Retrofitting towards climate neutrality

D5.2 Technical Requirement & Development Guide

Program: HORIZON EUROPE

Grant agreement number: 101096522

Project acronym: Green Marine

Project title: Retrofitting towards climate neutrality

Prepared by: PDM

Date: 29/02/2024

Report version: v1.0

Funded by the
European Union

Funding acknowledgement
Funded by the European Union; funding from the European Union's Horizon Europe research and innovation program under grant agreement No. 101096522

UK participation in Green Marine project is co-funded by Innovate UK funding scheme

Disclaimer: The views expressed in this deliverable are those of the authors and are disclosed in a non-reliance basis and for information purposes only. The authors, the **Green Marine** consortium and partner employees and directors do not express or implied representation and/or warranty as to the accuracy and completion of the information set out in this report (including in any opinion, estimate, forecast and/or projection disclosed herein) and shall have no liability to the readers for any reliance placed on this report. The readers will rely solely upon their own independent investigation and evaluation of the information provided herein. Finally, this deliverable is based in information, legislation and data available as at the date of its creation and the authors shall have no responsibility for updating the information contained in this report and for correcting any inaccuracies in it, which may become apparent. The content of this deliverable does not reflect the official opinion of the European Union.

HISTORY OF CHANGES		
Version	Publication Date	Changes
0.1	[20/11/2023]	First draft
1.0	[30/01/2024]	Discussion with CMMI
1.0	[30/01/2024]	Feedback from all partners
1.0	[09/02/2024]	Discussion with CMMI
1.0	[15/02/2024]	Discussion with CMMI
2.0	[23/02/2024]	Feedback from reviewers
2.0	[29/02/2024]	Submission to EC portal

DETAILS	
Grant Agreement No.	101096522
Project acronym	Green Marine
Project full title	Retrofitting towards climate neutrality
Dissemination level	Public
Due date of deliverable	M13
Actual submission date	M13
Deliverable name	D5.2 Technical Requirement & Development Guide
Type	Report
Status	Initial version
WP contributing to the deliverable	WP5
Author(s)	Nuno Pedrosa & Carlos Marques (PDM)
Other Contributor(s)	Ashok Kumar (CMMI) George Mallouppas (CMMI) Giovanni Mazzuto (UPM) Torbjørn Pettersen (SINTEF)
Reviewer(s)	Giovanni Mazzuto (UPM), Torbjørn Pettersen (SINTEF), Filippo Emanuele Ciarapica (UPM)
Keywords	Green Marine, Development, Specifications, Design Parameters, Functional Requirements, Coding Standards, System Architecture, Performance, Integration, Testing, Documentation, Prototyping,Version Control, Maintenance, Technical Constraints, Security, Agile, User Interface, Database, Backup

Green Marine

1 **Executive Summary** **8**

2 **Introduction** **9**

2.1 Purpose of the Guide 9
2.2 Scope and Objectives 10

3 **Technical Requirements** **10**

3.1 System Architecture 10
3.2 Hardware Requirements 11
3.3 Software Requirements 11

4 **Development Process** **12**

4.1 Methodology 12
4.1.1 Core Values 13
4.1.2 Key Principles 13
4.1.3 Common Agile Frameworks 13
4.1.4 Roles and Ceremonies 13
4.1.5 Benefits of Agile 13
4.1.6 Implementing Agile 13
4.2 Coding Standards 14
4.2.1 PSR-1 Basic Coding Standard 14
4.2.1.1 Overview 14
4.2.1.2 Files 15
PHP Tags 15
Character Encoding 15
Side Effects 15
4.2.1.3 Namespace and Class Names 16
4.2.1.4 Class Constants, Properties, and Methods 16
Constants 16
Properties 16
Methods 16
4.2.2 PSR-4 Autoloader 17
4.2.2.1 Specification 17
4.2.2.2 Examples 17
4.3 Testing Procedures 17
4.3.1 Unit Testing 18
4.3.2 Feature Testing 18
4.3.3 Best Practices 18
4.4 Deployment Process 18
4.4.1 Preparation for Deployment 18
4.4.2 Deployment Process 19
4.4.3 Server Configuration 19
4.4.4 Post-Deployment Checks 19

4.4.5 Rollback Plan ... 19

5 User Interface Design ... 20

5.1 User Experience (UX) Guidelines ... 20
5.1.1 Understand Your Users ... 20
5.1.2 Design with Consistency ... 20
5.1.3 Simplicity ... 20
5.1.4 Feedback and Response Time ... 20
5.1.5 Easy Navigation ... 20
5.1.6 Error Prevention and Handling ... 20
5.1.7 Usability Testing ... 21
5.1.8 Visual Hierarchy and Readability ... 21
5.2 Visual Design Principles ... 21
5.3 Accessibility Requirements ... 21

6 Data Management ... 22

6.1 Database Design ... 22
6.1.1 Understanding Laravel with PostgreSQL ... 22
6.1.2 Design Principles ... 22
6.1.3 Best Practices ... 22
6.2 Data Security Measures ... 23
6.2.1 Authentication and Authorization ... 23
6.2.2 Encryption ... 23
6.2.3 Network Security ... 23
6.2.4 Auditing and Monitoring ... 23
6.3 Data Backup and Recovery ... 23
6.3.1 Backup Techniques ... 24
6.3.2 Point-in-Time Recovery (PITR) ... 24
6.3.3 Best Practices for Backup and Recovery ... 24

7 Integration and APIs ... 24

8 Performance Optimization ... 25

8.1 Load Testing ... 25
8.2 Code Optimization ... 26
8.3 Scalability Considerations ... 26

9 Security Measures ... 27

9.1 Authentication and Authorization ... 27
9.2 Encryption Standards ... 28
9.3 Security Audits and Monitoring ... 29

10 Documentation Guidelines ... 30

10.1 Code Documentation ... 30
10.1.1 Commenting Code ... 30
10.1.2 Documenting Methods and Functions ... 30
10.1.3 Class Documentation ... 30

10.1.4 Versioning and Deprecation Notices 30
10.1.5 Readability and Consistency 31
10.1.6 External Documentation 31
10.1.7 Tools and Integrations 31
10.2 User Manuals 31
10.2.1 Understanding the Audience 31
10.2.2 Structure and Content 31
10.2.3 Visual Aids 32
10.3 Release Notes 32
10.3.1 Importance of Release Notes 32
10.3.2 Key Components 32

11 Conclusions 32

1 EXECUTIVE SUMMARY

This document is deliverable "D5.2 Technical Requirement & Development Guide" of the European Union project "Retrofitting towards climate neutrality" (herein referred to as "**Green Marine**"), with **grant agreement No. 101096522**.

The deliverable presents a comprehensive guide on the technical requirements and development strategies for deploying robust, scalable, and secure the web applications that support the software tools and simulators. It outlines the essential software components, hardware requirements, and best, required, practices in the database design, application development, and deployment strategies, with a focus on leveraging modern technologies such as Laravel, PostgreSQL, Docker, and Nginx.

Key Components and Technologies:

- **Database Management:** Implementation of PostgreSQL as the primary database system, offering advanced features, reliability, and compatibility with complex data structures.
- **Application Development:** Utilization of Laravel, a PHP framework known for its elegant syntax and rich ecosystem, facilitating rapid development and maintainability.

A significant focus is placed on performance optimization techniques, including load testing and code optimization strategies to ensure applications can handle expected user loads efficiently.

Security is addressed through encryption standards in Laravel, data security measures with PostgreSQL, and general best practices for securing web applications. Regular security audits and monitoring are recommended to identify and mitigate potential vulnerabilities.

The importance of thorough documentation is stressed, covering code documentation, user manuals, and release notes to aid developers and end-users.

2 INTRODUCTION

The main objective of **Green Marine** is to significantly accelerate climate neutrality of waterborne transport through retrofitting existing fleets with cost and emission control solutions. To support decision makers retrofitting protocols and a software tool catalogue that gathers knowledge will be developed and validated. We will demonstrate these tools and the innovative solutions aimed at carbon capture mineralization, which also aids in deacidifying our seas; energy savings for HVAC systems through air-reuse; carbon and water as a side product capture with membranes, and the use of excess engine heat to produce a syngas to save on fuel consumption. An ultra-sound technology will be tailored to suit vessels allowing air-reuse saving energy for HVAC systems and operated as pre-treatment enhancing a membrane carbon capture process. The Ca/Mg – alkali solvent capture process is capable of removing 75% of the CO_2 from flue gases. All solutions will be theoretically evaluated before demonstration on a land-based engine followed by the selection of the most suitable solution for a demonstration on a waterborne vessel. The (land-based) demonstrations will be representative for the operation of a majority of vessel engines in use currently. By developing retrofitting protocols, simulations of the solutions, data generated at the demonstrations a software catalogue tool will be developed. Through engagement activities this tool will gain more users and more knowledge, its value and effectiveness will increase for all users. The project aims to bring the different solutions to TRL 8. The demonstrations, the software tool catalogue, and the dissemination and exploitation activities ensure that project results will be replicated globally. The consortium consists of 10 partners from 7 countries with 4 research institutes, 1 shipping company, which will host a demo as end user and 5 SMEs.

The objectives of WP5 – Software tool catalogue for GHG-emission reduction solutions are to integrate data and modelling sources and tools into the software tool catalogue to develop and implement a modelling software tool suitable for all stakeholders: 1) allowing for federated learning; 2) create value chains; 3) to support a bottom-up characterization and simulate the cost-benefits of alternative options for GHG-emission avoidance, for a wide range of ship types/classes 4) to be a decision support tool for industry and communities.

- Develop open APIs, security, privacy features, cloud etc. and app tools for the catalogue tool and validate them.
- To collaborate with the ship owners in developing their tool so as learnings can be exchanged.

2.1 PURPOSE OF THE GUIDE

Task 5.2 deals with the engineering design solutions for GHG emission reduction solutions. There are two sub tasks as mentioned below.

Subtask 5.2.1 Air circulation - Utilizing existing CFD HVAC modelling tools to aid in determining what happens to air circulation. Engineer positive pressurised cabins, with exhaust air being disinfecting before leaving the cabins. While not all cabins (staff and guests) could be arranged this way, a number of cabins capable of producing such a positive pressure are considered. Optimization in recirculation of air is needed using special sensors, in order to limit energy consumption, the ship's emissions. The air should be sterilized with SepaRaptor and UV (or other solutions like electrostatic, chemicals etc.). Also, investigated is its implementation with air recirculation and fresh air or oxygen addition through membrane separation. This solution calls for a marinized system, that needs to be positioned inside the

existing vessel seamlessly, therefore a study of air quality, energy and air flows must be simulated before implementing it on the vessel.

Subtask 5.2.2 **Engines and their flue gas streams** - Similar to above, and less intensive as it is straight forward (engine to stack).

2.2 Scope and Objectives

This document is intended to serve as a comprehensive resource for the development team, project managers, and stakeholders involved in the planning, development, and deployment phases of the software project. This document outlines the technical specifications, system architecture, development methodologies and coding standards in use in the project.
The primary objectives are to:

1. **Ensure Clarity and Consistency:** Provide a clear and consistent framework for the development team to follow, reducing ambiguity and ensuring that all team members have a common understanding of the project's technical requirements and objectives.
2. **Facilitate Effective Planning:** Aid in the planning and estimation process, enabling efficient allocation of resources and timelines.
3. **Promote Quality and Performance:** Establish standards and practices that promote the development of high-quality, high-performance software that meets the needs of the users and stakeholders.
4. **Support Scalability and Flexibility:** Outline a system architecture and development approach that supports scalability and flexibility, allowing for future enhancements and integration with other systems.
5. **Enhance Collaboration and Communication:** Serve as a reference point for the development team, project managers, and stakeholders, enhancing collaboration and communication throughout the project lifecycle.
6. **Mitigate Risks:** Identify potential technical challenges and risks early in the project, allowing for the implementation of mitigation strategies to avoid delays and cost overruns.

3 Technical Requirements

3.1 System Architecture

The base platform encompasses several complementary and stand-alone analysis modules that can be invoked and run in different combinations. The users may access their own dedicated area within the platform to run, store and share simulations and results.
The platform is structured in layers, having a common knowledge base (KB) that provides base data to all modules, through a core functionalities module that provides base services.

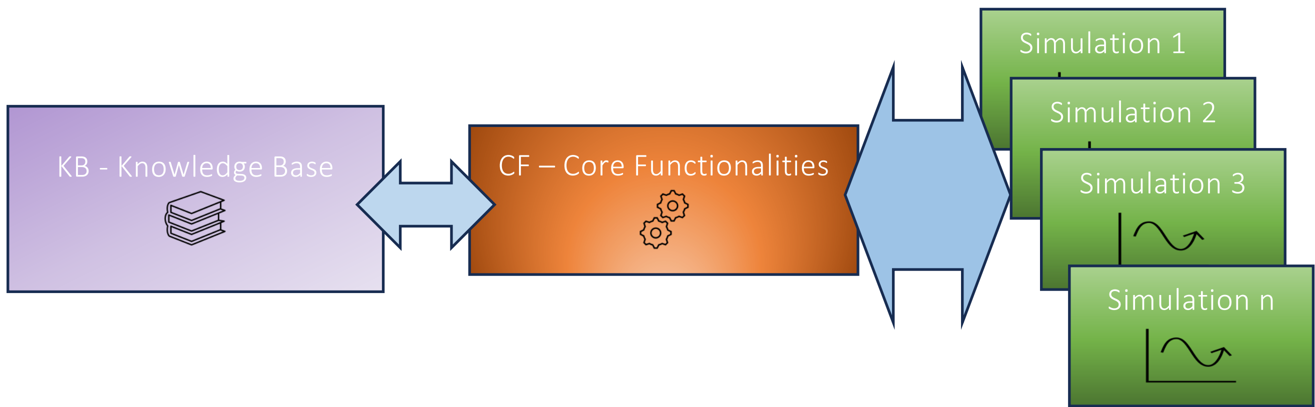

Figure 1- High level architecture

All the technical details of the equipment, the engine characteristics, fuels characteristics and fluids properties are retrieved from the KB, so that all common information is contained in a single source.

3.2 Hardware Requirements

The **COCO** (CAPE-OPEN to CAPE-OPEN) simulator requires WindowsXP or later and a suitable server.

The simulations being developed by UPM, require:

- Intel Core i9-11900KF or better
- 16+GB DDR4 RAM
- 1TB SSD
- Nvidia Quadro Graphics Card

For server deployment, the following specifications are required:

- Quad core CPU or better
- 8GB RAM
- 200GB storage
- Broadband connection

3.3 Software Requirements

The core functionalities module requires Python 3.8.0+. The packages needed to run the CF module are listed below:

-numpy=1.20.3
-pandas=1.3.4
-geopy=2.20
-pvlib-python=0.9.0
-urllib3=1.26.7
-beautifulsoup4=4.10.0
-matplotlib=3.4.3
-requests=2.26
-mpld3
-pydantic

The User Interface uses Laravel, requiring at least PHP 8.1.

Simulation modules will use Python 3.8.0+.

Up until now, the simulation models developed by UPM have been designed to require manageable computational resources in line with current processing capabilities. The

programming language Python has been adopted by UPM. This choice was motivated by the flexibility and power of Python in handling complex simulation algorithms.
With the evolution of technologies and the implementation of new solutions, more advanced machine learning and artificial intelligence algorithms may be necessary for future models. The integration of these technologies may require additional computational resources, as more complex algorithms and larger datasets could be involved.
For web deployment, the choice is Nginx server due to its high performance, stability, rich feature set, and low resource consumption. Nginx is an open-source web server that also serves as a reverse proxy, load balancer, and HTTP cache. It is known for its high scalability and ability to handle a large number of simultaneous connections with a minimal memory footprint, making it an ideal choice for serving static content, directing dynamic requests to application servers, and improving the overall efficiency and reliability of web applications.
Deployment is done through Docker. Docker is an open-source platform that simplifies the process of building, shipping, and running applications using containerization technology. By encapsulating applications and their dependencies into containers, Docker enables seamless deployment across different environments, reducing the "it works on my machine" problem and streamlining the development lifecycle.
PostgreSQL 16 is the database of choice. It is a powerful, open-source object-relational database system known for its strong reputation for system integrity, robust feature set, and support for advanced data types and functionality. Its versatility makes it suitable for a wide range of applications, from simple web applications to sophisticated data warehousing solutions with massive volumes of data.
For the membrane simulations, by SINTEF, the in-house models are implemented in the programming language Python as stand-alone functions which can be run on their own to simulate the performance of a single stage membrane module.
These Python models can be used in any cape open compliant process simulators using the commercially available Python Unit Operation model provided by AmsterCHEM[1]. This will allow for membrane modules to be included as unit operations in a steady state flowsheet simulation.
Within the Green Marine project SINTEF has implemented process flowsheet models of various membrane-based CO2 capture configurations in the cape-open compliant simulation environment COCO/COFE[2] which is free for download and free for use.

4 Development Process

Development in Green Marine starts from several different points, as each partner is focused on maturing their technology and simulation models start their life as replicas of their practical technology development.
In this context, the software development will follow the Agile methodology.

4.1 Methodology

The Agile methodology is a project management and product development approach that is iterative, incremental, and focuses on collaboration, customer feedback, and small, rapid releases. It is built on the foundation of flexible planning, early delivery, and continuous improvement, all with an eye toward being able to respond to change quickly and efficiently.

[1] https://www.amsterchem.com/pythonunitoperation.html
[2] https://www.cocosimulator.org/

Agile methodologies are particularly popular in software development but can be applied to various types of projects. Here are some key aspects of using Agile methodology:

4.1.1 Core Values

Agile methodology is defined by four core values as outlined in the Agile Manifesto:

1. **Individuals and interactions** over processes and tools.
2. **Working software** over comprehensive documentation.
3. **Customer collaboration** over contract negotiation.
4. **Responding to change** over following a plan.

4.1.2 Key Principles

Beyond the core values, Agile methodology is guided by 12 principles which include customer satisfaction through early and continuous delivery, welcoming changing requirements, frequent delivery of working software, close daily cooperation between business people and developers, building projects around motivated individuals, face-to-face conversation as the best form of communication, and maintaining a constant pace indefinitely.

4.1.3 Common Agile Frameworks

Several frameworks exist under the Agile umbrella, including:

- **Scrum**: Uses fixed-length iterations called sprints, with a focus on delivering a potentially shippable product increment at the end of each sprint.
- **Kanban**: Focuses on visualizing the workflow and limiting work in progress to improve flow and reduce cycle time.
- **Extreme Programming (XP)**: Emphasizes technical practices like test-driven development, continuous integration, and refactoring to improve software quality and responsiveness to changing customer requirements.

4.1.4 Roles and Ceremonies

In Scrum, which is one of the most widely used Agile frameworks, roles include the Product Owner, Scrum Master, and Development Team. Regular ceremonies such as Sprint Planning, Daily Stand-ups, Sprint Reviews, and Sprint Retrospectives facilitate communication, planning, teamwork, and reflection.

4.1.5 Benefits of Agile

- **Flexibility**: Agile allows for change and adapts to stakeholders' feedback throughout the development process.
- **Transparency**: Regular check-ins and demos with stakeholders keep everyone informed and ensure alignment with business goals.
- **Risk Management**: Frequent iterations expose issues early, reducing the risks associated with development.
- **Customer Satisfaction**: By involving the customer in the development process, the final product is more likely to meet their needs.
- **Employee Satisfaction**: Agile often leads to higher team morale and engagement due to its emphasis on autonomy, mastery, and purpose.

4.1.6 Implementing Agile

When implementing Agile methodology, it's important to:

- Ensure that the entire team, including stakeholders, understands the Agile mindset and principles.
- Start with a simple framework like Scrum or Kanban before adapting and customizing the process to fit your organization's unique needs.

- Use tools like digital boards and project management software to track progress and facilitate communication.
- Embrace the iterative nature of Agile, and be prepared to learn and adapt as you go.

Agile is more than just a set of practices; it's a mindset that encourages teams to work efficiently, deliver value quickly, and adapt to change smoothly. Adopting Agile requires commitment to continuous improvement and a willingness to embrace change, both of which are crucial in today's fast-paced business environment.

4.2 Coding Standards

The main application uses the Laravel Framework[3]. Laravel is a PHP web application framework with expressive, elegant syntax, that strives to follow the **PSR-3** and **PSR-1** coding standards, with the addition of the following conventions:

- a class namespace declaration must be on the same line as `<?php`.
- the class opening bracket "{" must be in the same line as the class name.
- functions and control structures must use ***Allman*** style braces, meaning that the brace associated with a control statement is placed on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.
- indent with tabs, align with spaces.

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in RFC 2119[4].

4.2.1 PSR-1 Basic Coding Standard

The PSR-1 basic coding standard comprises what should be considered the standard coding elements required to ensure a high level of technical interoperability between shared PHP code.

4.2.1.1 Overview

- Files MUST use only `<?php` and `<?=` tags.
- Files MUST use only UTF-8 without BOM for PHP code.
- Files SHOULD *either* declare symbols (classes, functions, constants, etc.) *or* cause side-effects (e.g. generate output, change **.ini** settings, etc.) but SHOULD NOT do both.
- Namespaces and classes MUST follow an "autoloading" PSR: [PSR-4[5]].
- Class names MUST be declared in `StudlyCaps`.
- Class constants MUST be declared in all upper case with underscore separators.
- Method names MUST be declared in `camelCase`.

[3] https://laravel.com
[4] https://www.ietf.org/rfc/rfc2119.txt
[5] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md

4.2.1.2 Files

PHP Tags

PHP code MUST use the long `<?php ?>` tags or the short-echo `<?= ?>` tags; it MUST NOT use the other tag variations.

Character Encoding

PHP code MUST use only UTF-8 without BOM.

Side Effects

A file SHOULD declare new symbols (classes, functions, constants, etc.) and cause no other side effects, or it SHOULD execute logic with side effects, but SHOULD NOT do both.
The phrase "side effects" means execution of logic not directly related to declaring classes, functions, constants, etc., *merely from including the file*.
"Side effects" include but are not limited to: generating output, explicit use of **`require`** or **`include`**, connecting to external services, modifying ini settings, emitting errors or exceptions, modifying global or static variables, reading from or writing to a file, and so on.
The following is an example of a file with both declarations and side effects; i.e, an example of what to avoid:

```
<?php
// side effect: change ini settings
ini_set('error_reporting', E_ALL);

// side effect: loads a file
include "file.php";

// side effect: generates output
echo "<html>\n";

// declaration
function foo()
{
    // function body
}
```

The following example is of a file that contains declarations without side effects; i.e., an example of what to emulate:

```
<?php
// declaration
function foo()
{
    // function body
}

// conditional declaration is *not* a side effect
if (! function_exists('bar')) {
    function bar()
    {
        // function body
    }
}
```

4.2.1.3 Namespace and Class Names

Namespaces and classes MUST follow an "autoloading" PSR: [PSR-4].
This means each class is in a file by itself and is in a namespace of at least one level: a top-level vendor name.
Class names MUST be declared in **StudlyCaps**.
Code written for PHP 5.3 and after MUST use formal namespaces.
For example:

```
<?php
// PHP 5.3 and later:
namespace Vendor\Model;

class Foo
{
}
```

Code written for 5.2.x and before SHOULD use the pseudo-namespacing convention of Vendor_ prefixes on class names.

```
<?php
// PHP 5.2.x and earlier:
class Vendor_Model_Foo
{
}
```

4.2.1.4 Class Constants, Properties, and Methods

The term "class" refers to all classes, interfaces, and traits.

Constants

Class constants MUST be declared in all upper case with underscore separators. For example:

```
<?php
namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const DATE_APPROVED = '2012-06-01';
}
```

Properties

This guide intentionally avoids any recommendation regarding the use of **$StudlyCaps**, **$camelCase**, or **$under_score** property names.
Whatever naming convention is used SHOULD be applied consistently within a reasonable scope. That scope may be vendor-level, package-level, class-level, or method-level.

Methods

Method names MUST be declared in **camelCase()**.

4.2.2 PSR-4 Autoloader

This PSR describes a specification for autoloading classes from file paths. It is fully interoperable, and can be used in addition to any other autoloading specification, including [PSR-0](PSR-0). This PSR also describes where to place files that will be autoloaded according to the specification.

4.2.2.1 Specification

1. The term " " refers to classes, interfaces, traits, and other similar structures.
2. A fully qualified class name has the following form:
   \<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>
   i. The fully qualified class name MUST have a top-level namespace name, also known as a "vendor namespace".
   ii. The fully qualified class name MAY have one or more sub-namespace names.
   iii. The fully qualified class name MUST have a terminating class name.
   iv. Underscores have no special meaning in any portion of the fully qualified class name.
   v. Alphabetic characters in the fully qualified class name MAY be any combination of lower case and upper case.
   vi. All class names MUST be referenced in a case-sensitive fashion.
3. When loading a file that corresponds to a fully qualified class name ...
   i. A contiguous series of one or more leading namespace and sub-namespace names, not including the leading namespace separator, in the fully qualified class name (a "namespace prefix") corresponds to at least one "base directory".
   ii. The contiguous sub-namespace names after the "namespace prefix" correspond to a subdirectory within a "base directory", in which the namespace separators represent directory separators. The subdirectory name MUST match the case of the sub-namespace names.
   iii. The terminating class name corresponds to a file name ending in .php. The file name MUST match the case of the terminating class name.
4. Autoloader implementations MUST NOT throw exceptions, MUST NOT raise errors of any level, and SHOULD NOT return a value.

4.2.2.2 Examples

The table below shows the corresponding file path for a given fully qualified class name, namespace prefix, and base directory.

Fully Qualified Class Name	Namespace Prefix	Base Directory	Resulting File Path
\Acme\Log\Writer\File_Writer	Acme\Log\Writer	./acme-log-writer/lib/	./acme-log-writer/lib/File_Writer.php
\Aura\Web\Response\Status	Aura\Web	/path/to/aura-web/src/	/path/to/aura-web/src/Response/Status.php
\Symfony\Core\Request	Symfony\Core	./vendor/Symfony/Core/	./vendor/Symfony/Core/Request.php
\Zend\Acl	Zend	/usr/includes/Zend/	/usr/includes/Zend/Acl.php

4.3 Testing Procedures

This section outlines the key testing procedures and best practices within the Laravel framework, ensuring high-quality software development through comprehensive test coverage.

4.3.1 Unit Testing

Unit tests focus on testing small, isolated parts of the application, typically individual methods or functions. Laravel is built with PHPUnit, a powerful and versatile testing framework for PHP, making it straightforward to write and run unit tests.

- **Writing Tests:** Unit tests in Laravel are stored in the **tests/Unit** directory. To create a new test case, use the Artisan command **php artisan make:test TestName --unit**.
- **Test Methods:** Each test method should be prefixed with **test**, such as **testUserCreation()**. Laravel also supports the **@test** annotation in docblocks to denote test methods.
- **Assertions:** Utilize PHPUnit's wide range of assertions to verify that the application behaves as expected. Laravel enhances PHPUnit with additional assertions specific to Laravel features, such as database and authentication assertions.

4.3.2 Feature Testing

Feature tests examine the application's response to HTTP requests, covering a broader scope than unit tests. These tests interact with the application's routes, controllers, middleware, and more, providing a way to test the application's overall behavior.

- **Writing Tests:** Feature tests are typically stored in the **tests/Feature** directory. Create a feature test using the Artisan command **php artisan make:test TestName**.
- **Making Requests:** Using methods like **get**, **post**, **put**, **delete**, etc., to simulate HTTP requests to your application. Then, assert the expected responses using assertions such as **assertStatus** or **assertSee**.
- **Database Testing:** Laravel provides features like database transactions and the database seeding to test database interactions without persisting data, ensuring tests do not interfere with each other.

4.3.3 Best Practices

- **Test-Driven Development (TDD):** Whenever possible a TDD approach is used, where tests are written before the actual code, ensuring every code change is tested.
- **Coverage:** Aim should be for high test coverage but prioritizing meaningful tests over merely increasing the coverage metric.
- **Refactoring:** Using tests as a safety net for refactoring, ensuring that improvements or optimizations do not break existing functionality.

Adhering to these testing procedures within Laravel not only ensures that your application functions as intended but also facilitates a development process that is more efficient, reliable, and maintainable. By incorporating unit, feature, and browser tests into your development workflow, and leveraging continuous integration for automated testing, you can significantly enhance the quality and robustness of your Laravel applications.

4.4 Deployment Process

This section outlines the key considerations, tools, and best practices for effectively deploying Laravel applications. The same procedures are used when deploying simulation modules or other server features.

4.4.1 Preparation for Deployment

- **Environment Configuration:** Ensure that the **.env** file, or equivalent, on the production server is correctly configured for the application's environment, including database connections, mail drivers, and any other external service configurations.

- **Dependency Management:** Use Composer to manage PHP dependencies. Ensure all dependencies are correctly installed on the production server by running **composer install --optimize-autoloader --no-dev** to install without development dependencies.
- **Asset Compilation:** When using Laravel Mix for asset compilation, run **npm run production** to compile and minify CSS and JavaScript assets.

4.4.2 Deployment Process

1. **Version Control:** Git is used to manage application's source code. Deployments can be triggered by pushing to a specific branch, such as **main**.
2. **Automated Deployment Tools:** Several tools can automate the steps of the deployment process, such as pulling the latest code, running migrations, and optimizing the application.
3. **Application Optimization:** Laravel provides several commands to optimize the application for production:
    - **php artisan config:cache** to cache the configuration files,
    - **php artisan route:cache** for routing optimization,
    - **php artisan view:cache** to compile and cache Blade templates.
4. **Database Migrations:** Use **php artisan migrate** to apply database migrations on the production server. Ensure this is done as part of the deployment process to keep the database schema up to date.

4.4.3 Server Configuration

- **Web Server:** Configure the web server (Nginx or Apache) to point to the Laravel application's **public** directory. Ensure that URL rewriting is correctly set up to route requests through the **index.php** file.
- **PHP Version:** Ensure the server is running a supported PHP version for the version of Laravel. If needed, check the Laravel documentation for the required PHP version.
- **Security:** Implement SSL/TLS to secure data transmission. Set appropriate file permissions for your application files and directories to protect them from unauthorized access.

4.4.4 Post-Deployment Checks

- **Environment Testing:** After deployment, thorough testing of the application in the production environment should be done, to ensure all functionalities work as expected and the environment is correctly configured. Focus on added features and basic usage flow.
- **Monitoring and Logging:** Monitoring and logging tools track the application's performance and detect errors or issues promptly. Laravel's built-in logging features can be integrated with external monitoring services for comprehensive monitoring.

4.4.5 Rollback Plan

- **Backup:** Regular backups of the application's database and files allow for recovery if strictly needed. Before deploying a new version, ensure recent backups are available to restore if needed.
- **Rollback Procedure:** There must be a clear procedure for rolling back to a previous version of the application in case of deployment failure or critical post-deployment issues. **This is a critical last resort.** Extensive upgrades may require extensive changes to the database, that may not be easy to roll back. If a critical upgrade is planned, consider locking the application, to avoid external changes to the database that may be lost upon roll back. Ex: REST APIs that update data should be

unavailable during the upgrade, even if they are not related to the new features being added.

Successful deployment of applications is a critical step in the development lifecycle, requiring careful planning and execution. By following these guidelines and utilizing the appropriate tools and practices, we can ensure that applications are deployed smoothly, securely, and efficiently, minimizing downtime and providing a seamless experience for the end-users.

5 USER INTERFACE DESIGN

5.1 USER EXPERIENCE (UX) GUIDELINES

Creating a positive user experience (UX) is essential for the success of any application. UX encompasses all aspects of the end-user's interaction with the company, its services, and its products. The goal is to enhance customer satisfaction and loyalty by improving the usability, ease of use, and pleasure provided in the interaction between the customer and the product. Below are the key guidelines for designing our UX:

5.1.1 Understand Your Users

- Conduct user research to understand their behaviors, needs, motivations, and pain points.
- Create personas to represent the different user types that might use your service, product, or site.
- Map user journeys to identify all possible interactions users can have with your product.

5.1.2 Design with Consistency

- Use consistent layouts and visual elements to improve learnability and reduce cognitive load.
- Employ a uniform language, tone, and terminology across the product.

5.1.3 Simplicity

- Keep the interface simple, with a clear hierarchy and minimal clutter to avoid overwhelming the user.
- Focus on primary tasks and present only the necessary information that will guide users to complete those tasks efficiently.

5.1.4 Feedback and Response Time

- Provide immediate feedback in response to user actions to keep them informed of results or errors.
- Optimize response times; users should not have to wait excessively long for any action they take.

5.1.5 Easy Navigation

- Design an intuitive navigation system that allows users to quickly find what they're looking for.
- Use familiar navigation patterns and include search functionality to expedite the discovery process.

5.1.6 Error Prevention and Handling

- Design the UI in such a way that potential errors are minimized.

- Clearly communicate error messages with actionable solutions to help users resolve any issues.

5.1.7 Usability Testing

- Test your design with real users to gather feedback and observe interaction patterns.
- Conduct usability testing at multiple stages of the design process to identify and fix issues early on.

5.1.8 Visual Hierarchy and Readability

- Use size, color, contrast, and alignment to create a clear visual hierarchy that guides users through your content.
- Ensure text readability with appropriate fonts, sizes, spacing, and paragraph formatting.

By adhering to these UX guidelines, we can create products that not only meet the users' needs but also provide an enjoyable and effective interaction. But, a good UX design is an ongoing process of learning and improvement, requiring continuous user feedback and iterative development, so it's only natural that changes to the UX may come at a later stage, after the users are able to have their hands on the system.

5.2 Visual Design Principles

5.3 Accessibility Requirements

Design is for all people including those with disabilities. The Web Content Accessibility Guidelines (WCAG) will be followed to ensure the platform is accessible.

As much as possible, the platform will provide alternative text for images, using sufficient contrast for text and background colors, and ensuring navigation is possible via keyboard.

The Web Content Accessibility Guidelines (WCAG) are part of a series of web accessibility guidelines published by the Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C), the main international standards organization for the Internet. They are designed to make web content more accessible to people with a wide range of disabilities, including visual, auditory, physical, speech, cognitive, language, learning, and neurological disabilities.
WCAG has several versions, with WCAG 2.0 and WCAG 2.1 being the most commonly referenced. The guidelines are organized around the following four principles, which lay the foundation necessary for anyone to access and use web content:
Perceivable: Information and user interface components must be presented to users in ways they can perceive. This means that users must be able to perceive the information being presented (it can't be invisible to all of their senses).
Operable: User interface components and navigation must be operable. This means that users must be able to operate the interface (the interface cannot require interaction that a user cannot perform).
Understandable: Information and the operation of the user interface must be understandable. This means that users must be able to understand the information as well as the operation of the user interface (the content or operation cannot be beyond their understanding).
Robust: Content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technologies. This means that users must be able to access the

content as technologies advance (as technologies and user agents evolve, the content should remain accessible).
Each principle is addressed through guidelines, and for each guideline, there are testable success criteria, which are at three levels: A (lowest), AA, and AAA (highest). WCAG 2.0 and 2.1 have 12-13 guidelines organized under these principles, with a total of 78 success criteria across all levels.

6 Data Management

6.1 Database Design

This section outlines key considerations and best practices for designing a database using Laravel and PostgreSQL, explaining the global application database approach.

6.1.1 Understanding Laravel with PostgreSQL

Laravel's Eloquent ORM and PostgreSQL: Laravel's Eloquent ORM provides an elegant, ActiveRecord implementation for working with databases. When paired with PostgreSQL, it leverages PostgreSQL's advanced features, such as full-text search, JSON support, and concurrency control, while maintaining ease of development and code readability.
Migration and Schema Builder: Laravel's migrations offer version control for your database schema, allowing the definition and sharing of the database's layout across development teams. The Schema Builder is an intuitive way to define database tables and columns in PHP code, ensuring your database schema is also part of your application's version-controlled codebase.

6.1.2 Design Principles

Normalization and PostgreSQL's Advanced Types: A normalized database design reduces redundancy and ensures data integrity. PostgreSQL's support for advanced data types, including arrays and JSONB, allows for efficient representation of structured data, giving the flexibility to denormalize where performance benefits outweigh the purity of normalization.
Indexes and Performance: PostgreSQL excels in its indexing capabilities, including B-tree, hash, GIN, and GiST indexes. These are used with advantage by indexing columns that are frequently queried or used in join conditions. Laravel's migration system allows to easily define indexes alongside the table structures.
Foreign Key Constraints: PostgreSQL's robust support for foreign key constraints within Laravel migrations enforce referential integrity at the database level. This ensures relationships between tables remain consistent, a critical aspect of database design that prevents orphaned records and data anomalies.
Utilizing Laravel Relationships: Eloquent ORM makes it simple to define relationships (e.g., one-to-one, one-to-many, many-to-many) between models, which correspond to the relationships between tables in PostgreSQL. Defining these relationships in Laravel models enables writing more expressive and concise code when fetching related data.

6.1.3 Best Practices

Using Migrations for Evolution: Laravel's migrations are used to evolve the database schema over time. This approach ensures changes are applied consistently across development, staging, and production environments, reducing the risk of discrepancies and deployment issues.
Leveraging PostgreSQL Features: Full advantage is taken of PostgreSQL-specific features where appropriate. This includes using JSONB columns for flexible data storage, exploiting

the full-text search capabilities for efficient searching, and utilizing PostgreSQL's powerful aggregation and window functions for complex data analysis.

Optimizing Queries with Eloquent: While Eloquent simplifies data manipulation and retrieval, it's crucial to remain mindful of potential performance pitfalls, especially the "N+1" query problem. Eloquent's eager loading feature (**with()**) can optimize related model data fetching. Leveraging PostgreSQL's materialized views for complex aggregations allows further optimization in data access.

Regularly Review and Optimize: Regularly reviews to the database schema and queries for optimization opportunities. PostgreSQL provides extensive logging and analysis tools like **EXPLAIN** to help identify and optimize slow queries. Laravel's database query log can also be a valuable tool in understanding how your application interacts with the database.

6.2 Data Security Measures

This section highlights key data security measures when using PostgreSQL in application development.

6.2.1 Authentication and Authorization

- **Role-Based Access Control (RBAC):** PostgreSQL utilizes a sophisticated role-based access control system. Defining roles for users and groups with specific privileges, ensures that individuals can only access data and perform operations pertinent to their role.
- **Strong Password Policies:** Enforcing strong password policies using PostgreSQL's password authentication mechanisms.

6.2.2 Encryption

- **SSL/TLS for Data in Transit:** Secure data in transit between the application and PostgreSQL server using SSL/TLS encryption. Always enforce SSL connections to prevent data interception and ensure confidentiality.

6.2.3 Network Security

- **Firewall Configuration:** Access to the PostgreSQL server is restricted from unauthorized networks by configuring firewall rules. Only connections from trusted application servers and administrative locations are allowed. No direct internet connections are allowed.
- **Secure Connection Settings:** PostgreSQL's **pg_hba.conf** file defines access policies, including which IP addresses can connect to which databases and which authentication methods are allowed.

6.2.4 Auditing and Monitoring

- **Logging and Audit Trails:** Detailed logging in PostgreSQL is used to keep an audit trail of database activities. Tools like **pgAudit** for more granular audit logging, allow to track and analyze access and changes to sensitive data.
- **Regular Security Assessments:** Periodically perform security assessments and vulnerability scans on the PostgreSQL installation. The database server and software are kept up to date with the latest security patches and updates.

6.3 Data Backup and Recovery

This section outlines essential practices for data backup and recovery in the PostgreSQL environment used in this project.

6.3.1 Backup Techniques

Logical Backups (in use):

- **pg_dump and pg_dumpall:** We use **pg_dump** for backing up a single database and **pg_dumpall** for all databases in a PostgreSQL cluster. These tools generate SQL scripts that can recreate the database by replaying the commands.
- **Advantages:** Logical backups are portable and can be restored on any PostgreSQL server, regardless of the architecture or version (with some version considerations).
- **Use Case:** Ideal for smaller databases or when migrating data across different PostgreSQL versions.

Physical Backups:

- **File System-Level Backup:** Directly copy the database files from the file system. This requires the database to be shut down or a consistent snapshot mechanism to be in place (e.g., using LVM snapshots).
- **pg_basebackup:** A tool for taking a base backup of a PostgreSQL database cluster, allowing for hot backups (backups taken while the database is running).
- **Advantages:** Faster backup and restore times compared to logical backups, especially for large databases.
- **Use Case:** Suitable for large databases and when minimizing downtime is critical.

6.3.2 Point-in-Time Recovery (PITR)

PITR extends physical backups by allowing the database to be restored to a specific moment in time. This is achieved by combining a base backup (ex: done the previous night) with the write-ahead log (WAL) files that record all changes made to the database after the backup (ex: during the day).

- **Continuous Archiving:** Configure continuous archiving of WAL files using the **archive_mode** and **archive_command** settings in **postgresql.conf**.
- **Restore Process:** Use **pg_restore** or directly copy the base backup files, and replay WAL files up to the desired point in time.

6.3.3 Best Practices for Backup and Recovery

- **Regular Backups:** Regular backups are scheduled, to reduce data loss risk. The frequency will be based on the ammount of data modifications and number of users.
- **Monitor Backups:** Automated backup monitoring to verify that backups complete successfully and alert on failures.
- **Secure Backups:** Encrypted backup files and secure transfer to off-site storage or cloud services to protect against unauthorized access and data breaches.
- **Test Recovery Procedures:** Regularly test recovery procedures to ensure that backups can be restored successfully and within the required time frames. This helps identify issues in the backup process and reduces downtime during actual recovery scenarios.

7 Integration and APIs

This section is the subject of deliverable **D5.1 Report on API specification**.

8 PERFORMANCE OPTIMIZATION

8.1 LOAD TESTING

Load testing is a critical component of performance optimization, designed to evaluate how a system behaves under both normal and peak load conditions. This process involves simulating a high number of users accessing the application simultaneously to identify bottlenecks, determine the system's capacity, and ensure stability and reliability under various load scenarios. This section outlines the importance, methodologies, tools, and best practices for conducting effective load testing as part of the technical requirements and development guide.

The objectives are:

- **Identify Performance Bottlenecks:** Discover areas in the application where performance issues may arise under heavy load, allowing for targeted optimizations.
- **Validate Scalability:** Confirm that the application can handle expected user growth and peak usage periods without degradation in performance.
- **Ensure Reliability and Stability:** Verify that the application remains stable and responsive under high load, preventing downtime and ensuring a positive user experience.
- **Optimize Infrastructure Costs:** Understand the system's capacity to optimize resource allocation and infrastructure costs, avoiding over-provisioning while ensuring performance targets are met.

Load Testing is done in several ways, following the order:

1. **Baseline Testing:** Establish a performance baseline by testing the application under normal load conditions. This baseline serves as a reference point for comparing the effects of optimizations and understanding performance under peak load.
2. **Stress Testing:** Incrementally increase the load until the application or infrastructure components fail. This identifies the upper limits of the system's capacity and uncovers how the system fails under extreme conditions.
3. **Endurance Testing:** Subject the application to a significant load over an extended period. This helps identify issues like memory leaks, resource depletion, and performance degradation over time.
4. **Peak Load Testing:** Simulate the maximum expected number of users accessing the application simultaneously. This test assesses the system's behavior during peak usage periods and ensures it can handle sudden spikes in traffic.
5. **Scalability Testing:** Evaluate the application's ability to scale up or out by adding resources (e.g., CPU, RAM, servers) and measuring the impact on performance. This helps determine the most effective scaling strategies.

Several open-source tools are available for conducting load testing, ranging from open-source to commercial solutions. Popular options include:

- **JMeter:** An open-source tool designed for load testing web applications. It can simulate multiple users with concurrent requests and supports various protocols.
- **Gatling:** An open-source load testing tool known for its high performance and support for complex scenarios. It uses a DSL for test script creation.
- **Locust:** An open-source load testing tool written in Python, allowing for writing test scenarios in Python code. It is lightweight and scalable.

The best practices on load testing, involve:

- **Realistic Test Scenarios:** Design test scenarios that closely mimic real-world usage patterns, including different types of user interactions and data.
- **Continuous Testing:** Integrate load testing into the continuous integration/continuous deployment (CI/CD) pipeline to regularly assess performance as the application evolves.
- **Monitor and Analyze:** Use monitoring tools to collect metrics during load tests. Analyze these metrics to identify bottlenecks and understand the system's behavior under load.
- **Iterative Optimization:** Conduct load testing iteratively throughout the development cycle. Use the results to guide performance optimizations and retest to validate improvements.

By systematically conducting load testing and analyzing the results, we can identify and address performance bottlenecks, ensuring that the application meets performance goals, remains stable under peak loads, and delivers a seamless user experience. Integrating load testing into the development process supports informed decision-making regarding infrastructure investments and optimization strategies, ultimately contributing to the success and scalability of the application.

8.2 Code Optimization

Code optimization is the process of modifying code to improve its efficiency and performance without altering its functionality. This is crucial for enhancing the speed, reducing resource consumption, and improving the overall user experience of software applications. By optimizing code, we can ensure that applications run smoothly, respond quickly to user inputs, and operate effectively on limited resources. It's crucial to approach optimization thoughtfully, based on actual performance data and considering the overall impact on code maintainability and readability. It’s important to focus first on having a working solution and only after using benchmarks and profiling to identify actual performance bottlenecks, to tackle possible optimizations. Avoid premature optimization without evidence that a specific piece of code is a performance issue.

8.3 Scalability Considerations

Docker is a leading platform for developing, shipping, and running applications inside containers, playing a pivotal role in addressing scalability challenges in modern software architectures. Containers encapsulate applications with their dependencies, ensuring consistency across different environments and facilitating easy scaling. This section discusses key scalability considerations when using Docker, offering insights into how Docker can be leveraged to build highly scalable and resilient applications.
The objectives are to:

- **Enable Seamless Scaling:** Utilize Docker to scale applications horizontally, adding or removing instances based on demand, without affecting the overall system stability or user experience.
- **Maintain Performance Under Load:** Ensure applications maintain high performance and responsiveness as they scale, handling increased loads efficiently.
- **Simplify Deployment and Management:** Leverage Docker's ecosystem to automate and simplify the deployment, scaling, and management of containerized applications.

To properly handle scalability, some considerations need to be taken:

- **Stateless vs. Stateful Applications:** Design applications to be stateless whenever possible, as stateless applications are easier to scale with Docker. For stateful

applications, consider strategies to externalize state management using databases or persistent storage solutions that can handle scaling independently.

- **Resource Allocation:** Carefully manage container resources through Docker's CPU and memory allocation features. Properly configuring these can prevent any single container from monopolizing system resources, ensuring a balanced distribution of resources among all containers.
- **Load Balancing:** Implement load balancing to distribute incoming traffic evenly across multiple container instances. Docker Swarm mode and Kubernetes offer built-in load balancing solutions that work seamlessly with containerized applications.
- **Service Discovery:** As applications scale, keeping track of which service is running where becomes challenging. Utilize service discovery mechanisms available in container orchestration platforms like Docker Swarm or Kubernetes to dynamically manage connections between services.
- **Orchestration Tools:** Adopt container orchestration tools such as Docker Swarm or Kubernetes for managing large-scale container deployments. These tools provide essential features for scalability, including automatic scaling, self-healing, and rolling updates.
- **Monitoring and Logging:** Implement comprehensive monitoring and logging solutions to track the performance and health of applications as they scale. Tools like Prometheus for monitoring and Elasticsearch for logging, integrated with Docker, provide valuable insights for scaling decisions.
- **Efficient Image Management:** Optimize Docker images for size and build efficiency. Smaller images reduce deployment times and resource consumption, which is crucial for scaling operations. Utilize multi-stage builds and remove unnecessary dependencies to minimize image size.

The following best practices considerably improve success when implementing scalability:

- **Immutable Containers:** Adopt an immutable infrastructure approach where containers are never modified after they are deployed. Instead, new container instances are created from updated images and replaced, simplifying scaling and deployment processes.
- **Automate Everything:** Use CI/CD pipelines and automation tools to streamline the building, testing, and deployment of containerized applications. Automation ensures consistency and reduces the risk of errors during scaling operations.
- **Plan for Failover and Redundancy:** Design the system with failover and redundancy in mind to ensure high availability. This includes deploying services across multiple nodes or data centers to prevent downtime during scaling actions or infrastructure failures.

Scalability is a critical consideration in modern application development, and Docker offers a robust set of features and practices to address these challenges effectively.

9 Security Measures

9.1 Authentication and Authorization

Laravel offers mechanisms for user authentication and "login" capabilities. Laravel aims to provide developers with the necessary tools to implement authentication swiftly, securely, and effortlessly.

At the heart of Laravel's authentication system lie "guards" and "providers". Guards are responsible for determining the method of user authentication for each request. For instance, Laravel includes a session guard that utilizes session storage and cookies to manage state.
Providers are tasked with fetching user information from your persistent storage solutions. By default, Laravel supports user retrieval via Eloquent and the database query builder, though it allows for the creation of additional providers to suit your application's specific needs.
The authentication configuration can be found in the **config/auth.php** file. This file offers a variety of well-explained options that allow for customization of Laravel's authentication features.
Beyond its built-in authentication capabilities, Laravel simplifies the process of authorizing user actions for specific resources. Although a user might be authenticated, they might not have the permission to modify or remove certain Eloquent models or database records within your application. Laravel's authorization tools offer a streamlined method for conducting these authorization verifications.
Laravel includes two main mechanisms for action authorization: **gates** and **policies**. Analogous to routes and controllers, **gates** provide a straightforward, closure-based method for authorization, whereas **policies** organize authorization logic around a specific model or resource.
Gates are ideally suited for actions not associated with a model or resource, such as accessing an admin dashboard. On the other hand, **policies** are recommended for authorizing actions concerning a specific model or resource.

9.2 Encryption Standards

Laravel employs strong encryption standards to ensure data security and confidentiality. This is the main application standard, used over or across the system modules, whenever secure communication is required beyond the context of the server. The framework uses the OpenSSL library to provide **AES-256** and **AES-128** encryption. These standards are widely recognized for their robustness and are considered secure for protecting sensitive information. Laravel's encryption facilities are built around these algorithms to safeguard data at rest and during transmission.
Here's a brief overview of the encryption standards in Laravel:

- **AES-256**: Stands for Advanced Encryption Standard with a 256-bit key. It is one of the most secure encryption methods used in modern encryption algorithms, protocols, and technologies. Laravel uses AES-256 as its default encryption cipher.
- **AES-128**: Similar to AES-256 but uses a 128-bit key. It offers a slightly lower level of security compared to AES-256 but is still considered highly secure and is faster in terms of performance. Laravel supports AES-128, allowing developers to choose between the level of security and performance that best suits their application's needs.

Laravel's encryption configuration is defined in the **config/app.php** file, where you can specify the default cipher (**AES-256-CBC** or **AES-128-CBC**) and set an application key (**APP_KEY**). This key serves as the encryption key and should be kept secret. Laravel uses this key to encrypt and decrypt data securely. The **APP_KEY** can be generated using the artisan command **php artisan key:generate**, ensuring it is sufficiently random and secure.

Laravel provides a simple, clean API for encrypting and decrypting data through its **Crypt** facade. Encrypted values are signed with a message authentication code (MAC) to detect any modifications to the encrypted string.

The following considerations are essential for a proper secure implementation:

- The encryption key (**APP_KEY**) is crucial to the security of encrypted data. It should be stored securely and never be exposed to unauthorized users.
- Laravel's encryption mechanism is designed to be seamless, but it is necessary to understand when and how to apply encryption to protect sensitive data appropriately.
- Regularly reviewing and updating security practices, including encryption standards and key management procedures, is essential for maintaining data security.

9.3 Security Audits and Monitoring

Ensuring the security of a server application is a continuous process that involves regular audits and proactive monitoring. Security audits help identify vulnerabilities within the application, while monitoring ensures that any security threats or anomalies are detected and addressed in real time. Implementing a comprehensive strategy for security audits and monitoring is crucial for maintaining the integrity, confidentiality, and availability of the application and its data.

Security audits involve a systematic examination of the application's codebase, dependencies, configuration settings, and deployment environment to identify security weaknesses and non-compliance with best practices. Key components of a security audit include:

- **Code Review:** Manual and automated review of the application's source code to detect security flaws, such as SQL injection vulnerabilities, cross-site scripting (XSS) vulnerabilities, and insecure direct object references.
- **Dependency Analysis:** Regularly checking third-party packages and libraries for known vulnerabilities using tools like Composer's **composer audit** command, Laravel-specific and Python specific packages designed for security auditing.
- **Configuration and Environment Review:** Verifying that configuration settings, both within Laravel and in the server environment, are optimized for security. This includes checking for correct permission settings, secure database connections, and proper encryption key management.

Monitoring involves tracking the application's operations and analyzing logs to detect and respond to security incidents. Effective monitoring can alert administrators to unauthorized access attempts, potential vulnerabilities being exploited, or unusual application behavior that could indicate a security issue. Key aspects of monitoring include:

- **Log Analysis:** Utilizing Laravel's logging capabilities to capture and analyze logs. This can be enhanced with external tools like Logstash, Elasticsearch, and Kibana (the ELK stack) for more sophisticated log analysis and visualization.
- **Real-Time Alerting:** Implementing real-time alerting mechanisms to notify administrators of potential security incidents. This can be achieved through custom Laravel notifications or integration with external monitoring services.
- **Performance Metrics:** Monitoring application performance metrics, as a sudden change in performance can sometimes indicate a security issue or an ongoing attack.

The best practices include:

- **Automate Security Processes:** Where possible, automate security auditing and monitoring processes to ensure they are performed consistently and efficiently.
- **Stay Informed:** Subscribe to security bulletins and update dependencies regularly.
- **Educate and Train:** Ensure that the development and operations teams are aware of security best practices and the importance of security within the development lifecycle.

Security audits and monitoring are essential components of na application's security posture. Regularly auditing the application for vulnerabilities and continuously monitoring its operations can help prevent security breaches and minimize the impact of any incidents that do occur.

10 DOCUMENTATION GUIDELINES

10.1 CODE DOCUMENTATION

This subsection outlines best practices for documenting application's code in the project.

10.1.1 Commenting Code

- **Inline Comments:** Use inline comments sparingly to explain "why" behind complex or non-obvious code logic. Avoid stating "what" the code does, unless it's not immediately clear from the code itself.
- **Clarity:** Keep comments concise and relevant. Avoid stating the obvious.
- **DocBlocks:** Utilize DocBlock comments for all classes, methods, and functions. These should describe the purpose of the element, parameters, return types, and any exceptions thrown. Laravel follows the **PHPDoc** standard for these comments, facilitating better understanding and integration with IDEs for auto-completion and code analysis.
- **Docstrings:** Python's docstrings offer a built-in method of documenting Python classes, methods, functions, and modules:
    - **Format:** Use triple double quotes **"""** to start and end a docstring.
    - **Content:** Include a concise description of the function's purpose, parameters, return values, and any exceptions it raises.

10.1.2 Documenting Methods and Functions

- **Purpose:** Start with a brief description of what the method/function does.
- **Parameters:** List each parameter, its type, and a short description. Include whether the parameter is optional or required.
- **Return Types:** Clearly specify the return type and describe what is being returned.
- **Exceptions:** Note any exceptions that can be thrown by the method/function.

10.1.3 Class Documentation

- **Class Overview:** Provide an overview of the class at the beginning of each class file, including its role within the application.
- **In python,** document classes using docstrings immediately below the class definition. Include an overview of the class and docstrings for each method.
- **Properties:** Document each property, especially if its usage isn't inherently clear. Include types and any default values.
- **Usage Examples:** Where appropriate, include snippets or examples of how to use the class or its methods, particularly for libraries, helpers, or complex components.

10.1.4 Versioning and Deprecation Notices

- **Versioning:** Document the version of the application or API that introduced each significant piece of functionality. This practice is particularly important for APIs and libraries.
- **Deprecation Notices:** Clearly mark deprecated methods or classes, providing alternatives when possible. Include the version in which the deprecation occurred and the expected removal version, if known.

10.1.5 Readability and Consistency

- **Language:** Use clear, concise, and simple language. Aim for accessibility, assuming readers have varying levels of expertise.
- **Format and Style:** Adhere to a consistent format and style for your documentation. Consider adopting widely used standards within the Laravel community or your development team.
- **Update Regularly:** Documentation should evolve with your codebase. Regularly review and update documentation to reflect changes, removals, or additions to the codebase.

10.1.6 External Documentation

- **ReadMe and Wikis:** Use Markdown files (e.g., README.md) for project-level documentation, including installation instructions, usage examples, and contribution guidelines.

10.1.7 Tools and Integrations

- **Automated Documentation Generators:** Utilize tools like PHPDocumentor, Sami or Sphinx to generate API documentation automatically from your DocBlocks and docstrings. These tools can save time and ensure consistency across your documentation.
- **Code Reviews:** Incorporate documentation quality into your code review process. Peer reviews should include checks for adequate and accurate documentation.

Well-documented code is as important as the code itself. It ensures that developers, both current and future, can understand, use, and contribute to the project effectively. By following these guidelines for code documentation within your Laravel application, you create a more maintainable, understandable, and user-friendly codebase.

10.2 User Manuals

This subsection outlines best practices for creating effective user manuals.

10.2.1 Understanding the Audience

- **User Profiling:** Tailor the manual to the audience's knowledge level and needs. Differentiate between novice, intermediate, and advanced users, if necessary, by providing layered instructions or separate sections.
- **Language and Tone:** Use clear, straightforward language that is accessible to your target audience. Avoid jargon and technical terms unless you are targeting a technically savvy audience, and even then, provide explanations.

10.2.2 Structure and Content

- **Introduction:** Begin with an overview of the software, including its purpose, key features, and benefits. This section sets the context for the rest of the manual.
- **Getting Started:** Provide a quick start guide for users to perform basic operations or setup. This section is crucial for ensuring a positive first experience.
- **Features and How-Tos:** Detail each feature of the software, including step-by-step instructions on how to use them. Include screenshots, diagrams to illustrate these steps when possible. Clearly state the limits and assumptions on simulations, so that users are fully aware of the context that the simulation expects (**ex:** zero friction, temperature ranges, etc)

- **Troubleshooting:** Offer a troubleshooting section addressing common issues and their solutions. This empowers users to solve problems without reaching out for support.
- **FAQs:** Include a Frequently Asked Questions (FAQ) section to cover common queries. This section can evolve based on the feedback and inquiries from users.

10.2.3 Visual Aids

- **Screenshots:** Visual aids can significantly enhance understanding. Use high-quality screenshots to complement text instructions.
- **Diagrams and Flowcharts:** Use diagrams to explain workflows or architecture, and flowcharts for decision-making processes within the software.

10.3 Release Notes

This subsection outlines best practices for crafting informative and user-friendly release notes.

10.3.1 Importance of Release Notes

- **Transparency:** Release notes communicate the changes in each version, fostering trust and transparency between the developers and the user community.
- **Upgrade Decisions:** They help users and administrators decide whether to upgrade by detailing the benefits and any potential impacts of the new release.

10.3.2 Key Components

- **Version Number and Release Date:** Clearly state the version number and release date at the beginning of the release notes to easily identify the software update.
- **New Features:** List new features and enhancements. Provide a brief description of each feature and its potential benefits to the user.
- **Improvements:** Mention improvements to existing features, focusing on how they enhance the user experience or performance of the software.
- **Bug Fixes:** Summarize key bug fixes, including the issue they resolve. Avoid overly technical descriptions unless the audience demands it.
- **Known Issues:** If applicable, list any known issues and their potential impact on users. Include workarounds or links to more information if available.
- **Breaking Changes:** Clearly highlight any breaking changes or compatibility issues. Provide guidance on how to address these changes, such as updating configurations or modifying code.
- **Acknowledgments:** Optionally, acknowledge contributors or thank the community for their feedback and support, reinforcing the collaborative nature of the project.

Release notes should be distributed in a text/Markdown file, along with the code.

11 Conclusions

This document has provided a comprehensive guide covering the essential aspects of technical requirements, development practices, and documentation standards to be used in the software side of the Green Marine project. From outlining the foundational server and hardware requirements to delving into the intricacies of database design, performance optimization, and security, the goal is to equip developers and project managers with the knowledge needed to navigate the complex landscape of modern software development, within the scope of this project, providing a common frame of reference.

These guidelines should be seen as a starting point. As the project evolves, some procedures may also evolve, to better adapt to the working needs.