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

# 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. Thes 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 in terms of 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. This is additional 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. While 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 which must be authorized, 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.

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)