Woody: A Software System for the Design and Production of Doors and Windows

Orsolya Máthé
Babeş–Bolyai University
Cluj–Napoca, Romania
mathe_orsolya@yahoo.com

Botond Miklós
Babeş–Bolyai University
Cluj–Napoca, Romania
myklosbotond@gmail.com

István Bege
Codespring
Cluj–Napoca, Romania
bege.istvan@codespring.ro

Attila Farkas
Codespring
Cluj–Napoca, Romania
farkas.attila@codespring.ro

Csaba Sulyok
Babeş–Bolyai University
Cluj–Napoca, Romania
csaba.sulyok@gmail.com

***Abstract*—The Woody project aims to make the daily production process easier and more efficient. It provides a platform for keeping track of all orders and makes it possible to generate workshop drawings, component lists and other required documents for each order. The workshop drawing contains the proportional blueprints for each product, while the list of all components needed for making them is also generated for the entire project. The software can also suggest an optimal strategy for cutting out these components, thus reducing the waste of resources. The definition of the products is done using a dedicated domain-specific language (DSL).**

The responsive web interface makes it possible for the application to be used in the field at the time of recording an order. In this way, pictures of the installation site can be uploaded using a mobile device in order to aid the manufacturing process.

I. Introduction

Due to the countless variations a client could ask for when ordering a door or window, parts of the design process are still done on paper today. It is hard to formulate a set of rules applicable to every situation. Despite its flexibility, designing on paper has significant drawbacks. It is time-consuming, and sometimes produces erroneous results, which lead to unexpected costs, additional work, and a delayed delivery date. Moreover, the formulas used in the design stage may change due to the client ordering a previously non-existent type of product, the introduction of new machines into the manufacturing process, or other unforeseen factors. On top of this, the carpenter faces a complex problem when trying to minimize the waste of resources in stock.

The purpose of the Woody software system is to make the above mentioned process easier and more efficient. Through a web platform, the orders can be managed, along with the products, clients and other relevant information. After the appropriate parametrization of the products in an order, it is possible to generate the shop drawing containing the proportional blueprints of each product, as well as the list of all components needed to manufacture them. The application also provides a one-dimensional lengthwise optimal strategy for cutting out these components, taking into account the sizes of the resources in stock.

This software system is a successor to a similar desktop application named FaTherm, written by one of the authors of the current paper in 2004. Although still in use today[1], it uses outdated and no longer supported technologies, making software maintenance too cumbersome. Woody replaces and extends this software in a way that would enable other workshops to use it as well. Shifting from the desktop to the web makes it possible for the application to be used simultaneously from multiple locations. The revised data model facilitates the addition of new product types without the intervention of a developer, while the optimal cutting strategy aims to reduce the expenses of the company.

[1]Workshop accessible at `https://www.farkas.ro/`

II. Functionalities

The application defines a single user role, therefore every functionality is available for anyone working in the workshop.

Every user can enter new clients and projects into the system. To create a client, only a few bits of personal information are needed. When creating a project, a client and a deadline must be specified; images and windows can be added optionally. The saved clients and projects may be listed.

The most important functionalities of the Woody project lie in the interpretation of the data representing a project. The shop drawing, the component list, an optimal cutting strategy and the list of the attached images are the views that a user can request regarding every project. The shop drawing contains the dynamically generated and proportionally scaled figures of the windows, while the component list enumerates all the parts from which the windows of a given project can be built. These two views are printable; the last page of the printed version shows an overview of the project to promote transparency during the production phase.

The optimal cutting shows how the wood parts belonging to a given project can be cut out from the boards in the workshop with minimal loss. The cutting strategy is visualized by dynamically generated vector graphics.

The images of the project location are meant to help to prepare the installation process.

The user can modify the data of already existing clients and orders, so that they always remain realistic. During the maintenance of a project, images can be deleted, added, and windows can be updated, removed or created. When creating a window, values have to be assigned to the variables defined in its type. The formulas that describe the window can be reached only through the window type.

Furthermore, a price offer/weight estimation can be requested for every project, which contains the individual

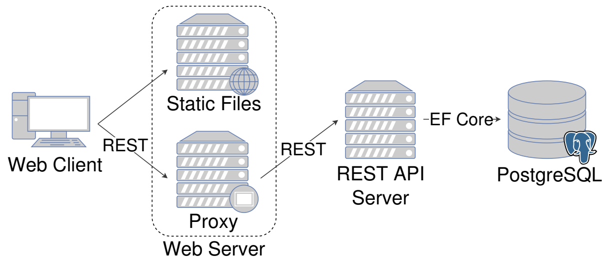


Fig. 1: A depiction of the communication between the client, the web server, the application server and the database.

price/weight of each window from the project, as well as their sum.

III. The Structure of the Woody Application

The Woody application has two main parts: the web interface and the application server (see Figure 1). The application server provides RESTful Application Programming Interface (API) endpoints for accessing and changing data. First introduced by Roy Fielding in his 2000 PhD. dissertation [1], REST (REpresentational State Transfer) is an architectural style for web-based projects. A RESTful service has the following main characteristics: it provides a uniform interface for accessing data, it is stateless, responses are cacheable, the communication follows the client-server model, and the system can be layered and restructured without the necessity for client-side intervention.

Due to the structure of the software system, requests from the outside can only reach the web server responsible for serving the static files needed by the web interface. This server also acts as proxy, forwarding any API requests to an application server located in its local network; API request paths are differentiated by the `/api/` prefix. The application server handles the request, querying or modifying the data in the PostgreSQL relational database, then sends a response containing the result or an error, both in JSON (JavaScript Object Notation) format.

IV. The Application Server

A. Architecture

The server is modularized as seen in Figure 2.

The *Persistence Model*, *Domain Model* and *DTO (Data Transfer Object)* components represent three variants of the domain information (see Sections IV-C and IV-D).

The task of the *Controllers* component is to accept and answer client requests. The incoming DTOs are converted to domain models, conversely outgoing data is formed into DTOs with the *Domain–DTO Mappers*.

The *Service layer* contains the business logic. It accepts requests transmitted by *Controllers*, and creates the correct answers for them by communicating with the data access layer.

The *Repository* facilitates CRUD (create, read, update, delete) operations, communicating with the database. The received domain model from the *Service layer* is converted into

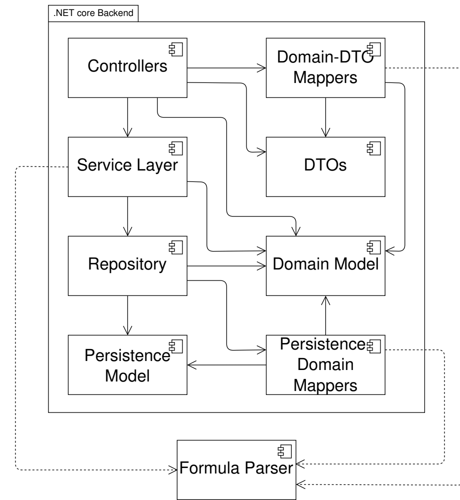


Fig. 2: The components of the server.

the persistence model variant with the *Persistence–Domain Mapper*.

The *Formula Parser* module creates the possibility of storing the formulas in the data structures, and evaluating them when needed.

B. Formula Parser

The purpose of the *Formula Parser* is to interpret, evaluate the rules and to generate the dynamic drawings. The rules can be simple formulas, which describe some quality of the components; e.g. the `(width + height) / 2` formula defines a size. These formulas may be stored in the database and can be evaluated for different variable inputs at any time; e.g. if the `width=3` and `height=5`, then the *Parser* can determine the final result, in this case `4`.

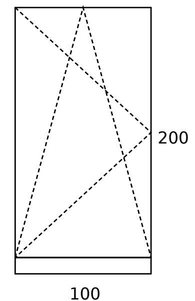
In addition, visual element commands may also be used for drawing; e.g the `Line[0, 0, 100, 100]` draws a line starting at the `(0, 0)` coordinate, and ending at `(100, 100)`.

Since dynamically generated drawings are necessary for the project, the two mentioned functionalities are combined into a domain-specific language (DSL) [2] called *Nyílászáró leíró nyelv (NyZLNy)*[2]. This language defines commands to simplify describing/generating window drawings. For example,

[2]"Language describing doors/windows" in Hungarian

```
SVG[
    width + 30, height + 20,
    WindowPane[opening, 0, 0,
        width, height * 0.92],
    Fill["none",
        Rect[0, height * 0.92,
            width, height * 0.06]
    ],
    Text[width + 5, height / 2, height],
    Text[width * 0.4, height + 15, width]
]
```

(a) Snippet from the *NyZLNy* language



(b) The resulting drawing

Fig. 3: Example of how the DSL works.

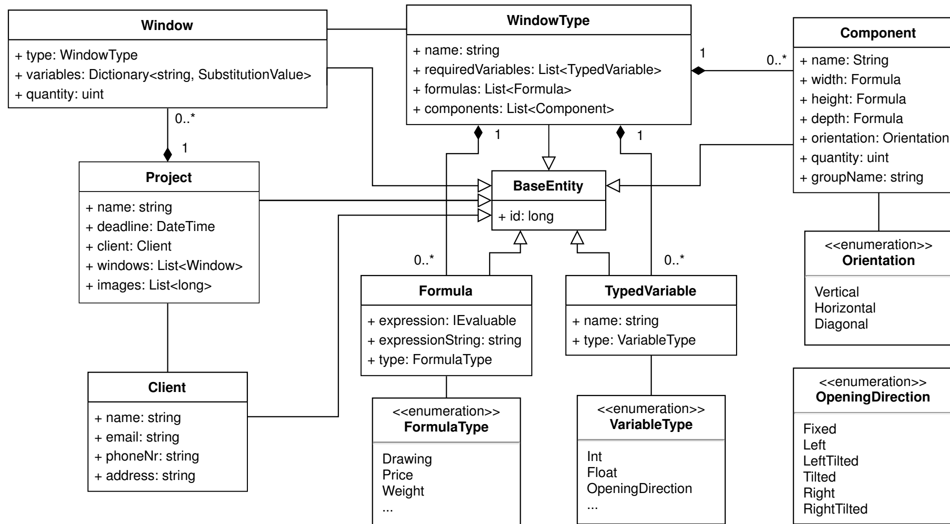


Fig. 4: The structure of the Domain Model

the `WindowPane` command (see snippet in Figure 3a) uses the following parameters to describe a window: its opening direction, the coordinates of its upper left corner, and its size. The command generates the rectangle with the dashed lines inside it (indicating the opening directions) shown in Figure 3b.

C. Domain Model

Of the three different server-side data model variants, the *Domain Model* is built around aiding business logic implementation. The *DTOs* represents the models that are used by the *Controllers* during the communication with the client.

The structure of the *Domain Model* is shown in Figure 4. The `BaseEntity` is the base class of each entity describing class, it has only one `long` typed `id` field, which is needed for identifying the objects.

The `Client` class contains the data of a client: name, e-mail, phone number and address. The `Project` incorporates information about orders, such as a `Client` typed client. While the list that represents the windows is made of `Window` typed objects, the list of the images contains only their identifier. Since the business logic does not use the binary data of the images, it would be a waste of resources to store it in the project entity. Contrary to this, various operations are made on the windows, and hence the whole object list is well-founded.

For expandability reasons, the formulas are not hard-coded. The window types visible to the client are represented in the `WindowType` entity on the server side, which contains a name and three lists for formulas, required variables and components.

The `Formula` class makes it possible for the `WindowType` to describe any window type. Its `type` attribute differentiates between formula types such as drawing, price, weight or size. The formula itself is stored both as plain text (`expressionString`), and in an evaluable form (`expression`). The conversion form text to the `IEvaluable` type is made by the *Formula Parser*.

The `Component` entity defines the components from which a window is built, e.g. the wood pieces of the frame. The `width`, `height`, `depth` fields are formulas, this way every piece of the window has the same adaptability. Besides these, every component has a name, an orientation, quantity, and a group name; grouping the wood elements helps in the production.

The `Window` model is for storing the window instances. An instance has a type (`WindowType`), quantity and variable values (the substitution values for the variables that are specified in the type). The `variables` field is a dictionary mapping the required variable names (from the window type) to substitution values; e.g. `(width, 3)`. These variable values are substituted into the formulas of the window type with the help of the *Formula Parser*. This is how the components of the right size, the dynamic figures and the right price estimation are created.

D. Persistence Model

The communication between the server and the database is realized with the help of an ORM framework (Object Relational Mapper), which makes it possible to handle data through objects. The objects are defined in the *Persistence Model*, which differs only in few aspects from the *Domain Model*.

The repository layer contains an `Images` entity, which stores the binary data that represent the images, and has a `projectId` foreign key field pointing to a project. In the *Persistence Model*, the `Project` contains not only the `ids` that point to the images, but also the whole image objects.

In the `Window` entity, the variables are stored in a `Variable` typed list. This new type has a `string` field for storing the value of the variable, and two foreign keys: one is pointing to the window instance, the other to a `TypedVariable`. This `TypedVariable` is mapped to a `Dictionary` in the domain model.

These differences are needed, because the project uses a *code first database* [3]., i.e. Entity Framework Core generates

database tables from the exact provided models, if they do not already exist. The *Domain Model* is added to the project so that these conventions would not affect the business logic.

E. Optimal Cutting

The wood elements from the frame are cut from boards with fixed height and depth, but varying width. During the cutting process the question of an optimal cutting strategy emerges. This is only a one-dimensional question, yet the answer is an NP-hard problem [4].

In Woody, a randomized, heuristic algorithm is used for searching for the optimal cutting of the boards for a project. The algorithm is not guaranteed to find the single most optimal strategy, but it significantly reduces the material losses of the workshop.

Algorithm 1 Pseudocode of the algorithm used in finding the optimal cutting strategy.

$W \leftarrow$ Wood pieces that need to be cut out
$B \leftarrow$ Board lengths from the storage
$elems \leftarrow []$
$waste \leftarrow 0$
$randomSort(W)$
while W is not empty **do**
  $minList \leftarrow W$
  $minElem, minWaste \leftarrow cutBestGreedy(B_0, minList)$
  for all $boardLength$ in B starting from the second item **do**
    $currentList \leftarrow W$
    $currentElem, currentWaste \leftarrow$
      $cutBestGreedy(boardLength, currentList)$
    if $currentWaste < minWaste$ **then**
      $minWaste \leftarrow currentWaste$
      $minElem \leftarrow currentElem$
      $minList \leftarrow currentList$
    end if
  end for
  $W \leftarrow minList$
  add $minElem$ to $elems$
  $waste = waste + minWaste$
end while

The wood elements from the order are grouped by their height and depth, the algorithm runs separately on these groups. First, the elements are randomized and the loss is calculated for this order. The number of possible elements cut is calculated for each board length in stock (the priority is based on the random order), and the loss is measured, retaining the better case. This is repeated until there is no wood element left uncut. These steps are repeated multiple times for different random orders, and strategy yielding the minimal loss is kept. The amount of repetitions is constant, set in this case to 1000. To illustrate the result, the DSL generates an SVG format image.

V. The Web Client

A. Architecture

The web client is composed of four main modules: *Types*, *API*, *Components* and *Stores*.

The *Types* module contains the client-side representation of the data entities.

The *API* module is responsible for communicating with the server. The requests are sent asynchronously and the responses are handled using callback functions, ensuring dynamic data queries. This module also handles the conversion between entities and DTOs.

The *Stores* handle the client state, i.e. each entity has a corresponding store which contains data and related operations. It uses the *API* module for querying or modifying data.

The *Components* module houses all the web components used by the user interface. It calls upon the *Stores* component for handling user input and re-renders the contents of a component when the data being represented changes in the stores.

B. Single-Page Applications and Routing

The advantage of Single-Page Applications (SPA) is their speed due to the fact that after the initial page load they only update the parts which have changed, and do not require reloads of the entire page [5]. The required HTML, CSS and JavaScript source files are loaded on first access, after which the only communication is the transfer of data between the client and the server.

However, SPAs lose the advantage of user progress tracking through URLs, therefore client-side routing is used to dynamically change the URL based on the currently used view. This allows the browsers to save the different states in their history, making stepping back or searching for a particular page easier. Routing helps in choosing the visible components as well, thus the user can be greeted by the same view after a potential reload of the page.

C. Components

The user interface is composed of smaller units named components. The final view is constructed by nesting many components, which allows the reuse of general purpose components in different views. These components can be stateless or stateful; the latter type automatically re-renders its contents whenever data changes in its state.

The dynamic updating of components based on the data they represent makes building complex interfaces easier. The updating process can happen solely on the client side without the need for further requests after the initial data load. An example of this would be the addition of a new window to a project. In this case, the selected window type specifies what variables are needed for that window. Based on this list of required variables, an input form is generated with an input field for each variable. The type of each input field is determined based on the type of the variable it represents, which can result in a numeric input, a text field or a drop-down list.

VI. Technologies

The application server is written in C# using the ASP.NET Core framework [6], chosen for its efficiency and cross-platform capabilities. For enforcing the various constraints in the data model, the relational PostgreSQL database [7]

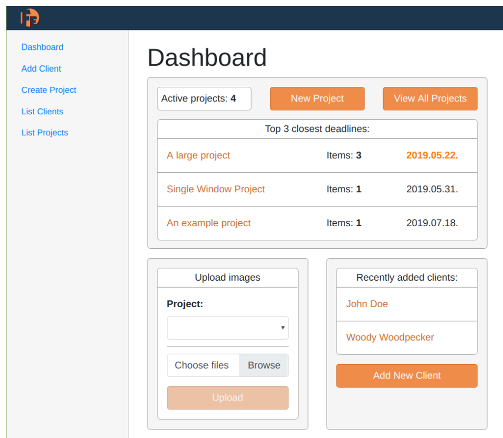


Fig. 5: The view greeting the user upon accessing the website

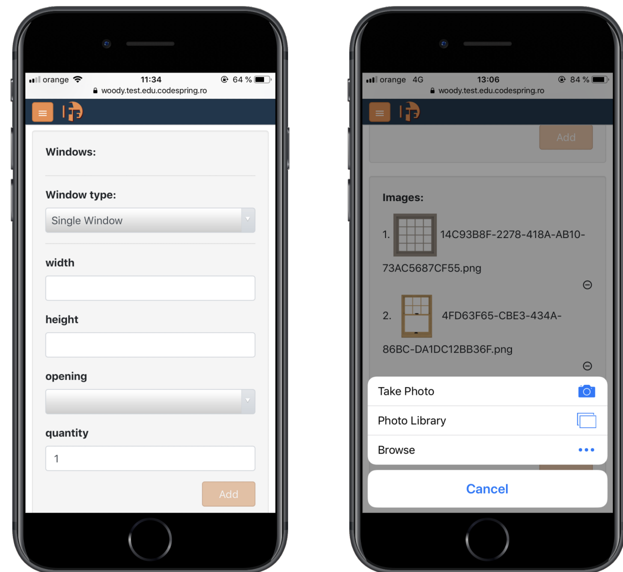
is responsible for persisting the data. The communication between the server and the database is handled by Entity Framework Core [8].

The web interface is built using the React library [9], which helps in building reusable components written in the XML-like JSX (JavaScript eXtension). The URL routing is achieved with the help of React Router. For type safety and an object-oriented approach TypeScript is used instead of JavaScript. MobX [10] is responsible for the centralized management of the application state, extracting the state from each component into stores centered around each entity. InversifyJS provides an inversion of control (IoC) container for achieving the dependency injection (DI) design pattern [11] on the front-end. The Bootstrap [12] CSS framework provides a grid system for making the web application responsive, while styled-components manages the styling of UI elements on a React component level.

Git [13] was chosen as a distributed version control system, with GitLab [14] providing a remote repository and project management interface. For deploying the application, Docker [15] is used for its efficient virtualization capabilities. Webpack serves as a static bundler for aggregating the source files and their dependencies. npm and nuGet are the dependency management tools used in the front-end and back-end, respectively. For quality assurance purposes, NUnit is used for testing, while TSLint and StyleCop are static code analysis tools for enforcing coding conventions.

VII. The Use of the Woody Application

When visiting the webpage, the user is greeted with the *Dashboard*, which provides easy access to the most used functionalities of the application (see Figure 5). The first card contains a link to the top 3 projects with the deadline closest to the current date. The two additional buttons redirect to the project creation and project list views. The next widget aims to make uploading photos to a project easier. After choosing the project from a drop-down list, the user can attach one or more pictures using the Browse button. Below these lies a third card listing links pointing to the most recently added clients, as well as a button, which navigates to the client addition form.



(a) Variable input fields generated for the chosen window type

(b) Taking and uploading photos to a project

Fig. 6: The webpage as viewed on a mobile device

The *Dashboard*, much like the rest of the application, has a responsive design which makes using the webpage just as comfortable on smaller devices as on larger ones (see Figure 6). On a mobile device, for example, the three cards are situated below each other.

The main navigation element of the webpage is the sidebar, located on the left side of the screen. On smaller screens, like mobile devices, the sidebar is hidden and can be accessed through the "hamburger" button in the top left corner. Menu items include the aforementioned dashboard, the client addition page, creation of a new project, listing clients and listing projects.

When adding a new client to the application, one must fill a form with relevant client details, with mandatory input fields marked with an asterisk. The *Add* button becomes enabled only when all entered data is valid: all the mandatory fields are filled out, and the phone number/e-mail address are not yet present in the system (in order to avoid client duplication). Success or failure of the asynchronous addition call is communicated to the client through appropriately stylized notification boxes in the lower right corner.

When listing the clients, the information regarding each is displayed in a tabular form, together with an *Edit* button, which enables data modification.

By choosing the *Create Project* menu item, the user is presented with a form built for this purpose. Each order must have a client that can be chosen from the list of registered clients using a drop-down list. A name and deadline are also compulsory for an order; the deadline can be chosen using a date picker input.

When creating a new project, one can also add windows or upload photos as well. After choosing a window type, the user is presented with a set of input fields representing the required variables related to that specific product type. Figure 6a shows

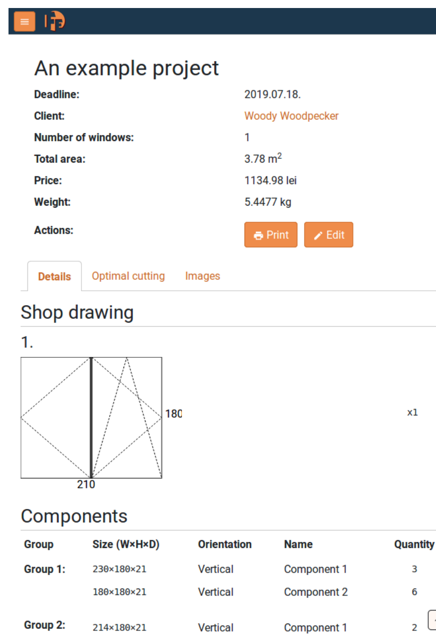


Fig. 7: *Details* - Detailed information of a project with the workshop drawing and list of components

the inputs belonging to the *Single Window* type. Only after filling out every field can the user click on the *Add* button, after which the window is added to the "basket" associated with the order. This basket lists all the windows added to the project, together with a deletion option.

By clicking on the *Browse* button, the images to be uploaded to the project can be selected. In the case of mobile devices, taking a photo also becomes an option, as seen in Figure 6b. Clicking the *Add* button adds the image to a basket similar to the one described in the case of windows.

The last menu item is the *List Projects*, containing some information about each project (name, client, deadline, number of products) and a *Details* button. Clicking this button accesses a detailed view that can be seen on Figure 7. At the top of the page, the information about the project is listed, followed by a tab bar listing different views.

The *Details* tab is the default view, listing the blueprints of each window in the project, followed by the grouped list of components. This can also be printed by clicking the *Print* button. The printed format contains an overview table with the information present in the header. The second tab is titled *Optimal Cutting* and shows the results of the cutting strategy. The various boards are grouped by depth and height, with the amount wasted being shown for each group, as well as the total waste on the bottom of the page. This view can be printed as well. By choosing the third tab, *Images*, the photos attached to the project can be viewed. Throughout each tab view, an *Edit* button can be seen, which leads to a form for modifying the current project.

VIII. Conclusion and Future Work

The result of development on the Woody project presented in this paper is a software system that provides a web interface for managing woodworking workshop orders and designing their products.

One of the main ideas of the project is to create a software solution for the wood industry which models the manufacturing process of doors and windows in a generic way. This provides an easily maintainable and expandable application, which does not require rebuilding of the source code.

Other workshops can therefore easily apply and benefit from this software by using the presented domain-specific language to create their own product templates.

Based on the needs of the inquired workshop and the ideas that arose during development, there are several possibilities for improvement, such as:

- interactive graphical user interface for creating new window types with refreshable layout drawing and variable parametrization;
- material needs view that lists the raw materials needed by type, including more aggregates, such as iron fittings ordering table, glass ordering table, sealing gel quantity, paint quantity or installation accessories (screw, polyurethane foam, etc.).
- creating remarks to clients that can be linked to projects, special requests and bids;

References

[1] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
[2] M. Fowler, *Domain-Specific Languages*, ser. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.
[3] M. Bojkowski. (2018) Code-First Database Design with Entity Framework and PostgreSQL. [Online]. Available: https://www.compose.com/articles/code-first-database-design-with-entity-framework-and-postgresql/
[4] G. Scheithauer and J. Terno, "A branch&bound algorithm for solving one-dimensional cutting stock problems exactly," *Applicationes Mathematicae*, vol. 23, no. 2, pp. 151–167, 1995.
[5] M. Mikowski and J. Powell, *Single Page Web Applications: JavaScript End-to-end*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2013.
[6] What is ASP.NET Core? A cross-platform web development framework. [Online]. Available: https://dotnet.microsoft.com/learn/web/what-is-aspnet-core
[7] Official PostgreSQL documentation. [Online]. Available: https://www.postgresql.org/about/
[8] Microsoft Docs: The Official Overview of Entity Framework Core. [Online]. Available: https://docs.microsoft.com/en-us/ef/core/
[9] Official React.js documentation. [Online]. Available: https://reactjs.org/tutorial/tutorial.html#what-is-react
[10] Official MobX documentation: Introduction. [Online]. Available: https://mobx.js.org/index.html#introduction
[11] M. Fowler. (2004) Inversion of Control Containers and the Dependency Injection pattern. [Online]. Available: https://martinfowler.com/articles/injection.html
[12] Official Bootstrap documentation: Introduction. [Online]. Available: https://getbootstrap.com/docs/4.3/getting-started/introduction/
[13] S. Chacon and B. Straub, *Pro Git*, 2nd ed. Berkely, CA, USA: Apress, 2014.
[14] Official GitLab documentation: What is GitLab? [Online]. Available: https://about.gitlab.com/what-is-gitlab/
[15] J. Turnbull, *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014.