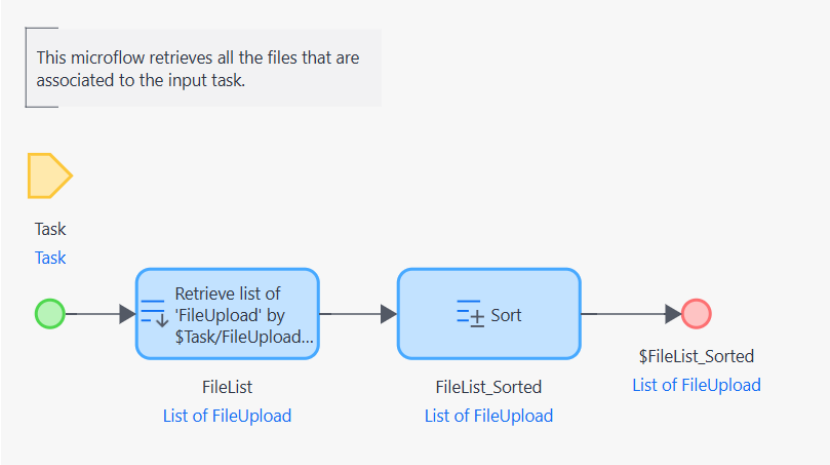


6. The FileUpload has a Dataview on the one hand and a ListView on the other. The Dataview is responsible for the file upload. Here the entity ‘FileUpload’ is initialised in the microflow ‘CreateNewFile’. This is followed in the Dataview by the File Manager which can be added from the Toolbox. Here the type is set to ‘Upload’ and the maximum file size is set to 5 MB. The action button ‘Upload File’ calls the new microflow ‘UploadFile’ which saves the file.

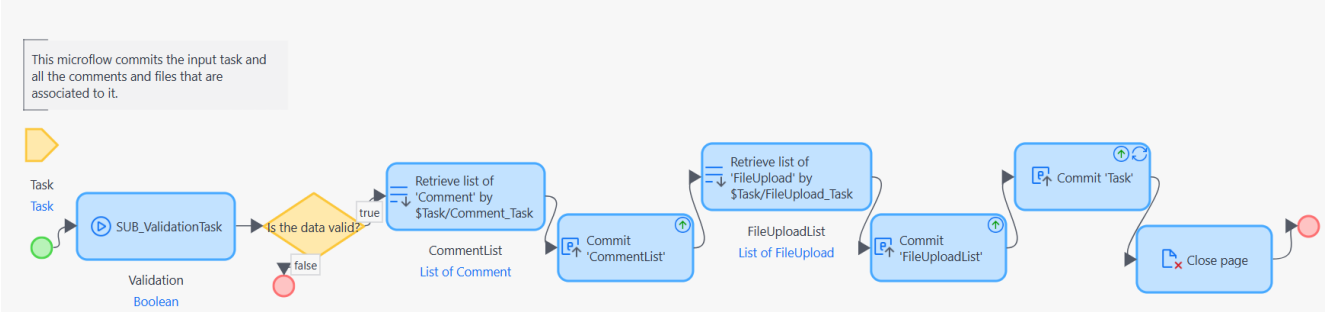




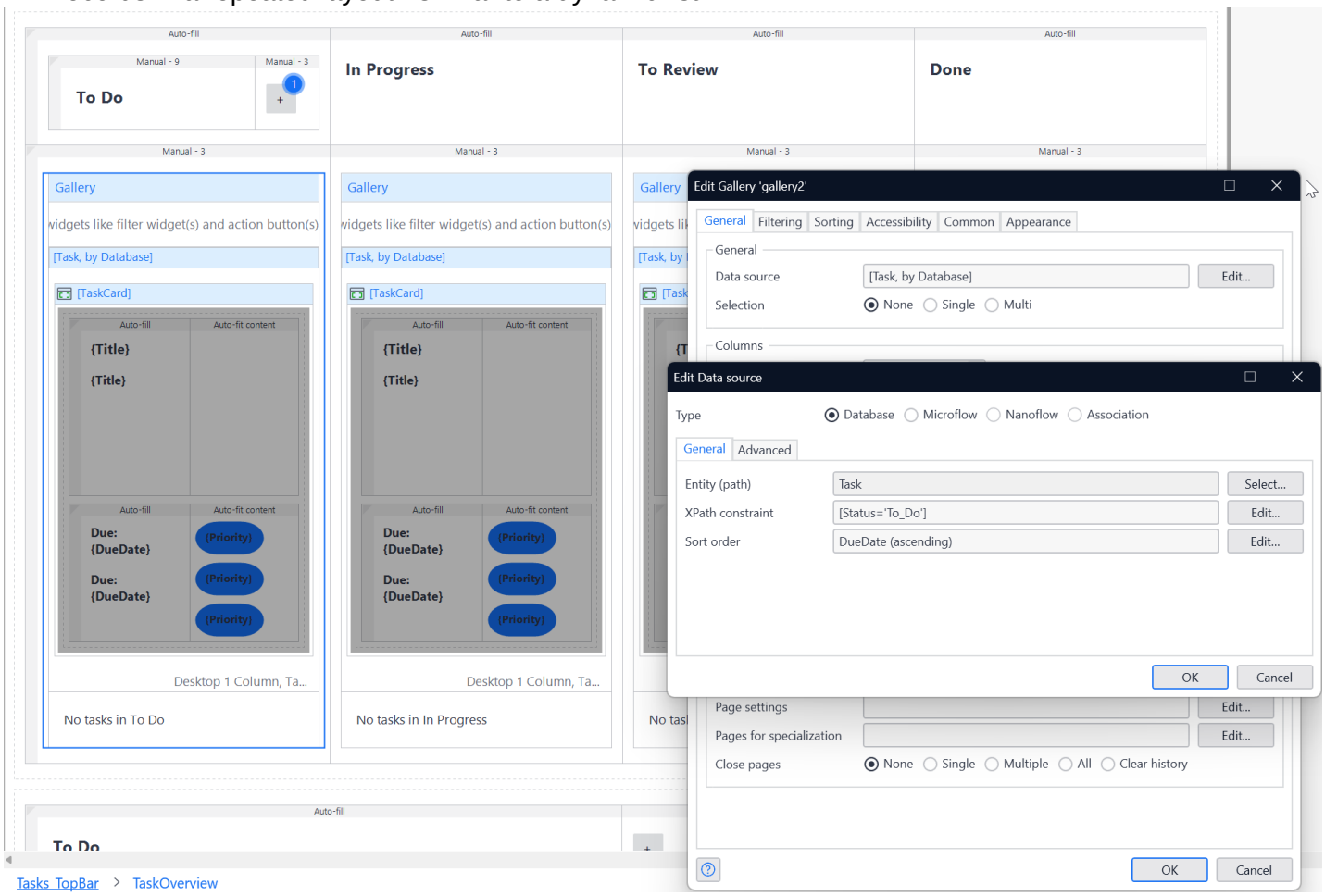
The List View contains the files that have already been uploaded to this task. A new microflow ‘Retrieve Files’ is selected as the data source here.



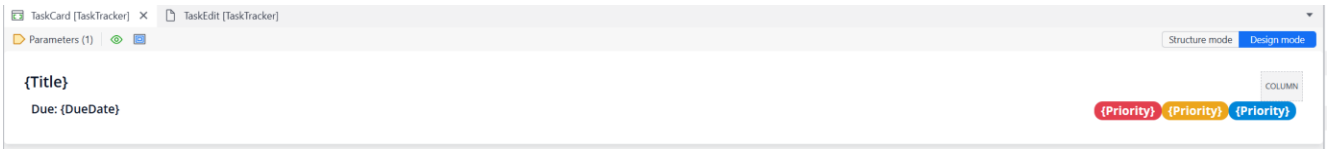
8. Last but not least, there is the option of canceling in 'TaskEdit' with 'Cancel changes' and saving the new task with 'Save changes'. The task is then saved using the new microflow 'SaveTask'.



9. Back on the 'TaskOverview' page, the next 4 columns now follow, which basically represent the created tasks. A 'Gallery' from the toolbox is used for each column. The advantage of this is that for each column only those tasks are displayed that match the column. An XPath constraint such as '[Status='To_Do']' is set to ensure this. The Gallery widget in Mendix is used to display several data records in a repeated layout - similar to a dynamic list.



10. The 'TaskCard' must still be created, which is used for the display on the main page. In Mendix, the layout for a single data record (e.g. a single TaskCard) is often separated from the display of multiple data records (e.g. via a Gallery).



Use Case 3 - Java/Angular/MySQL

For the comparison, an existing project from GitHub is used, which was modified by the author to be comparable with the Mendix application. Small changes were made to the backend and frontend. For example, it is now only necessary to create a Kanban board, and the tasks have more attributes than before. A progress bar has also been added.

Original Java/Angular Kanban-Board: https://github.com/wkrzywiec/kanban-board

Forked Project: https://github.com/stevo9061/mse-projects/tree/steve/Use-Case-3

Backend: Developed with Java Spring Boot (Jdk 11) for RESTful APIs.

Frontend: Implemented with Angular v7 for a reactive user interface.

Database: Postgres v9.6 for relational data and liquibase

Containerization: Docker v4.35.1

This project is based on a containerized infrastructure with Docker Compose, which orchestrates the frontend, backend and database in separate containers. Additional Dockerfiles exist for the frontend and backend to enable specific configurations for the build and runtime environment. The aim of this implementation is to define an equivalent application to Mendix. Since Angular version 7 works for this application purpose without any problems, no update to a newer version was considered here.

Dependencies

The following dependencies are used in the **Spring Boot** project, which are defined via **Maven** in pom.xml:

- Spring Boot Starter Web
  Purpose: Provision of RESTful APIs.
- Spring Boot Starter Data JPA
  Purpose: Communication with the database via JPA.
- Postgresql
  Purpose: Connection to a PostgreSQL database.
- Lombok
  Purpose: Reduction of boilerplate code.
- Spring Boot Maven Plugin
  Purpose: Support for the build and deployment of Spring Boot applications.
- Liquibase-core
  Purpose: Datenbankmigrationen
- Springfox-swagger
  Purpose: Documentation & UI für REST APIs

In addition, a short list of dependencies for Angular:

- @angular/core, @angular/router
- Purpose: Basic functionality and routing
- @angular/material
- Purpose: UI component library
- Rxjs
  Purpose: A library for reactive programming with observables.
- zone.js
  Purpose: Change Detection

Further development dependencies (tests, linting, CLI) can be viewed in the package.json file under devDependencies.

Implementation: A project was found on GitHub that is very similar to the requirements of Mendix. This was used and a few small details were added/adapted to meet the requirements. First, the initial development steps are recreated in order to record the work steps. The changes made are also discussed.

1. Set up the projects in Angular and Spring Boot with the respective dependencies and docker files. A Docker-Compose file must also be created, which provides the respective infrastructure for each container.

   Let's start with the backend first, the directory structure is as follows:
   /main/java/app/web/kanban-app

   ```
   ├── config          # configures Swagger 2 to automatically generate API documentation
   ├── controller      # Contains REST controllers
   ├── model           # Data models (e.g., Kanban, KanbanDTO)
   ```

├── repository # Database repository interfaces

├── service # Business logic (e.g., KanbanServiceImpl)

├── resource # The files shown define the database structure of the Kanban application in the form of versioned SQL scripts and are controlled centrally by Liquibase via the db.changelog-master.xml file.

Create the entities for the database

Task.java

- Fields: id, title, description, color, status, dueDate, prioritystatus, uploadedFileNames
- This entity represents a single task within a Kanban board, with the attributes just mentioned. Status and Prioritystatus are defined as enums, the files are saved as an element collection.

TaskDTO.java

- Fields: title, description, color, status, dueDate, prioritystatus, uploadedFileNames
- This DTO encapsulates all relevant data of a task for display or processing, including metadata such as due date, priority and file attachments.

Kanban.java

- Fields: id, title, tasks
- This entity represents a Kanban board that contains a title and a list of assigned tasks (1:n relationship). Tasks are loaded and added dynamically via addTask().

KanbanDTO.java

- Fields: title
- Data Transfer Object (DTO) to represent a new Kanban board when it is created via the API, reduced to the title field.

PriorityStatus.java

- Fields: LOW, MEDIUM, HIGH
- Enumeration for defining the task priority with three possible levels.

TaskStatus.java

- Fields: TODO, INPROGRESS, TOREVIEW, DONE
- Enumeration to represent the status of a task within the Kanban process.

2. The repository interfaces are based on CrudRepository from Spring Data JPA and enable access to the database via generic CRUD methods as well as individually defined queries (e.g. findByTitle).

KanbanRepository.java

- This repository provides CRUD operations for the Kanban entity and also makes it possible to find a Kanban board by title using the findByTitle(String title) method.

TaskRepository.java

- This repository offers CRUD functionality for task objects and provides two additional methods: findByTitle(String title) for searching for the title and findById(Long id) for the specific query by ID (the latter is optional, as findById is already included in the CrudRepository).

3. The basic functions for the backend are now complete. The service classes now follow. The TaskService interface defines the central methods for managing tasks. The TaskServiceImpl class implements the TaskService interface and contains the business logic for task management. The KanbanService interface defines the central methods for managing Kanban boards and assigning tasks.

TaskService.java & TaskServiceImpl.java

- getAllTasks()
  Retrieves all tasks from the repository and returns them as a list.
- getTaskById(id)
- Uses taskRepository.findById() to query a task by its ID.
- getTaskByTitle(title)
  Uses taskRepository.findByTitle() to search for the title.
- saveNewTask(taskDTO)
  Converts the DTO into a task object and saves it via the repository.
- updateTask(oldTask, newTaskDTO)
  Performs an update of the existing task based on the transferred DTO values (only if values are set).
- deleteTask(task)
  Deletes the transferred task from the database.

Help methods

- convertDTOToTask(TaskDTO)
  Creates a new task object from a DTO (e.g. for saving new tasks).
- updateTaskFromDTO(Task, TaskDTO)
  Only overwrites set fields from the DTO in the existing task.

KanbanService.java & KanbanServiceImpl.java

- getAllKanbanBoards()
  Retrieves all Kanban boards via the repository and returns them as a list.
- getKanbanById(id)
  Fetches a specific Kanban board based on the ID (optional).
- saveNewKanban(kanbanDTO)
  Converts the DTO into a Kanban object and saves it.
- deleteKanban(kanban)
  Removes the transferred Kanban board from the database.
- addNewTaskToKanban(kanbanId, taskDTO)
  Finds a Kanban board via the ID, converts the TaskDTO into a task object, adds it and saves the board again.

Help methods:

- convertDTOToKanban(KanbanDTO)
  Creates a new Kanban object from a DTO.
- convertDTOToTask(TaskDTO)
  Converts a TaskDTO into a Task object (e.g. for transfer to addTask()).

FileStorageService.java & FileStorageServiceImpl.java

- init()
  Creates the directory folder uploads at startup if it does not exist (@PostConstruct).
- storeFile(file, taskId)
  Saves the file under the name taskId_filename, adds it to the associated task entity and saves the task again.
- loadFileAsResource(fileName)
  Loads the file from the uploads folder and returns it as a Spring Resource.
- deleteFile(fileName)

  Physically deletes the file from the file system, if present

4. Now the controller classes are still missing. The following endpoints are created:

   KanbanController

   - /kanbans/: Returns a list of all Kanban boards.
   - /kanbans/{id}: Returns a Kanban board based on the ID.
   - /kanbans?title=: Returns a Kanban board based on the title.
   - /kanbans/ (POST): Creates a new Kanban board.
   - /kanbans/{id} (PUT): Updates an existing Kanban board with the specified ID.
   - /kanbans/{id} (DELETE): Deletes the Kanban board with the specified ID.
   - /kanbans/{kanbanId}/tasks/: Returns all tasks of a Kanban board.
   - /kanbans/{kanbanId}/tasks/ (POST): Creates a new task and assigns it to a Kanban board.

TaskController

- /tasks/: Returns a list of all tasks.
- /tasks/{id}: Returns a task based on the ID.
- /tasks?title=: Returns a task based on the title.
- /tasks/ (POST): Creates a new task.
- /tasks/{id} (PUT): Updates an existing task with the specified ID.
- /tasks/{id} (DELETE): Deletes the task with the specified ID.

5. Basically, the implementation is almost complete, only Liquibase still needs to be set up. To do this, the following line is added to application.properties: 'spring.liquibase.change-log=classpath:/db/changelog/db.changelog-master.xml.' This XML file defines exactly how the database structure should be set up. Four SQL files and a further XML file are created. Theoretically, it is also sufficient to create just one SQL file and the corresponding XML, but Liquibase enables clean versioning of the database.
   This means that as soon as changes to the database are required in the future, a new SQL script is simply created and referenced in the changelog XML. An example: The file '04_add_due_date_and_priority.sql' was added to supplement the due_date and prioritystatus columns. This file must then be included in the db.changelog-master.xml file. Liquibase automatically recognizes this change during the next run and executes it.
   The files mentioned can be viewed in the repository [6] if you are interested:

   - db.changelog-master.xml
   - 01_init_kanban.sql
   - 02_init_task.sql
   - 03_kanban_column.sql
   - 04_add_due_date_and_priority.sql

6. As the front end has already been set up in the first step, we can now continue with the actual implementation. The entry point of the application is the file home.component.ts. If a Kanban board does not yet exist, a new one can be created at this point. To do this, the kanban-dialog.component.html including the associated TypeScript file is called. The home component has been adapted so that only a single Kanban board can be created. Originally, it was possible to create several boards and choose between them. The home.component.html file has also been adapted accordingly. A Kanban class is defined in kanban.ts, which contains the attributes id of type number, title of type string and tasks as an array of task objects. The task class in task.ts has the fields id (number), title (string), description (string), color (string), status (string), dueDate (string), prioritystatus with the possible values 'LOW', 'MEDIUM' or 'HIGH' as well as uploadedFileNames as an array of type string, which is initially defined as an empty array.

7. The file kanban.component.html and the associated TypeScript file have been adapted so that not only the title of the tasks is now displayed, but also the due date (dueDate) and the priority (Priority). An additional column with the designation “To Review” has also been added.

8. Additional fields have been added to the task-dialog.component.html file, while the original color field has been removed. Instead, the attributes Due Date, Priority and Status have been added, from which the user can now select. The due date is selected using a date picker, while priority and status are mapped using enumerations. Tasks are now colored dynamically based on priority, with the color automatically adjusting depending on the priority status set.

   The application is now fully prepared and can be started via the Docker Compose file. The entire environment can be started with the docker-compose up command in the root directory of the project, where the docker-compose.yml file is also located.

Bibliography

[1] Gartner forecast. [Online] https://ninox.com/de/blog/gartner-prognose-die-nutzung-von-low-code-technologien-boomt-weiter [Accessed: 18.01.2025]

[2] [ONLINE] - https://docs.mendix.com/appstore/modules/forgot-password/ [Accessed: 17.11.2024]

[3] [ONLINE] - https://www.mendix.com/pricing/ [Accessed: 08.12.2024]

[4] [ONLINE] - https://docs.mendix.com/quickstarts/responsive-web-app/ [Accessed: 17.11.2024]

[5] [ONLINE] https://marketplace.mendix.com/link/component/108938

[6] [ONLINE] https://github.com/stevo9061/mse-projects