👋 Hello!

Welcome to the front-end development skills test! We're excited to have you showcase your abilities and understanding of front-end development.

Objective:
The primary aim of this test is to assess:

1. **Code Structuring**: Your ability to structure code in an organised, scalable, and maintainable manner.
2. **Reusable UI Components**: Your proficiency in creating UI components that are modular and reusable across different parts of an application.

Task Overview:
You may find in this File:

1. A dashboard related to tools and equipment management.
2. A breakdown of the UI components, showing their interrelationships, properties, and data fetch methods.

Your task is to recreate the dashboard using **React** or **NextJS** with **typescript** preferably. The final output should closely resemble the provided design, both in appearance and functionality.

To aid in this process, you may use dummy data that mimics the information you'd typically fetch using the methods shown in the breakdown.

Guidelines:

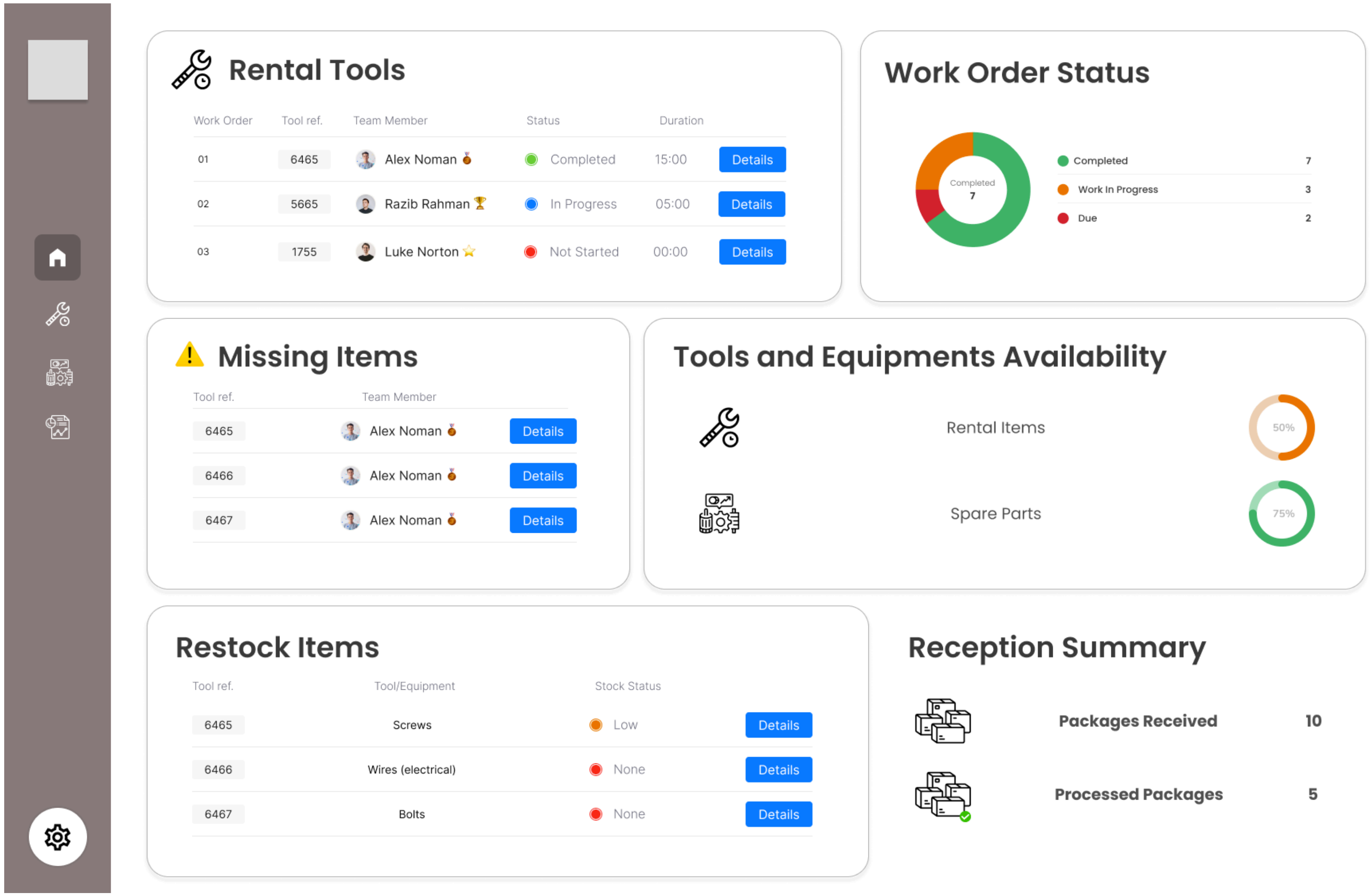
1. While aesthetics are important, we are more interested in your approach to component architecture, data management, and overall code quality.
2. Make sure the components you create are reusable. For instance, the table components you see in the design could potentially be used in other sections of the application with different data.
3. Make use of dummy data effectively to demonstrate how you would handle real data fetches and updates in a live environment.
4. You might notice some empty cells in the provided UI component props tables. Part of this test is assessing your understanding of the test requirement file. Please fill out these cells with appropriate properties and values as you see fit. This will also give us insight into how you might build similar requirement files in the future.

Submission:

1. Fork the provided GitHub repository.
2. Create a new branch for your work.
3. Develop the dashboard interface based on the Figma design.
4. Commit your changes to your branch after every development session.
5. Push your branch to the forked repository.
6. Submit a pull request to the original repository.

Best of luck! We're eager to see how you tackle this challenge.

Dashboard to be replicated:



Rental Tools

Work Order	Tool ref.	Team Member	Status	Duration	
01	6465	Alex Noman	● Completed	15:00	Details
02	5665	Razib Rahman	● In Progress	05:00	Details
03	1755	Luke Norton	● Not Started	00:00	Details

Work Order Status

Completed 7

● Completed	7
● Work In Progress	3
● Due	2

Missing Items

Tool ref.	Team Member	
6465	Alex Noman	Details
6466	Alex Noman	Details
6467	Alex Noman	Details

Tools and Equipments Availability

Rental Items	50%
Spare Parts	75%

Restock Items

Tool ref.	Tool/Equipment	Stock Status	
6465	Screws	● Low	Details
6466	Wires (electrical)	● None	Details
6467	Bolts	● None	Details

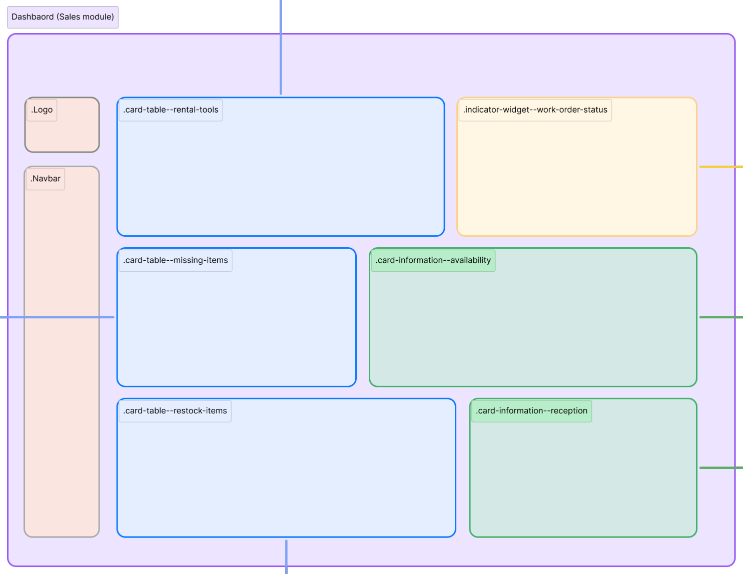
Reception Summary

Packages Received	10
Processed Packages	5

Lo-Fi UI Breakdown

cardTable props

	A	B	C
1	**Name**	**Type**	**Values**
2	.card-table--{purpose}	'div' 'container' 'table'	:dynamic:
3	.card-table--rental-tools	'div' 'container' 'table'	:data: getCheckedOutRentals()
4	.card-table--missing-items	'div' 'container' 'table'	:data: getMissingRentals()
5	.card-table--restock-items	'div' 'container' 'table'	:data: getItemToRestock()


Dashbaord (Sales module)
.Logo
.card-table--rental-tools
.indicator-widget--work-order-status
.Navbar
.card-table--missing-items
.card-information--availability
.card-table--restock-items
.card-information--reception

indicatorWidget props

	A	B	C
1	**Name**	**Type**	**Values**
2	.indicator-widget--{purpose}	'div' 'container' 'pieChart'	:dynamic:
3			:data: getWorkOrderStatus()

cardInformation props

	A	B	C
1	**Name**	**Type**	**Values**
2	.card-information--{purpose}	'div' 'container'	:dynamic:
3			:data: getToolEquipmentAvailability()
4			:data: getReception()

Development Guidelines

Open-Source Dependencies
> Requiring open-source dependencies can have several benefits, including leveraging existing solutions, fostering collaboration, and reducing development costs.
> It's important to clearly define the criteria for selecting open-source dependencies, such as licensing compatibility, community support, and code quality.
> Need to establish a process for evaluating and approving new dependencies to ensure they meet the required standards.

Development guidelines
> May vary depending on the solutions to be developed. However, a good folder structure is mandatory where everything is built as individual components. Every component should be functional on its own.

Folder Structure and Componentisation:
> A well-defined folder structure promotes organisation and modularity, making it easier to manage and maintain the codebase.
> Building components that are functional on their own enhance reusability, testability, and maintainability.
> Modular architecture pattern, such as micro-services or a component-based architecture.
> Document the expected folder structure, naming conventions, and guidelines for organising code within the components.

Frameworks and Libraries
> Front-end development (i.e. web applications, user interface development); NextJS is the preferred front-end framework
> Python development to be packaged as Poetry project; ease of dependency management; helps in establishing a solid project structure.
> NodeJS to be used on assessment of solutions to be developed.
> MUI: material UI library for NextJS; user interface components.
> ||Assessment of the solutions to be developed will define the framework and libraries to be used. ||

**Coding Standards **
Naming Conventions
> Use descriptive and meaningful names for variables, functions, classes, and other identifiers.
> Follow a consistent naming convention, such as camelCase or snake_case, throughout the codebase.
> Naming convention for each type of development (i.e. API dev, front-end dev...)
> Avoid abbreviations or overly cryptic names that can make code harder to understand.

Code Formatting
> Use consistent indentation (e.g., tabs or spaces) to improve code readability.
> Prettify all scripts improving readability.
> Apply proper spacing around operators, commas, and parentheses to enhance code clarity.

Commenting and Documentation
> Document the purpose, inputs, outputs, and any important considerations for functions and methods.
> Include comments to explain complex algorithms, non-obvious code, or potential pitfalls.
> Regularly update and review documentation to ensure it remains accurate and up-to-date.

Error Handling and Exception Management
> Implement proper error handling mechanisms to gracefully handle exceptions and errors.
> Use appropriate exception types and provide informative error messages to aid in troubleshooting and debugging.
> Avoid catching generic exceptions unless necessary, as it may hide potential issues.

Code Reusability and Modularity
> Discourage code duplications, encourage the reuse of code by breaking it into smaller, reusable functions or modules.
> Aim for single-responsibility principle, where each function or module focuses on one specific task.
> Minimise code duplication by extracting common functionality into reusable utility functions or libraries.
> Apply same principles when building user interface where reusable components are always welcome

Performance Optimisation
> Profile and benchmark critical code sections to identify potential performance bottlenecks.
> Always assume there's a better way of achieving the more complex algorithms.
> Perform unit testing to point out potential components inefficiencies.
> Bottlenecks could be found when testing components that communicates with each other.

Version Control
> Follow recommended version control practices, such as committing atomic changes and providing meaningful commit messages.
> Example of commit messages:
> "fix: *action*" → fix: to specify that this commit is to fix a certain component or function

> "feat: *feature*" → feat: to specify the addition of a new feature.
> Regularly pull the latest changes from the repository and resolve conflicts promptly.
> Use branching strategies effectively to isolate features or bug fixes and merge them back to the main branch when ready.

****Testing Practices and Quality Assurance****
> Write automated unit tests to verify the correctness of individual functions and modules.
> Practice test-driven development (TDD) by writing tests before implementing the functionality.
> Test Components relationships and efficiency.
> Aim for high code coverage to ensure that critical parts of the code are thoroughly tested.
****Integration Test: ****
> Deploy in test environment with access to specified users to throughly test prior to releasing into production
> **Bug reporting, Triaging, and tracking.**
> **Bug reports**
> Automated bug report logs from interactions with the solutions developed.
> Bug Report Submission by end-users description of the issue, steps to reproduce it, and any relevant screenshots or error messages.
> **Bug prioritisation**
> Status based on severity, impact and urgency
> Number of users affected, impact on features and functionality or is a prerequisite for another feature/functionality.
> **Tracking and Updates:**
> The bug's progress is tracked throughout its lifecycle, usually through Github and Linear (a tool for tracking Github project issues)
> Updates are provided to stakeholders, including the reporter, as the bug moves through various stages like development, testing, and resolution.

Github Rules

Main/Branch: The main branch represents the production-ready code. It should only contain stable and thoroughly tested code. Developers should not directly commit to this branch, and it should be protected from force pushes. Only push to this branch once all test have been conducted to then push into staging then production.
Development/Branch: This branch serves as the integration branch for ongoing development work. Developers create feature branches from this branch and merge their changes back into it once they are complete.
Feature/Branch: Each new feature or task can have its own feature branch. Developers create a branch from the development branch, work on their specific feature or task, and then merge it back into the development branch upon completion. This approach allows for isolated development and easier code reviews.

Release/Branch: When preparing for a release, a release branch can be created from the development branch. This branch is used for finalising the release by performing any necessary bug fixes or last-minute adjustments. Once the release is ready, it can be merged into the main branch and tagged with a version number.
Hotfix/Branch: In case of critical issues or bugs discovered in the main branch, a hotfix branch can be created from the main branch. Developers can quickly address the issue in this branch and merge it back into main or development branch.
Bug fix/Branch: Bug fix branches can be created from the development branch to address non-critical bugs. These branches focus on fixing specific issues and are merged back into the development branch once the bug is resolved.
Experimental/Branch: For experimental or research-oriented work, developers can create experimental branches to explore new ideas or approaches. These branches allow for independent experimentation without affecting the main codebase. Not to merge directly into the main branch without thorough testing and review.