“RepairMart”

A dissertation submitted in partial fulfilment of

the requirement for the degree of

MASTER OF SCIENCE in Software Development

in

The Queen’s University of Belfast

By

Ciaran Neeson

04-May-2025

Contents

[Acknowledgements i](#_Toc197441255)

[Abstract ii](#_Toc197441256)

[Introduction 1](#_Toc197441257)

[Understanding the Problem 1](#_Toc197441258)

[User interface design 1](#_Toc197441259)

[Architecture Design 1](#_Toc197441260)

[Testing 1](#_Toc197441261)

[Evaluation and Conclusion 1](#_Toc197441262)

[Understanding the Problem 2](#_Toc197441263)

[Problem Specification 2](#_Toc197441264)

[Current options 2](#_Toc197441265)

[Waste 2](#_Toc197441266)

[Resale 3](#_Toc197441267)

[Repair 3](#_Toc197441268)

[Solution 3](#_Toc197441269)

[Users/Stakeholders 4](#_Toc197441270)

[Requirements elicitation 4](#_Toc197441271)

[Adaptation to the proposed solution 4](#_Toc197441272)

[Languages, libraries & frameworks 5](#_Toc197441273)

[Advantages Over Other Frameworks 6](#_Toc197441274)

[User Interface Design 8](#_Toc197441275)

[Design Process 8](#_Toc197441276)

[Prototyping 8](#_Toc197441277)

[Investigation/examination of similar solutions 8](#_Toc197441278)

[Final Design 9](#_Toc197441279)

[Architecture Design 17](#_Toc197441280)

[Laravel development framework 17](#_Toc197441281)

[Front end 18](#_Toc197441282)

[File storage 19](#_Toc197441283)

[Database development 19](#_Toc197441284)

[UML diagrams 19](#_Toc197441285)

[Entity-Relationship diagrams 20](#_Toc197441286)

[Development & Testing 22](#_Toc197441287)

[Considerations for Production transition 23](#_Toc197441288)

[Database population 23](#_Toc197441289)

[Static data 23](#_Toc197441290)

[Database relationships 26](#_Toc197441291)

[Seeding 27](#_Toc197441292)

[Code base 28](#_Toc197441293)

[Directory Structure and MVC Implementation 28](#_Toc197441294)

[Performance Optimisation 30](#_Toc197441295)

[View Layer & Front-End Technologies 32](#_Toc197441296)

[Routing 33](#_Toc197441297)

[User Email Verification 33](#_Toc197441298)

[Laravel Migration files 34](#_Toc197441299)

[Scheduled Jobs 34](#_Toc197441300)

[User Messaging 35](#_Toc197441301)

[Factories & Seeders 37](#_Toc197441302)

[Tests 37](#_Toc197441303)

[Testing 38](#_Toc197441304)

[Manual Testing 38](#_Toc197441305)

[Automated Testing 38](#_Toc197441306)

[Evaluation and Conclusion 46](#_Toc197441307)

[Improvements to existing features 46](#_Toc197441308)

[Scope expansion 46](#_Toc197441309)

[Use of AI 47](#_Toc197441310)

[References i](#_Toc197441311)

[Appendices i](#_Toc197441312)

[Appendix 1: Original Figma UI designs i](#_Toc197441313)

[Appendix 2: UML diagrams iii](#_Toc197441314)

[Appendix 3: Manual Testing vi](#_Toc197441315)

# Acknowledgements

I would like to thank my supervisor, Olga Pishchukhina, for her guidance and advice during the development of this project.

I would also like to thank the following colleagues in my employer’s organisation for showing an interest in my project and volunteering to review developments and provide feedback along the way: Jennifer Mahoney, Ryan O’Reilly, and Scott Lewis.

I would also like to express my thanks to Queen’s University Belfast, and particularly all of the teaching staff on the EEECS school’s MSc Software Development (part-time) programme for the organisation of this course, and for giving me the opportunity to pursue it.

# Abstract

# Introduction

An outline of this project’s structure is below, giving the key components of each section.

### **Understanding the Problem**

This section outlines the problem area and motivation for the current project, highlighting the current solution gap and contrasting it with analogous solutions in other areas. Following this, the software choices used for developing this project are outlined, with justification of the choices made.

### **User interface design**

In this section, existing user interfaces are examined and used to inform the design for this project. Rationale is provided for the interface design choices in this project, and an overview of the interface is given.

### **Architecture Design**

This section gives a high-level overview of the project’s architecture, covering the different sections within it: (1) the front-end UI framework, (2) the system file storage, (3) the back-end database management system, and finally (4) the code base used to connect the aforementioned sections within the application. Justification is given for the design choices made and the technologies used, and an overview of the code base is given to present the most important elements and several noteworthy configurations.

### **Testing**

This section describes the project’s testing methodology, with examples of manual testing....

### **Evaluation and Conclusion**

This section assesses the degree to which the final developed solution meets the outlined requirements, and discusses areas for potential future improvements and developments.

# Understanding the Problem

### **Problem Specification**

Waste from electronic consumer goods (e-waste) is a big problem in today’s world (World Health Organization, 2024). Consumers frequently dispose of their electronic products in an inefficient manner when they no longer function correctly, while in reality these products may be repairable and their useful life may be extendable. Market conditions mean that it is frequently cheaper to buy a brand new product and throw away a broken one, creating excess waste and undesirable secondary effects (in addition to causing people to spend more than they need to). The current consumer economy is primarily oriented towards profit, and this largely excludes easily repairing faulty items: although ‘Right to Repair’ laws have been introduced in the UK and the EU in July 2021 (Conway, 2021) and April 2024 (García Molyneux & Oberschelp de Meneses, 2024), respectively, the new legislation is limited in scope, and in duration (European Commission, 2023). Furthermore, when seeking repairs, consumers are directed towards a single point of contact (the manufacturer or vendor of their non-functioning product), which limits their options in terms of budget, location, viability, etc.

The aim of this project is to develop a repair ‘marketplace’ (an ‘eBay’ equivalent for electronics repairs) where consumers can create a request to repair a non-functioning product, uploading details so that their request is visible to multiple repair specialists (e.g. large manufacturers, small businesses or independent individuals with the relevant expertise). The specialists will be able to review requests, ask any relevant questions for clarification on the product and/or its defects, and submit a quote for repair to the consumer. Consumers should then be able to review a list of quotes received and choose the most suitable option for them – based on price, location, estimated turnaround, etc. – making the process of repairing their product easier and cheaper, and resulting in the extension of their product’s useful life, a reduction in electronic waste, and an overall cultural shift away from throwaway consumerism.

### **Current options**

When dealing with electronic goods that no longer work correctly, consumers can either (a) dispose of them as waste, (b) potentially try to re-sell a second-hand item for spare parts, or (c) try to repair the item.

### **Waste**

There are numerous websites giving recycling instructions and allowing consumers to find their nearest recycling points. A number of problems exist, though:

* these websites are not well-publicised: many people do not know about their existence
* they often have limited (geographical) scope, serving local communities/council areas while lacking coordination across different localities/jurisdictions
* the lack of a ‘recycling culture’ among some demographics can be difficult to overcome
* the lack of incentives for recycling (and of penalties for failing to properly dispose of electronic goods) causes unnecessary and avoidable waste

### **Resale**

Consumers can try to re-sell their faulty/broken goods, often using online marketplaces such as eBay, however the return on doing so is often negligible and may not compensate for having to deal with things like non-standard postage & packaging (as well as handling potential refunds/returns for unsatisfied customers).

### **Repair**

Repairs have become somewhat less prohibitive for certain classes of electronic goods thanks to the aforementioned new legislation, however the cost is often still prohibitively expensive since customers typically only deal with a very small number of potential experts, only receiving one or two quotes (often from the items’ manufacturers, who tend to charge significantly more for their services and also prohibit competition by voiding product warranties if consumers seek the same service outside of a manufacturer-approved network (Brannon, 2024)).

### **Solution**

Creating a centralised marketplace where consumers can submit requests for repair can solve many of the above-mentioned problems, since the onus can be placed on repair specialists to offer their services in a more transparent manner, lowering prices for consumers and creating a greater incentive to avoid electronic waste through extending the useful life of products (in contrast to the lack of incentive that currently exists, failing to discourage the common consumer ‘throwaway’ reflex). Such a solution, if developed correctly, could also be pitched to government/local authorities who may be interested in lowering the burden on them for waste disposal and helping to reduce the carbon footprint inherent in the current ‘throwaway’ economy (from manufacturing new products and importing them from far-away locations).

The solution to this problem is envisaged as an online platform containing some elements which can be found in websites for purchasing consumer electronics (e.g. [Amazon](amazon.co.uk)) and for the resale of used goods (e.g. [eBay](ebay.co.uk), [Gumtree](gumtree.com), [DoneDeal](donedeal.ie), etc.).

While there are websites currently offering similar services (e.g. <https://uk.electronic.partners>), these only offer contact with a single repair specialist (or single organisation) at a time: they do not centralise a range of repair options in the manner proposed for this project.

The advantage of a centralised platform to consumers is that instead of having to send multiple enquiries about the desired repair to multiple specialists – all via different channels, potentially having to register/create an account on multiple platforms, and without necessarily ever receiving a response to an inquiry – the proposed solution will provide one central marketplace and point of contact through which a consumer may register once and be contacted by multiple repair specialists, making the selection process much easier while also providing transparency over order management, status updates, feedback/customer satisfaction, etc.

The advantages to suppliers (i.e. repair specialists) include the ability to reach a wider number of customers, and the removal of a potential hurdle in no longer needing to set up and maintain their own proprietary website for handling repair enquiries and orders.

## **Users/Stakeholders**

The target user audience will consist of:

1. regular consumers who prefer to repair their electronic goods at a reasonable cost, with the intention to (a) avoid having to buy a replacement item, and (b) reduce/avoid the waste associated with the disposal of electronic items
2. electronics repairs specialists who wish to offer their services to a wider market, providing their services through the platform

Other potential interested parties (while not being direct users) may be government departments and/or local authorities who wish to promote a greater culture of re-using/refurbishing/recycling electronic goods, to reduce the cost and burden placed on them when it comes to the disposal of such goods.

## **Requirements elicitation**

Requirements have been gathered by investigating some well-known existing websites:

1. electronic goods retailers
2. marketplaces for second-hand goods
3. electronics repair specialists

The proposed solution will contain selected elements from each of the above categories, adapted for our proposed solution.

|  |  |
| --- | --- |
| **Website category** | **Features** |
| Electronic goods retailers (e.g. Amazon) | Account creation/management  Search/filter according to keyword, manufacturer, product type |
| Second-hand goods marketplaces (e.g. eBay, Gumtree) | Account creation/management  Account type differentiation (seller/buyer)  Search/filter according to keyword, manufacturer, product type  Auction system allowing bids from customers  Listing creation & management (incl. image/video upload)  System-generated email notifications (email verification, confirmation of different user actions, status updates, etc.)  Feedback system attached to user profiles  User-to-user messaging (e.g. for queries on listings) |
| Electronics repair specialists | Account creation/management  Assessment of product’s defects and receipt of quote with the option to accept/decline |

## **Adaptation to the proposed solution**

The below table gives greater detail on the features outlined above, and how these will be incorporated into the proposed solution.

|  |  |
| --- | --- |
| **Functionality group** | **Details** |
| User registration | Registration via email confirmation after providing user details (min. requirements: email address & password)  Password reset functionality |
| Account management | Account setup – addition of account type (consumer/repair specialist) & name/location details (these will be required for in-person collection/drop-off of items and/or postal information delivery  Ability to update account details (password, name, location, and potentially also email address) |
| Listing creation & management | Consumers |
| Search/review listings | Searching & sorting listings on the platform using keywords, with the ability to filter results by location, date (age of listing), product type, manufacturer |
| Repair quote creation & management | Repair Specialists |
|  |  |

In addition to the above, and as with most standard web applications in use today, we will aim to permit user registration via email, with password reset functionality.

## **Languages, libraries & frameworks**

In order to choose the right development framework for the proposed solution, a number of options have been considered. The table below contains details of these.

| **Option** | **Advantages** | **Disadvantages** | **Examples** |
| --- | --- | --- | --- |
| **React (JavaScript)**  Front-end library developed by Facebook for building user interfaces, particularly single-page applications.  It uses a component-based architecture and a virtual DOM to optimize rendering. | **Component Reusability**: Promotes code reuse and modularity.  **High Performance**: Virtual DOM ensures efficient updates.  **Strong Ecosystem**: Extensive libraries and tools. | **SEO Challenges**: Requires server-side rendering for better SEO.  **Learning Curve**: Requires understanding of JSX and modern JavaScript. | **Facebook** [[1]](https://www.intelivita.com/blog/popular-websites-built-with-react/)  **Airbnb** [[2]](https://www.cmarix.com/blog/most-popular-websites-built-with-react/)  **Instagram** [[1]](https://www.intelivita.com/blog/popular-websites-built-with-react/) |
| **Angular (TypeScript)**  Comprehensive front-end framework developed by Google. It uses TypeScript and provides a full suite of tools for building dynamic web applications. | **Two-Way Data Binding**: Simplifies synchronization between model and view.  **Comprehensive Toolset**: Includes everything needed for front-end development.  **Strong Community Support**: Backed by Google. | **Complexity**: Can be overwhelming for small projects.  **Performance Issues**: Can be slower compared to other frameworks for large applications. | **Gmail** [[3]](https://seclgroup.com/10-best-examples-of-websites-and-apps-built-with-angular/)  **Upwork** [[3]](https://seclgroup.com/10-best-examples-of-websites-and-apps-built-with-angular/)  **PayPal** [[3]](https://seclgroup.com/10-best-examples-of-websites-and-apps-built-with-angular/) |
| **Vue.js (JavaScript)**  Progressive front-end framework that can be integrated incrementally into projects. It is known for its simplicity and flexibility. | **Easy to Learn**: Simple syntax and structure.  **High Performance**: Lightweight and fast.  **Flexibility**: Can be used for both small and large projects. | **Smaller Community**: Compared to React and Angular.  **Limited Enterprise Adoption**: Less common in large-scale enterprise projects. | **Alibaba** [[4]](https://trio.dev/websites-using-vue/).  **Xiaomi** [[4]](https://trio.dev/websites-using-vue/).  **Behance** [[5]](https://vuejsfeed.com/blog/8-popular-websites-that-use-the-vue-js-framework). |
| **Django (Python)**  Django is a high-level back-end framework for building web applications quickly and with clean, pragmatic design. It follows the "batteries-included" philosophy. | **Rapid Development**: Built-in admin panel and ORM.  **Security**: Comes with built-in security features.  **Scalability**: Suitable for both small and large applications. | **Monolithic Framework**: Less flexibility compared to microservices architecture.  **Learning Curve**: Requires knowledge of Python and Django’s conventions. | **Instagram** [[6]](https://codment.com/famous-websites-built-with-the-django-framework/).  **Pinterest** [[6]](https://codment.com/famous-websites-built-with-the-django-framework/).  **Disqus** [[7]](https://djangostars.com/blog/10-popular-sites-made-on-django/). |
| **Ruby on Rails (Ruby)**  Ruby on Rails is a full-stack web application framework that emphasizes convention over configuration. It is known for its developer-friendly syntax and rapid development capabilities. | **Convention Over Configuration**: Reduces the number of decisions developers need to make.  **Rapid Development**: Scaffolding and built-in tools.  **Strong Community**: Extensive libraries and plugins. | **Performance**: Can be slower compared to other frameworks.  **Scalability Issues**: May require additional optimization for large-scale applications. | **Shopify** [[8]](https://sumatosoft.com/blog/top-20-websites-built-on-ruby-on-rails).  **GitHub** [[9]](https://prograils.com/top-10-famous-sites-built-with-ruby-on-rails).  **Airbnb** [[9]](https://prograils.com/top-10-famous-sites-built-with-ruby-on-rails). |
| **Laravel (PHP)**  PHP framework that follows the Model-View-Controller (MVC) architectural pattern. It is designed to streamline web development by providing a structured and expressive syntax. | **Elegant Syntax and Expressive Code**: Enhances productivity and reduces the learning curve.  **MVC Architecture**: Improves code organization, maintainability, and scalability.  **Built-In Authentication and Authorisation**: Simplifies user registration, login, and role-based access control.  **Eloquent ORM**: Simplifies database interactions with an intuitive syntax.  **Robust Security**: Protects against common vulnerabilities such as SQL injection, XSS, and CSRF.  **Scalability**: Suitable for both small and large-scale projects.  **Community and Ecosystem**: Large and active community with extensive documentation and third-party packages.  **Cost-Effective**: Pre-built components and tools reduce development time and costs. | **Performance Overhead**: The middleware pipeline and various components can slow down execution speed compared to leaner, micro-frameworks or custom-built solutions[[2]](https://courseya.com/disadvantages-of-laravel/).High-traffic applications may require extensive optimization to achieve optimal performance.  **Frequent Updates:** Laravel's active development community leads to frequent updates. While updates are essential for security and new features, they can pose challenges for projects that prioritize stability and continuity[[1]](https://www.sinelogix.com/laravel-disadvantages/).  **Scalability Challenges**: While Laravel can handle large-scale applications, it may require additional optimization and architectural considerations to scale effectively[[1]](https://www.sinelogix.com/laravel-disadvantages/). | **Invoice Ninja**: An open-source invoicing solution[[10]](https://infostride.com/sites-built-with-laravel/).  **Alison**: An e-learning platform offering online courses[[10]](https://infostride.com/sites-built-with-laravel/).  **Neighborhood Lender**: A financial services platform[[11]](https://codetheorem.co/blogs/websites-built-with-laravel-framework/). |

From the above, Laravel/PHP stands out as a strong choice for the development of the proposed solution. The key advantages which stand out in the context of this project are:

* built-in authentication and role-based access control (since the proposed app will behave differently for consumers vs. for repair specialists)
* comprehensive documentation: this will be vital in aiding developer understanding while creating the solution
* scalability: high-traffic considerations will not be an immediate concern for this project, but if the final product is successful it may lead to higher user numbers and roll-out to a larger audience
* MVC architecture will allow us to build a well-organised code base easily favouring modularity, maintainability and extensibility

### **Advantages Over Other Frameworks**

|  |  |
| --- | --- |
| **vs. React/Angular/Vue** | Laravel provides a full-stack solution with built-in backend capabilities, whereas React, Angular, and Vue are primarily frontend frameworks that require additional backend setup. |
| **vs. Django/Ruby on Rails** | Laravel’s syntax and structure are often considered more intuitive and easier to learn. Additionally, PHP is widely supported across various hosting environments, making deployment easier. |
| **vs. Symfony/CodeIgniter** | Laravel offers a more comprehensive set of features out-of-the-box, including built-in authentication, Eloquent ORM, and a powerful templating engine (Blade). |

The disadvantages of Laravel (outlined above) are not significant enough to deter us from using this framework. The performance overhead is not a major concern for this kind of website and is a minor concern, while any potential scalability issues would only occur if the developed solution becomes very popular and widely-adopted (which would likely take a long time) – even if this were to happen in the future, it should be welcomed as an opportunity for optimisation rather than being viewed as a constraint. Similarly, new versions of Laravel should be welcomed rather than avoided and should not deter us from using this framework.

# User Interface Design

User interface (UI) design is paramount for the success of any online platform, and very important for a website like an online marketplace which is constantly open to potential new users. Capturing and retaining user attention is a significant challenge: a well-crafted UI is a critical factor directly impacting user engagement, conversion rates, and ultimately, the platform's overall success. For the application being developed in this project, the UI must effectively guide users through the process of describing repair needs, selecting an appropriate service provider, and managing the repair order. Key design principles to be considered include intuitive navigation, clear visual hierarchy, consistent branding, and accessible design to accommodate diverse user needs. The sections below detail the UI design choices implemented in this project in order to provide a pleasant and effective user experience.

## **Design Process**

### **Prototyping**

In the first instance, [Figma](https://www.figma.com/) was used as a tool to create mock-ups the different pages that make up the application (see [Appendix 1](#_Appendix_1:_Original)). The original designs allowed features and processes to be identified, aiding the construction of a data model for the application. The final UI design varies significantly from the Figma mock-ups, however: the mock-ups only loosely relied on HTML/CSS design standards and also suggested the need for unnecessary technical complexity (e.g. implying nested window management where this was not required). User feedback along the way also favoured a more simplistic design (this is discussed further below).

### **Investigation/examination of similar solutions**

The [requirements elicitation](#_Requirements_elicitation) section mentions some of the key functionality/features from existing websites operating in a similar domain that this project aimed to incorporate (with some adaptation).

Some long-established and extremely popular websites from adjacent domains (e.g. eBay, Amazon) were examined, however the overhead in creating a ‘clone’ of these websites – which have been established for over 2 decades, and are supported by huge budgets and teams of developers – was deemed to be prohibitive to the completion of this project. Instead, it was considered more realistic to emulate many of the features of more modern, simplistic marketplace-style website such as the below examples:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Figure 1: [gumtree.com](https://www.gumtree.com/for-sale) listings page | Figure 2: [donedeal.ie](https://www.donedeal.ie/all) listings page | Figure 3: [marktplaats.nl](https://www.marktplaats.nl/) listings page |

These marketplace websites offer most of the required features identified in the [requirements elicitation](#_Requirements_elicitation) section, using simple card-based UI design without complicated animations, overly-elaborate colour schemes and other features that more mature websites with massive user bases have been able to incorporate. They all include an upper navbar that is visible throughout the user experience, with relevant search filters on the left of the page where appropriate, and greater detail displayed when individual listed items are selected. Individual listings pages contain many options, and not all of these are incorporated into the RepairMart site (for example, sharing a listing via WhatsApp, Twitter, etc. and showing breadcrumb navigation were not deemed to be key requirements, and user feedback during the design process did not highlight these functionalities – although they could be included as possible future developments). The design also took into account the need to avoid having too many buttons/options on screen at one time (Thorne, 2021).

### **Final Design**

Having considered the above, this project has been designed in a more simplistic manner using card-based presentations with paginated results and a basic colour scheme which presents the necessary information clearly, concisely and intuitively. An analogous design for a car-selling marketplace-style website (adapted from a programming course offered at [www.thecodeholic.com](http://www.thecodeholic.com)) has been chosen and adapted for the purposes of this project. The car-selling website can be found at [www.grabacar.xyz](http://www.grabacar.xyz).

|  |
| --- |
|  |
| Figure 4: Home Page from www.grabacar.xyz |

The HTML and CSS template for the above website has been made publicly available (Sekhniashvili, 2025), and it has been adapted for the purposes of this project. It uses the [Ubuntu font](https://design.ubuntu.com/font/), which is supported by positive experiences of using the [Ubuntu operating system](https://ubuntu.com/): a modern, sharp typeface which has advantages over other options in terms of clarity (Rodriguez, 2019) and has been made open source by the creator (Beckwith, 2023). The site’s main colour palette uses an orange-based option, chosen for its attention-grabbing quality (McLean, 2023) and its association with affordability (Raj, 2024) – which is obviously relatable for an online marketplace aimed towards reducing the cost of product repairs.

The UI design process also involved gathering feedback from a selection of volunteers along the way. Feedback was given on existing developments/layouts presented to potential users (and was therefore reactive rather than prescriptive). Comments were gathered on the site’s user friendliness, intuitiveness and overall appeal and suggestions were, within reason, incorporated – although pragmatism was needed due to technical and time-based constraints. This is, however, a normal part of the design process (Craig, n.d.).

|  |  |
| --- | --- |
|  |  |
| Figure 5: RepairMart logo & landing page | Figure 6: RepairMart user dashboard and signup/login pages |

The landing page uses a [hero slider](https://www.npmjs.com/package/hero-slider) presentation with slides to signal the site’s purpose to the 2 main user categories (customers and repair specialists), and relevant links to sign up/create a new listing. The slider contains a minimalist diagram showing the use of a soldering iron to repair a circuit board – capturing the overall idea of the site. A logo for the site generated by the Opera browser’s [Aria AI](https://www.opera.com/features/browser-ai) has been incorporated into the navbar, and also contains a basic representation of the site’s purpose in the form of various electronics items. Both the slider image and the site’s logo complement the overall colour scheme and simplistic visual design.

The navbar contains Login and Signup links (for users browsing as guests) and functional links in a dropdown menu (see Figure 6: RepairMart user dashboard and signup/login pages, top-right) for logged-in users. The ‘Search Listings’ and ‘Add new Listing’ links remain in the navbar for all users (although a user must first create an account and assign the ‘customer’ role to themselves before being allowed to proceed with creating a new listing).

|  |  |
| --- | --- |
|  |  |
| Figure 7: RepairMart sign-in with Google / password reset form | Figure 8: RepairMart account management |

The signup page (Figure 6: RepairMart user dashboard and signup/login pages) allows users to create an account with the site using their Google or Facebook account and sign in to the site using one of those accounts. Upon creating an account via one of these methods, the user will be directed to their account management page where they need to choose at least one role (customer or repair specialist) and enter address details before they are allowed to create listings or quotes in the site.

|  |  |
| --- | --- |
|  |  |
| Figure 9: RepairMart search results/watchlist view | Figure 10: Individual listing details |

As shown in Figure 5: RepairMart logo & landing page, the lower part of the landing page also contains a display of recent listings posted in the app, and a search form which takes users to the search results page (Figure 9: RepairMart search results/watchlist view).

Clicking on a listing card, either from the landing page or from the search results page, takes users to the individual listing details (Figure 10: Individual listing details) where they can see full details of the listing: title, description, published date, expiry date, manufacturer, customer budget/currency, product categories (of which up to 3 are allowed per listing), a link to the customer’s profile page, and role-dependent action buttons: here, customers may edit their own listing or delete it, or view a summary of quotes that have been received for it, and repair specialist users may send a message to the customer (e.g. if they need clarification on some details) or submit a repair quote to the customer for the listing being viewed and/or view a list of any other quotes that they have previously submitted for the same listing.

Users also have the option to add a listing to their Watchlist view, by clicking on the star icon – either on the listing’s card in the search results page, or on the individual listing details page. This feature allows quick navigation when returning to a listing (e.g. if a repair specialist needs to gather information before submitting a repair quote).

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Figure 11: Customer listings summary |  | Figure 12: Quotes/Orders summary pages |

Customers can view a summary of listings that they have created in the site (Figure 11: Customer listings summary) and both customers and specialists have summary views for Quotes and Orders (Figure 12: Quotes/Orders summary pages) which show the main details for each record and include some contextual links:

* to view the listing/order/quote
* to edit/delete the record
* to relist (in the case of an expired listing)
* to view a summary of quotes received for a specific listing
* to navigate quickly to the listing associated with a quote/order record
* to view the profile of a customer/repair specialist who has been assigned to an order

The Quotes and Orders summary pages have specific tabs for the ‘customer’ and ‘repair specialist’ roles (Quotes Received/Quotes Submitted and Customer Orders/Specialist Orders). In the case where a user has one role, they will only see one of these tabs, but if they have assigned both roles to their account, they will see both tabs as indicated in Figure 12: Quotes/Orders summary pages.

|  |  |
| --- | --- |
|  |  |
| Figure 13: Listing creation | Figure 14: Quote creation |

The Listing and Quote creation screens allow users to enter details for the records they are creating, and contain validation for inputs (Figure 13: Listing creation) and the ability to upload attachments (which are also validated according to their size and MIME type). When creating a quote, a user will see brief summary details for the associated listing and customer at the top of the page.

|  |  |
| --- | --- |
|  |  |
| Figure 15: Order creation | Figure 16: Order view including comment history with status updates |

The order creation screen follows the design of the quote creation screen (Figure 14: Quote creation), listing summary details of the related listing/quote and allowing the user to add attachments and a comment when creating an order.

The order view screen also has a similar layout, and contains a section at the bottom giving context-specific actions to the customer and repair specialist who are assigned to the order. The order is initially created with a status of ‘Created’ and specific status updates are permitted to one or other party according to the current status (as can be seen in Figure 16: Order view including comment history with status updates).

Upon completion of an order, the status will be marked as ‘Closed’ and at this point both the customer and the repair specialist can add a feedback rating and comment to it. The feedback ratings for customers and repair specialists can then be seen on their profile page (visible to other logged-in users) – see Figure 17: User profile view. This is useful to give assurance to future potential customers/specialists about the other party with whom they are dealing.

|  |  |
| --- | --- |
|  |  |
| Figure 17: User profile view | Figure 18: User profile search |

In addition to feedback ratings, the user profile view contains summary information about their location, the role(s) assigned to their account, and their activity in the site (in terms of listings/quotes/orders – shown as appropriate according to their role).

An admin-only view (Figure 18: User profile search) exists for administrator users to search for specific users in the site and edit/delete their profiles, if necessary, or message them through the site’s messaging facility.

|  |  |
| --- | --- |
|  |  |
| Figure 19: Compose message/Read message view | Figure 20: My Messages view |

Individual message compose/view screens are shown above with links to the sender profile (for the recipient) and the listing/quote (for the sender) in cases where the user has viewed a listing or a quote and used the ‘send message’ option in relation to it.

Each user has a ‘My Messages’ view (Figure 20: View Message / My Messages view) where messages sent within the site are shown (separated by tabs into received/sent items, and shown with a read/unread status). Note: these are not always the same as external email notifications that are sent by the site, although there is some overlap. Not every external email notification will generate an internal message (the rationale behind external email notification is discussed later in this report).

# Architecture Design

The RepairMart project has been developed in Laravel and consists of:

* Front-end user interface
* Back-end database framework
* File storage system

The Laravel codebase interacts with these different parts of the architecture to authorise, validate and facilitate the different user actions and perform CRUD operations in the back-end database.

The following diagram outlines the high-level system architecture. The choices made in relation to architecture options are briefly discussed below.

|  |
| --- |
|  |
| Figure 21: RepairMart system architecture |

## **Laravel development framework**

[Laravel 11](https://laravel.com/docs/11.x/releases) was selected as the primary framework for developing the RepairMart application, due to several key advantages which it offers in relation to this type of project.

Laravel's elegant syntax and robust structure is well-suited to the development process, allowing efficient implementation of several essential features:

* Eloquent ORM (object-relational mapping) permits easy management of the relationships between different entities in the database (users, listings, quotes, orders and emails), and greatly simplifies the process of querying and updating database tables, optimising joins, filters and other database operations while using simple and logical syntax (Shukla, 2023)
* Migrations, Factories and Seeders are easy to define and run, and allow efficient database creation, population and version control (Gacovski, 2025) while also facilitating transferability between different environments (e.g. dev/test/prod) (Ditya, The Power of Laravel Factories: Efficient Database Seeding for Testing and Development, 2025) and deployment on a range of SQL database management systems (Dev, 2023)
* User authentication is critical for the RepairMart application, and greatly simplified in Laravel (Barot, 2025)
* Role-based permission management is also an important area of focus for the application, as the different actions allowed for customers and repair specialists need to be clearly separated. Laravel easily facilitates this (Katariya, 2025)

Its built-in routing, templating and database interaction were able to streamline the coding process, reducing development time and effort. Furthermore, Laravel's MVC (Model-View-Controller) architecture promotes a clean and organised codebase, enhancing maintainability and scalability. The framework’s extensive documentation and large, active community also proved to be an extremely valuable resource during the development process.

Although Laravel has some potential disadvantages, such as a slightly steeper learning curve compared to other frameworks and potential performance limitations at extremely large scales, these were not critical concerns for this project, given its current size and scope. The benefits of Laravel's structure, features, and community support far outweighed these minor drawbacks, making it a strong choice for developing the RepairMart application.

The entirety of the application architecture has been hosted locally on a Windows 11 machine: this is not a problem for development purposes, but other options need to be explored when moving towards a production release. <https://thecodeholic.teachable.com/courses/laravel-11-for-beginners/lectures/57190688>

## **Front end**

Laravel's Blade templating engine was chosen to develop the application’s front end. Unlike JavaScript frameworks such as React or Vue (which require separate build processes, API endpoints, and state management libraries), Blade templates operate within the same ecosystem as the application logic, eliminating the complexity of maintaining two separate codebases and the overhead of JSON serialisation/de-serialisation between the front and back ends (Gomez, 2024). For applications like the one being developed in this project, where user interfaces need to reflect database relationships and business logic directly, this unified approach significantly streamlines development and maintenance.

Blade templates allow an extension of PHP's capabilities without sacrificing its power or familiarity: control structures (@if, @foreach, @forelse) combined with template inheritance (@extends, @section, @include) allow interfaces to be built with minimal boilerplate. Blade also makes it easier to implement conditional rendering (e.g. displaying different action buttons based on user roles, listing status, etc.). A close coupling between data and presentation is especially valuable for applications where UI decisions frequently depend on the back-end state and authorisation rules (as is the case with this application, particularly as regards user and role authorisation). Enhanced Blade features in Laravel 11, such as components and slots, also provide many of the composition benefits of JavaScript frameworks without their conceptual overhead (e.g. virtual DOM and state management).

Since Blade involves server-rendered HTML, it delivers faster initial page loads compared to JavaScript frameworks that may require downloading, parsing, and executing significant amounts of JavaScript code before displaying content. Client-side frameworks are ideal for creating highly interactive single-page applications, but in the case of this application and the planned functionality, such a high degree of interactivity is not foreseen and Blade's server-rendering approach is more appropriate.

Security is another area where Blade has an advantage over JavaScript framework alternatives. Blade automatically escapes output to prevent XSS attacks, and Laravel's CSRF protection seamlessly integrates with Blade forms (Sling Academy, 2024). The server-rendered nature of Blade applications also means that critical business logic remains on the server, protected from client-side manipulation. Authentication and authorisation checks can be performed before any sensitive data is exposed via the front end, providing a robust security model that doesn't rely on client-side validation.

## **File storage**

Best practice has been followed for file management (e.g. images/videos added as attachments for quotes, listings and orders in the system) (Corey, 2021), which means that these files are not stored as Varbinary in the database itself, but in disk storage at a location which is accessible to the Laravel code via a folder link.

|  |
| --- |
|  |
| Figure 22: RepairMart file storage |

This configuration is easily transferable when deploying the system in different environments, and avoids several costs associated with the storage of Varbinary files in the database itself – namely, much larger database sizes and increased wait times on database backup and restore. Although this would be unlikely to cause problems for the application’s development, it could prove to be very significant for a production application with a large number of users and listings. Additionally, if/when such a large user base is acquired, the scaling of file storage space is much cheaper when using in-built server file management systems vs. Varbinary database storage.

## **Database development**

### **UML diagrams**

In the early phases of developing the application, UML (Unified Modelling Language) diagrams were valuable for visualising and planning the system's architecture. For this project, Use Case and Sequence diagrams were produced:

* Use Case diagrams helped to define user interactions and system functionalities, clarifying requirements from the perspectives of customers, repair specialists, and administrators.
* A Sequence diagram mapped out the interactions between different system components, such as the website, database, and email server, to identify potential bottlenecks and ensure smooth operational flows.

Diagrams were not needed for every workflow (e.g. the workflow for sending a message to another user in the system was simple enough to implement without one) but the main characteristics of the account creation and repair request listing processes are detailed in UML Use Case and Sequence diagrams in [Appendix 2](#_Appendix_2:_UML).

### **Entity-Relationship diagrams**

in parallel with the UML diagrams, a discovery exercise was carried out to identify the key data points from the [original UI designs](#_Appendix_1:_Original). The results from this exercise were used to create a preliminary database model (see below), and from this it was possible to create a reasonably accurate preliminary entity-relationship diagram. This was deemed sufficient to start database development work, and as a result it was decided that creating UML Class diagrams would not be necessary for this project.

|  |
| --- |
|  |
| Figure 23: Preliminary Entity-Relationship diagram |

The preliminary ERD was created in Microsoft SQL Server: using the data points identified in the discovery exercise, tables and stored procedures were created and tested, and the diagram shows the result of table definitions.

Following subsequent development in Laravel and changes to the SQLite database that was connected to the code base and used for development, the final ERD (below) contained several changes. Many of these related to naming conventions and data types, due to the way that these are handled in Laravel. For example:

* the camel case convention used for table names in the above diagram is replaced by snake case names in Laravel)
* default date columns in Laravel are created\_at / updated\_at instead of those used in the preliminary ERD above
* Laravel uses an “id” column by default as the primary key for any new table that is created, whereas the preliminary ERD diagram has used a different approach

Additionally, when database tables are created via Laravel migration files, datatypes are applied so that they are agnostic to the SQL database option that is being used – this means that, as long as a Laravel migration is being used to create a table (rather than direct execution of a database statement), column definitions are not tied to a particular SQL DBMS provider (Verma, 2025) and their definitions may not be the same as those prescribed in situations such as that reflected in the preliminary ERD.

Apart from the aforementioned differences between the preliminary and final ERDs, additional tables have been added where needed. These additions are discussed below in further detail.

|  |
| --- |
|  |
| Figure 24: Final Entity-Relationship diagram |

### Development & Testing

When developing an application in Laravel, a number of SQL database management systems are supported (Laravel, n.d.). For the purposes of the RepairMart application, SQLite was the option selected.

As a file-based relational database system, SQLite eliminates the need for a separate database server installation, significantly simplifying the development setup process. This self-contained nature means that repositories can be cloned and start working quickly without complex database configuration, reducing environment-related issues across different development machines (Team Gyata, 2024).

Performance is another benefit: for typical development workloads with small to moderate data volumes, SQLite delivers excellent response times with minimal resource consumption. The entire database exists as a single file on disk, resulting in reduced I/O operations compared to client-server database systems. This translates to faster testing cycles and a more responsive development experience, which is particularly valuable when frequently running migrations and seeders during iterative development.

A crucial consideration was the fact that SQLite also integrates very well with Laravel's testing framework, enabling fully isolated tests with in-memory databases. This dramatically speeds up test execution by eliminating disk I/O operations, while ensuring no interference between test cases (SQLite, n.d.). Laravel's database migrations and seeders work seamlessly with SQLite, and facilitate a consistent testing approach.

The combination of SQLite's transaction support and Laravel's database transaction testing helpers creates a powerful testing ecosystem. Test cases can be wrapped in database transactions that are automatically rolled back after each test, maintaining database integrity without manual cleanup – which is particularly valuable when testing complex features that touch multiple related database tables, since SQLite provides full support for Laravel's relationships, eager loading, and query building (HiBit, 2024).

### Considerations for Production transition

While SQLite excels in development environments, transitioning to production requires careful consideration. Laravel's database abstraction layer generally handles the switch between SQLite and production databases like MySQL or PostgreSQL smoothly, but platform-specific SQL features need to be carefully considered. As mentioned earlier, using Laravel's schema builder and query builder instead of raw SQL allows applications to maintain agnosticism within supported SQL varieties, however a migration to a different database management system has not been carried out as part of this project. Although there may be some issues to overcome, efforts have been made during development to maintain the agnostic approach, and it is felt that the above-detailed advantages offered by SQLite outweigh the disadvantages that may be posed by minor differences between development and production environments.

## Database population

### Static data

Before starting full solution development, a number of static data tables needed to be populated in the database. This was initially done using SQL insert scripts that were created in the MS SQL Server environment where the preliminary ERD was created – however these scripts were later replaced with proper Laravel migration files including the relevant data. The static data tables are described below.

* **countries & currencies**

A list of ISO country & currency codes and names, retrieved from [iso.org](https://www.iso.org/iso-4217-currency-codes.html).

* **manufacturers**

A list of electronic goods manufacturers: these are used to filter product searches on websites such as Amazon and eBay, and will be incorporated into listing details on the RepairMart application. The list has been gathered from [Wikipedia](https://en.wikipedia.org/wiki/List_of_electronics_brands), although it has been filtered to exclude many less well-known manufacturers. An ‘—Other-Unknown—’ value has also been included in this list, which can be selected when a manufacturer is not known or cannot be found. In such cases, it is envisaged that a data quality analysis task could be performed to examine products listed with an unknown manufacturer and potentially add new names to the list for future use.

* **products**

A product category/subcategory classification for use in repair request listings.

Consideration was given to how complicated the RepairMart product classification needed to be, and various options were examined. Since there is no know analogue at present which is fully equivalent to the solution that we intend to build with this project, no ready-made list of electronics product categories was available to copy (unlike the case of ISO countries and currencies).

In order to avoid overcomplicating both our database design and the user experience, it was decided to use a 2-tier product classification, which is a slimmed-down version of the Amazon classification found [here](https://www.asinspotlight.com/amz-categories-list-csv).

* **roles**

A fixed list of roles that can be assigned to users in the database. For almost all cases, users will have either a ‘customer’ or ‘specialist’ role, or both. These assigned roles are used to determine permissions for various activities and workflows. Additionally, an ‘admin’ role has been created and this allows access to most activities and workflows - although it is assumed that any admin user will also have access to the RepairMart database and may not need to perform all admin tasks through the front end interface.

* **feedback\_types**

Original Figma mock-ups included a detailed feedback view on user profiles, with access to individual comment history. However, given that some potential for abuse exists, and that there may also be some privacy concerns (e.g. negative feedback comments including personal information), it was decided to simplify the feedback process and display only positive/neutral/negative classifications without detailed comments.

* **listing\_statuses, quote\_statuses, order\_statuses**

The following statuses were identified for listings, quotes and orders:

***Listings***

|  |  |
| --- | --- |
| Open | For listings which have passed their published date, have not yet expired, have not been retracted by the customer, and whose customer has not yet accepted any quotes (leading to the creation of an order) |
| Closed-Expired | Listings will be marked as Closed-Expired when their listing period has expired and the customer has not retracted the listing or accepted any quotes to create an order |
| Closed-Retracted | If a customer chooses to withdraw a listing without it having expired, and without having accepted any quotes, it will be marked as Closed-Retracted |
| Closed-Order Created | When a customer chooses to accept a quote and create an order, the associated listing will be marked as Closed-Order Created |

***Quotes***

|  |  |
| --- | --- |
| Open | For quotes which have been created on Open listings. |
| Closed-Rejected | When a listing is closed for any of the reasons above, and a quote that has been submitted is not the quote that the customer has accepted for creating an order, the quote will be marked as Closed-Rejected |
| Closed-Retracted | If a repair specialist chooses to withdraw a quote without it having been accepted, it will be marked as Closed-Retracted |
| Closed-Order Created | When a customer chooses to accept a quote and create an order, the quote will be marked as Closed-Order Created |

***Orders***

|  |  |
| --- | --- |
| Created | The initial status when an order is created after a customer accepts a quote |
| Dispatched to Specialist | The customer can update the order’s status to let the repair specialist know that the product has been dispatched to them (N.B. this is not always necessary, depending on the delivery method being used) |
| Specialist Assessing | When the repair specialist receives the product, this status lets the customer know that they are assessing the defect |
| Price Adjustment Requested | If, having assessed the product, the repair specialist finds that the defect is not as described in the listing, they are able to change the order status and request a price adjustment |
| Price Adjustment Approved | If a price adjustment has been requested, it is up to the customer to approve this before the repair is carried out. |
| Price Adjustment Rejected | If the customer does not agree to adjust the price of the order, they can use this status to update it. |
| Specialist Repairing | This status confirms that the product is being repaired by the specialist. |
| Dispatched to Customer | This status confirms that the product is being returned to the customer. |
| Received by Customer | This status confirms that the product has been received by the customer. |
| Payment requested | After the customer confirms receipt, the repair specialist can update the status and request payment from the customer. |
| Payment received | This status confirms that the specialist has received payment for the product repair. |
| Closed | Once payment has been received, either the customer or the repair specialist may mark the order as closed. |
| Cancelled | If a price adjustment has been requested by the specialist, and the customer has rejected this request, the specialist may mark the order as cancelled. |

As an extension of the order\_statuses table, it was later decided to introduce an order\_status\_transitions table which defines which user (the customer or the specialist) may update the current order status, as well as defining the new order status options that they are allowed to use in their update.

* **deliverymethods**

A fixed list of delivery methods that repair specialists can choose in the system, when creating a quote in response to a repair request. These are: (1) Pick-up/Drop-off at Customer, (2) Drop-off/Pick-up at Repair Specialist, (3) Postage (tracked), and (4) Postage (untracked).

### Database relationships

Laravel model files use Eloquent ORM to manage a variety of database table relationships (One To One, One To Many, Many To Many, Has One Through, Has Many Through, etc.) and greatly facility the process of linking entities through clear definitions and syntax – also reducing the coding overhead and the risk of manual error when raw SQL queries are being written (Selvaraj, 2023).

Most relationships in the RepairMart database are One To One or One To Many, with a few Many To Many examples included where necessary. Commentary is given below on some of the more notable relationships.

#### Listing/Quote/Order

The key relationship between entities in the RepairMart application is between listings, quotes and orders. One listing can receive many quotes, but can only create one order. One order can only be linked to one quote and one listing. A listing is not required to have any quotes or orders, but quotes and orders must be associated with a listing, and orders must be associated with a quote.

#### Quote/Delivery Method

Choosing a delivery method for orders is one area which can be very complicated. When considering options and configuration to use, analogous examples such as eBay’s shipping method selection were considered, however mirroring eBay’s offering (which includes the ability to quote multiple delivery methods per listing, and pay for shipping & print out packaging labels directly from the eBay website) was deemed to be overly complex for the initial purposes of the RepairMart application.

A decision was made to simplify the process of quote creation: rather than permitting multiple delivery methods per quote submitted, allowing a variable quote amount which would be calculated based on the shipping method selected by a customer, it was decided to only allow a single delivery method per quote. In the event that a repair specialist wishes to offer multiple delivery methods to a customer, they will be able to submit more than one quote for the same listing, choosing a different delivery method for the new quote. In this case, the RepairMart system is not burdened with calculating the delivery cost for each method offered – instead, repair specialists are responsible for calculating the overall cost for their quote (including delivery fees) and quoting a single amount in the system.

#### Emails/Recipients

The database includes an emails\_recipients Many To Many table which is designed to allow multiple recipients for messages sent within the RepairMart application. The Many To Many relationship turned out to be unnecessary, as all the use cases identified for sending messages within the application only involve a single recipient. Nevertheless, it was decided to retain this relationship as it may be used in future development (e.g. when an admin user wants to send a message to multiple users).

#### Order Status Transitions

As mentioned in the [Static Data section](#_Static_data), it was necessary to introduce an extra table to control the permitted status transitions for different users (based on their role) and statuses of an order. This table uses the foreign keys: role\_id (customer or specialist), and from\_status\_id/to\_status\_id (the order statuses) to restrict the permitted updates on an order.

#### Attachments

It was originally planned to create a Many To Many relationship between attachments and listings/quotes/orders/emails. The idea behind this was to optimise storage by allowing the same attachments to be used across multiple listings/quotes/orders/emails. In the end, however, it was decided that due to the way in which Laravel saves and renames files in local storage, this more complicated configuration offered little benefit. As such, a single attachments table has been used instead, with 4 columns (listing\_id, quote\_id, order\_id and email\_id) which are used as foreign keys for the listings/quotes/orders/emails tables.

### Seeding

The Eloquent ORM relationships have been defined in Laravel model files such as the below Listing model. CRUD methods for users, listings, etc. were then included in Controller files.

|  |  |
| --- | --- |
|  |  |
| Figure 25: RepairMart Listing model file | Figure 26: RepairMart ListingController file |

These methods were used by Factory components and a DatabaseSeeder file to create test data which was then used to assist front-end development. The Factory files rely on the [static data](#_Static_data) mentioned above, and also use [Faker](https://fakerphp.org/) to create data items such as names, addresses, titles, descriptions, etc.

|  |  |
| --- | --- |
|  |  |
| Figure 27: RepairMart ListingFactory file | Figure 28: RepairMart DatabaseSeeder file |

The above-outlined method, in tandem with the Laravel migration files, allowed easy deconstruction and reassembly of the database and test data, which was important for iterative development. The same process was automated and re-used in testing, where it provided significant benefits.

## **Code base**

### Directory Structure and MVC Implementation

RepairMart is built using Laravel 11's opinionated directory structure, which follows the Model-View-Controller (MVC) architecture pattern to organise code logically and maintain separation of concerns.

The core application logic resides in the app directory, which contains several subdirectories that represent different architectural components.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Figure 29: RepairMart app directory | Figure 30: RepairMart Models folder | Figure 31: Use of scoping in the RepairMart Listing.php model |

Models like Listing.php are placed in the Models directory, and encapsulate data structures and relationships while implementing business logic through accessors, mutators, and query scopes. For instance, the Listing model defines relationships with users, quotes, and products while implementing domain-specific functionality like isExpired() and custom accessors like getExpiryDateAttribute().

|  |  |
| --- | --- |
|  |  |
| Figure 32: isExpired() function in RepairMart Listing model | Figure 33: getExpiryDateAttribute() function in RepairMart Listing model |

These models leverage Laravel's Eloquent ORM to provide an expressive interface for database operations without writing raw SQL.

|  |  |
| --- | --- |
|  |  |
| Figure 34: RepairMart Controllers folder | Figure 35: store() function in OrderController file |

Controller files in the Controllers directory handle HTTP requests and coordinate application logic, delegating responsibilities to models and returning appropriate responses. The OrderController exemplifies this by implementing request validation, authorisation checks, and complex business processes like updating related entities when orders are created (see above).

|  |  |
| --- | --- |
|  |  |
| Figure 36: Laravel’s built-in auth() (Authentication) middleware | Figure 37: Use of auth() middleware in RepairMart routing |

Laravel's middleware components provide a filtering mechanism for HTTP requests, handling concerns such as authentication and verification. This approach ensures that common request processing tasks are handled consistently across the application. For the purposes of this project, the vast majority of middleware comes in-built in Laravel 11 (see above, for example).

### Performance Optimisation

#### Caching & Scopes

Strategic use has been made of both caching and query scopes in some of the RepairMart Controller files. Although not everything has been optimised in this way, it was felt that the application’s home page, in particular, could benefit from this practice as it is the landing point for all users (and as such will see the most traffic), and as it includes a display of recent listings (which involves retrieval of information from the database), it was deemed to be a good candidate for optimisation.

|  |
| --- |
|  |
| Figure 38: Caching and scope use in the HomeController file |

RepairMart significantly reduces database load by storing the result of complex listing queries for five minutes, serving cached content to subsequent visitors without repeatedly executing resource-intensive database operations. The below images show query analysis output when the home page is initially loaded (top image) and when it is refreshed a few seconds later (bottom image) – confirming that the database load is greatly reduced as a result of this practice (just 1 query upon reload, compared to 14 with the initial load).

|  |
| --- |
|  |
| Figure 39: Query analysis output for the RepairMart home page on initial load (top) and refresh (bottom), showing the benefit of caching |

Complementing this approach is the use of Eloquent's custom query scope through Listing::active(), which encapsulates complex filtering logic for retrieving only published, open, and unexpired listings in a reusable, expressive method. This abstracts away multiple WHERE conditions into a single method call, improving code readability while ensuring consistent application of business rules throughout the application.

The implementation is further optimised through eager loading of multiple relationships (with(['country', 'customer', 'manufacturer', ...])) which prevents the N+1 query problem by prefetching associated data in a single database request rather than executing separate queries for each listing's relationships. The combination of time-based caching, expressive query scopes, and eager loading maintains excellent performance even under high traffic conditions, demonstrating clean code architecture with practical performance considerations for an optimal user experience.

#### Search Optimisation

The ListingController file uses a sophisticated approach to search optimisation through an adaptive, database-agnostic implementation of full-text search capabilities.

|  |  |
| --- | --- |
|  |  |
| Figure 40: SQL driver detection for full-text search in ListingController | Figure 41: SQLite use of full-text search and fallback to LIKE-based search |

The applyTextSearch() method detects the underlying database driver and applies the most efficient search strategy available – using FTS5 virtual tables in the case of SQLite. This design enables consistent search functionality across different deployment environments while maximising performance for each database platform. For SQLite specifically, the controller implements a robust fallback mechanism that uses LIKE-based searches when FTS (full-text search) is unavailable, ensuring search functionality remains operational even in constrained environments.

As with the home page, search results are further optimised through eager loading (with(['country', 'manufacturer', 'currency'...])) and strategic query caching, with a composite cache key derived from all search parameters. This prevents redundant database operations for identical searches within a five-minute window. Pagination is used to limit result sets, and the search function includes advanced filtering capabilities that narrow results based on multiple criteria simultaneously. This multifaceted approach show RepairMart's ability to deliver responsive, accurate search functionality regardless of the underlying infrastructure, while maintaining excellent performance.

### View Layer & Front-End Technologies

RepairMart's presentation layer is built with Laravel's Blade templating engine, with views stored in the resources/views directory.

|  |  |
| --- | --- |
|  |  |
| Figure 42: RepairMart ‘show’ blade template within resources/views/listings | Figure 43: JavaScript inclusion in RepairMart via Vite in RepairMart listings ‘create’ blade template |

Blade templates like listings/show.blade.php combine HTML markup with PHP control structures to dynamically generate content based on application state. The templates leverage Blade's component system and layout inheritance to maintain consistent styling and behaviour across the application, while reducing code duplication. JavaScript files such as listings-create-toggle-address-inputs.blade.php demonstrate how client-side interactivity is implemented with vanilla JavaScript directly embedded into Blade templates, allowing for progressive enhancement without requiring complex front-end frameworks.

The application leverages Laravel's integration with Vite for modern, efficient asset bundling, as evidenced in the create listing form (above) where specific JavaScript modules are selectively included with the @vite directive. This brings several significant advantages. Vite accelerates the development cycle by instantly reflecting changes in CSS and JavaScript without requiring a full page refresh, resulting in substantially improved developer productivity (Raed, 2023). Vite's modern build system provides automatic code splitting, efficient tree shaking to eliminate unused code, and proper handling of dependencies, providing optimised production assets that load faster for end users. Vite's native support for modern JavaScript and CSS features like ES modules and CSS preprocessors also allows cleaner, more maintainable code to be written without worrying about browser compatibility issues – as evidenced by RepairMart's semi-modular JavaScript structure, where functionality specific to certain views is separated into purpose-specific files that are bundled and optimised at build time (in addition to common overall functionality that is included in the main app.js file). This significantly enhances both developer experience during implementation and end-user experience through optimised production assets.

### Routing

To maintain a clear separation between back-end business logic and front-end presentation, RepairMart follows Laravel's convention of using route files in the routes directory to define URL endpoints and their corresponding controller actions.

|  |  |
| --- | --- |
|  |  |
| Figure 44: Main web.php route file showing import of separate logically-organised route files | Figure 45: orders.php route file showing use of ‘auth’ and ‘verified’ middleware to protect routes |

The web routes map URLs to controllers like ListingController and OrderController, serving as the entry point for all HTTP requests and providing a centralised location for registering middleware, naming routes, and organising logical groups based on functionality or authorisation requirements.

The above images show the importing of logically-organised route files into the main web.php file, as well as the use of middleware for authorisation and verification purposes.

### User Email Verification

Users must have a verified email address in order to use the RepairMart application. The EmailVerifyController uses Laravel 11's built-in email verification infrastructure in order to achieve this.

|  |
| --- |
|  |
| Figure 46: verify, notice and send methods in EmailVerificationRequest class |

The controller uses Laravel’s EmailVerificationRequest class, which automatically handles signature validation, expiration checking, and throttling protection without requiring custom implementation. This controller demonstrates Laravel's "convention over configuration" philosophy through the use of three endpoints:

1. the verify method that processes successful verifications through the framework's fulfill() method
2. a notice method that renders the verification instructions page
3. a send method that triggers verification email delivery

The controller delegates complex verification logic to Laravel's specialised request class, and focuses solely on directing application flow and providing user feedback through appropriate redirects and flash messages. The RepairMart implementation integrates with Laravel's authentication scaffolding and verification middleware, which automatically enforces verification requirements on protected routes throughout the application. This potentially-complicated configuration has been greatly simplified through using Laravel's sophisticated built-in features with minimal effort.

### Laravel Migration files

The database structure is defined through Laravel's migration system, with migration files stored in the migrations  directory. Migrations like the below example provide a version-controlled approach to database schema management, allowing for incremental changes to the database structure that can be applied, rolled back, or recreated consistently across different environments (Sharma, 2024).

|  |  |
| --- | --- |
|  |  |
| Figure 47: RepairMart migration file for the orders table, with datatypes, foreign key constraints and default value | Figure 48: RepairMart migration file for the manufacturers table, including static data population |

Each migration defines table structures with column types, constraints, and relationships, using Laravel's fluent schema builder to express database operations in a database-agnostic manner. This approach supports development with SQLite for local environments while remaining compatible with other SQL databases which may exist in different environments.

### Scheduled Jobs

Although the development environment for this project has not been continuously operating, processes have been configured to run under such circumstances, with a view to production deployment.

|  |
| --- |
|  |
| Figure 49: Custom ExpireListings class |

The ExpireListings class automates business processes through Laravel's command scheduler system. As a custom Artisan console command registered with the signature listings:expire, this component systematically identifies and processes listings that have reached the end of their visibility period, based on the expiry date chosen by the customer when the listing was created.

The command includes comprehensive error handling, transaction-based processing to ensure database integrity, fallback mechanisms for missing status configurations, and detailed logging of successful operations and exceptions. When executed, expired listings are transitioned to a "Closed-Expired" status while simultaneously updating all associated open quotes to "Closed-Rejected", maintaining consistency across related data models. The implementation sends personalised email notifications to listing owners and quote providers, and provides command-line feedback for monitoring. The command has been set up to run automatically every day via Laravel's task scheduler (in App\Console\Kernel.php), but can also be manually run through the “php artisan listings:expire” command.

### User Messaging

The application uses 2 different kinds of message notification.

The first of these is an internal messaging system hosted in the RepairMart database, which is for user-to-user messaging inside the application. This system can be accessed from (1) listing views, (2) quote views, and (3) user profile views, and allows specialists and customers to send questions in relation to specific listings and quotes. In these cases, the message subject line is pre-filled with the relevant information (e.g. the listing title – see below).

|  |
| --- |
|  |
| Figure 50: EmailController file containing pre-population of message subjects |

The second message notification system involves sending actual emails outside the RepairMart database, using SMTP configuration. The intention is for these notifications to advise users about events in the RepairMart system, and direct them to the relevant part of the application, while reserving more extensive details for the internal system. Not every action will generate an external email – for example, it has not been considered necessary to send an external confirmation email to a customer after they have created a listing (since they can already see a success message on screen when they do so). When a customer’s listing expires, however, they will not necessarily be aware of the event and in this case an external notification is sent.

The user signup process also involves external email notifications, since it requires verification of the user email address. The email verification process is handled through adaptation of in-built Laravel functionality, while other external notifications are custom components created specifically for RepairMart’s purposes. These make use of Laravel's [mailable](https://laravel.com/docs/11.x/mail) architecture through a dedicated Mail directory containing classes such as ListingExpired.

|  |  |
| --- | --- |
|  |  |
| Figure 51: EmailController file containing pre-population of message subjects | Figure 52: QuoteController file using send (rather than queue) for external email notifications |

These classes provide a link between data models and email templates. Each mailable class receives relevant model instances (e.g. Listing/User) through the constructor and defines presentation details through methods such as envelope() and content(). These classes are linked to corresponding Blade templates in the mail directory through the view parameter, allowing the templates to focus purely on presentation while the mailable classes handle data preparation, subject line composition, and delivery configuration.

It was originally planned to follow recommended practice by implementing the ShouldQueue interface and using the Queueable trait in Mail classes, to ensure asynchronous email processing and maintain application responsiveness (Suhail, 2023). Since this responsiveness is not an area of particular concern in the development environment, the practice has not been adhered to and instead the ‘send’ command is used (see above). However, this is something that requires an update before any potential production deployment.

### Factories & Seeders

Supporting the database layer are seeders and factories in the seeders/factories directories, which are discussed [above](#_Seeding). These generate test data for the development and testing environments, with realistic model instances with randomised attributes, relationships & states that mirror production data.

### Tests

The application's tests in the tests directory leverage the factories & seeders to create isolated test environments, with feature tests like ListingTest.php verifying that controllers, models, and views work together correctly to implement business requirements.

|  |
| --- |
|  |
| Figure 53: RepairMart ListingTest file with route access and view content tests |

This comprehensive approach ensures that the application's functionality remains consistent and reliable across updates and deployments.

# Testing

This section details the testing process undertaken to ensure the stability, security, accuracy, and performance of the RepairMart application. This is vital for ensuring the quality of the application, and the absence of defects (Zaidi, 2025).

For the RepairMart application a comprehensive approach to testing has been adopted, combining both manual and automated testing in order to identify and resolve potential issues across various aspects of the application.

### Manual Testing

As the development of this application was somewhat experimental (due to the use of a new framework, i.e. Laravel), it was decided that manual testing would be carried out throughout the stages of code development. This played an important role in improving the user experience and identifying combinations of circumstances that had not previously been considered.

Testing focused not only on confirming the absence of defects, but also on improving usability, functionality, and overall application flow from the perspective of a typical user, and feedback was sought and received on this from volunteer users who reviewed certain aspects of the application and gave informal feedback (and in some cases advised on best practice and directions to proceed in). Specific attention was paid to ensuring a seamless and intuitive experience across different user roles and scenarios, and exploratory testing was also used to uncover unexpected behaviours or vulnerabilities.

### Automated Testing

To ensure comprehensive and repeatable testing, automated tests were implemented using [Pest](https://pestphp.com/), a popular testing framework for PHP. Pest’s advantages include elegant syntax, minimal boilerplate code, and easy-to-understand output (Ditya, Mastering Testing in Laravel with Pest PHP: A Comprehensive Guide, 2025).

#### Feature Testing Architecture & Approach

A robust testing strategy has been implemented through Laravel's feature testing framework, using comprehensive test classes. Tests follow the "arrange-act-assert" pattern: each test sets up the required data and environment first, then performs actions that mirror real user interactions and verifies that the application responds correctly to these.

|  |
| --- |
|  |
| Figure 54: Pest’s it() syntax within tests/Feature/AuthTest.php |

The it() function syntax shown above is used by Pest, providing a more expressive, human-readable definition format compared to traditional PHPUnit tests (Lamé, 2021). This enhances readability and maintenance, allowing tests to serve as self-sufficient documentation of the application's expected behaviour.

#### Content and Response Validation

Tests assert the correct responses (with codes and messages) for given actions within the application (see below).

|  |  |
| --- | --- |
|  |  |
| Figure 55: Testing for correct response code and message within ListingTest | Figure 56: ListingTest showing detailed assertSeeInOrder test for Listing page |

The tests also go beyond status code validation, examining the content of responses and ensuring that both functionality and user experience meet specifications. Multiple assertion methods are used to verify that pages contain the expected elements in the expected order – including navigation links, form inputs, and third-party authentication options. The use of assertSee() and assertSeeInOrder() with HTML fragments confirms that the markup structure is correct for essential interface elements. This ensures that template changes don't inadvertently break critical user flows, acting as a form of regression testing for the application's interface.

#### Data Integrity & Business Rule Testing

Feature tests systematically validate that business rules are correctly enforced across the application. For example, tests defined in ListingTest.php verify the entire listing lifecycle, ensuring that listings are created & expired, and can be viewed & updated, according to business requirements.

|  |  |
| --- | --- |
|  |  |
| Figure 57: assertDatabaseHas() check on data integrity | Figure 58: positive/negative tests for authorised users in QuoteTest.php |

Tests also verify that expired listings are appropriately flagged, that attachments can be properly manipulated, and that only authorised users can perform sensitive operations.

This approach ensures that the application's core functionality remains reliable even as the application evolves. By testing against carefully constructed fixtures and factory-generated data, these tests validate that the application handles diverse data scenarios correctly.

#### Testing HTTP Request Validation and Form Processing

The application's feature tests methodically verify that form validation rules work correctly across all endpoints. Specific tests are constructed in order to submit deliberately invalid data and check that the application correctly identifies and reports each validation error. These tests confirm that the application not only rejects invalid inputs but does so with appropriate error messages directed to the relevant form fields. This serves as a critical safeguard against data corruption and security vulnerabilities that might arise from improperly validated inputs. The comprehensive coverage of validation rules in the test suite ensures that users receive clear guidance when completing forms throughout the application.

|  |
| --- |
|  |
| Figure 59: Form processing test for errors in QuoteTest.php |

#### Authentication and Authorisation Testing

The AuthTest.php file demonstrates a thorough approach to security testing, covering the complete authentication lifecycle from registration through login, password reset, and logout. A list of the tests carried out in this file is below.

|  |
| --- |
|  |
| Figure 60: AuthTest output |

These tests verify not only the ‘happy paths’ where users provide correct credentials, but also test edge cases and security boundaries where the system needs to reject invalid inputs. The tests also validate that appropriate feedback reaches users, ensuring the user experience remains intuitive even during authentication failures.

#### Home Page Testing

HomeTest.php is a simpler test configuration, confirming the absence and presence of listing details under the correct expected circumstances. The positive check for listings on the home page asserts that the correct number of items appears after the database has been seeded, with the correct corresponding information.

|  |
| --- |
|  |
| Figure 61: HomeTest output |

More thorough testing of this page can be implemented (for example to ensure that all the correct form elements are displayed for the listing search functionality, and that the correct routes can be accessed), however this is one area where time constraints restricted the thoroughness of testing – and where manual confirmation was carried out without documentation of the results.

#### User Profile Testing

ProfileTest.php tests user profile functionality, organised into three distinct testing domains: profile management, profile display, and profile search. The defined tests verify access control rules by confirming that unauthenticated users are properly redirected to the login page, while authenticated users can access their own profiles and view other users' profiles. Each test sets up the necessary user accounts with appropriate roles, simulating user authentication where needed, making HTTP requests to profile-related routes, and checking the response status codes. Profile search tests specifically validate role-based authorisation, ensuring that only admin users can access the user search functionality while regular users receive appropriate permission denial responses.

|  |
| --- |
|  |
| Figure 62: ProfileTest output |

While basic access controls are thoroughly covered, there is room for improvement in these tests. The tests lack assertions on the actual content displayed in profile pages, and testing of profile update functionality has not yet been included to ensure that users can modify their details with enforcement of appropriate data validation rules.

#### Watchlist Testing

WatchlistTest.php covers simple authorisation tests for access to the ‘My Watchlist’ page with guest and authenticated users, checking that the correct responses are received and the correct routes are accessed under two scenarios. As a less critical area of the application, less scrutiny was given to testing this workflow and there is room for expending the testing scope to include more detailed checks – not just on the Watchlist page itself, but also on the ability to add/remove items from individual listings pages and search results views.

|  |
| --- |
|  |
| Figure 63: WatchlistTest output |

#### Listings Testing

ListingTest.php uses a comprehensive testing strategy that covers the entire lifecycle of the Listing entity - the core business object of the RepairMart platform. The test suite methodically verifies each operation users can perform on listings, starting with creation permissions, proceeding through modification workflows, and ending with deletion safeguards. The tests map directly to user workflows, ensuring that all aspects of listing management are thoroughly tested.

The test suite contains dedicated tests to verify enforcement of appropriate access controls at every endpoint, so that only verified customers can create and manage listings, while specialists and guest users are appropriately restricted and unauthorised data manipulation is prevented.

The suite also includes extensive testing of data validation rules through tests like it('does not allow a Listing to be created with empty data') and it('does not allow a Listing to be created with invalid data'), meticulously verifying that the application enforces all validation rules and correctly identifies each validation issue. This approach ensures that listings cannot be created or updated with invalid manufacturer IDs, improper budget formats, non-existent countries, malformed phone numbers, or invalid product relationships. These validation rules ensure that data stored in the system maintains quality and consistency.

Sophisticated testing of complex business operations like relisting expired listings and managing attachments is also covered. The it('can relist an expired listing with a new published\_at date') test verifies the relisting workflow, confirming that expired listings can be brought back into active status with appropriate metadata updates. Similarly, attachment management tests verify that users can add, remove, and reorder attachments, with thorough verification of database state after each operation.

|  |
| --- |
|  |
| Figure 64: ListingTest output |

While the ListingTest suite provides excellent coverage of CRUD operations and permissions, some areas could be enhanced in future iterations – for example, more edge case testing for listing expiry. The suite could also benefit from more thorough testing of the search & filtering functionality (especially with entries containing special characters). Prior to any production deployment, performance testing for larger listing and attachment volumes could be used to ensure that the application maintains responsiveness under heavy use conditions.

#### Quotes Testing

QuoteTest.php again uses a highly detailed approach, with over 1,000 lines of testing code meticulously validating every aspect of the quoting system. The test suite follows a logical progression through the entire quote lifecycle, from creation through viewing, editing, attachment management, and deletion. The tests have very broad coverage, ensuring that both successful operations and appropriate error handling are verified for each endpoint and action.

The quote system uses role-based access controls, and the test suite systematically validates permissions for every operation. Tests verify that specialists can create quotes but customers cannot, that users can only edit and delete their own quotes, and that quote visibility is properly restricted to the involved customer/specialist parties. This permission testing is especially important for a marketplace application like RepairMart, where privacy and transaction integrity are essential to building trust. The tests also verify appropriate redirects and error messages for unauthorised actions, ensuring the application clearly communicates permission boundaries.

Quote creation and editing involve validation requirements across multiple data types, and the test suite verifies these through negative test cases. Validation is carried out for each field type to confirm that the application rejects invalid currency IDs, negative amounts and turnaround times, malformed phone numbers, and non-existent country references. By testing empty and invalid values, the suite ensures that the application's robust data validation, which is essential for maintaining data integrity across the application.

Comprehensive business operations verification is also included, with tests for attachment management and checks for appropriate database changes and proper cleanup on deletion. Important checks also verify the prevention of requests to delete quotes that aren't in the "Open" state – vital to ensuring business process integrity.

|  |
| --- |
|  |
| Figure 65: QuoteTest output |

While core functionality is extensively covered, potential improvements include performance testing for scenarios with large numbers of quotes or attachments (a scenario also identified above for the ListingTest suite). Other aspects to consider may depend on the integration of additional features into the application - such as currency conversions and localised content, or changing quote price calculation to incorporate specific delivery prices into the system. Adding tests for quotes nearing expiry would further enhance reliability.

#### Orders Testing

The OrderTest.php suite contains extensive verification of the presentation layer, with assertSeeInOrder checks that validate the presence and the exact sequence of content elements on the main order management page and the ‘My Orders’ page. The test suite also covers the many relationships involved in orders, verifying that quotes, listings, customers, and specialists are all properly connected and accessible from the order context.

Authorisation testing is central to this suite, and it contains comprehensive verification that each user type can only perform appropriate actions based on their role and relationship to the order. This is a critical aspect of the RepairMart application – although no financial transactions are performed within the system, it is expected that invoices and payment details may be shared within the context of updating an order. This is obviously a sensitive area.

The order system contains specific rules about which status transitions are permitted by which roles, however the test suite does not yet incorporate these business rules. The creation of tests to automatically verify the correct behaviour of these transitions involves complex set-up, and it has not been possible to conclude this process as at the time of writing this report. Instead, manual end-to-end testing of the order lifecycle has been carried out to ensure that orders follow the expected business process and both parties understand the current status at each stage of the transaction. Manual testing screenshots can be found in [Appendix 3](#_Appendix_3:_Manual).

|  |
| --- |
|  |
| Figure 66: OrderTest output |

Aside from the limitations noted above in creating automated tests for status verification throughout the order lifecycle, current tests could also be expanded to include edge cases such as cancelled orders. The application does not have a dispute resolution system in place – this was deemed to be outside the initial development scope – and if/when such a system is implemented, it should impact this particular test suite. As with listings and quotes, performance testing for large volumes and orders with large numbers of comments would help ensure good performance under high loads. Given that orders represent the culmination of the marketplace transaction, more thorough testing of notification systems around order status changes would also be valuable to ensure all parties remain informed throughout the process.

UTC vs. ExpireListings

Socialite

Size limitation on individual/aggregate attachments

Order status screenshots

# Evaluation and Conclusion

This section should include a general evaluation of the success of the project measured against the criteria stated in the introduction and/or requirements. An evaluation of the hardware/software environment and language used may also be presented. Suggestions for further work should also be discussed. Do not be afraid to be critical or to draw a negative conclusion; not all projects will be successful. This section should provide a thorough and honest reflection on the process followed in the project and the results of that process. To do this well, the student should not leave any blind spots in their reflection and should identify the most and least successful aspects of the project. It should be written in such a way as to be helpful to a person seeking to adapt the project or to create a similar project in the future.

The aim of this project was to create a web application to act as a marketplace for facilitating the repair of consumer electronics. Creating a centralised marketplace where consumers can submit requests for repair can solve many of the above-mentioned problems, since the onus can be placed on repair specialists to offer their services in a more transparent manner, lowering prices for consumers and creating a greater incentive to avoid electronic waste through extending the useful life of products (in contrast to the lack of incentive that currently exists, failing to discourage the common consumer ‘throwaway’ reflex)

## Improvements to existing features

Product categories dropdown – text search

Query optimisation in other areas, e.g. messages

Smaller lists of countries/currencies

Address validation – Google Maps API

Emails – layout improvement, add more events to trigger external emails, use queueing

Testing – noted in section above

Production deployment prep – different SQL varieties, VPS hosting, API

Some minor UI improvements, e.g. email.show page

Options like setting up video chat for defect assessment – can be done, agreed between parties,

## Scope expansion

‘About Us’, Contact admin, Trustpilot, etc.

Specialists – ‘web shop’ vendor page similar to eBay

Disclaimers required on sign-up

Currency conversion

Invoicing / Payment integrations

Dispute resolution

Consideration of unrepairable items

## Use of AI

GitHub Copilot

Kulp.ai – re-evaluation of software choices? AI-prompted app creation has chosen React, Supabase

# References

Barot, S. (2025, February 12). *Laravel Authentication Packages: A Comparative Analysis*. Retrieved from Glowid IT Solutions: https://aglowiditsolutions.com/blog/laravel-authentication/

Beckwith, K. (2023, October 30). *Ubuntu’s new variable fonts*. Retrieved from typenetwork.com: https://typenetwork.com/articles/ubuntus-new-variable-fonts

Brannon, I. (2024, Spring). *A Criticism of ‘Right to Repair’ Laws*. Retrieved from CATO Institute: https://www.cato.org/regulation/spring-2024/criticism-right-repair-laws

Compass, B. (n.d.). *Color Theory: Orange as a Branding Color*. Retrieved from https://brandingcompass.com/branding/color-theory-orange-as-a-branding-color/

Conway, L. (2021, September 24). *Right to Repair Regulations.* Retrieved from UK Parliament - House of Commons Library: https://researchbriefings.files.parliament.uk/documents/CBP-9302/CBP-9302.pdf

Corey, T. (2021, February 4). *How Do I Store Images In SQL? In Other Database Types?* Retrieved from YouTube: https://www.youtube.com/watch?v=XnNbizvb1uM

Craig, W. (n.d.). *Finding the Middle Ground: How to Compromise with Clients*. Retrieved from WebFX - Digital Marketing that Drives Revenue: https://www.webfx.com/blog/web-design/compromise-with-clients/

Dev, K. (2023, June 3). *Comparing MySQL Database and Database Management in Laravel: Which One is Easier?* Retrieved from Medium: https://medium.com/@kangjessy.dev/comparing-mysql-database-and-database-management-in-laravel-which-one-is-easier-5650c10146c5

Ditya, Z. (2025, March 25). *Mastering Testing in Laravel with Pest PHP: A Comprehensive Guide*. Retrieved from Medium: https://medium.com/@zulfikarditya/mastering-testing-in-laravel-with-pest-php-a-comprehensive-guide-0d1a599f79f5

Ditya, Z. (2025, March 17). *The Power of Laravel Factories: Efficient Database Seeding for Testing and Development*. Retrieved from Medium: https://medium.com/@zulfikarditya/the-power-of-laravel-factories-efficient-database-seeding-for-testing-and-development-f8700c1b2083

European Commission. (2023, March 22). *‘Right to repair': Questions & Answers*. Retrieved from European Commission: https://ec.europa.eu/commission/presscorner/detail/es/qanda\_23\_1795

Gacovski, Z. (2025, January 6). *Leveraging Laravel Migrations for Seamless Cloud Deployments*. Retrieved from Kamatera Performance Cloud: https://www.kamatera.com/blog/leveraging-laravel-migrations-for-seamless-cloud-deployments/

García Molyneux, C., & Oberschelp de Meneses, A. (2024, June 10). *The EU Adopts Right to Repair Directive*. Retrieved from Inside Energy & Environment: https://www.insideenergyandenvironment.com/2024/06/the-eu-adopts-right-to-repair-directive/

Gomez, G. (2024, June 3). *Content rendering: Blade vs Vue and React*. Retrieved from https://ggomez.dev/blog/content-rendering-blade-vs-vue-and-react

HiBit. (2024, February 19). *Using Laravel with an SQLite database*. Retrieved from hibit.dev: https://www.hibit.dev/posts/148/using-laravel-with-an-sqlite-database

Katariya, J. (2025, February 12). *12 Key Laravel Benefits for Enterprise Development*. Retrieved from Moon Technolabs: https://www.moontechnolabs.com/blog/laravel-benefits/

Lamé, A. (2021, November 15). *From PHPUnit to Pest*. Retrieved from Medium: https://medium.com/@antoine.lame/from-phpunit-to-pest-b825b1d881f3

Laravel. (n.d.). *Database: Getting Started*. Retrieved from laravel.com: https://laravel.com/docs/12.x/database

McLean, S. (2023, 08 21). *The Color Orange: Essential Color Theory, Symbolism and Design Application*. Retrieved from Dunn Edwards: https://www.dunnedwards.com/pros/blog/the-color-orange-essential-color-theory-symbolism-and-design-application/

Raed, R. (2023, June 29). *Why is there Vite.js in Laravel*. Retrieved from dev.to: https://dev.to/roshdiraed/why-is-there-vitejs-in-laravel-1ag6

Raj, S. (2024, December 5). *The Color Psychology of Orange in Branding and Marketing*. Retrieved from Ofspace Digital Agency: https://www.ofspace.co/blog/orange-in-branding-and-marketing

Rodriguez, C. (2019, April 29). *Ubuntu's Use and Color Theory. Comparing to Helvetica*. Retrieved from Medium: https://medium.com/@rodriguez.cesar009/ubuntus-use-and-color-theory-2442a779aab1

Sekhniashvili, Z. (2025). *Laravel for Beginners Project HTML/CSS Template*. Retrieved from The Codeholic: https://resources.thecodeholic.com/laravel-for-beginners-html-css-template

Selvaraj, S. (2023, December 16). Database Management and Eloquent ORM. *Building Real-Time Marvels with Laravel*. Berkeley, CA: Apress. Retrieved from Building Real-Time Marvels with Laravel: https://doi.org/10.1007/978-1-4842-9789-6\_3

Sharma, A. (2024, May 31). *Level Up Your Laravel Development with Best Practices for Migrations*. Retrieved from Medium: https://medium.com/@asis-sharma/level-up-your-laravel-development-with-best-practices-for-migrations-44cd2d166894

Shukla, M. (2023, September 21). *Eloquent ORM in Laravel: Simplifying Database Interactions*. Retrieved from Medium: https://manoj-shu100.medium.com/eloquent-orm-in-laravel-simplifying-database-interactions-b0269942f190

Sling Academy. (2024, January 15). *Laravel Blade: How to escape dangerous HTML & JavaScript* . Retrieved from slingacademy.com: https://www.slingacademy.com/article/laravel-blade-escape-dangerous-html-javascript/

SQLite. (n.d.). *Appropriate Uses For SQLite*. Retrieved from sqlite.org: https://sqlite.org/whentouse.html

Suhail, M. (2023, May 10). *Sending queue emails in Laravel*. Retrieved from Medium: https://medium.com/@mosuhail084/sending-queue-emails-in-laravel-6ac24d84f23e

Team Gyata. (2024, August 8). *Database Tests*. Retrieved from Gyata AI: https://www.gyata.ai/laravel/database-tests

Thorne, A. (2021, July 27). *Website Button Best Practice for UX*. Retrieved from Factory Pattern: https://factorypattern.co.uk/blog/website-button-best-practice-for-ux/

Verma, R. (2025, January 29). *Mastering Laravel Migrations: Simplifying Database Management in 2025*. Retrieved from Bacancy Technology: https://www.bacancytechnology.com/blog/laravel-migrations

World Health Organization. (2024, October 1). *Electronic waste (e-waste)*. Retrieved from World Health Organization: https://www.who.int/news-room/fact-sheets/detail/electronic-waste-(e-waste)

Zaidi, S. (2025, February 11). *Why There’s a Need for Automated Web Testing‍*. Retrieved from opkey.com: https://www.opkey.com/blog/why-theres-a-need-for-automated-web-testing

# Appendices

Appendix 1: Original Figma UI designs

A sample of the original Figma UI designs for this project are shown below.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

The full set of pages can be viewed at the following links:

[Presentation of individual pages](https://www.figma.com/proto/IMPhnXOBdeJeiRnOl32IMa/RepairMart?node-id=5-3&p=f&t=UGb5pkvM5wl9wqY0-0&scaling=min-zoom&content-scaling=fixed&page-id=0%3A1&starting-point-node-id=5%3A3)

[Overall design](https://www.figma.com/design/IMPhnXOBdeJeiRnOl32IMa/RepairMart?node-id=0-1&t=ywKro3jD78llIxHB-1)

Appendix 2: UML diagrams

UML use case diagram for Account Creation

|  |
| --- |
|  |

UML use case diagram for repair request listing

|  |
| --- |
|  |

Sequence diagram for repair request listing

|  |
| --- |
|  |

Appendix 3: Manual Testing

The below images show screenshots from each stage of the Order management lifecycle, from creation to closure. Screen recordings of the full listing 🡪 quote 🡪 order lifecycle have also been made to document the process as part of manual testing. These can be viewed [here](https://qubstudentcloud-my.sharepoint.com/:f:/g/personal/15615014_ads_qub_ac_uk/Eltq1KP7G6lMvM_Z_0oHrjUBxTP2SY-DfG5RA2xT_CBqWQ?e=ofQov9).