CLERP – Clever Enterprise Resource Planning

# Terms

CRUD: Create Read Update Delete  
App: Application  
EAN: European Article Number, also known as International Article Number  
EF: Entity Framework  
DTO: Data Transfer Object  
DB: Databases

# About the product

CLERP – Clever Enterprise Resource Planning is an enterprise resource planning solution for small to large businesses. It’s aimed at firms which assemble their products in-house, based on parts provided by third parties. The application can administrate the departments and employees as well the specific roles from the departments and employees from the customer firm. Additionally, this software can manage the product types / categories from the firm, for example graphics cards CPUs etc. It can register individual products via a QR-Code scanner built into the app. On registration of a new product the storage location must be selected by the scanning employee and will afterwards saved in the system.

# General Information

To start the frontend from vs code, use the command ‘npm run start’.

# Functional Requirements

* CRUD for a product
* CRUD for product type
* Each product and each product type can have multiple parent- and/or child-products
* All products can be presented with their tree of sub/parent products
* CRUD for business partners
* CRUD address
* Each business partner can have multiple addresses
* CRUD for order
* Incoming deliveries and their products can be recorded via barcodes
  + product type (EAN)
  + serial number (SN)
* All barcodes can be recorded manually, if necessary
* CRUD warehouse
* CRUD shelf
* CRUD compartment
* Each product has a physical location in a warehouse (warehouse -> shelf -> compartment)
* On incoming order check availability of required products
  + If products available: option to reserve them
  + Else: option to automatically order the required amount

# Non-Functional Requirements

## General

* CLERP will be a web application, as a single page application

## Frontend

* The frontend design is primarily optimized for desktop and tablet devices. The architectural design should contain the possibility for future mobile implementations.
* The frontend will be developed using the angular framework in the version 7.
* The angular 7 framework is using:
  + HTML 5
  + TypeScript, which will be converted to JS ES5
  + SASS, which will be converted to CSS 3
* The frontend will use additionally:
  + Bootstrap V4 (customized)
  + Font Awesome

## Backend

* The backend will be an ASP.NET Core Version 2.2 Application implemented as a RESTful API
* The backend will have C# as main language
* The EF Core will be used for data access (ORM)
* The database will be generated and designed using the Code-First Principle (Exceptions: Views and other unsupported elements in EF Core)
* The database will run on a MSSQL (-Express) server
* The database will contain specific tables which are indexed

# Security

## Overall API Security

In the backend we’re using an ASP.NET Core REST API. The asp.net core framework provides by default already useful helpers and automated procedures to secure the app, such as automated token validation, require https request etc. Our app is configured in a way, that only HTTPS request are accepted instead of unsecure HTTP requests. We did this to provide a decent base protection against man in the middle attacks (the content of HTTPS request is encoded and not submitted plain-text as with HTTP). We realised this HTTPS enforcement with the HSTS options and HTTPSRedirect features provided by the asp.net core.[[1]](#footnote-1)

We didn’t use the frameworks built in CSRF protection features, because we don’t need it in our use case. A CSRF attack aims to leverage cookies, especially cookies containing a Session-Id. Because we have a REST API in the backend, which is stateless, we don’t have any sessions. Neither do we use any cookies in the front end. If there are no cookies or session there’s also no room for potential CSRF attacks (discussion about this topic[[2]](#footnote-2)).

Due the API for this should be private (only used by the front end) we disabled CORS in the production environment. Its only enabled in the development environment. This way we can ensure that only requests from our front-end can reach the API and not any other potentially dangerous requests from external websites.

The API is configured in a way, that you must be authorized and possess the required role to successfully perform a request. The exception to that rule is, when a controller or an action is decorated with the “AllowAnoymous”-attribute, for example the employee login action.

## Authentication

For the authentication (auth) we chose a token-based approach. For this our app uses JWT (JSON Web Token) tokens. These tokens are a well-tested and widely used standard today. The benefits are, that the information stored in a token (claims) is read-only because every token is signed with a verification hash. If someone would change the content of the token, the verification wouldn’t match anymore, and the token would be invalid. In our tokens we store: the employee-guid, the employees username and all the roles this employee has. Additionally, every token has a unique id (JTI-Claim), that the token can be identified with. This will enable us to store the tokens in a lookup-table inside the database, but this feature may not be implemented at launch. Every token generated by our app has a lifetime of two days until it expires. Our configuration forces the token to be a signed token (e.g. “read-only-token”) and requires the token to be transmitted via HTTPS.

If you want to generate a JWT-token, you must provide a private key (cryptographical secret) which will be used to generate tokens and validate existing tokens against. In our app we use a RSASecurityKey which uses a random 2048-bit strong byte key generated with the RSACryptoServiceProvider (both key and service provider are implementations from the asp.net core). This key will be newly generated on every start-up of the app and will be registered as a singleton throughout the app. This is beneficial because the key exists only inside the memory and is not saved persistently (in a config file for example, or, even worse, as plain text in the code). This ensures that it cannot be stolen via decompiling our code or by stealing the config file. For the hashing of the sign- in-credentials we use the RSASecretKey and the RSASha256 algorithm.

## Password Hashing

In our app all passwords are saved hashed and salted in the database. We achieve this using a hasher from ASP.NET Identity. This implementation uses a Key Derived Function (KDF – in our case the Rfc2898DerivedBytes from .Net Core) to generate a hash and includes the salt with it. The KDF is configured to generate a 10-byte salt value and go over 1’000 iterations. For further information about this implementation please see <https://stackoverflow.com/a/20622428>.

## SQL-Injections

For data-access our app uses EF Core. To prevent any kind of SQL-Injections the EF core framework automatically escapes all data included in generated SQL-queries which are going to the database.

## Directory Traversal

Per default, directory traversal isn’t possible with ASP.NET Core, because there isn’t any folder structure which is exposed. But for general resources which are accessible per URL (Images, or other content, which has the id of the element in the URL), we chose to have GUID’s instead of normal Id’s as primary keys for our entities. That way a directory traversal is made considerably more difficult, since a GUID can’t be guessed as easily as an Id. This is in addition to the auth/role security we already have in place.

## Token Theft

The access-token (JWT) which is generated and returned after an employee has himself logged in, will be stored in the front end. In this location its relatively vulnerable against JS attacks (from browser extensions for example). These extensions could steal the access-token from the employee and get themselves access to the application. To prevent this issue, we implemented an IP-lookup feature. On token generation the current IP-address from the login request will be saved in the token as a separate claim. With a global authorization policy, we ensured that on every request that requires authorization, the IP from the token and the current IP will be compared. If they don’t match the request will be automatically rejected and a response with the code 403 will be returned.

# Architecture

## Bildergebnis fÃ¼r vertical slice architectureBackend

We started off the project using the common approach using the onion architecture. This architecture splits up the application in different layers like data access, business, persistence etc. But in the first two weeks we learned about this fairly new approach, the vertical slice architecture, which is gaining a fair amount of popularity in the industry now. We then carefully compared these two approaches with each other. We did our research about the topic and came to the following pros and cons for the vertical slice architecture compared with the onion architecture:

Illustration - scheme vertical slice architecture

* Pros:
  + Learning effect: this approach was completely new to us and we wanted to get more familiar with it, so that we could use our gained knowledge in future projects
  + Separation of concerns: since this architecture is feature based, every feature does exactly what it’s supposed to do.
  + Due the fact every slice is encapsulated from every other slice, each slice can take the optimal approach for solving his request.
  + Abstraction: because of the fact every slice is very specific, there’s no need for developing fancy abstractions like repositories or factories etc.
  + With the encapsulation of the slices comes also a higher maintainability due the fact everything is grouped up in features.
* Cons:
  + With the strict encapsulation within the slices, comes a lot of “duplicate code”. Because the slices shouldn’t use each other there’s little of reusable code.
  + Data collecting from multiple slices can be far more complex in the presentation layer.
  + The business logic in the slices can easily get very messy due the lack of defined abstractions. Because of this refactoring is unavoidable.

After careful evaluation of these pros and cons, we decided to go with the vertical slice approach. From our point of view this is far more interesting, and we could learn a great deal by implementing this architecture. This decision required a complete rewrite of the backend which required about one day of work.

We limited the implementation of the vertical slice architecture to the backend. In our implementation every feature has a specific request class/type, a handler which handles the request and a specific response class/type. To facilitate the implementation of the pattern, we depend on the MediatR-Framework. This framework is responsible for the complete mapping from the request to the corresponding handler and for automatically registers all handlers in the IoC (Inversion of Control) container from .NET Core. In our implementation every request and response type are automatically the request or the response DTO, used to communicate with the frontend via the controller. These are all confined within the namespace from the slice.

## Ein Bild, das Screenshot enthält. Automatisch generierte BeschreibungFolder Structure

Illustration - folder structure backend

In the root level of our backend we have four main folders:

* Domain
* Features
* Infrastructure
* Migrations

In the domain folder are all our domain / business models stored. These are standard POCOs (Plain Old CLR Objects) which represent all our business classes as well as the database structure.

Features folder contains the core of our application. In it there all features stored version based. Each version has its own namespace and its own definitions for validation. Inside the version folder there are areas for every domain model. Inside these areas there are the controller, default response DTO and a folder for every feature. Every feature has its own namespace and mustn’t use any other slice / feature. Inside the feature there’s at least a request class and a handler class. Additionally, there can be a response class and a validator class.

The infrastructure folder is filled with utilities used for the core functionalities of our app and for customizing the asp.net core. Additionally, there’s also our configuration classes for each domain model for the design of the DB.

The migration folder contains all migrations done with the database and the current snapshot of the state.

# Database

## ERD

![Ein Bild, das Text enthält.

Automatisch generierte Beschreibung]()

Illustration 3 – entity relation diagram clerp db

## Approach

For creating the database, we used the code first approach. We have chosen this approach due the fact, we expected it that we’d be far faster simply writing the classes than completely setup a DB. For configuring the single tables and their relations we created for each domain model a configuration class which implements IEntityTypeConfiguration. Inside this classes we used the FluentApi to configure the tables and associations. Also, inside these classes is the written seed data. Theses configuration files can be found in the infrastructure folder.

# Retrospective

We started off the project with a huge plan about what the final product should be able to do. The problem was, that we massively underestimated the effort of building even the backbone of the project. Even small things took a factor n more time as initially expected. This meant, that the final product is a lot smaller in scope than initially planned.

A big positive aspect with this project was the teamwork. All members were very reliable and highly motivated. Sometimes it was even kind of a problem, since we were overambitious and had to curb our enthusiasm. While implementing certain features we tended to focus too much on the small details and forgot to see the big picture.

In summary we found the project very interesting. We could gain a huge amount of knowledge, especially due the fact that we had to work with technologies, frameworks and even architectures, that were completely new to us. One critical factor was time, time management indeed. We had a little bit of stress in the final phase of the project, because our time management wasn’t a hundred percent mature. Nevertheless, a proper time management is something very hard to achieve, especially in the informatics industry.

# Source dictionary

Illustration 1: [https://cdn-images-1.medium.com/max/1600/1\*JHlN7ixDbPFgyjZh1GQIFQ.png](https://cdn-images-1.medium.com/max/1600/1*JHlN7ixDbPFgyjZh1GQIFQ.png)  
Illustration 2 & 3 : Screenshots of the product

# Illustration dictionary

[Illustration 1 - scheme vertical slice architecture 4](file:///C:\VisualStudio\Projects\CLERP\Documentation\CLERP-Doc.docx#_Toc11592389)

[Illustration 2 - folder structure backend 5](file:///C:\VisualStudio\Projects\CLERP\Documentation\CLERP-Doc.docx#_Toc11592390)

[Illustration 3 – entity relation diagram clerp db 6](#_Toc11592391)

1. [https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-2.1&tabs=visual-studio](https://docs.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-2.1&tabs=visual-studio#http-strict-transport-security-protocol-hsts) [↑](#footnote-ref-1)
2. <https://security.stackexchange.com/questions/166724/should-i-use-csrf-protection-on-rest-api-endpoints> [↑](#footnote-ref-2)