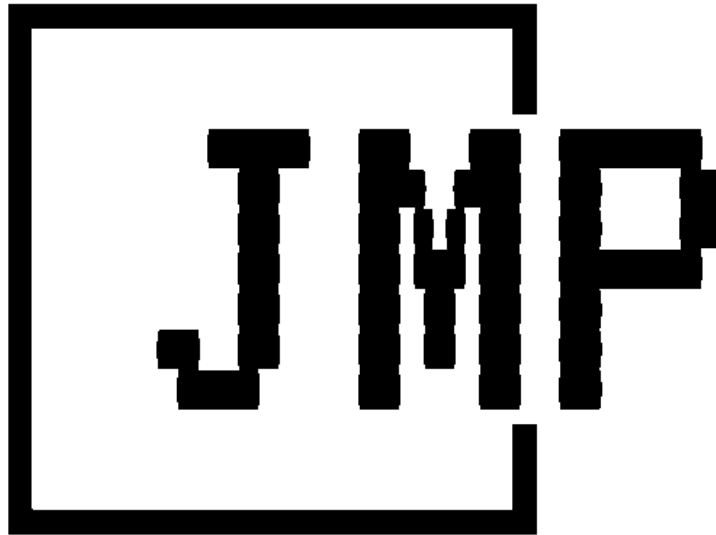


JMP Designdocument



Made by:

Mitchel Meskes

Joshua de Bruijn

Pedro Eduardo Cardoso

Date: 11 February 2024

Table of Contents

-	Table of Contents	2
-	Introduction	3
-	Important roles	4
-	Plan of requirements	5
-	Corporate identity	6
-	User stories	8
-	Wireframes	18
-	Mockups	30
-	Database model	41
-	ERD	46
-	Design patterns	48 t/m 67
-	Design Smells	68, 69
-	SOLIDs	70
-	UML Diagrams	71 t/m 87
-	Scrum	88, 89
-	Unit Tests	90
-	Security	91
-	Performance considerations	92, 93
-	Installation and Implementation Guide	94, t/m 96
-	Maintenance and Future Developments	97 t/m 100

Introduction

Welcome to our Blue Box coding project!

In an era where information management is crucial for the success of transportation companies, we at JMP have a vision to enhance these processes. Therefore, we are excited to announce our latest project, the Blue Box Tool. Our ambition is to transform the current Excel sheet system by developing an advanced software solution using PHP and Laravel.

The Blue Box Tool will serve as an all-in-one platform for information management, designed to improve the efficiency, accuracy, and flexibility of our operational processes.

With a focus on replacing the outdated Excel spreadsheets, we aim to create a streamlined and accessible web application that incorporates all the functionalities of the current system.

Important roles

Costumer

BlueBox

The Product Owners

Joshua de Bruijn

SCRUM Master

Joshua de Bruijn

Development Team

Pedro Eduardo Cardoso

Joshua de Bruijn

Mitchel Meskes

Plan of requirements

Key Features

Chosen coding language for Application Development:

- PHP

Chosen Framework:

- Laravel

The key functionalities of the application include:

- Translating all functionalities of the Excel sheet into a clear and accessible website
- Role-based access control with associated permissions
- Communication between management

Plan of requirements

The project has several key development points:

Transfer of all functionalities from the Excel sheet to a clear and accessible website. This includes simplifying complex data input and reporting processes, enabling users to easily navigate and work with the data.

Implementation of roles and associated permissions within the system. By defining user roles and assigning access rights, we can ensure data security and confidentiality while promoting productivity through targeted access to specific functions.

Improvement of communication between management and other stakeholders within the company. Through advanced communication modules, managers will be able to seamlessly communicate with various teams and departments, enhancing collaboration and accelerating decision-making.

With the Blue Box Tool, we aim to set the standard for information management in the transportation industry. We are committed to delivering a powerful and intuitive platform that not only transforms our own processes but also strengthens our competitive position.

Corporate identity

Color pallet / Font

The color palette for the Blue Box Tool will consist of the original company colors:

Used colors:

Background: (White) #FFFFFF

Details:  #066AB4

Font: Default font will be used.

These official company colors convey simplicity and professionalism.

Buttons

The buttons will be blue with white letters on the inside.

Background button:  #066AB4

Details button / text: (White) #FFFFFF

User story's

ID	Title	Work Item Type	Description	Acceptance Criteria	Priority	Effort	Business Value	Value Area	Tags
28586	1 Login Page	Epic	A login page is critical for securing access to online platforms and applications, providing users with a secure and convenient way to authenticate their identities.		1	8	90	Business	admin; management; sales; user
28612	1.1 Interface	Feature			2			Business	
28615	1.1.1: Design Login Interface	User Story	As a user, I want to have a visually appealing and intuitive login interface, so that I can easily access my account.	Non Functional * Design a clean and user-friendly login form layout. * Pop-up with login successful. * Possible pop-up with login failed. Functional * Include fields for username/email and password with appropriate labels and placeholders. * Implement responsive design to ensure compatibility across different devices and screen sizes. * A button for confirming the information to login. * A button for password forgotten. Prestate * N.V.T.	2			Business	
28616	1.1.2 Implement Form Styling	User Story	As a user, I want the styling for the login form to match the overall look and feel of the application, so that I as a user have a consistent experience on the application.	Non Functional * Apply CSS styles and typography to enhance the visual appeal of the login form. Functional * Ensure consistency with the application's branding and design guidelines. * Test the form's appearance on various browsers and devices to ensure uniformity. Performance * N.V.T.	3			Business	
28613	1.2 Backend	Feature			2			Business	
28617	1.2.1 Develop Backend Authentication Logic	User Story	As a Admin, I want there to be backend logic to authenticate user credentials during the login process, so that only authorized users can access the system.	Non Functional * N.V.T. Functional * Create API endpoints or controllers to handle login requests. * Verify the provided credentials against stored user data in the database. * Generate and return a session token or authentication token upon successful login. Performance * N.V.T.	1			Business	
28618	1.2.2 Secure Password Storage	User Story	As a Admin, I want to ensure that user passwords are securely stored in the database, so that sensitive information remains protected.	Non Functional * N.V.T. Functional * Hash user passwords using a strong cryptographic hashing algorithm (e.g., bcrypt). * Implement salted hashing to mitigate against rainbow table attacks. * Store hashed passwords securely in the database with appropriate access controls. Prestate * N.V.T.	1			Business	
28614	1.3 Database	Feature			2			Business	
28619	1.3.1 Set up Database Schema for User Accounts	User Story	As a Database administrator, I want the database schema to store user account information efficiently, so that login data can be managed effectively.	Non Functional * N.V.T. Functional * Design and implement database tables for storing user accounts and authentication details. * Define appropriate data types and constraints for each field to ensure data integrity. Prestate * Set up indexes for optimized query performance, especially for login-related operations.	3			Business	
28617	1.4 Features	Feature			2			Business	
28510	1.4.1 Recover my account when I forget my password.	User Story	As a User, I want to be able to recover my account for when I forget my password so that my data won't be lost. User gets an e-mail with the account credentials. The user will log back in and change the new password to his own password.	Non Functional * N.V.T. Functional * User gets an e-mail with the account credentials. Prestate * N.V.T.	2			Business	user
28509	1.4.2 Get a notification when a user forgot their password	User Story	As an Admin I want to get notified when a user forgot their password. Admin gets the notification on his dashboard when the user presses on the "forget password" button.	Non Functional * Pop-up with the notification of the reset password request. Functional * Admin gets an e-mail with the account of the user that wants to reset their password. Prestate * N.V.T.	3			Business	admin
28511	1.4.3 Reset a users password when they requested it	User Story	as an Admin, I want to be able to reset a users password when they requested it so that they can log in again. If the user forgets its password he needs to be able to reset it, to make this more safe, this goes via the admin. When the admin clicks on the message of the user wanting a password reset he resets the password, a password is generated and sent to the user and the database is updated with the newly generated password.	Non Functional * N.V.T. Functional * Admin is able to e-mail the user that wants to reset their password a new password. Prestate * N.V.T.	2			Business	admin
28591	2 Registration Page	Epic	The development of a registration page is a pivotal aspect of any online platform or application, serving as the gateway for users to create accounts and access the offered services.		3	8	10	Business	user
28621	2.1 Interface	Feature			3			Business	

28624	2.1.1 Design Registration Form	User Story	As a user, I want to see a clear and user-friendly registration form, so that I can easily sign up for the service.	Non Functional * Design a visually appealing and intuitive registration form layout. Functional * Include fields for essential user information such as username, email, password, etc. Prestatie * Implement client-side validation to provide feedback on input errors.	3				Business	
28625	2.1.2 Implement Form Validation	User Story	As a Admin, I want to ensure that the registration form includes robust validation mechanisms, so that we can prevent invalid or malicious input.	Non Functional * Display meaningful error messages to guide users in correcting input errors. Functional * Validate email format, password strength, and other relevant fields according to specified criteria. Prestatie * Implement server-side validation to enforce data integrity and security.	2				Business	
28626	2.2 Backend	Feature			2				Business	
28626	2.2.1 Develop Backend Registration Logic	User Story	As a Admin, I want there to be the necessary backend logic to handle user registration requests, so that new users can be securely added to the system.	Non Functional * N.V.T. Functional * Set up API endpoints or controllers to handle registration requests. * Hash passwords using industry-standard encryption algorithms before storing them in the database. Prestatie * Validate incoming data and sanitize inputs to prevent security vulnerabilities.	3				Business	
28627	2.2.2 Integrate Authentication Service	User Story	As a Database administrator, I want there to be a robust authentication service with the backend, so that registered users can securely access the system.	Non Functional * N.V.T. Functional * Integrate authentication middleware or libraries to validate user credentials during registration and login. * Implement token-based authentication for secure user sessions. Prestatie * Ensure compatibility with industry-standard authentication protocols like OAuth or JWT.	3				Business	
28627	2.3 Database	Feature			3				Business	
28628	2.3.1: Set up Database Schema for User Accounts	User Story	As a Database administrator, I want there to be a database schema to store user account information efficiently, so that registration data can be managed effectively.	Non Functional * N.V.T. Functional * Design and implement database tables for storing user accounts and associated information. * Define appropriate data types and constraints for each field to ensure data integrity. Prestatie * Set up indexes and foreign key relationships for optimized query performance and data consistency.	3				Business	
29318	2.4 Features	Feature			3				Business	
28505	2.4.1 Able to generate users	User Story	As an Admin, I want to be able to register user so that the user can use his account on the website.	Non Functional * N.V.T. Functional * A temporary password and the role off the user and when the account is made. * the user should be notified via email that he can log in. Prestatie * N.V.T.	2				Business	admin
28506	3 Home Page	Epic	The development of a home page is crucial for setting the tone and providing a gateway to the content and features of a website or application.		1	13	45		Business	admin; management; sales; user
28628	3.1 Interface	Feature			2				Business	
28629	3.1.1 Implement Homepage Layout	User Story	As a user, I want to see an appealing and intuitive layout on the homepage, so that I can easily navigate through the website.	Non Functional * Design and implement a responsive layout for the homepage. * Ensure consistency with the overall branding and style guide. Functional * Include prominent sections for featured content, navigation, and calls-to-action. Prestatie * N.V.T.	2				Business	
28630	3.2 Backend	Feature			2				Business	
28631	3.2.1 Implement Backend Functionality for Homepage	User Story	As a Admin, I want to ensure that the backend supports the necessary functionality for the homepage, so that the frontend can seamlessly interact with data and services.	Non Functional * Ensure proper error handling and logging for backend processes. Functional * Set up backend APIs or services to provide data for dynamic content on the homepage. Prestatie * Implement caching mechanisms to optimize performance.	2				Business	
29319	3.3 Features	Feature			2				Business	
28504	4 User List Page	Epic	A user list page is essential for providing administrators or authorized users with an overview of all registered users within a system.		2	13	25		Business	admin; management
28604	4.1 Interface	Feature			2				Business	
28605	4.1.1: Design User List Interface	User Story	As a user, I want to have a visually appealing and easy-to-navigate user list interface, so that I can view and interact with a list of users.	Non Functional * Design a user-friendly layout for displaying user information in a list format. Functional * Ensure responsive design to accommodate various screen sizes and devices. Prestatie * Include features such as pagination, sorting, and filtering options for better usability.	3				Business	
28606	4.1.2: Implement User List Component	User Story	As the client, I want the application to consist of reusable components for rendering the user list, so that the codebase remains modular and maintainable.	Non Functional * Implement functionality for sorting and filtering users based on different criteria. Functional * Develop React or Vue components for displaying user information in the user list. Prestatie * Write unit tests to ensure the correctness of the user list component.	3				Business	
28607	4.2 Backend	Feature			2				Business	
28609	4.2.1 Retrieve User Data from Backend	User Story	As a database administrator, I want the user data fetched from the backend API to populate the user list, so that users can be displayed accurately.	Non Functional * Handle error scenarios gracefully and provide feedback to the frontend in case of failures. Functional * Create API endpoints or services to retrieve user data from the backend database. * Implement RESTful API calls or GraphQL queries to fetch user information. Prestatie * N.V.T.	1				Business	
28610	4.2.2 Optimize API Calls for User List	User Story	As a database administrator, I want the API calls for fetching user data optimized, so that the user list page loads quickly and efficiently.	Non Functional * N.V.T. Functional * Monitor API performance and fine-tune query optimization strategies as needed. Prestatie * Implement pagination and lazy loading techniques to minimize the amount of data fetched at once. * Utilize caching mechanisms to reduce database load and improve response times.	3				Business	
28608	4.3 Database	Feature			2				Business	
28611	4.3.1 Configure Database Schema for User Data	User Story	As a database administrator, I want to set up the database schema to store user data efficiently, so that it can be retrieved and displayed accurately on the user list page.	Non Functional * N.V.T. Functional * Design and implement database tables for storing user information such as username, email, etc. * Normalize the database schema to eliminate redundancy and improve data consistency. Prestatie * Define appropriate indexes and constraints for efficient querying and data integrity.	2				Business	
29320	4.4 Features	Feature			2				Business	
29321	5 Contract Page	Epic			1	13	70		Business	
29322	5.1 Interface	Feature			1				Business	
29326	5.1.1: Design Contract Interface	User Story	As a user, I want to have a visually appealing and easy-to-navigate Contract page interface, so that I can view and interact with a list of contracts I am assigned to.	Non Functional * Design a user-friendly layout for displaying contract information. Functional * Ensure responsive design to accommodate various screen sizes and devices. Prestatie * Include features such as pagination, sorting, and filtering options for better usability.	2				Business	
29329	5.1.2: Implement Contract Component	User Story	As the client, I want the application to consist of reusable components for rendering the contracts, so that the codebase remains modular and maintainable.	Non Functional * Implement functionality for sorting and filtering contracts based on different criteria. Functional * Develop React or Vue components for displaying contracts. Prestatie * Write unit tests to ensure the correctness of the component.	3				Business	
29323	5.2 Backend	Feature			2				Business	
29327	5.2.1 Retrieve Contract Data from Backend	User Story	As a database administrator, I want to fetch contract data from the backend API, so that contracts can be displayed accurately.	Non Functional * Handle error scenarios gracefully and provide feedback to the frontend in case of failures. Functional * Create API endpoints or services to retrieve contract data from the backend database. * Implement RESTful API calls or GraphQL queries to fetch contract information. Prestatie * N.V.T.	1				Business	
29328	5.2.2 Optimize API Calls for Contracts	User Story	As the database administrator, I want the API calls for fetching contract data optimized, so that the contracts load quickly and efficiently.	Non Functional * N.V.T. Functional * Monitor API performance and fine-tune query optimization strategies as needed. Prestatie * Implement pagination and lazy loading techniques to minimize the amount of data fetched at once. * Utilize caching mechanisms to reduce database load and improve response times.	3				Business	
29324	5.3 Database	Feature			2				Business	
28506	5.3.1 Able to add parents to the database.	User Story	As an admin, I want to be able to add parents to the database, so that the application stays up to date with the growth of the company.	Non Functional * N.V.T. Functional * The admin user should be the only one who can populate the parent table. Prestatie * N.V.T.	1				Business	admin
28507	5.3.2 Able to remove parents from the database.	User Story	As an admin, I want to be able to remove parents so that the interface stays up to date and room for errors are minimized.	Non Functional * N.V.T. Functional * The admin is the only one who can delete items from the parent table. Prestatie * N.V.T.	2				Business	admin
29325	5.4 Features	Feature			2				Business	
28522	Able to write a actions so that we can execute strategies.	User Story	As a sales, I want to be able to write a actions so that we can execute strategies.	sales can make actions based on the strategies of their parents.	2				Business	sales
28518	I want to add customer divisions to a parent	User Story	As a parent can be divided into different sections, for example customer divisions so: as a sales I want to add customer divisions to a parent to take action on different parts of the parent company.	the admin should be able to navigate to a parent and add divisions through text input field	2				Business	admin; sales

28519	I want to be able to mail in one click with my sales partner	User Story	As a parent are always 2 sales employees, those two need to be able to contact each other easily so: as a sales I want to be able to mail in one click with my sales partner on that parent so <u>contact will go easy</u>	there is a button on the screen when a sales is navigated to a parent and they can click that button to open the preferred mail application to send a new mail to the other sales on this parent	2			Business	sales
28516	I want to add countries to a parent	User Story	As a parent can spread to different nations and the sales have to divide the company up in those countries to make detailed strategies so: as a sales I want to add countries to a parent so that I know where we're doing business	the admin should be able to navigate to a parent and add countries they can choose out of the selection in the countries table	2			Business	admin; sales
28513	I want to be able to add clients to parents	User Story	the admin should be able to do the same as sales so: as an admin I want to be able to add clients to parents so that I can <u>help sales</u>	the admin should be able to navigate to a parent and add clients this is through a text input field	2			Business	admin
28603	I want to be able to edit contracts (if not signed)	User Story	As an Admin I want to be able to edit contract if the contract is not signed, so that when a contract has the incorrect information that is initially needed the admin will be able to <u>reassign it</u>	Admin is able to edit the unsigned contract, it will show up white, if the contract is signed it will be grey.	2			Business	admin
28595	I want to be able to assign sales to parents	User Story		As an Admin I want to be able to assign sales to parents so that parents won't be left without a sales	2			Business	admin
28504	I want to receive a mail when I get assigned as a resource to a strategy	User Story	As a user, I want to receive a mail when I get assigned as a resource to a strategy, so that I know what I have to do.	when a user is assigned to a strategy, an automatic noreply mail should be generated and send to the user. the two sales on the parent should be added in the cc	2			Business	user
28523	Able to update actions statuses.	User Story	As a sales I want to be able to update actions statuses so that we know what the status is of the action. One of the following states: <u>proposed, completed, canceled, suspended, active</u>	sales can update action statuses based on the strategies of their parents	2			Business	sales
28517	I want to add categories	User Story	As a parent can have different categories, that is why as a sales I want to add categories so that I know what kind of business <u>we're doing</u>	the admin should be able to navigate to a parent and add categories through text input field	2			Business	admin; sales
28520	Able to make strategies so that we can improve efficiency.	User Story	As a sales I want to be able to make strategies so that we can improve efficiency.	Sales can populate the strategy table but only of their parents	2			Business	sales
28521	Able to update the status of the strategies.	User Story	As a sales I want to be able to update the status of the strategies so that we know the status of the strategy. One of the following states: <u>proposed, completed, canceled, suspended, active</u>	sales can alter the status column of the strategies table but only the strategies of their parents	2			Business	sales
28674	6 Design Document	Epic			1			Business	
28680	Documentatie	Feature			2			Business	
28677	6.1 Wireframes	Feature			2			Business	
28681	6.1.1 Login Page	User Story			2			Business	
28682	6.1.2 Register Page	User Story			2			Business	
28683	6.1.3 Home Page	User Story			2			Business	
28684	6.1.4 Member List Page	User Story			2			Business	
29386	6.1.5 Contract Page	User Story			2			Business	
28685	6.1.6 Header	User Story			2			Business	
28686	6.1.7 Footer	User Story			2			Business	
28678	6.2 Mockups	Feature			2			Business	
28691	6.2.1 Login Page	User Story			2			Business	
28692	6.2.2 Register Page	User Story			2			Business	
28693	6.2.3 Home Page	User Story			2			Business	
28694	6.2.4 Member List Page	User Story			2			Business	
29385	6.2.5 Contract Page	User Story			2			Business	
28679	6.3 ERD	Feature			2			Business	
28690	6.3.1 Version 1	User Story			2			Business	

User story's

General Stories

The user stories are created using Azure DevOps/Github Projects

How we create our user stories:

We select an EPIC, which is then broken down into several Sprints, and within one of those sprints, we then pick up parts of that EPIC so that each person works on a part of the project.

(See the following example)

Step 1

Choose an EPIC. (User Registration and Authentication)

Step 2

Then we break it down into parts and each person picks a story.

Step 3

Then each person selects one of the following points and starts working on it.

For each EPIC we tackle, we rotate through all roles so that everyone works on all aspects, including backend, frontend, and databases.

User story's

EPIC 1: Login Page

User Story 1.1.1: Design Login Interface

As a user, I want to have a visually appealing and intuitive login interface, so that I can easily access my account. **Acceptance Criteria:**

- Design a clean and user-friendly login form layout.
- Include fields for username/email and password with appropriate labels and placeholders.
- Implement responsive design to ensure compatibility across different devices and screen sizes.

User Story 1.1.2: Implement Form Styling

As a user, I want the styling for the login form to match the overall look and feel of the application, so that I as a user have a consistent experience on the application. **Acceptance Criteria:**

- Apply CSS styles and typography to enhance the visual appeal of the login form.
- Ensure consistency with the application's branding and design guidelines.
- Test the form's appearance on various browsers and devices to ensure uniformity.

User Story 1.2: Develop Backend Authentication Logic

As an Admin, I want there to be backend logic to authenticate user credentials during the login process, so that authorized users can access the system. **Acceptance Criteria:**

- Create API endpoints or controllers to handle login requests.
- Verify the provided credentials against stored user data in the database.
- Generate and return a session token or authentication token upon successful login.

User Story 1.2.2: Secure Password Storage

As an Admin, I want to ensure that user passwords are securely stored in the database, so that sensitive information remains protected. **Acceptance Criteria:**

- Hash user passwords using a strong cryptographic hashing algorithm (e.g., bcrypt).
- Implement salted hashing to mitigate against rainbow table attacks.
- Store hashed passwords securely in the database with appropriate access controls.

EPIC 2: Registration Page

User Story 2.1.1: Design Registration Form

As a user, I want to see a clear and user-friendly registration form, so that I can easily sign up for the service. **Acceptance Criteria:**

- Design a visually appealing and intuitive registration form layout.
- Include fields for essential user information such as username, email, password, etc.
- Implement client-side validation to provide feedback on input errors.

User Story 2.1.2: Implement Form Validation

As a user, I want to ensure that the registration form includes robust validation mechanisms, so that we can prevent invalid or malicious input. **Acceptance Criteria:**

- Display error messages to guide users in correcting input errors.
- Validate email format, password strength, and other relevant fields according to specified criteria.
- Implement server-side validation to enforce data integrity and security.

User Story 2.2.1: Develop Backend Logic for Registration

As an Admin, I want there to be necessary backend logic to handle user registration requests, so that new users can be securely added to the system. **Acceptance Criteria:**

- Set up API endpoints or controllers to handle registration requests.
- Hash passwords using industry-standard encryption algorithms before storing them in the database.
- Validate incoming data and sanitize inputs to prevent security vulnerabilities.

User Story 2.2.2: Integrate Authentication Service

As a Database administrator, I want there to be a robust authentication service within the backend, so that registered users can securely access the system. **Acceptance Criteria:**

- Integrate authentication libraries to validate user credentials during registration and login.
- Implement token-based authentication for secure user sessions.
- Ensure compatibility with industry-standard authentication protocols.

EPIC 3: Home Page

User Story 3.1.1: Implement Home Page Layout

As a user, I want to see an appealing and intuitive layout on the homepage, so that I can easily navigate through the website. **Acceptance Criteria:**

- Design and implement a responsive layout for the homepage.
- Ensure consistency with the overall branding and style of the application.
- Include prominent sections for featured content, navigation, and calls-to-action.

User Story 3.2.1: Integrate Backend Logic for Homepage

As an Admin, I want to ensure that the backend supports the necessary functionality for the homepage, so that the frontend can seamlessly interact with data and services. **Acceptance Criteria:**

- Ensure proper error handling and logging for backend processes.
- Set up backend APIs or services to provide data for dynamic content on the homepage.
- Implement caching mechanisms to optimize performance.

EPIC 4: User List Page

User Story 4.1.1: Design User List Interface

As a user, I want to have a visually appealing and easy-to-navigate user list interface, so that I can view and manage users in a list format. **Acceptance Criteria:**

- Design a user-friendly layout for displaying user information in a list format.
- Ensure responsive design to accommodate various screen sizes and devices.
- Include features such as pagination, sorting, and filtering options for better usability.

User Story 4.1.2: Implement User List Component

As the client, I want the application to consist of reusable components for rendering the user list, so that the codebase remains modular and maintainable. **Acceptance Criteria:**

- Implement functionality for sorting and filtering users based on different criteria.
- Develop React or Vue components for displaying contracts.
- Write unit tests to ensure the correctness of the user list component.

EPIC 5: Contact Page

User Story 5.1.1: Design Contact Interface

As a user, I want to have a visually appealing and easy-to-navigate Contact page interface, so that I can easily interact with a list of contacts. **Acceptance Criteria:**

- Design a user-friendly layout for displaying contact information.
- Ensure responsive design to accommodate various screen sizes and devices.
- Include features such as search, sorting, and filtering options for better usability.

User Story 5.1.2: Implement Contact Component

As the client, I want the application to consist of reusable components for rendering the contact list, so that the codebase remains modular and maintainable. **Acceptance Criteria:**

- Implement functionality for sorting and filtering contacts based on different criteria.
- Develop React or Vue components for displaying contacts.
- Write unit tests to ensure the correctness of the component.

User Story 5.2.1: Retrieve Contact Data from Backend

As a database administrator, I want to fetch contact data from the backend API, so that contacts can be displayed accurately. **Acceptance Criteria:**

- Handle error scenarios gracefully and provide feedback to the frontend in case of failures.
- Create API endpoints or GraphQL queries to fetch contact data from the backend database.
- Ensure proper validation and sanitation of fetched contact data.

User Story 5.2.2: Optimize API Calls for Contact Data

As a database administrator, I want the API calls for fetching contact data optimized, so that the contacts load quickly and efficiently. **Acceptance Criteria:**

- Monitor API performance and fine-tune optimization strategies as needed.
- Implement lazy loading techniques to minimize the amount of data fetched at once.
- Utilize caching mechanisms to reduce database load and improve response times.

User Story 5.3.1: Add Parents to the Database

As an admin, I want to be able to add parents to the database, so that the application stays up to date with the growth of the company. **Acceptance Criteria:**

- The admin user should be the only one who can populate the parent table.

User Story 5.3.2: Add Divisions to a Parent

As an admin, I want to be able to remove parents from the interface by using a text box, so that errors are minimized. **Acceptance Criteria:**

- The admin is the only one who can remove items from the parent table.

User Story 5.4.1: Retrieve User Data from Backend

As a database administrator, I want to fetch user data from the backend API to populate the user list, so that users can be displayed accurately. **Acceptance Criteria:**

- Handle error scenarios gracefully and provide feedback to the frontend in case of failures.
- Create API endpoints or RESTful API calls or GraphQL queries to fetch user data from the backend database.
- Ensure proper validation and sanitation of fetched user data.

User Story 5.4.2: Optimize API Calls for User List

As a database administrator, I want the API calls for fetching user data optimized, so that the user list page loads quickly and efficiently. **Acceptance Criteria:**

- Monitor API performance and fine-tune optimization strategies as needed.
- Implement lazy loading techniques to minimize the amount of data fetched at once.
- Utilize caching mechanisms to reduce database load and improve response times.

User Story 5.4.3: Configure Database Schema for User Data

As a database administrator, I want to set up the database schema to store user data efficiently, so that it can be reviewed and displayed accurately on the user list page. **Acceptance Criteria:**

- Design and implement database schema for storing user information such as username, email, etc.
- Normalize the database schema to eliminate redundancy and improve data consistency.
- Define appropriate indexes and constraints for efficient querying and data integrity.

User Story 5.4.4: Recover my Forgotten Password

As a user, I want to be able to recover my account when I forget my password so that I can get back into the system. **Acceptance Criteria:**

- User gets an e-mail with the account credentials.

User Story 5.4.5: Get a Notification when a User Forgets their Password

As an Admin, I want to get notified when a user forgets their password so that admins see when users press the “forget password” button. **Acceptance Criteria:**

- Pop-up with the notification of the reset

Wireframe

Header



This will be shown on the top of every page of the website. You will see the BlueBox logo on the top left and on the top right there will be navigation, the navigation will be shown with a dropdown menu.

Admin wireframes

Admin contract panel

A wireframe for an admin contract panel. It features a light gray background. At the top, the text "Find a user" is centered in a large, bold font. Below this is a rounded rectangular search bar with the placeholder text "Searchbar". Underneath the search bar, the text "Results" is centered in a large, bold font. At the bottom, there is a large rounded rectangular box containing the text "John Doe" and "Users email" stacked vertically.

As an admin, you can search for users registered on the website. Admins can use the "search bar" to find a user and view the results in the list under "Results".

Contract title

payment terms

"Text Input"

Country

"Selector with dropdown"

Rebate

"Text Input"

Rebate end

DD/MM/YYYY

Contract end

DD/MM/YYYY

"Update Button"

As an admin, you can edit the contracts. You can modify the payment terms, country, rebate, rebate end date, and contract end date. To save these changes, you need to press the "Update" button.

Review

Review base

"Text Input"

Review start

DD/MM/YYYY

Review end

DD/MM/YYYY

CTO value

"Text Input"

Type

"Selector with dropdown"

"Update Button"

As an admin, you are able to edit the review details, including the start and end dates, the CTO value, and the type.

Action Title

Title

Strategy



Status



Who



Support



When

Update

As an admin you are able to edit the Actions, you are able to edit the title, strategy, the status of the action, who did the action, whoever supported with the action and when the action started

Strategy

Summary

"Text Input"

Today

"Text Input"

Tomorrow

"Text Input"

How

"Text Input"

Resource

"Selector with dropdown"

Internal alignment

"Text Input"

"Update Button"

As an admin, you are able to edit strategies. Within the strategies, you can modify the summary, today's input, tomorrow's input, the approach (how), the resource, and the internal alignment.



Within the strategy page, there are actions that can also be edited by the admin. Additionally, actions can be added using the button at the top right.

Admin parent panel

Contracts of (this parent)

"List with Contracts"

Contract

Lorem ipsum

Contract

Lorem ipsum

As an admin, you can edit the contracts of a certain parent.

Assigned employees

Sales Support

"Selector with dropdown"

Sales Admin

"Inputfield for text"

"Update Button"

As an admin you are able to edit the assigned employees to a certain parent, you can select the support through a dropdown menu. For the admin you can write the sales admins name on the input field.

As an admin, you can search for parents, and the search results will be displayed in the list below the search bar.

Strategies

+

"List with Strategies"

Strategy title

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor

"Add Strategy button"

Strategy title

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor

As an admin, you have the ability to add or remove strategies assigned to a parent, with all strategy details displayed in the list.

Admin parents panel

Add parent

Submit Button

Find a Parent

Search-bar

List with each parent company registerd

As an admin, you have the ability to add parents by entering their name, category, partner, as well as filling in the input fields for sales support and sales administrators.

Add category

"Text field for a name"

"Text field for description"

"Submit Button"

As an admin, you have the ability to add categories by entering the text and providing a description.

Mockups

Admin mockups

Admin Parents Panel

The mockup shows a web browser window with a 'New Tab' and a search bar. The page header includes the 'BLUE BOX PARTNERS' logo and navigation links for 'Contact', 'Parents', and 'Users'. The main content area is divided into three panels:

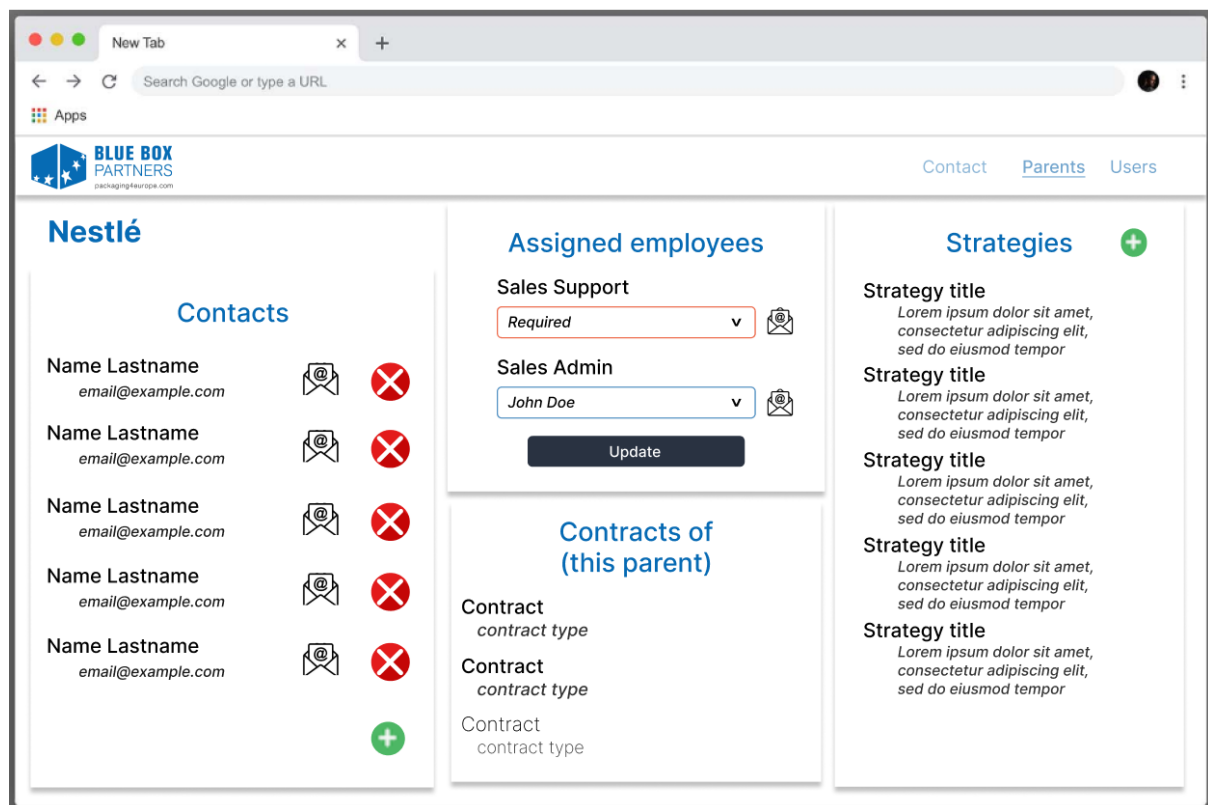
- Find a Parent:** Features a search bar with a magnifying glass icon. Below it, a list of parents is shown: 'Nestle inc. (action required)', 'Other Parent (action required)', and three 'Other Parent' entries. The first two entries are highlighted in red.
- Add parent:** Contains four input fields: 'Name (required)', 'Category (required)' (with a dropdown arrow), 'Partner (required)' (with a note '//more than one is possible'), and 'Sales support' (with a dropdown arrow). Below these is a 'Sales administrator' dropdown and an 'Add parent' button.
- Add category:** Contains a 'Name (required)' input field and a 'Description' text area. Below these is an 'Add category' button.

As an admin, you can find a parent listed below the search bar. Whenever a parent requires action, it will be highlighted in red and marked with "(action required)".

You can easily add a parent by filling in the name, category, and partner fields, and selecting the appropriate options from the "Sales Support" and "Sales Administrator" dropdowns.

You can also add a category by entering the name and the description below it.

Admin Parent Panel



As a parent, you have access to view all contacts associated with the parent account. You can add a contact by pressing the green button at the bottom of the “contacts” section, and you can remove a contact by pressing the red button.

Additionally, you can assign roles to employees, such as "sales support" and "sales admin".

You can also view contracts linked to the parent account, including the contract details and types.

Additionally, you can view strategies, which include the strategy title and description. You can also add strategies by pressing on the green button.

Admin edit action

New Tab

Search Google or type a URL

Apps

BLUE BOX PARTNERS
packaging4europe.com

Contact Parents Users

Action title

Title
title

Belongs to
Strategy ▼

Status
Pending ▼

Who
John Doe ▼

Support
Niccollette Claassen ▼

When
dd/mm/yyyy

Update

As an admin, you can update the actions created by parents. You can adjust the title, the assigned strategy, the action's status, the responsible person, support, and schedule.

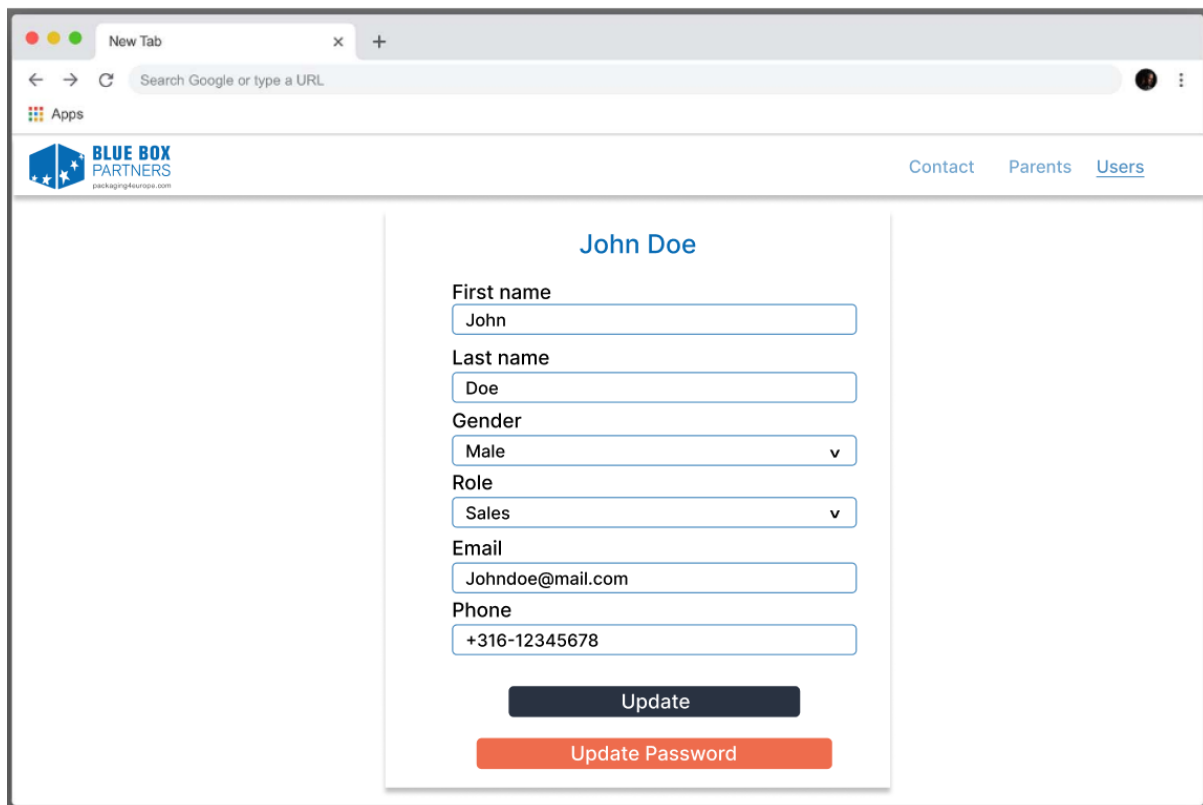
Admin Edit Strategy

The screenshot displays a web application interface for editing a strategy. The browser window shows a 'New Tab' with a search bar. The page header features the 'BLUE BOX PARTNERS' logo and navigation links for 'Contact', 'Parents', and 'Users'. The main content area is split into two panels. The left panel, titled 'Strategy', contains several sections with text input fields: 'Summary' (with placeholder text), 'Today', 'Tomorrow', 'How', 'Resource' (with a dropdown menu showing 'Nicolette Claassen'), and 'Internal alignment'. An 'Update' button is located at the bottom of this panel. The right panel, titled 'Actions', has a green '+' button and a list of actions with status labels: 'Action Active', 'Action Pending', 'Action Canceled', and 'Action Completed'.

As an admin, you can edit a strategy by modifying the summary, updating the current and future strategies, adjusting the resources, and managing the internal alignment.

You can also see if an action is active, pending, canceled, or completed. By pressing the green button, you can add a new action.

Admin edit users



The screenshot shows a web browser window with a 'New Tab' and a search bar. The page is titled 'BLUE BOX PARTNERS' with a logo and a URL 'packaginghours.com'. The navigation menu includes 'Contact', 'Parents', and 'Users'. The main content area displays the user details for 'John Doe' with the following fields:

- First name: John
- Last name: Doe
- Gender: Male (dropdown menu)
- Role: Sales (dropdown menu)
- Email: Johndoe@mail.com
- Phone: +316-12345678

At the bottom of the form, there are two buttons: a dark blue 'Update' button and a red 'Update Password' button.

As an admin, you can edit user details, including their first and last name, gender, role, email, and phone number.

You can also update their password by pressing the red button at the bottom.

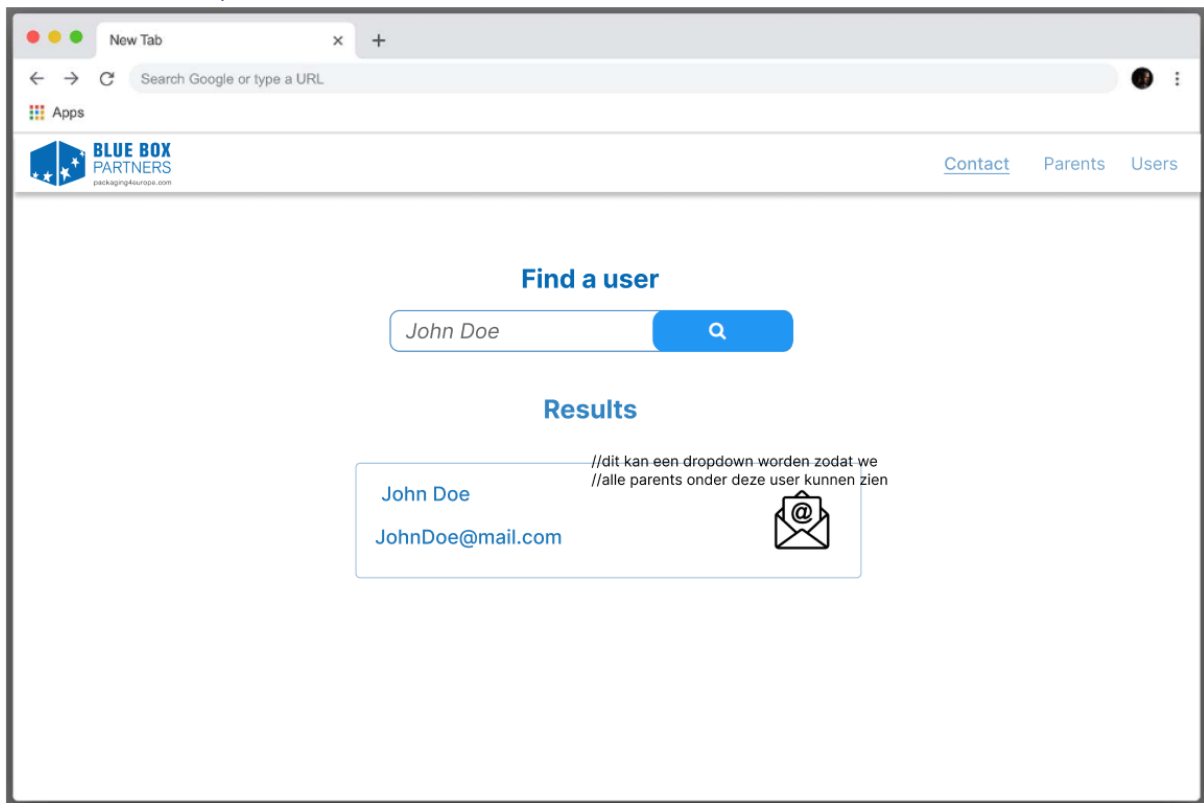
Admin edit contract

The screenshot shows a web browser window with a 'New Tab' and a search bar. The page header includes the 'BLUE BOX PARTNERS' logo and navigation links for 'Contact', 'Parents', and 'Users'. The main content area is divided into two panels. The left panel, titled 'Contract title', contains the following fields: 'payment terms' (text input), 'Country' (dropdown menu), 'Rebate' (text input), 'Rebate end' (DD/MM/YYYY date input), and 'Contract end' (DD/MM/YYYY date input), with an 'Update' button at the bottom. The right panel, titled 'Review', contains the following fields: 'Review base' (text input), 'Review start' (DD/MM/YYYY date input), 'Review end' (DD/MM/YYYY date input), 'CTO value' (text input), and 'Type' (dropdown menu), with an 'Update' button at the bottom.

As an admin, you can edit contracts, including changing the payment terms, country, rebate start and end dates, and contract end date.

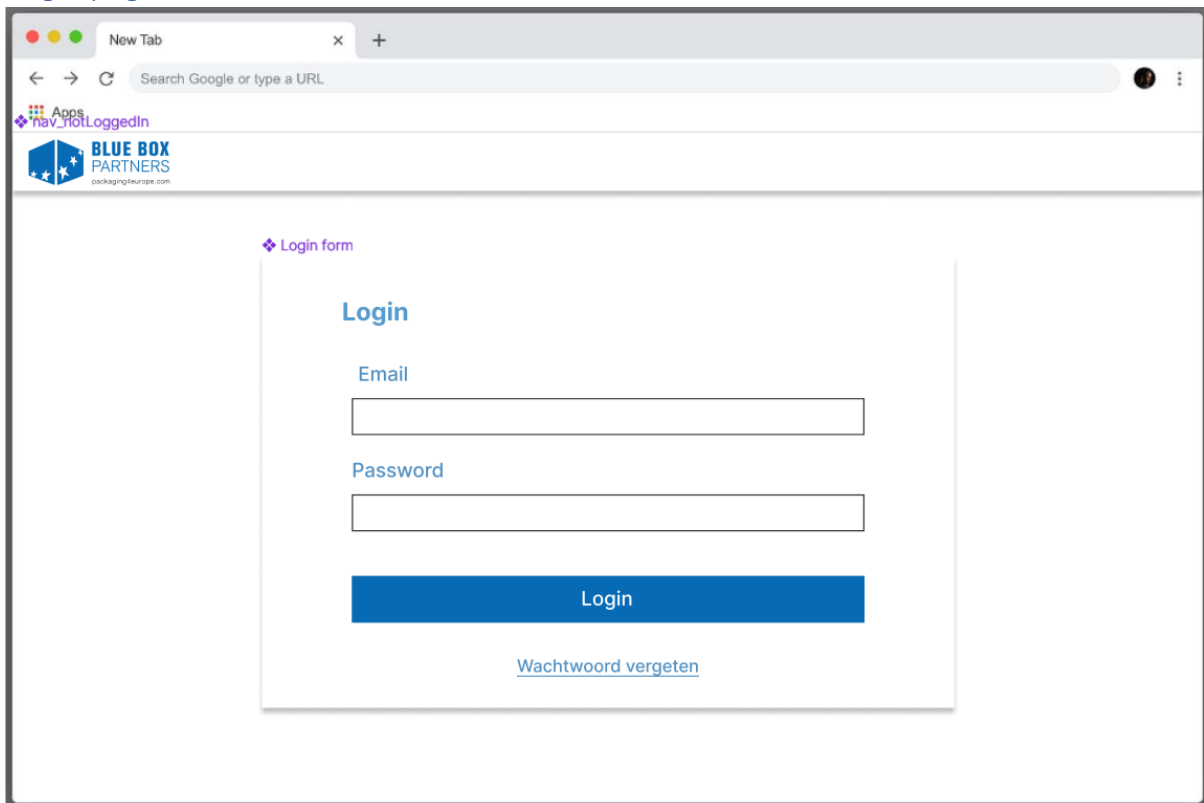
You are also able to edit reviews, including modifying the review base, review start and end dates, CTO value, and type.

Admin contact panel



As an admin, you can find a user in the contacts panel. When you type the user's name, their information will be displayed.

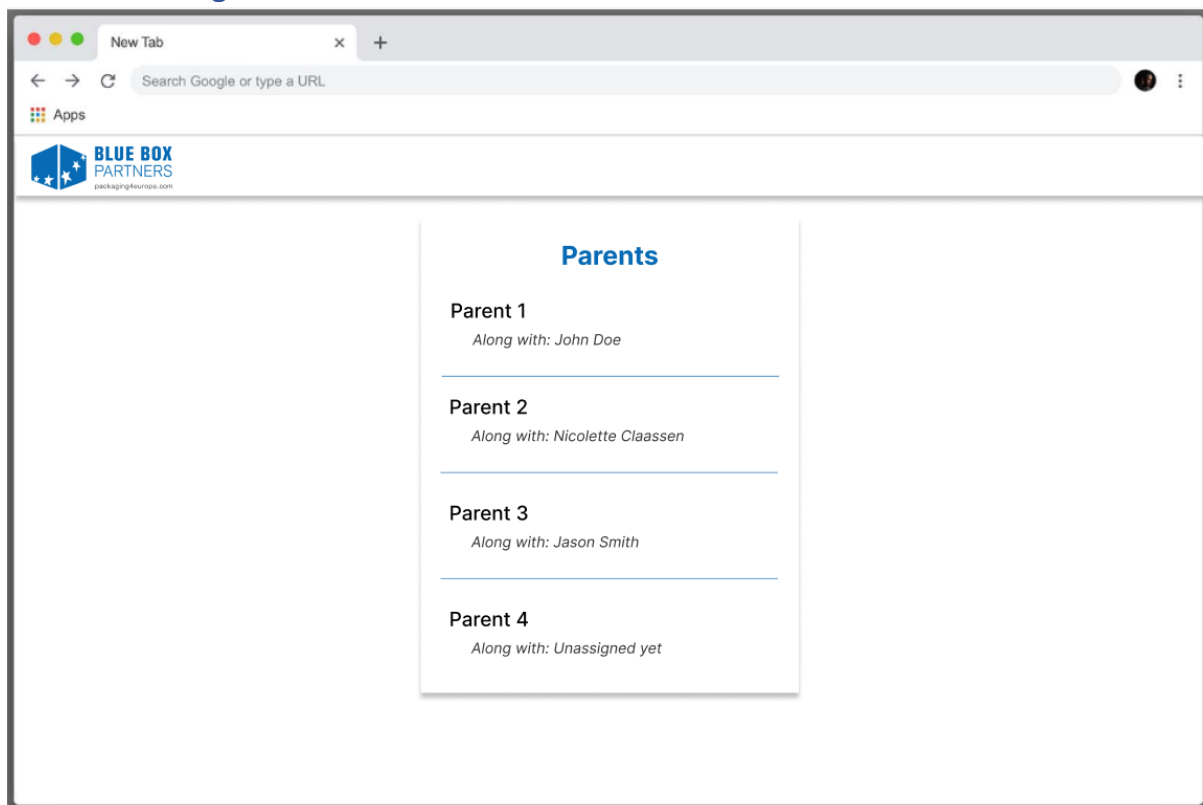
Login page



The screenshot shows a web browser window with a single tab titled "New Tab". The address bar contains the text "Search Google or type a URL". The page header features a navigation bar with a logo on the left that says "BLUE BOX PARTNERS" and "packaging4europe.com" below it. To the right of the logo, there is a status indicator "nav_notLoggedIn" and a small "Apps" icon. The main content area is titled "Login form" and contains a "Login" heading. Below the heading are two input fields: "Email" and "Password". A blue "Login" button is positioned below the password field. At the bottom of the form, there is a link that says "Wachtwoord vergeten".

As a user, you can log in once an admin has created your account.

Sales Home Page



As a sales user, you can see the parent accounts and the people assigned to each parent account.

Sales Parent Page

The screenshot shows a web browser window with the following elements:

- Browser Header:** "New Tab", "Search Google or type a URL", and "Apps" button.
- Page Header:** "BLUE BOX PARTNERS" logo and "Home Parents" navigation links.
- Main Content Area:**
 - Nestlé Section:**
 - Colleague:** John Doe (with email icon).
 - Contracts of (this parent):** A list of three contracts, each with "Contract" and "contract type" labels.
 - Contacts Section:** A list of five contacts, each with "Name Lastname" and "email@example.com". Each contact has a red 'X' button for removal and a green '+' button for addition.
 - Strategies Section:** A list of four strategies, each with "Strategy title" and "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor" text. Each strategy has a green '+' button for addition.

As a sales user, you can see the colleagues and the contracts associated with the parent account. You can add contacts to a parent account by pressing the green button and remove them by pressing the red button. Additionally, you can add strategies by pressing the green button.

Sales Update / Edit Strategies

The screenshot shows a web browser window with a 'New Tab' and a search bar. The application header includes the 'BLUE BOX PARTNERS' logo and navigation links for 'Home' and 'Parents'. The main content is divided into two panels. The 'Strategy' panel on the left contains several form fields: 'Summary' (a large text area with placeholder text), 'Today' (a text field), 'Tomorrow' (a text field), 'How' (a text field), 'Resource' (a dropdown menu with 'Nicolette Claassen' selected), and 'Internal alignment' (a text field). An 'Update' button is at the bottom of this panel. The 'Actions' panel on the right, titled 'Actions' with a green plus icon, lists four actions: 'Action Active', 'Action Pending', 'Action Canceled', and 'Action Completed'.

When you press the green button in the "strategies" section on the parent page, you will be redirected to a page where you can update and edit strategies. On this page, you can edit the summary, input details for today and tomorrow, specify and select resources using the dropdown menu, and write the internal alignment.

You are also able to add actions by pressing on the green button on the top right.

Database model

Entities and Attributes and the explanation for each attribute.

General Concepts:

- **PK (Primary Key):** Uniquely identifies each record in the table.
- **FK (Foreign Key):** References the primary key in another table to establish a relationship.

Tables and Their Fields:

Actions

- **ID (PK):** Ensures each action is uniquely identifiable.
- **Strategy ID (FK):** Links the action to a specific strategy.
- **Title:** Provides a brief label for the action.
- **Description:** Offers detailed information about the action.
- **User ID (Support) (FK):** Connects the action to a support user.
- **Status:** Tracks the current state of the action.
- **When (Date):** Records a relevant date for the action.

BBP_Employees

- **ID (PK):** Unique identifier for each employee.
- **Gender ID (FK):** Links to the **Genders** table to specify the employee's gender.
- **Role (FK):** Links to the **Roles** table to specify the employee's role.
- **First Name:** The employee's first name.
- **Last Name:** The employee's last name.
- **Email:** The employee's email address.
- **Phone:** The employee's phone number.

Categories

- **ID (PK):** Unique identifier for each category.
- **Name:** The name of the category.
- **Description:** A detailed description of the category.

Competitors

- **ID (PK)**: Unique identifier for each competitor.
- **Competitor**: The name of the competitor.
- **Volume**: The volume of business or transactions related to the competitor.

Competitors_Parents

- **ID (PK)**: Unique identifier for each record.
- **Competitor ID (FK)**: Links to the **Competitors** table.
- **Parent ID (FK)**: Links to the **Parents** table.

Contacts

- **ID (PK)**: Unique identifier for each contact.
- **Gender ID (FK)**: Links to the **Genders** table.
- **Role**: The role of the contact.
- **First Name**: The contact's first name.
- **Last Name**: The contact's last name.
- **Email**: The contact's email address.
- **Phone**: The contact's phone number.
- **Location**: The location of the contact.

Contacts_Parents

- **ID (PK)**: Unique identifier for each record.
- **Contact ID (FK)**: Links to the **Contacts** table.
- **Parent ID (FK)**: Links to the **Parents** table.

ContractDetails

- **ID (PK)**: Unique identifier for each contract detail record.
- **Contract ID (FK)**: Links to the **Contracts** table.
- **Country ID (FK)**: Links to the **Countries** table.
- **End Date**: The end date of the contract.
- **Payment Terms**: The payment terms of the contract.
- **Rebate**: Any rebate associated with the contract.
- **Rebate End (Date)**: The end date for the rebate.
- **Paper Review (Bool)**: Indicates if a paper review is required (Boolean).

ContractDetails_Country

- **ID (PK)**: Unique identifier for each record.
- **Country ID (FK)**: Links to the **Countries** table.
- **Area**: The area within the country.

Contracts

- **ID (PK)**: Unique identifier for each contract.
- **Parent ID (FK)**: Links to the **Parents** table.
- **Contract Type**: The type of contract.
- **Has Contract (Bool)**: Indicates if a contract exists (Boolean).
- **Is Local (Bool)**: Indicates if the contract is local (Boolean).

Countries

- **ID (PK)**: Unique identifier for each country.
- **Code**: The countrycode.
- **Name**: The name of the country.

Customers

- **ID (PK)**: Unique identifier for each customer.
- **Parent ID (FK)**: Links to the **Parents** table.
- **Name**: The name of the customer.

Divisions

- **ID (PK)**: Unique identifier for each division.
- **Division**: The name or identifier of the division.

Employees_Strategies

- **ID (PK)**: Unique identifier for each record.
- **Employee ID (FK)**: Links to the **BBP_Employees** table.
- **Strategy ID (FK)**: Links to the **Strategies** table.

Genders

- **ID (PK)**: Unique identifier for each gender.
- **Gender**: The name or type of gender.

Parents

- **ID (PK)**: Unique identifier for each parent record.
- **Partner**: The name of the partner.
- **Category**: The category associated with the parent.
- **Employee ID (Sales) (FK)**: Links to the **BBP_Employees** table for the sales employee.
- **Employee ID (Support) (FK)**: Links to the **BBP_Employees** table for the support employee.

Parents_Sales

- **ID (PK)**: Unique identifier for each record.
- **Parent ID (FK)**: Links to the **Parents** table.
- **BBP Sales ID (FK)**: Links to the **BBP_Employees** table for the sales employee.

Partners

- **ID (PK)**: Unique identifier for each partner.
- **Name**: The name of the partner.

Reviews

- **ID (PK)**: Unique identifier for each review.
- **Contract ID (FK)**: Links to the **Contracts** table.
- **Review Start (Date)**: The start date of the review.
- **Review End (Date)**: The end date of the review.
- **Review Base (String)**: The base or criteria of the review.

Roles

- **ID (PK)**: Unique identifier for each role.
- **Role**: The name or description of the role.

Strategies

- **ID (PK)**: Unique identifier for each strategy.
- **Parent ID (FK)**: Links to the **Parents** table.
- **BBP Volume ID (FK)**: Links to a **Volumes** table.
- **Resource ID (FK)**: Links to a **Resources** table.
- **Summary**: A summary of the strategy.
- **Today**: Information about today's strategy or actions.
- **Tomorrow**: Information about tomorrow's strategy or actions.
- **How**: Details on how the strategy will be implemented.
- **When (Internal Agreement)**: Internal agreement date or schedule.
- **Resources (FK)**: Links to a **Resources** table.

Statuses

- **ID (PK)**: Unique identifier for each status.
- **Title**: The title of the status.
- **Description**: A detailed description of the status.

Volumes

- **ID (PK)**: Unique identifier for each volume record.
- **Customer ID (FK)**: Links to the **Customers** table.
- **Country ID (FK)**: Links to the **Countries** table.
- **Division (Nullable)**: The division which can be null.
- **Factory**: The factory associated with the volume.
- **Last Year (Volume)**: Volume data from the previous year.
- **BBP Volume**: The BBP-specific volume.
- **Total Company Volume**: The total volume for the company.
- **BBP Share**: The BBP share of the total volume.
- **Potential Volume**: The potential volume.

ERD

ERD stands for Entity-Relationship Diagram. It is a type of diagram used in database design to visually represent the relationships within a system.

ERDs depict the structure of a database, showing the entities, the attributes within those entities, and the connections between those entities.

Components of an ERD:

Entities: These are the objects or concepts that have data stored about them. Entities are typically nouns, like in our case "Parents," "Partners," or "Contracts."

Attributes: These are the pieces of information that describe an entity. For example, our "Parents" entity has attributes like "parent_ID," "partner," "category," "name," "employee_id(sales)," and "employee_id(support)"

Relationships: These describe how entities are related to one another. Like how a Parent company is assigned to a certain contract.

Notations in ERDs:

Rectangles represent entities.

Ovals represent attributes.

Diamonds or lines represent relationships.

Connecting lines with different symbols (crow's foot, straight line, etc.) indicate the cardinality of the relationship (one-to-one, one-to-many, many-to-many).

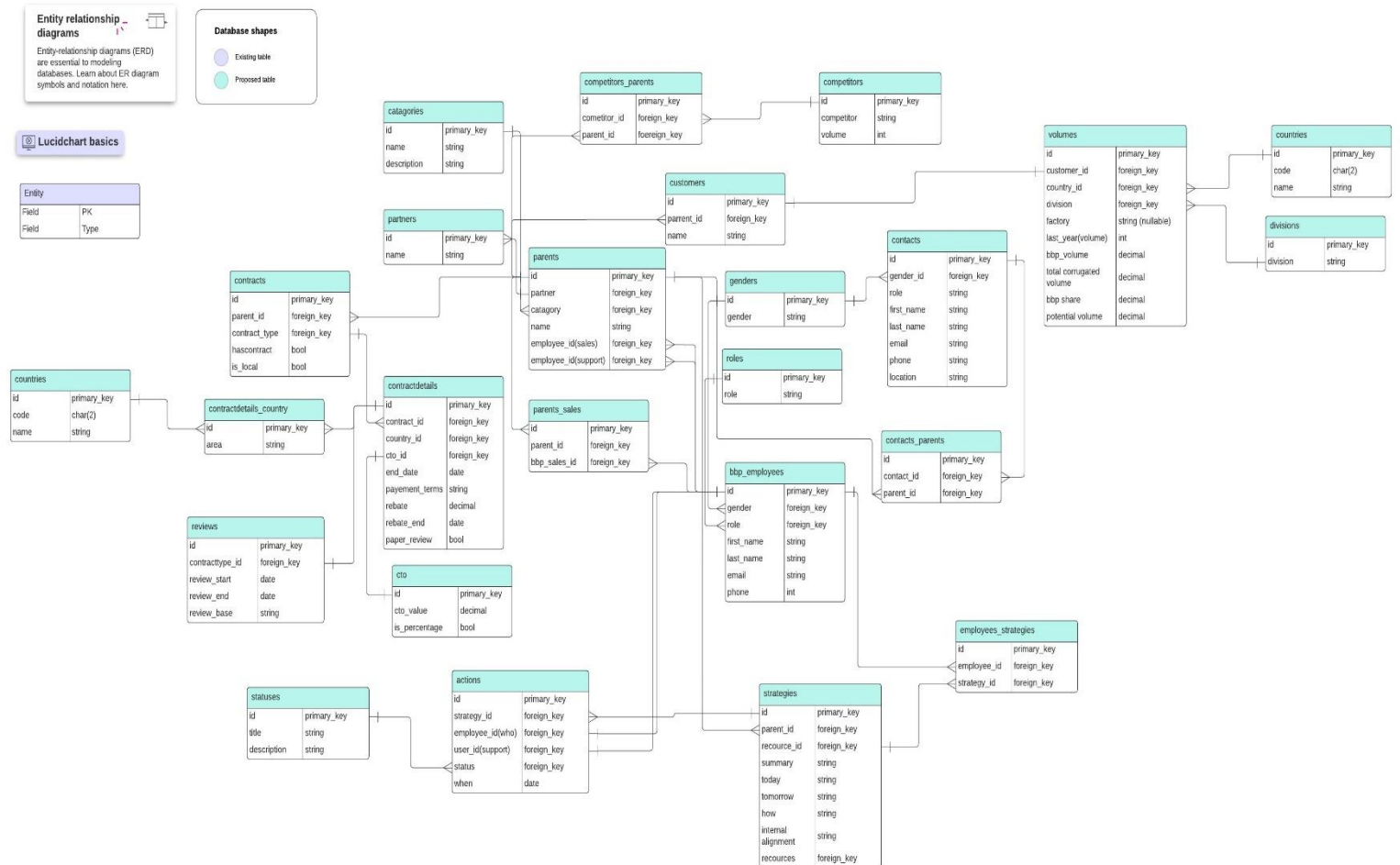
Purpose of an ERD:

To provide a clear and structured visual representation of the data.

To facilitate communication between stakeholders about the system's data model.

To serve as a blueprint for designing and creating a database.

ERD diagram



Design Patterns

Creational Patterns

Creational patterns in software development are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Creational patterns aim to provide flexibility in the instantiation process while promoting the reuse of existing code and adhering to principles like encapsulation and separation of concerns.

1. **Singleton Pattern:** Ensures that a class has only one instance and provides a global point of access to that instance. It's useful when exactly one object is needed to coordinate actions across the system.
2. **Factory Method Pattern:** Defines an interface for creating an object but allows subclasses to alter the type of objects that will be created. It provides a way for a class to delegate the instantiation logic to subclasses.
3. **Abstract Factory Pattern:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes. It's useful when a system should be independent of how its objects are created, composed, and represented.
4. **Builder Pattern:** Separates the construction of a complex object from its representation so that the same construction process can create different representations. It's useful when the construction process must allow for different representations of the object that's being built.
5. **Prototype Pattern:** Creates new objects by copying an existing object, known as the prototype. This pattern allows for reducing the cost of creating objects compared to creating them from scratch.
6. **Object Pool Pattern:** Maintains a pool of reusable objects for use by multiple clients. It's useful when the cost of initializing a class instance is high or when instantiation and destruction of many instances would degrade system performance.
7. **Dependency Injection Pattern:** Provides a technique to inject dependencies into an object rather than having the object create its dependencies. It promotes loose coupling and allows for easier testing and configuration management.

These creational patterns provide solutions to various problems encountered during object creation and initialization, helping developers to write more maintainable, flexible, and efficient software systems. Choosing the appropriate creational pattern depends on the specific requirements and constraints of the system being developed.

Behavioural Patterns

Behavioral patterns in software development are design patterns that focus on how objects interact and communicate with each other. These patterns define patterns of communication between classes and objects, emphasizing the assignment of responsibilities between objects and how they interact to achieve desired behaviors. Behavioral patterns help in making the system more flexible and easier to understand by promoting loose coupling and high cohesion.

1. **Observer Pattern:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It's useful when there is a need for a consistent state between multiple objects.
2. **Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from clients that use it. It's useful when different variations of an algorithm are required.
3. **Command Pattern:** Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations. It's useful for implementing undo functionality, auditing, and logging.
4. **Chain of Responsibility Pattern:** Allows a set of objects to handle a request one by one, providing a flexible way to pass a request along a chain of handlers. It's useful when the system should process a request through multiple handlers, and the specific handler to process the request is not known a priori.
5. **Iterator Pattern:** Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It's useful when there is a need to traverse a collection of objects without knowing its internal structure.
6. **State Pattern:** Allows an object to alter its behavior when its internal state changes. The object appears to change its class. It's useful when an object's behavior depends on its state, and it must change its behavior dynamically based on that state.
7. **Visitor Pattern:** Defines a new operation to a set of objects without changing their class. It's useful when there is a need to perform operations on elements of a complex structure without adding new methods to their classes.
8. **Interpreter Pattern:** Defines grammar for interpreting sentences in a language and provides a way to evaluate sentences in a language. It's useful when there is a need to interpret sentences or commands in a language.
9. **Mediator Pattern:** Defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly and allows for a more flexible and reusable design.

These behavioral patterns provide solutions to various challenges encountered in designing systems where objects need to communicate and collaborate effectively. Choosing the appropriate behavioral pattern depends on the specific requirements and constraints of the system being developed.

Concurrency Patterns

Concurrency patterns in software development are design patterns that address the challenges related to managing concurrent execution and synchronization in multi-threaded or distributed systems. These patterns help in designing systems that efficiently handle multiple tasks running simultaneously, ensuring proper synchronization and coordination between concurrent operations.

1. **Thread Pool Pattern:** Pre-creates a pool of threads to execute tasks concurrently, avoiding the overhead of creating and destroying threads for each task. It's useful when there's a need to limit the number of concurrent threads and manage resources efficiently.
2. **Producer-Consumer Pattern:** Decouples the production of tasks (produced by one or more threads) from their consumption (consumed by one or more threads), using a shared data structure such as a queue. It's useful when there's a need to balance the workload between producers and consumers, preventing resource contention.
3. **Readers-Writers Pattern:** Manages access to a shared resource that can be read by multiple threads simultaneously but only written to by one thread at a time. It's useful when there are multiple readers and occasional writers accessing a shared resource, and it's necessary to prioritize access to ensure data consistency.
4. **Barrier Pattern:** Synchronizes a group of threads at a specific point in their execution, forcing them to wait until all threads reach the barrier before proceeding. It's useful when there's a need to coordinate multiple threads to perform a task in stages.
5. **Semaphore Pattern:** Controls access to a shared resource using a counter that limits the number of threads allowed to access the resource concurrently. It's useful when there's a need to limit access to a shared resource or control the number of threads accessing it.
6. **Mutex Pattern:** Provides mutual exclusion to shared resources by allowing only one thread to access the resource at a time. It's useful when there's a need to prevent multiple threads from concurrently modifying shared data, ensuring data integrity.
7. **Monitor Pattern:** Uses synchronization primitives such as locks to protect critical sections of code and coordinate access to shared resources. It's useful when there's a need to ensure exclusive access to shared resources while minimizing contention and deadlock.
8. **Actor Pattern:** Models concurrent computations as independent actors that communicate through message passing. Each actor has its own state and processes messages asynchronously, enabling scalable and fault-tolerant systems.
9. **Futures and Promises Pattern:** Represents the result of an asynchronous operation that may not yet be available, allowing the program to continue executing other tasks while waiting for the result. It's useful for asynchronous programming and parallel computation.

These concurrency patterns provide solutions to various challenges encountered in designing concurrent and parallel systems, helping developers to write scalable, responsive, and efficient software. Choosing the appropriate concurrency pattern depends on the specific requirements and constraints of the system being developed, such as the level of parallelism required, the nature of shared resources, and the desired synchronization mechanisms.

Structural Patterns

Structural patterns in software development are design patterns that focus on organizing classes and objects to form larger structures, making it easier to manage complex systems and relationships between components. These patterns help in defining how classes and objects are composed to create larger structures while keeping them flexible and efficient.

1. **Adapter Pattern:** Allows incompatible interfaces to work together by providing a bridge between them. It's like using an adapter to plug a European device into a North American outlet.
2. **Bridge Pattern:** Separates an abstraction from its implementation so that the two can vary independently. It's like building a bridge that connects two different islands but can be built using different materials.
3. **Composite Pattern:** Composes objects into three structures to represent part-whole hierarchies. It's like organizing files and folders on a computer where folders can contain both files and other folders.
4. **Decorator Pattern:** Dynamically adds new functionality to objects by wrapping them with other objects. It's like adding toppings to a pizza without changing its base ingredients.
5. **Facade Pattern:** Provides a simplified interface to a complex subsystem, making it easier to use. It's like using a remote control to operate various devices instead of dealing with each device's controls individually.
6. **Flyweight Pattern:** Shares common objects to reduce memory usage, especially when dealing with a large number of similar objects. It's like sharing resources such as pens and paper in a classroom rather than giving each student their own set.
7. **Proxy Pattern:** Provides a placeholder for another object to control access to it. It's like using a security guard to control access to a building, allowing or denying entry based on certain criteria.
8. **Mixin Pattern:** Allows objects to inherit functionality from multiple sources without requiring multiple inheritance. It's like adding specific traits to a character in a game to enhance their abilities.
9. **Module Pattern:** Organizes code into modules or namespaces to improve maintainability and encapsulation. It's like dividing a large book into chapters and sections to make it easier to navigate and understand.

These structural patterns provide solutions to various challenges encountered in designing software systems, helping developers to create well-organized, scalable, and maintainable code. Choosing the appropriate structural pattern depends on the specific requirements and constraints of the system being developed, such as the need for flexibility, scalability, or performance optimization.

Used Design Patterns

Creational Patterns

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Singleton Pattern

- **Description:** Ensures a class has only one instance and provides a global point of access to it. This pattern is useful when exactly one object is needed to coordinate actions across the system.
- **Application Area:** Managing a single instance of configuration or logging.
- **Example:**

```
// Config.php
class Config {
    private static $instance = null;
    private $settings = [];

    // Private constructor to prevent multiple instances
    private function __construct() {}

    // Static method to get the single instance
    public static function getInstance() {
        if (self::$instance == null) {
            self::$instance = new Config();
        }
        return self::$instance;
    }

    // Set a configuration value
    public function set($key, $value) {
        $this->settings[$key] = $value;
    }

    // Get a configuration value
    public function get($key) {
        return $this->settings[$key] ?? null;
    }
}

// Usage
$config = Config::getInstance();
$config->set('siteName', 'My Website');
echo $config->get('siteName'); // Outputs: My Website
```

Explanation:

- The 'Config' class has a private constructor to prevent direct instantiation.
- The 'getInstance' method checks if an instance already exists. If not, it creates one.
- The 'set' method allows setting configuration values, and the 'get' method retrieves them.
- This ensures that only one 'Config' instance exists throughout the application, centralizing configuration management.

Factory Method

- **Description:** Defines an interface for creating an object but lets subclasses alter the type of objects that will be created. This pattern is useful for creating objects when the exact type of the object might not be known until runtime.
- **Application Area:** Creating objects like User, Role, and Gender.

- **Example:**

```
// UserFactory.php
class UserFactory {
    public static function create(array $data): User {
        return new User($data);
    }
}

// Usage in Controller
class UserController extends Controller {
    public function createUser(Request $request) {
        $userData = $request->only(['name', 'email', 'password']);
        $user = UserFactory::create($userData);
        $user->save();
        return response()->json(['message' => 'User created successfully']);
    }
}
```

- **Explanation:**

The 'UserFactory' class provides a static method create to instantiate User objects.

This 'factory' method encapsulates the instantiation logic, making it easier to manage and modify.

The 'UserController' uses this factory method to create and save a new User, simplifying the controller's responsibilities.

Builder Pattern

- **Description:** Separates the construction of an object from its representation, allowing the same construction process to create various representations. This is particularly useful for constructing complex objects with multiple optional fields.
- **Application Area:** Creating user forms with various fields.
- **Example:**

```
// UserBuilder.php
class UserBuilder {
    private $name;
    private $email;
    private $role;

    public function setName($name): UserBuilder {
        $this->name = $name;
        return $this;
    }

    public function setEmail($email): UserBuilder {
        $this->email = $email;
        return $this;
    }

    public function setRole($role): UserBuilder {
        $this->role = $role;
        return $this;
    }

    public function build(): User {
        return new User($this->name, $this->email, $this->role);
    }
}

// Usage
$builder = new UserBuilder();
$user = $builder->setName('John')->setEmail('john@example.com')->setRole('Admin');
```

- **Explanation:**

The 'UserBuilder' class provides methods to set various properties of a User.

Each method returns the builder object itself, allowing for method chaining.

The 'build' method constructs the final User object using the provided properties.

This pattern makes it easy to create complex objects step-by-step, with clear and readable code.

Behavioural Patterns

Behavioral patterns deal with communication between objects, focusing on how they interact and fulfill their responsibilities.

Strategy Pattern

- **Description:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern lets the algorithm vary independently from the clients that use it.
- **Application Area:** Implementing different strategies for handling requests.
- **Example:**

```
// StrategyInterface.php
interface StrategyInterface {
    public function execute($data);
}

class ConcreteStrategyA implements StrategyInterface {
    public function execute($data) {
        // Specific algorithm implementation for strategy A
        return "Strategy A executed with data: " . json_encode($data);
    }
}

class ConcreteStrategyB implements StrategyInterface {
    public function execute($data) {
        // Specific algorithm implementation for strategy B
        return "Strategy B executed with data: " . json_encode($data);
    }
}

class StrategyContext {
    private $strategy;

    public function setStrategy(StrategyInterface $strategy) {
        $this->strategy = $strategy;
    }

    public function executeStrategy($data) {
        return $this->strategy->execute($data);
    }
}

// Usage in StrategyController
class StrategyController extends Controller {
    public function executeStrategy($type, $data) {
        $context = new StrategyContext();
        if ($type == 'A') {
            $context->setStrategy(new ConcreteStrategyA());
        } else {
            $context->setStrategy(new ConcreteStrategyB());
        }
        return response()->json(['result' => $context->executeStrategy($data)]);
    }
}
```

- **Explanation:**

The StrategyInterface defines a common interface for all strategies.

ConcreteStrategyA and ConcreteStrategyB are implementations of this interface, each providing a different algorithm.

The StrategyContext class holds a reference to a strategy and delegates the execution to the strategy object.

The StrategyController sets the appropriate strategy based on the input and executes it, allowing for flexible and interchangeable algorithms.

Observer Pattern

- **Description:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Application Area:** Monitoring user updates.
- **Example:**

```
// Observer.php
interface Observer {
    public function update($eventData);
}

class EmailNotifier implements Observer {
    public function update($eventData) {
        // Send email notification
        echo "EmailNotifier: Notified with data: " . json_encode($eventData) . "\n";
    }
}

class LogNotifier implements Observer {
    public function update($eventData) {
        // Log the event
        echo "LogNotifier: Notified with data: " . json_encode($eventData) . "\n";
    }
}

class Subject {
    private $observers = [];

    public function attach(Observer $observer) {
        $this->observers[] = $observer;
    }

    public function notify($eventData) {
        foreach ($this->observers as $observer) {
            $observer->update($eventData);
        }
    }
}

// Usage in ProfileController
class ProfileController extends Controller {
    private $subject;

    public function __construct() {
        $this->subject = new Subject();
        $this->subject->attach(new EmailNotifier());
        $this->subject->attach(new LogNotifier());
    }

    public function update(ProfileUpdateRequest $request): RedirectResponse {
        $request->user()->fill($request->validated());
        if ($request->user()->isDirty('email')) {
            $request->user()->email_verified_at = null;
        }
        $request->user()->save();

        $this->subject->notify(['user' => $request->user(), 'event' => 'profile-updated']);

        return Redirect::route('profile.edit')->with('status', 'profile-updated');
    }
}
```

- **Explanation:**

The Observer interface defines a common method for receiving updates.

EmailNotifier and LogNotifier are concrete implementations of the Observer interface, each performing a specific action when notified.

The Subject class manages a list of observers and notifies them of any state changes.

The ProfileController uses the subject to notify observers when a profile is updated, demonstrating how to decouple the notification logic from the main business logic.

Command Pattern

- **Description:** Encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations. This pattern is useful for logging changes, undoing operations, and more.
- **Application Area:** Encapsulating requests in the ActionController.
- **Example:**

```
// Command.php
interface Command {
    public function execute();
}

class CreateActionCommand implements Command {
    private $action;

    public function __construct(Action $action) {
        $this->action = $action;
    }

    public function execute() {
        $this->action->save();
        echo "Action created: " . json_encode($this->action) . "\n";
    }
}

class UpdateActionCommand implements Command {
    private $action;

    public function __construct(Action $action) {
        $this->action = $action;
    }

    public function execute() {
        $this->action->update();
        echo "Action updated: " . json_encode($this->action) . "\n";
    }
}

class CommandInvoker {
    private $commands = [];

    public function addCommand(Command $command) {
        $this->commands[] = $command;
    }

    public function executeCommands() {
        foreach ($this->commands as $command) {
            $command->execute();
        }
    }
}

// Usage in ActionController
class ActionController extends Controller {
    private $invoker;

    public function __construct() {
        $this->invoker = new CommandInvoker();
    }

    public function store(StoreActionRequest $request) {
        $action = new Action($request->validated());
        $command = new CreateActionCommand($action);
        $this->invoker->addCommand($command);
        $this->invoker->executeCommands();
    }
}
```

- **Explanation:**

The Command interface defines a common method for executing actions.

CreateActionCommand and UpdateActionCommand are implementations of this interface, each encapsulating a specific request.

The CommandInvoker class manages a list of commands and executes them.

The ActionController uses the invoker to encapsulate the creation and execution of commands, making it easier to manage and extend action-related logic.

Concurrency Patterns

Concurrency patterns deal with multi-threaded programming paradigms.

Thread Pool Pattern

- **Description:** Manages a pool of threads to perform tasks concurrently. This pattern improves performance by reusing existing threads rather than creating new ones for each task.
- **Application Area:** Processing multiple user registrations.
- **Example:**

```
// ThreadPool.php
class ThreadPool {
    private $pool;

    public function __construct($size) {
        $this->pool = new SplQueue();
        for ($i = 0; $i < $size; $i++) {
            $this->pool->enqueue(new Worker());
        }
    }

    public function execute(callable $task) {
        if (!$this->pool->isEmpty()) {
            $worker = $this->pool->dequeue();
            $worker->run($task);
            $this->pool->enqueue($worker);
        }
    }
}

class Worker {
    public function run(callable $task) {
        $task();
    }
}

// Usage in RegisterController
class RegisterController extends Controller {
    public function bulkRegister(array $users) {
        $threadPool = new ThreadPool(5);

        foreach ($users as $user) {
            $threadPool->execute(function() use ($user) {
                User::create($user);
            });
        }
    }
}
```

- **Explanation:**

The ThreadPool class manages a fixed number of Worker threads.

The execute method assigns tasks to available workers from the pool.

The Worker class runs the tasks assigned to it.

The RegisterController uses the thread pool to handle multiple user registrations concurrently, improving performance by reusing threads.

Mutex Pattern

- **Description:** Ensures that only one thread accesses a resource at a time, preventing race conditions. This pattern is essential for maintaining data consistency when multiple threads access shared resources.
- **Application Area:** Safely updating user profiles.
- **Example:**

```
// Mutex.php
class Mutex {
    private $lock = false;

    public function acquire() {
        while ($this->lock) {
            usleep(100); // Wait for the lock to be released
        }
        $this->lock = true;
    }

    public function release() {
        $this->lock = false;
    }
}

// Usage in ProfileController
class ProfileController extends Controller {
    private $mutex;

    public function __construct() {
        $this->mutex = new Mutex();
    }

    public function update(ProfileUpdateRequest $request): RedirectResponse {
        $this->mutex->acquire();
        try {
            $request->user()->fill($request->validated());
            if ($request->user()->isDirty('email')) {
                $request->user()->email_verified_at = null;
            }
            $request->user()->save();
        } finally {
            $this->mutex->release();
        }

        return Redirect::route('profile.edit')->with('status', 'profile-updated');
    }
}
```

- **Explanation:**

The Mutex class controls access to a shared resource by acquiring and releasing a lock.

The acquire method waits until the lock is available, ensuring only one thread can access the resource.

The release method frees the lock for other threads.

The ProfileController uses the mutex to ensure that profile updates are performed safely, preventing concurrent modifications that could lead to inconsistent data.

Read-Write Lock Pattern

- **Description:** Allows multiple threads to read a resource but only one to write to it, balancing the need for concurrency with data integrity.
- **Application Area:** Managing category data.
- **Example:**

```
// ReadWriteLock.php
class ReadWriteLock {
    private $readers = 0;
    private $writer = false;

    public function acquireRead() {
        while ($this->writer) {
            usleep(100);
        }
        $this->readers++;
    }

    public function releaseRead() {
        $this->readers--;
    }

    public function acquireWrite() {
        while ($this->readers > 0 || $this->writer) {
            usleep(100);
        }
        $this->writer = true;
    }

    public function releaseWrite() {
        $this->writer = false;
    }
}

// Usage in CategoryController
class CategoryController extends Controller {
    private $lock;

    public function __construct() {
        $this->lock = new ReadWriteLock();
    }

    public function store(Request $request) {
        $this->lock->acquireWrite();
        try {
            $request->validate([
                'category_name' => 'required',
                'description' => 'nullable',
            ]);

            Category::create([
                'name' => $request['category_name'],
                'description' => $request['description']
            ]);
        } finally {
            $this->lock->releaseWrite();
        }

        return redirect()->route('parents.index')->with('success', 'Category');
    }

    public function index() {
        $this->lock->acquireRead();
        try {
            $categories = Category::all();
        } finally {
            $this->lock->releaseRead();
        }

        return view('categories.index', ['categories' => $categories]);
    }
}
```

- **Explanation:**

The ReadWriteLock class manages concurrent access to a resource by allowing multiple readers but only one writer.

The acquireRead method ensures that no writers are active before allowing a thread to read.

The acquireWrite method ensures that no readers or writers are active before allowing a thread to write.

The CategoryController uses this lock to manage concurrent access to category data, ensuring that reads and writes are performed safely.

Structural Patterns

Structural patterns deal with object composition and typically involve ways to compose objects to form larger structures.

Adapter Pattern

- **Description:** Converts the interface of a class into another interface that clients expect. This pattern allows classes to work together that couldn't otherwise because of incompatible interfaces.
- **Application Area:** Adapting third-party service responses.
- **Example:**

```
// ThirdPartyService.php
class ThirdPartyService {
    public function getUserData($id) {
        // Returns data in a format that doesn't match our application
        return [
            'id' => $id,
            'full_name' => 'John Doe',
            'email_address' => 'john@example.com'
        ];
    }
}

// UserAdapter.php
class UserAdapter {
    private $thirdPartyService;

    public function __construct(ThirdPartyService $service) {
        $this->thirdPartyService = $service;
    }

    public function getUser($id) {
        $data = $this->thirdPartyService->getUserData($id);
        return [
            'id' => $data['id'],
            'name' => $data['full_name'],
            'email' => $data['email_address']
        ];
    }
}

// Usage in ProfileController
class ProfileController extends Controller {
    public function show($id) {
        $service = new ThirdPartyService();
        $adapter = new UserAdapter($service);
        $user = $adapter->getUser($id);
        return view('profile.show', ['user' => $user]);
    }
}
```

- **Explanation:**

The ThirdPartyService class provides user data in a format that doesn't match our application's expected format.

The UserAdapter class adapts this data to match the format our application expects.

The ProfileController uses the UserAdapter to retrieve user data in the correct format, enabling seamless integration with the third-party service.

Facade Pattern

- **Description:** Provides a unified interface to a set of interfaces in a subsystem. This pattern simplifies interactions with complex subsystems by providing a higher-level interface.
- **Application Area:** Simplifying interactions with user profile management.
- **Example:**

```
// ProfileFacade.php
class ProfileFacade {
    public static function updateProfile(User $user, array $data) {
        $user->fill($data);
        if ($user->isDirty('email')) {
            $user->email_verified_at = null;
        }
        $user->save();
    }

    public static function deleteProfile(User $user) {
        Auth::logout();
        $user->delete();
        session()->invalidate();
        session()->regenerateToken();
    }
}

// Usage in ProfileController
class ProfileController extends Controller {
    public function update(ProfileUpdateRequest $request): RedirectResponse {
        ProfileFacade::updateProfile($request->user(), $request->validated());
        return Redirect::route('profile.edit')->with('status', 'profile-updated');
    }

    public function destroy(Request $request): RedirectResponse {
        $request->validateWithBag('userDeletion', [
            'password' => ['required', 'current_password'],
        ]);

        ProfileFacade::deleteProfile($request->user());
        return Redirect::to('/');
    }
}
```

- **Explanation:**

The ProfileFacade class provides static methods to handle profile updates and deletions, encapsulating the complex logic involved.

The ProfileController uses these facade methods to perform profile operations, reducing its own complexity and improving readability.

The facade pattern simplifies the interaction with the profile subsystem, making the controller code cleaner and easier to maintain.

Decorator Pattern

- **Description:** Adds behavior to objects dynamically without affecting the behavior of other objects from the same class. This pattern is useful for extending the functionalities of classes in a flexible and reusable way.
- **Application Area:** Adding logging functionality to controllers.
- **Example:**

```
// logger.php
interface Logger {
    public function log($message);
}

class FileLogger implements Logger {
    public function log($message) {
        file_put_contents('log.txt', $message.PHP_EOL, FILE_APPEND);
    }
}

class LoggerDecorator implements Logger {
    protected $logger;

    public function __construct(Logger $logger) {
        $this->logger = $logger;
    }

    public function log($message) {
        $this->logger->log($message);
    }
}

class CategoryControllerLogger extends LoggerDecorator {
    public function log($message) {
        parent::log("CategoryController: " . $message);
    }
}

// Usage in CategoryController
class CategoryController extends Controller {
    private $logger;

    public function __construct() {
        $this->logger = new CategoryControllerLogger(new FileLogger());
    }

    public function store(Request $request) {
        $request->validate([
            'category_name' => 'required',
            'description' => 'nullable',
        ]);

        $this->logger->log('Creating new category: ' . $request['category_name']);

        Category::create([
            'name' => $request['category_name'],
            'description' => $request['description']
        ]);

        return redirect()->route('parents.index')->with('success', 'Category added');
    }
}
```

- **Explanation:**

The Logger interface defines a common logging method.

The FileLogger class implements this interface to log messages to a file.

The LoggerDecorator class extends the functionality of any Logger implementation by adding additional behavior.

The CategoryControllerLogger class adds specific logging behavior for the CategoryController.

In CategoryController, the decorator is used to add logging functionality, enhancing the class without modifying its core logic.

Design Smells

Design smells, also known as design anti-patterns or code smells, refer to certain structures in the design or implementation of software that are indicative of potential problems. These smells often suggest poor design choices that can lead to issues such as decreased maintainability, readability, scalability, or performance. Identifying and addressing these smells is crucial for maintaining high-quality code.

Design Smells in RegisterController.php

1. Long Method:

The create and update methods are relatively long and contain nested conditionals, which can be broken down into smaller methods to improve readability and maintainability.

2. Duplicated Code:

The repeated code for fetching roles and genders and rendering views can be refactored into helper methods or use a facade.

3. Inappropriate Intimacy:

- The update method accesses User model methods directly and manipulates user data, which should be handled by a service or repository layer to reduce coupling between the controller and model.

4. God Class:

The controller is handling multiple responsibilities, such as user creation, updating, and view rendering. This violates the Single Responsibility Principle (SRP).

Design Smells in SingleMasterController.php

1. Long Parameter List:

The index method fetches multiple related models, which can be simplified by using a service to handle the data fetching logic.

2. Inappropriate Intimacy:

The controller accesses Master and User models directly. It would be better to delegate this to a service layer.

Design Smells in StrategyController.php

1. Long Parameter List:

The index method fetches a single model, which is appropriate, but consider using a service for data fetching.

2. Inappropriate Intimacy:

Directly accessing the Strategy model can be delegated to a service layer.

Design Smells in ActionController.php

1. Empty Methods:

The controller contains several empty methods, which may indicate incomplete implementation. Each method should have a clear purpose and contain the necessary logic or be removed if not needed.

Design Smells in CategoryController.php

1. Inappropriate Intimacy:

The controller directly accesses the Category model for data manipulation. Consider using a service or repository to handle this logic.

SOLIDs

SOLID is an acronym representing five design principles intended to make software designs more understandable, flexible, and maintainable.

The RegisterController

- **Single Responsibility Principle (SRP):** The RegisterController now focuses only on handling HTTP requests and responses. The user creation and update logic is moved to the UserService.
- **Open/Closed Principle (OCP):** The UserService can be extended with new functionalities without modifying existing methods.
- **Dependency Inversion Principle (DIP):** The controller depends on the UserService interface, making it easier to change the implementation without modifying the controller.

The UserService

- **Single Responsibility Principle (SRP):** The UserService handles all business logic related to user management.
- **Open/Closed Principle (OCP):** The service can be extended with new functionalities (e.g., sending confirmation emails) without modifying existing methods.
- **Dependency Inversion Principle (DIP):** The controller depends on the UserService interface, which can be swapped out with a different implementation if needed.

UML Diagrams

Unified Modeling Language (UML) diagrams are graphical representations used in software engineering to visualize, specify, construct, and document software-intensive systems. UML provides a standard way to model software systems, making it easier for developers, analysts, and stakeholders to communicate ideas, understand system behavior, and design software effectively.

There are several types of UML diagrams, each serving a specific purpose:

Use Case Diagrams: These describe the interactions between users (actors) and the system, showing the various ways the system can be used.

Class Diagrams: These illustrate the structure of the system by depicting classes, their attributes, methods, and relationships between them.

Sequence Diagrams: These depict interactions between objects in a sequential order, showing how messages are exchanged over time.

Activity Diagrams: These represent workflows or processes within the system, showing the flow of control from one activity to another.

State Diagrams: Also known as state machines, these model the states of objects and transitions between states based on events.

Component Diagrams: These show the physical components of the system and their relationships.

Deployment Diagrams: These depict the physical deployment of artifacts on nodes, such as hardware or software components.

UML diagrams are used in software development for several reasons:

Visualization: They provide a clear and concise way to represent complex systems visually, making it easier for stakeholders to understand and discuss system requirements and designs.

Specification: UML diagrams serve as blueprints for software systems, documenting the system's structure, behavior, and interactions.

Design: They aid in the design process by helping developers to plan, organize, and refine the architecture of the software system.

Communication: UML diagrams facilitate communication among team members, allowing them to share ideas, clarify requirements, and collaborate effectively.

Analysis: They support analysis activities, such as identifying potential design flaws, performance bottlenecks, or inconsistencies in requirements.

Documentation: UML diagrams serve as valuable documentation for future reference, providing insights into the system's design and rationale behind design decisions.

Overall, UML diagrams play a crucial role in software development by improving communication, fostering collaboration, and aiding in the design and documentation of software systems.

Use Case Diagram

Find a Parent: The user searches for a parent using the search bar. This action can only be performed after at least one parent has been created.

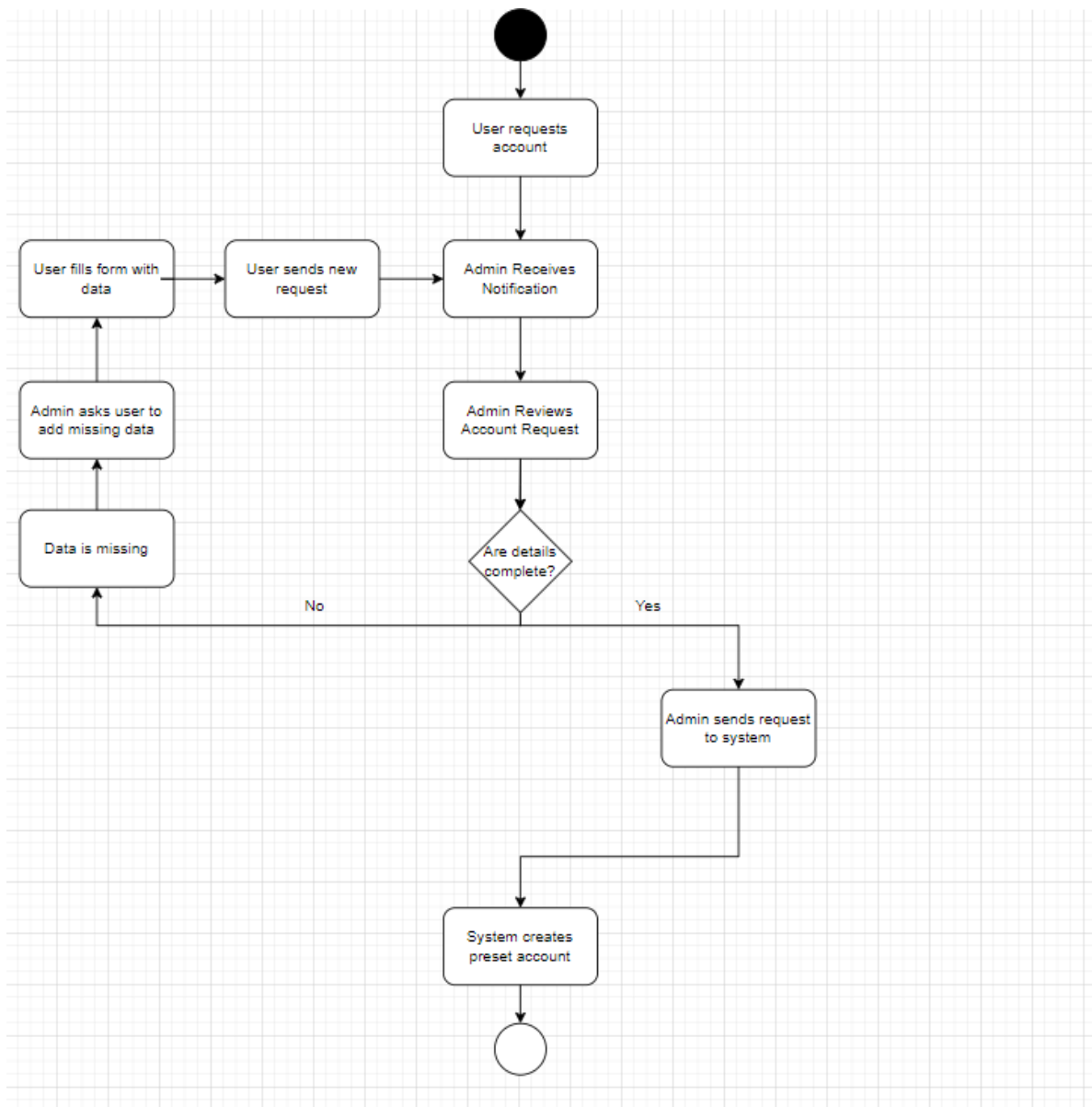
Add Parent: The user fills out the form to add a new parent, which includes selecting or entering:

- Category
- Partner
- Sales support
- Sales administrator The system validates each of these entries before creating the parent. This action can only be performed if there is at least one category available.

Add Category: The user enters a category name and description, then submits the form to add a new category.



Activity Diagram



User Account Request

This activity diagram illustrates the process flow for handling user account requests. It details the steps from a user requesting an account to the system creating a preset account, including interactions with an admin for validation and data completeness checks.

User Requests Account:

The process begins when a user requests a new account. This is indicated by the initial black circle leading to the "User requests account" activity.

User Fills Form with Data:

The user fills out a form with the necessary data for the account request.

User Sends New Request:

After filling out the form, the user submits the new account request.

Admin Receives Notification:

The admin receives a notification of the new account request.

Admin Reviews Account Request:

The admin reviews the details provided in the account request.

Decision Point: Are Details Complete?:

The admin assesses whether all necessary details are complete:

No Path:

Data is Missing:

If details are incomplete, the admin notes that data is missing.

Admin Asks User to Add Missing Data:

The admin requests the user to provide the missing information.

User Fills Form with Data:

The user updates the form with the missing data and resends the request.

Yes Path:

Admin Sends Request to System:

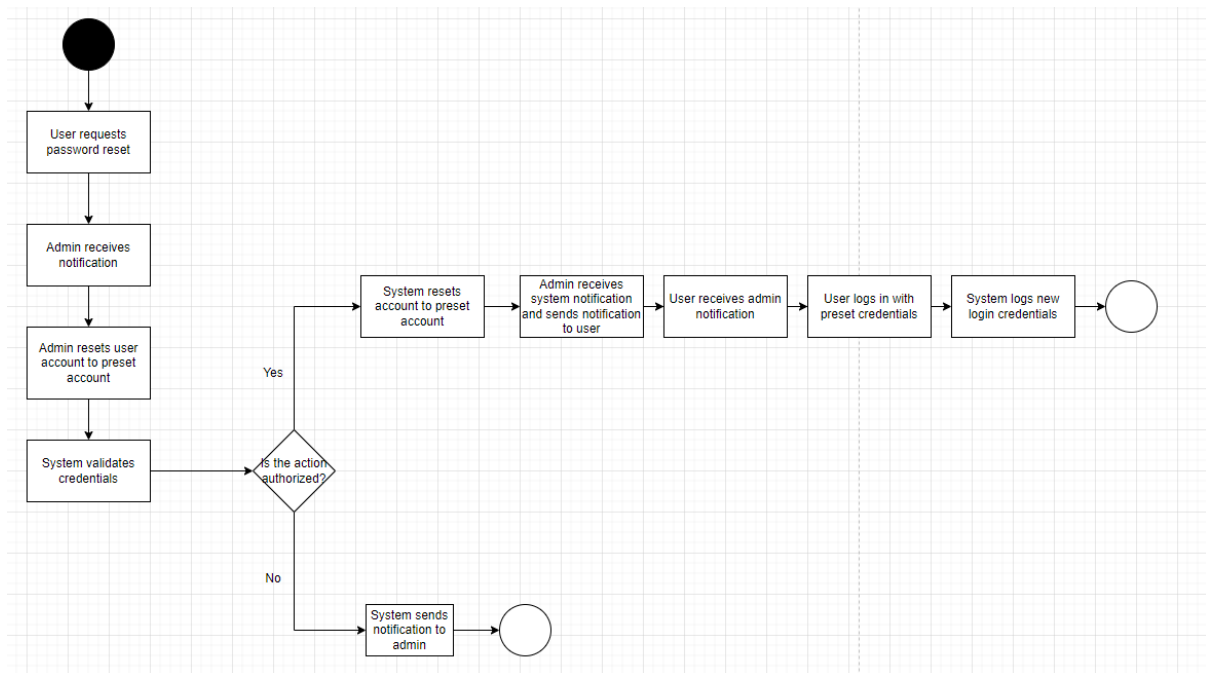
If the details are complete, the admin forwards the request to the system for processing.

System Creates Preset Account:

Upon receiving the complete request, the system creates a preset account.

End:

The process ends here, marked by the final circle, indicating the completion of the account creation process.



User Password Reset Activity Diagram

This diagram illustrates the process for handling user password resets. It details the steps from the user requesting a password reset to the admin sending the reset information to the system.

User requests password reset

The process begins when a user requests a password reset. This is indicated by the black circle that leads to the "User requests password reset" activity.

Admin receives notification

The admin receives a notification from the user who wants to reset their password.

Admin resets user account to preset account

The admin sends an account reset request to the system.

System validates credentials

The system checks if the admin is authorized to send account reset requests.

Decision Point: Is the action authorized?

The system assesses whether the admin's request is valid.

Yes path:

System resets account to preset account

The action from the admin was authorized, so the system resets the user's account to a preset state.

Admin receives system notification and sends notification to user

The system notifies the admin that the account has been reset. The admin then sends a notification to the user.

User receives admin notification

The user receives account credentials from the admin's message.

User logs in with preset credentials

The user logs in with the preset credentials and then changes the credentials to their preference.

System logs new login credentials

When the user logs in with the new credentials, the system records the new credentials. This marks the end of the activities, shown by the white circle.

No Path:**System sends notification to admin**

If the action is not authorized, the system sends the admin a notification indicating that the action failed.

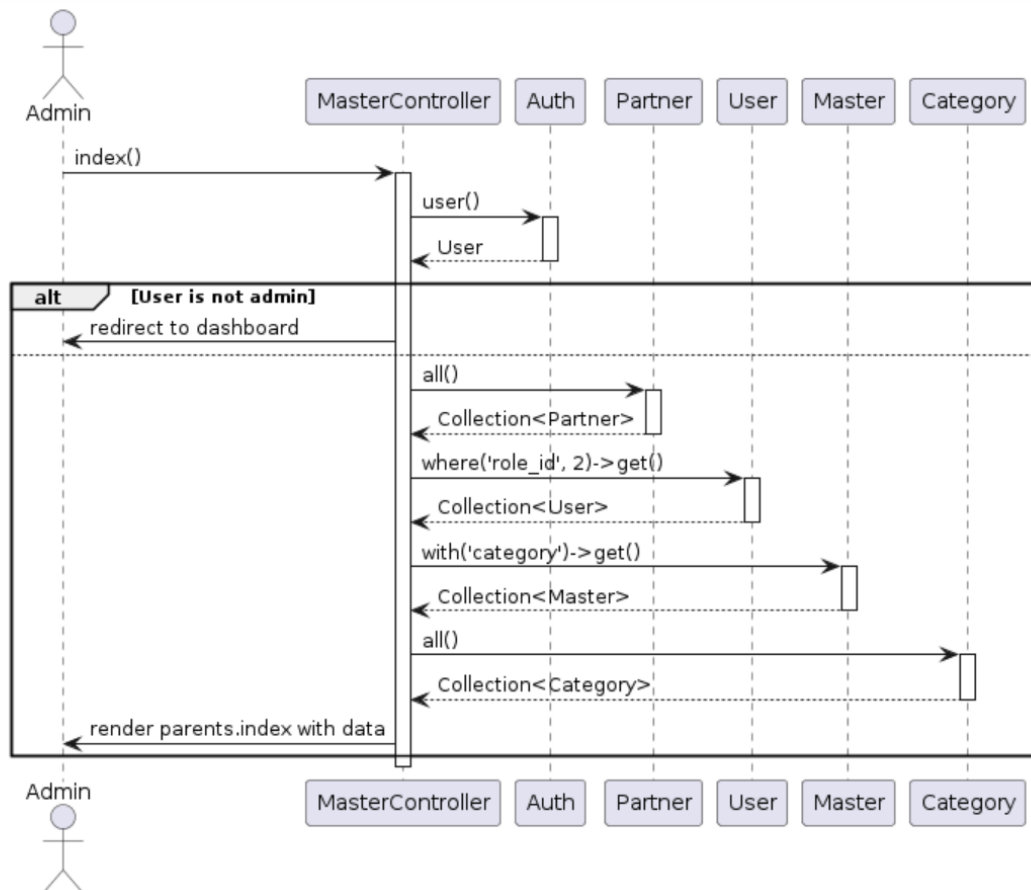
This is also the end of the activities, shown by the white circle.

Sequence Diagram

Sequence Diagram for index()

Description: Lists parents and related data.

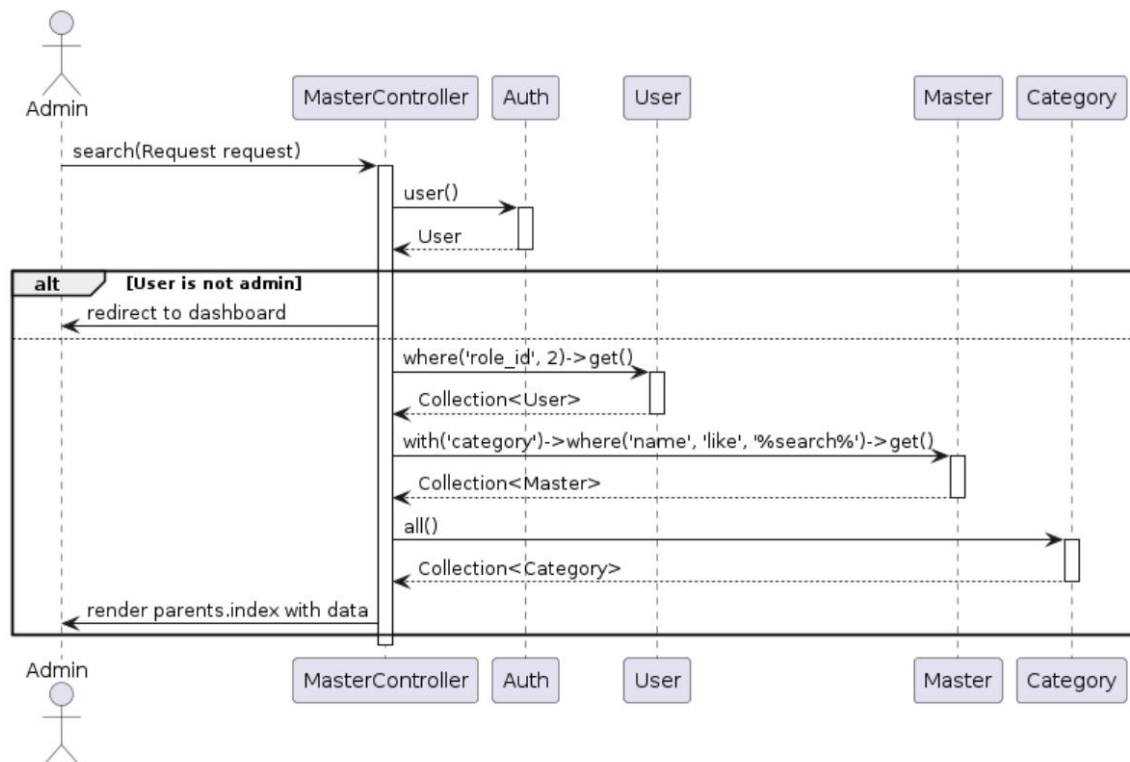
1. **Admin** calls the **index** method on the **MasterController**.
2. **MasterController** checks if the user is an admin.
3. If not, it redirects to the dashboard.
4. If the user is an admin:
 - Fetches all **Partner** records.
 - Fetches all **User** records with **role_id** 2.
 - Fetches all **Master** records with their associated **Category**.
 - Fetches all **Category** records.



Sequence Diagram for search

Description: Searches for parents based on criteria.

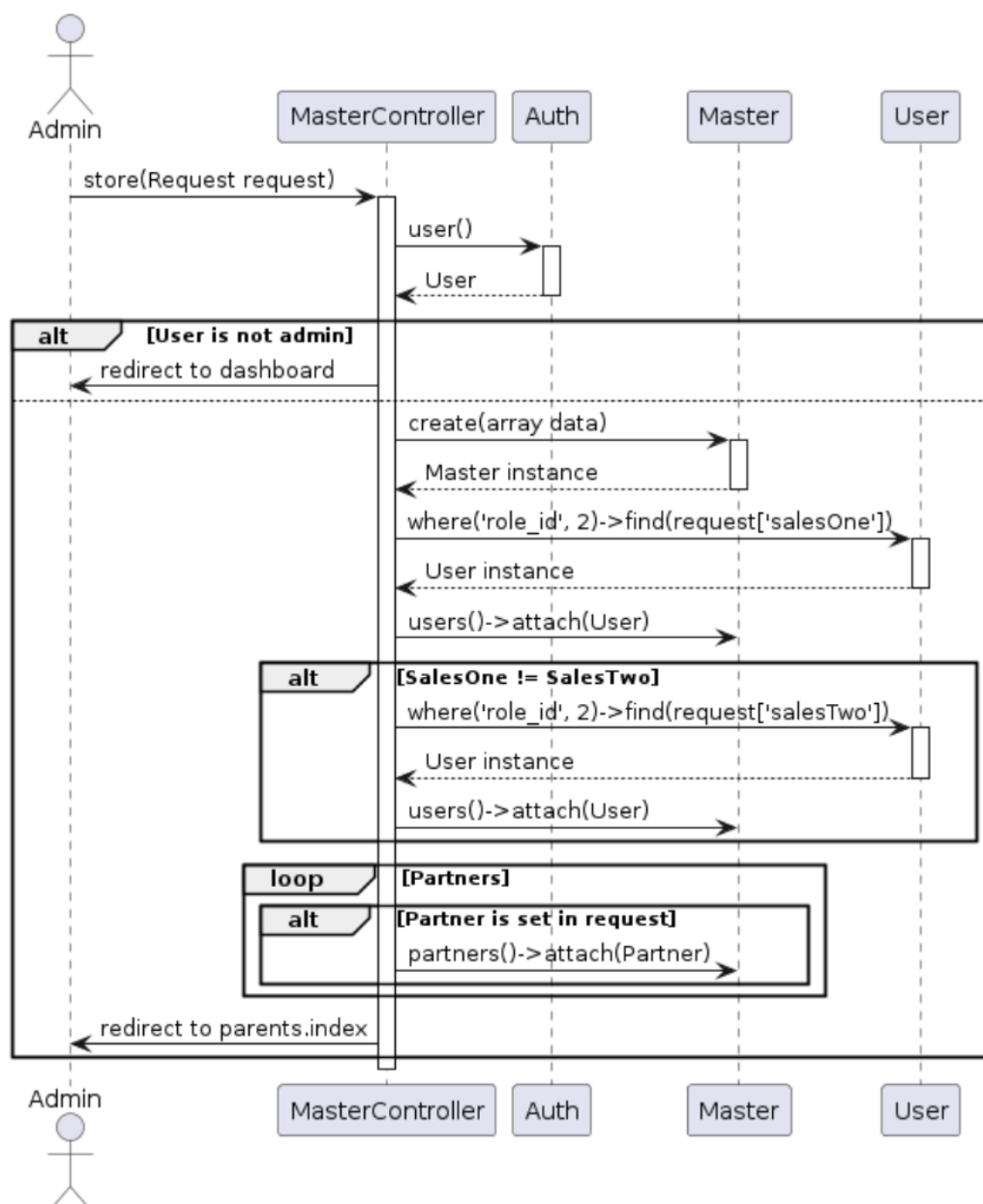
1. **Admin** calls the **search** method on the **MasterController** with a search request.
2. **MasterController** checks if the user is an admin.
3. If not, it redirects to the dashboard.
4. If the user is an admin:
 - Fetches all **User** records with **role_id** 2.
 - Fetches all **Master** records with the associated **Category** where the name matches the search criteria.
 - Fetches all **Category** records.



Sequence Diagram for store

Description: Stores a new parent.

1. **Admin** calls the **store** method on the **MasterController** with a request.
2. **MasterController** checks if the user is an admin.
3. If not, it redirects to the dashboard.
4. If the user is an admin:
 - Creates a new **Master** record.
 - Attaches the selected **User** records (sales agents) to the **Master**.
 - Attaches the selected **Partner** records to the **Master**.



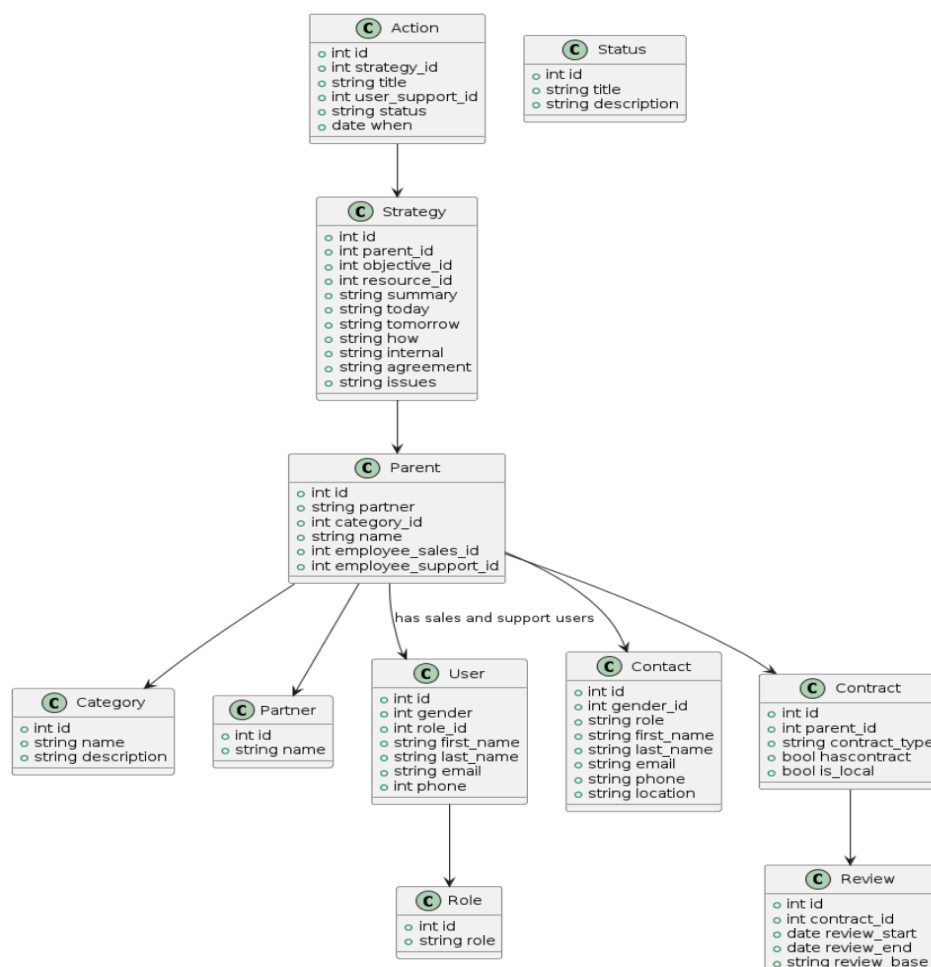
Class Diagram

This diagram outlines the structure of various entities in the system and their relationships, providing a high-level overview of how data is organized and interconnected within the system.

1. **Category:** Represents a classification or grouping. It has attributes such as ID, name, and description.
2. **Partner:** Represents entities that are associated with the business. Each Partner has attributes like ID and name.
3. **Parent:** Represents the parent entity in the system. It has attributes such as ID, name, a reference to a category, references to sales and support users, and references to employee IDs for sales and support. It appears to be some sort of organizational unit or entity that has relationships with other entities in the system.
4. **User:** Represents users of the system. It has attributes like ID, gender, role ID, first name, last name, email, and phone number.
5. **Role:** Represents roles or permissions within the system. It has attributes like ID and role name.
6. **Contact:** Represents contact information. It has attributes like ID, gender ID, role, first name, last name, email, phone number, and location.
7. **Contract:** Represents a contractual agreement. It has attributes like ID, parent ID, contract type, a boolean indicating if there's a contract, and a boolean indicating if it's a local contract.
8. **Review:** Represents reviews associated with contracts. It has attributes like ID, contract ID, review start date, review end date, and review base.
9. **Status:** Represents status information. It has attributes like ID, title, and description.
10. **Action:** Represents actions taken within the system. It has attributes like ID, strategy ID, title, user support ID, status, and date.
11. **Strategy:** Represents strategies within the system. It has attributes like ID, parent ID, objective ID, resource ID, summary, today's plan, tomorrow's plan, how-to, internal notes, agreement details, and issues.

The relationships:

- **Parent** has relationships with several entities:
 - Each Parent is associated with exactly one Category.
 - Each Parent is associated with one or more Partners.
 - Each Parent has one or more Users for sales and support roles.
 - Each Parent is associated with one or more Contacts.
 - Each Parent is associated with one or more Contracts.
- **User** is associated with a Role, indicating the role or permission level of the user.
- **Contract** is associated with a Review, indicating reviews associated with contracts.
- **Action** is associated with a Strategy, indicating actions associated with specific strategies.
- **Strategy** is associated with a Parent, indicating the parent entity to which the strategy belongs.

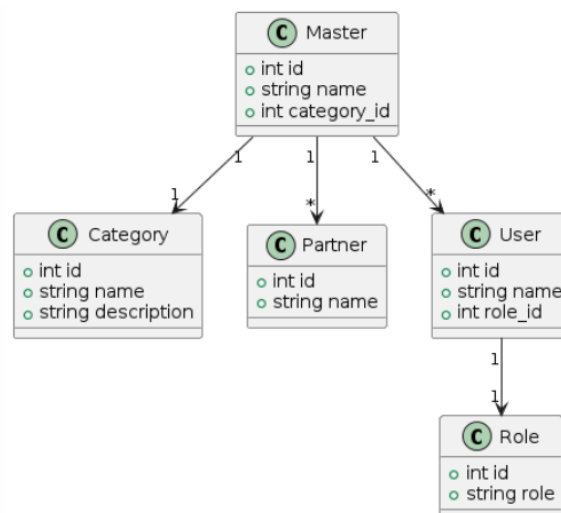


This class diagram represents a simple data model for a system that manages Masters, Categories, Partners, Users, and Roles.

1. **Master:** Represents a main entity in the system. It has attributes such as ID, name, and a reference to a category. This implies that each Master belongs to exactly one Category.
2. **Category:** Represents a classification or grouping. It has attributes like ID, name, and description.
3. **Partner:** Represents entities that are associated with a Master. Each Master can have multiple Partners.
4. **User:** Represents users of the system. It has attributes like ID, name, and a reference to a role. Each User belongs to exactly one Role.
5. **Role:** Represents roles or permissions within the system. It has attributes like ID and role name.

The relationships:

- Each Master is associated with exactly one Category (1-to-1 relationship).
- Each Master can have multiple Partners (1-to-many relationship).
- Each Master can have multiple Users associated with it (1-to-many relationship).
- Each User belongs to exactly one Role (1-to-1 relationship).



This diagram gives a structured view of how controllers, models, and Laravel classes interact in the application, along with the relationships between various entities.

MasterController: This controller class seems to handle operations related to the Master model. It contains methods for typical CRUD (Create, Read, Update, Delete) operations:

- **index():** Displays a list of Masters.
- **show(int id):** Shows details of a specific Master identified by its ID.
- **search(Request request):** Searches for Masters based on certain criteria.
- **store(Request request):** Stores a new Master.
- **edit(Request request):** Displays a form to edit a specific Master.
- **update(Request request):** Updates an existing Master.

Models:

- **Category:** Represents a category entity with attributes such as ID, name, and description.
- **Master:** Represents the main entity with attributes like ID, name, and a reference to a category.
- **Partner:** Represents entities associated with Masters. It seems to be a many-to-many relationship as indicated by the association arrow.
- **Role:** Represents roles or permissions with attributes like ID and role name.
- **User:** Represents users with attributes like ID, name, last name, and role ID.

Http:

- **Request:** Represents an HTTP request. It has methods like **all()** to retrieve all request parameters and **input(string key)** to retrieve a specific input parameter.

Facades:

- **Auth:** Facade class for authentication-related operations. It provides a method **user()** which returns the authenticated User object.

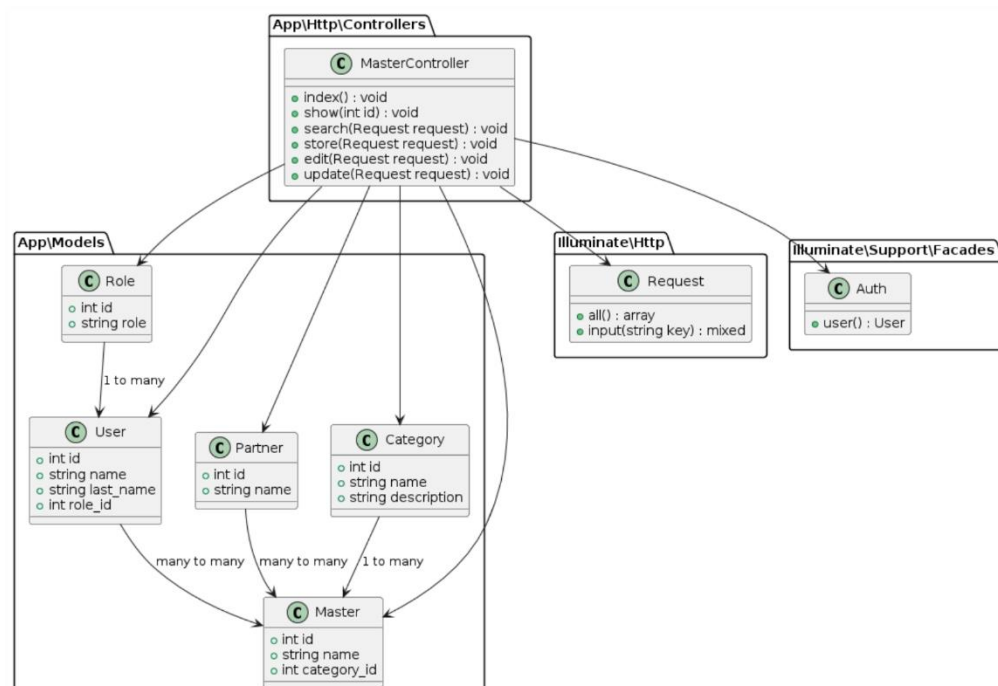
The relationships:

MasterController is associated with various model classes and Laravel classes:

- It interacts with the **Category**, **Master**, **Partner**, **Role**, and **User** models, suggesting that it manages operations related to these entities.
- It uses the **Request** class for handling HTTP requests.
- It utilizes the **Auth** class, presumably for handling authentication and authorization.

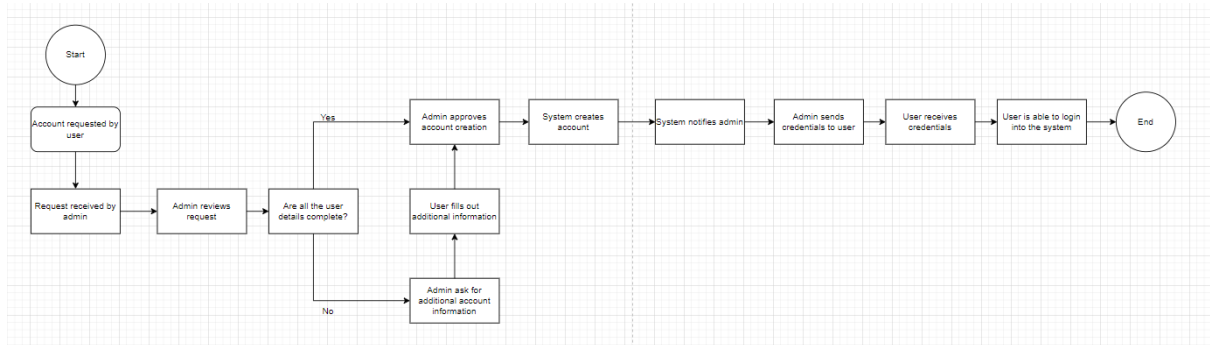
There are implied relationships between models:

- Each **Master** belongs to a **Category** (1-to-many).
- There's a many-to-many relationship between **Partner** and **Master**.
- Each **Role** has many **Users** (1-to-many).
- There's a many-to-many relationship between **User** and **Master**.



Flow Diagram

User Account Request



Start: Initiates the process.

User Fills Out Form: User provides necessary information for the account request.

User Submits Request: User submits the filled-out form.

Admin Receives Notification: Admin gets notified of a new account request.

Admin Reviews Request: Admin checks the details provided by the user.

Are Details Complete?: Decision point to check if the details are complete.

No:

Admin Requests Additional Information: Admin asks the user to provide more information.

User Fills Out Form: Loop back to the user filling out the form with additional details.

Yes:

Admin Approves Account Creation: Admin approves the account request.

System Validates Information: The system verifies the provided information.

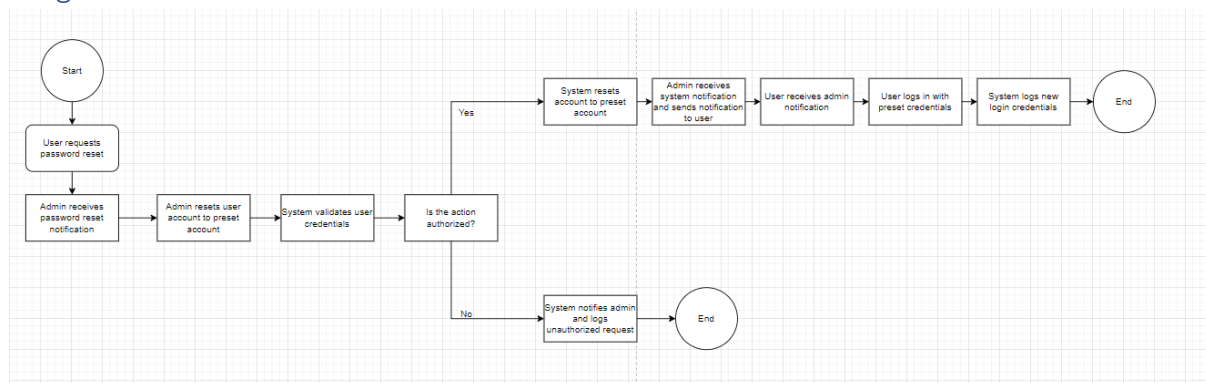
System Creates Account: The system creates the user account in the database.

System Sends Confirmation to User: User receives an email confirming account creation.

System Notifies Admin: Admin gets notified that the account has been successfully created.

End: The process is complete.

Forget Password



Start: The process begins when the user initiates a password reset request.

User Requests Password Reset: The user clicks on the "Forgot Password" or similar option to start the process of resetting their password.

Admin Receives Password Reset Notification: The system notifies the admin that a password reset has been requested by the user.

Admin Resets User Account to Preset Account: The admin resets the user's account to a preset default state. This might involve setting a temporary password or restoring default settings.

System Validates User Credentials: The system validates the user's credentials to ensure the request is legitimate and authorized.

Is the Action Authorized?: This decision point checks whether the admin's action is authorized:

Yes:

System Resets Account to Preset Account: The system proceeds to reset the user's account to the preset state as specified by the admin.

Admin Receives System Notification and Sends Notification to User: The system notifies the admin that the reset was successful, and the admin (or system) sends a notification to the user informing them about the reset.

User Receives Admin Notification: The user receives the notification about the reset and the new credentials or instructions.

User Logs In with Preset Credentials: The user logs in using the new preset credentials provided.

System Logs New Login Credentials: The system logs the user's new login credentials for auditing and security purposes.

No:

System Notifies Admin and Logs Unauthorized Request: If the action is not authorized, the system logs the unauthorized attempt and notifies the admin of the security breach or error.

End: The process ends here if the action is unauthorized.

End: The process concludes once the account reset has been completed, either successfully or with an unauthorized attempt logged.

Scrum

Scrum Opzet

Wij hebben de hele planning uitgewerkt met Azure DevOps/Github Projects zoals te zien is in de volgende kopjes.

En communicatie ging via Discord en Whatsapp.

Way of Working:

Scrum Rules

 Renas Khalil 4 okt 2023

Priority scaling 1 = Highest priority / 5 = Lowest priority

Effort = Fibonnachi scaling 0 / 1 / 2 / 3 / 5 / 8 / 13 etc 1 effort = 1 hour of work.

Business value = 1 / 100.

100 Meaning its important for the product / 1 meaning its not important.

Backlogbeheer:

Aan het begin van de dag gaan we in Discord en houden wij onze dagelijkse stand-up vergaderingen en sprint review waar wij bespreken wat we gaan behandelen van de backlog, en delen wij die groteren EPICS op in meerdere sprints die vervolgens in meerdere User Storys omgezet worden.

Sprint Oplevering/ Sprintplanning:2

De groep levert de tot zover werkende software op aan het einde van elke sprint.

En melden vervolgens ook welke problemen zij hebben opgelopen met de opgepakte user story's of welke ze allemaal afgemaakt hebben met succes.

Definition of Done:

Onze D.O.D. Bestaat uit:

Stap 1

Het begint uiteraard met het schrijven van de code.

Stap 2

En dan moet de code gecommenen worden.

Stap 3

Het testen van die code in de aparte brench in zichzelf.

Stap 4

Het dan samenvoegen met de Main brench en het geheel testen.

Stap 5

Vervolgens meld je bij de groep dat je klaar bent en ga je verder met het volgende item op de sprint.

Stap 6

Aan het einde van die dag gaan we in onze afsluit stand up en gaan we over alle veranderingen en progressie die er gemaakt is zover en bespreken wij de zowel geschreven code als comments zodat we alles samen dubbelchecken.

En het dan allemaal definitief goedkeuren en als alles goed is sluiten wij die dag af.

Maar als wij tegen een probleem zijn gelopen bijvoorbeeld met een push conflict zullen wij elkaar in een call in discord gelijk alles verhelpen zodat iedereen productief blijft en weer verder kan.

Unit Test

We have chosen to do unit testing during our production. For example, pressing non valid keys and testing the performance on different pcs.

Security

Ensuring the security of the Blue Box Tool is paramount to protecting sensitive data and maintaining user trust. Our security strategy involves a multi-layered approach to safeguard the application from potential threats and vulnerabilities. Below are the key security measures implemented:

1. Authentication and Authorization

- **Role-Based Access Control (RBAC):**

We implement role-based access control to ensure that users can only access the functionalities and data relevant to their roles. This minimizes the risk of unauthorized access to sensitive information.

Roles and permissions are defined and managed within the system, allowing administrators to easily assign and revoke access as needed.

- **Secure Login Mechanisms:**

User passwords are securely stored using a strong cryptographic hashing algorithm (bcrypt) with salted hashing to mitigate against rainbow table attacks.

We utilize token-based authentication for secure user sessions, ensuring that user credentials are not exposed during transmission.

2. Data Protection

- **Encryption:**

All sensitive data, including personal information and credentials, is encrypted both in transit and at rest using industry-standard encryption protocols (e.g., TLS/SSL for data in transit and AES-256 for data at rest).

Regular audits and updates of encryption standards are conducted to ensure continued compliance with best practices.

- **Data Validation and Sanitization:**

Input validation and sanitization are implemented at both client-side and server-side to prevent common web vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).

Server-side validation includes strict type checking, length restrictions, and the use of prepared statements for database queries.

Performance

Optimizing the performance of the Blue Box Tool is essential to ensure a seamless and responsive user experience. Our performance strategy encompasses various techniques and best practices to maximize efficiency, reduce latency, and enhance scalability. Below are the key performance measures implemented:

1. Efficient Data Handling

- **Database Optimization:**

Indexing: Strategic indexing is applied to frequently queried fields to speed up data retrieval operations.

Query Optimization: Complex queries are optimized to minimize execution time and resource consumption. This includes using joins efficiently and avoiding unnecessary data fetching.

Caching: Results of expensive queries are cached to reduce the load on the database and improve response times for repeated requests.

- **Data Partitioning:**

Horizontal and vertical partitioning techniques are used to distribute large datasets across multiple storage nodes, ensuring balanced load distribution and faster access times.

Sharding: The database is partitioned into smaller, more manageable pieces (shards) to improve performance and scalability.

2. Application Optimization

- **Code Efficiency:**

Code Profiling: Regular profiling of the application code helps identify performance bottlenecks, allowing developers to refactor inefficient code segments.

Asynchronous Processing: Time-consuming operations, such as file uploads and data processing, are handled asynchronously to prevent blocking the main application flow.

- **Resource Management:**

Memory Management: Efficient memory usage is ensured by identifying and resolving memory leaks, optimizing data structures, and using garbage collection effectively.

Connection Pooling: Database connections are pooled and reused to reduce the overhead of establishing new connections, thus improving performance for high-traffic applications.

3. Front-End Optimization

- **Minification and Compression:**

CSS and JavaScript files are minified and compressed to reduce their size, leading to faster load times and reduced bandwidth usage.

Image Optimization: Images are compressed and served in modern formats like WebP to minimize load times without compromising quality.

- **Lazy Loading:**

Implementing lazy loading for images, videos, and other media ensures that these elements are loaded only when they are needed, reducing initial load times and improving perceived performance.

Installation

Setup Instructions

Follow the steps below to set up the project on your local machine:

Step 1: Clone the Repository

Clone the repository from GitHub:

```
git clone [repository-url]
```

Step 2: Install PHP Dependencies

Navigate to the project directory and install PHP dependencies using Composer:

```
composer install
```

Step 3: Install JavaScript Dependencies

Install the necessary JavaScript dependencies using npm:

```
npm install
```

Step 4: Configure Environment Variables

1. Locate the `.env` file in the project root directory.
2. Open the `.env` file in a text editor.
3. Change the `DB_DATABASE` value to your desired database name. For example:

```
DB_DATABASE=your_database_name
```

Step 5: Create the Database

Open your console and create a new database using the following command:

```
mysql -u [username] -p -e "CREATE DATABASE [database_name];"
```

Replace [username] with your MySQL username and [database_name] with the name you specified in the .env file.

Step 6: Run Database Migrations

Run the database migrations to set up the necessary tables:

```
php artisan migrate
```

Step 7: Compile Assets

Compile the front-end assets using npm:

```
npm run dev
```

Step 8: Start the Development Server

Start the Laravel development server:

```
php artisan serve
```

Implementation

Continuous Integration/Continuous Deployment (CI/CD):

- A CI/CD pipeline is set up to automate the build, test, and deployment processes. Tools like Jenkins, Travis CI, or GitHub Actions are used to facilitate this automation.
- Each code commit triggers an automated build and test process. Only code that passes all tests is deployed to staging or production environments, ensuring high reliability and reducing the risk of bugs.

Environment Configuration:

- Environment-specific configurations are managed using environment variables. This approach ensures that sensitive information, such as API keys and database credentials, is not hardcoded into the application.
- Configuration files are managed separately for development, staging, and production environments, allowing for smooth transitions and consistent behavior across different stages of the deployment pipeline.

Upkeep

Regular Maintenance

Scheduled Maintenance:

- Regular maintenance windows are scheduled to perform necessary updates and checks without disrupting user activities. These windows are communicated to users in advance.
- Maintenance tasks include database optimization, server health checks, and application updates.

Codebase Refactoring:

- Continuous code refactoring is conducted to improve code quality, maintainability, and performance. This involves removing deprecated code, optimizing algorithms, and adhering to coding standards.
- Technical debt is regularly assessed and addressed to prevent long-term issues.

Updates and Upgrades

Software Updates:

- The application and its dependencies are kept up-to-date with the latest versions to benefit from security patches, performance improvements, and new features.
- Automated tools are used to monitor and apply updates to libraries, frameworks, and tools.

Feature Enhancements:

- New features and improvements are regularly introduced based on user feedback and market trends. Feature requests are prioritized and implemented following a structured development process.
- Beta testing is conducted for major updates to ensure stability and gather user feedback before a full release.

Monitoring and Alerts

System Monitoring:

- Continuous monitoring of the application and its infrastructure is implemented using tools like New Relic, Nagios, or Prometheus. Key metrics such as response times, error rates, and resource usage are tracked.
- Real-time monitoring dashboards provide insights into system health and performance.

Alerts and Notifications:

- Automated alerts are configured to notify the support team of any anomalies or performance issues. These alerts help in early detection and resolution of potential problems.
- Alerts are set for critical events such as server downtime, high error rates, and security breaches.

Backup and Recovery

Regular Backups:

- Regular backups of the database and application data are performed to prevent data loss. Backups are stored securely and tested periodically to ensure integrity.
- Incremental backups are used to minimize storage requirements and ensure faster recovery times.

Disaster Recovery Plan:

- A comprehensive disaster recovery plan is in place to handle catastrophic events such as data breaches, hardware failures, or natural disasters. This plan includes procedures for data restoration, system recovery, and communication protocols.
- Regular disaster recovery drills are conducted to ensure preparedness and effectiveness of the recovery plan.

User Support and Documentation

User Support:

- A dedicated support team is available to assist users with any issues or questions. Support channels include email, chat, and a helpdesk ticketing system.
- Common issues and solutions are documented in a knowledge base, providing users with self-service support options.

Documentation Updates:

- User manuals, technical documentation, and help guides are regularly updated to reflect changes and new features in the application. Documentation is maintained in a version-controlled repository to track updates.
- FAQs and troubleshooting guides are expanded based on user feedback and support queries.

Future Developments

User Experience Enhancements

User Interface (UI) Overhaul:

Redesigning the user interface to improve aesthetics, usability, and accessibility. This includes adopting modern design principles, improving navigation, and ensuring a consistent user experience across all devices.

Conducting user experience (UX) research and testing to gather feedback and identify areas for improvement.

Personalization and Customization:

Introducing personalization features to tailor the user experience based on individual preferences and usage patterns. This includes customizable dashboards, themes, and personalized recommendations.

Development of a widget-based interface, allowing users to add, remove, and arrange components to create a personalized workspace.

Artificial Intelligence and Machine Learning

AI-Powered Features:

Integration of artificial intelligence (AI) and machine learning (ML) capabilities to enhance the tool's functionality. This includes developing AI-powered recommendations, automated workflows, and intelligent data analysis.

Implementation of natural language processing (NLP) for improved search functionality and conversational interfaces, such as chatbots.