Functional and Non-Functional Requirements Specifications:

* Functional Requirements:
  + A functioning Login page that redirects user correctly based on their roles in the database. (Approved)
  + A Sign-up page that redirects users to the login page if their sign up was successful and inserts their data in the database and also warns them if their sign up was a failure. (Approved)
  + A support request form that sends data to a database that will be used to help customers with their repairs (Approved)
  + A table in the same page to the users what are their requests. (Approved)
  + An inventory page for keeping track of the company’s spare parts (Approved)
  + A job scheduling page for the ease of scheduling the jobs of the company. (Approved)
  + An Analysis page to keep track of the company’s growth over the years and how it’s doing. (Approved)
* Non-Functional Requirements:
  + A good and readable UI. (Approved)
  + A good UX for the users. (Approved)
  + A highly performing Website. (Approved)
  + A good design. (Approved)
  + Good security (Will be implemented in future plans) (Approved)

Algorithm Design Documentation:

* Some examples of the Algorithms designs:
  + Backend Server pseudo code for the support request page (most of the backend servers follow a similar approach to this code, albeit this is the most detailed one with more features added to it.):

1. Import necessary modules (express, body-parser, fs, cors)

2. Initialize express application

3. Set port number

4. Configure express application to use body-parser and cors

5. Define paths for database file and nextId file

6. Read the initial value of nextId from the file

7. Define an endpoint to fetch data from the database:

- Read the database file

- If there's an error, log it and send an error response

- Otherwise, send the data as a response

8. Define an endpoint to handle form submissions and update the database:

- Read the form data from the request

- Read the database file

- If there's an error, log it and send an error response

- Otherwise, add the form data to the database with a new ID

- Write the updated database back to the file

- If there's an error, log it and send an error response

- Otherwise, update nextId in the file and send a success response

9. Start the server and listen on the specified port

* + Pseudo code of a frontend component that renders a form and listens to the backend Server (The same backend server that has its pseudo code above):

1. Import necessary modules and components (React, useState, Link, Helmet, SupporTable)

2. Define initial state for form data

3. Define a function to handle changes in form fields:

- Update the form data state with the new value of the changed field

4. Define a function to handle form submission:

- Check if any required fields are empty, if so, throw an error

- Send a POST request to the server with the form data

- If the response is OK, alert the user of successful submission and reset the form data

- If the response is not OK, throw an error

- Catch any errors during the process and alert the user

5. Render the component:

- Set the page title using Helmet

- Display a header with a link to the home page

- Display a form with fields for Requester, Request Type, Schulde a Repair, Parts Needed, and Problem Description

- Each field updates the form data state when its value changes

* + Lastly A pseudo-Code of a Database:

1. Initialize an empty list to store the database entries

2. For each entry in the database:

- Create a dictionary with the following keys and their corresponding values:

- "id": a unique identifier for the entry

- "Requester": the type of requester (either "Individual" or "Organization")

- "RequestType": the type of request (e.g., "Full Repair", "Optimization")

- "SchuldeARepair": the scheduled date for the repair

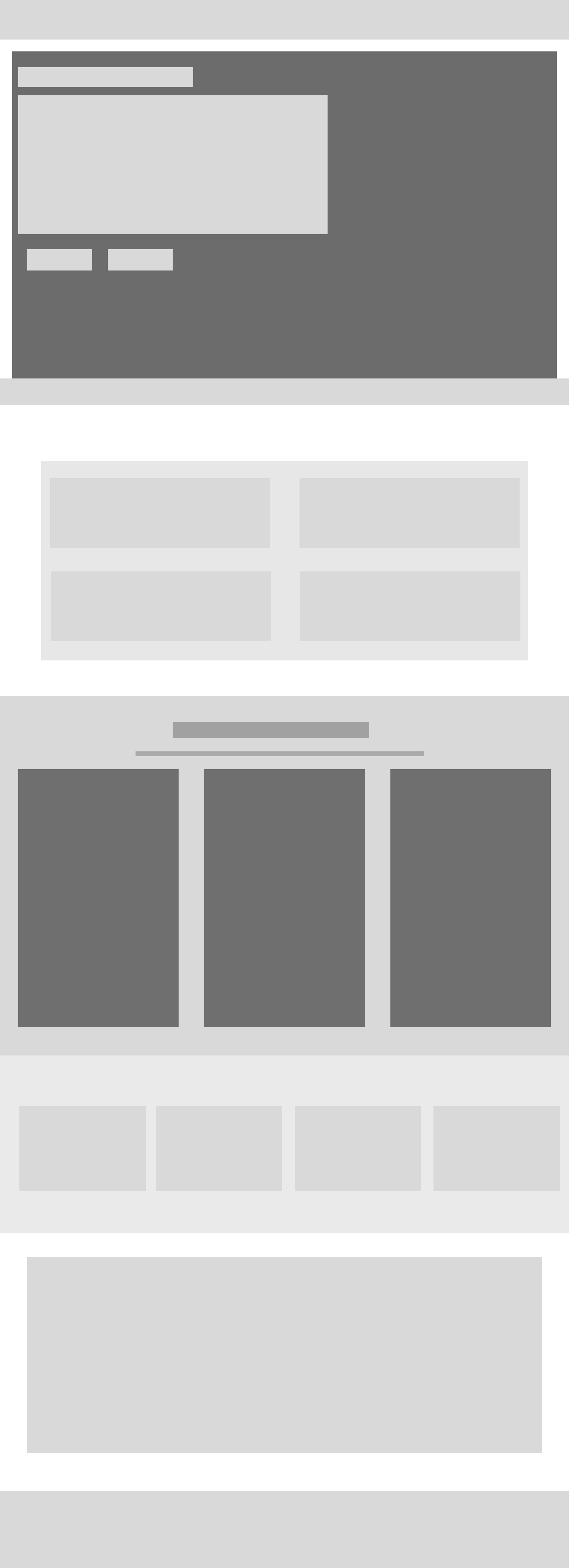
- "PartsNeeded": a description of the parts needed for the repair

- "ProblemDescription": a description of the problem

- Add the dictionary to the list

Visual Design Documentation:

* block outs:
  + Home Page:



* + Sign Up/ Sign In:

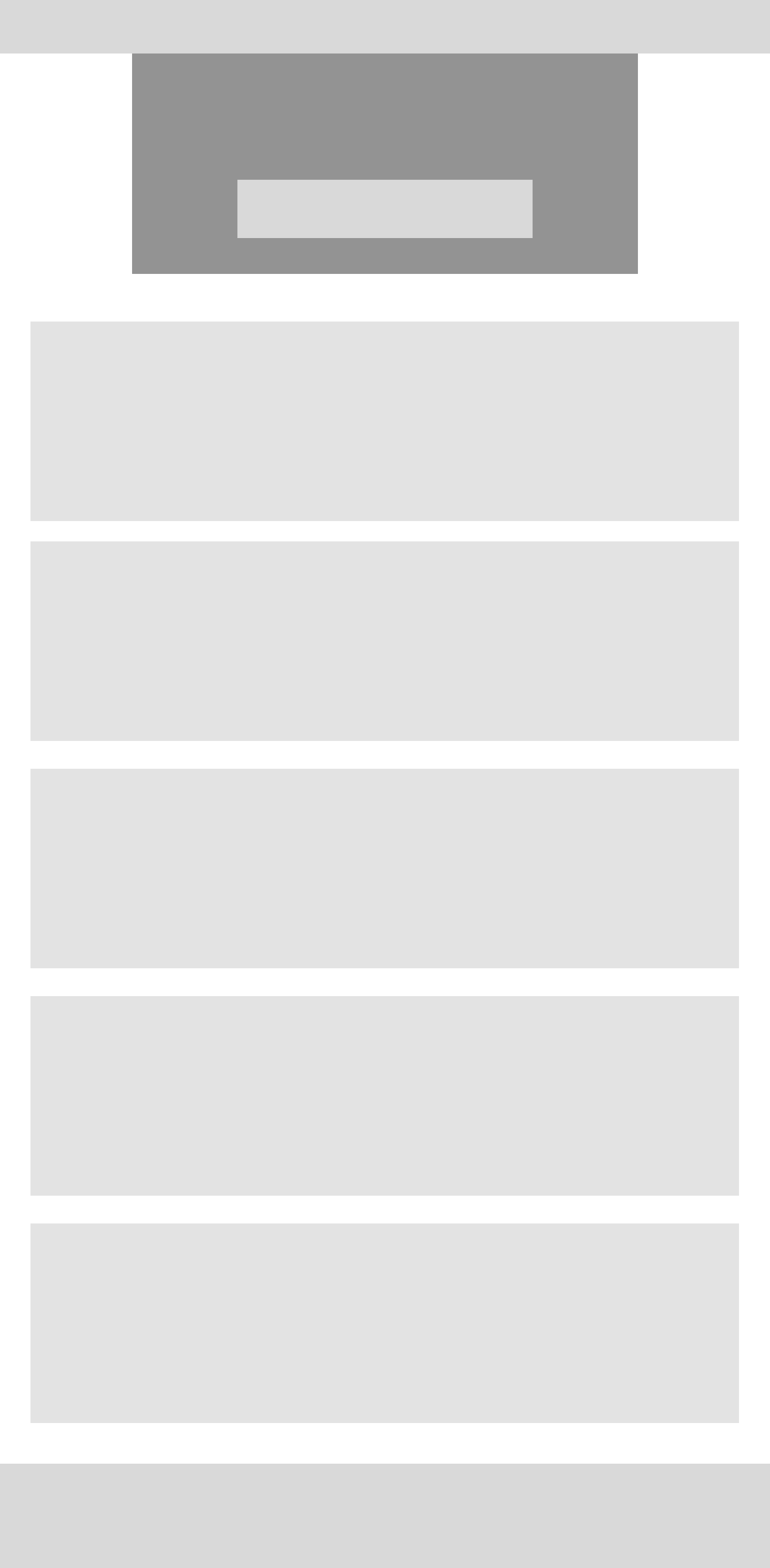




* + Support Request:



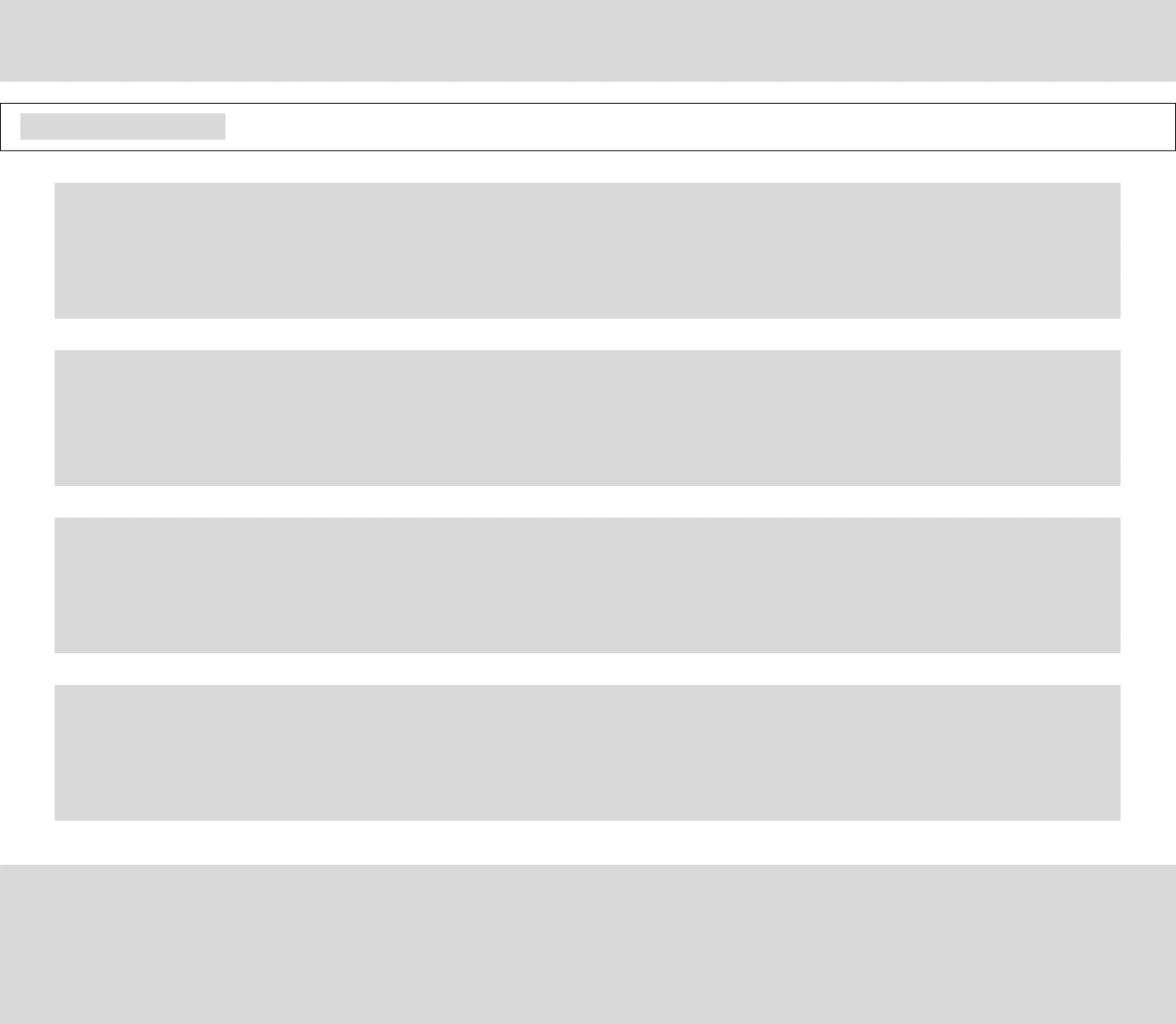
* + Knowledge Base:



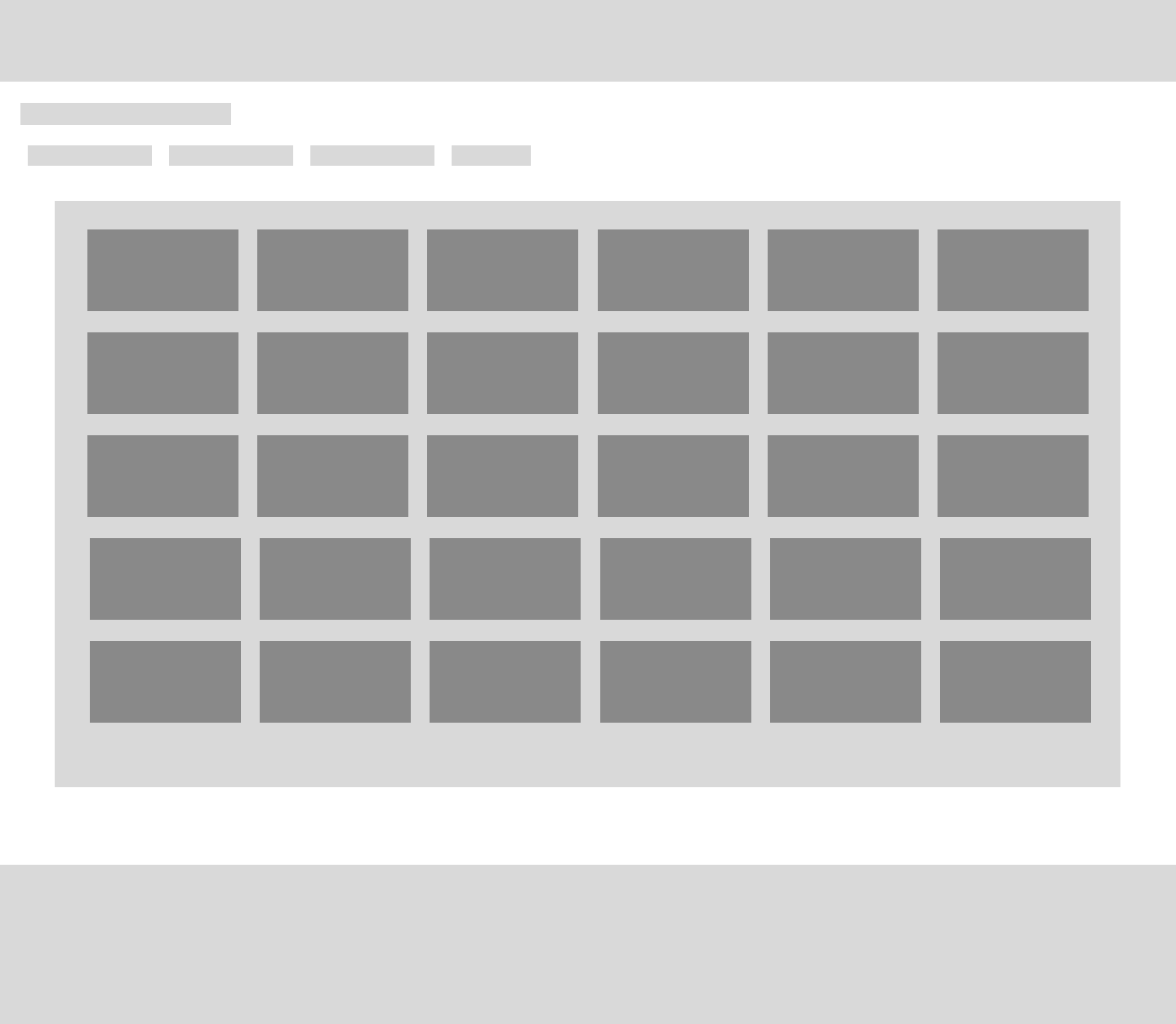
* + Contact us:



* + Inventory:



* + Schedule A Job:



* + Analysis:



Data Requirements Design:

* All Databases in the web application are independent of each other to solve any complexity that could happen and because they are mostly unrelated and thus it’s been decided that none of them would benefit from being connected with any other Database.
* Databases were made in MongoDB and are in the form of Json Files.
* An Example of A database is the of the login data:
  + [
    - {
      * "username": "Admin", (String type)
      * "password": "the Admin", (String type)
      * "email": "admin@gmail.com", (String type)
      * "phone": 1068452056, (int type)
      * "role": "admin" (String type)
    - },
    - {
      * "username": "regular user",
      * "password": "the regular user",
      * "email": "theRegularUser@gmail.com",
      * "phone": 1028052863,
      * "role": "user"
    - }
  + ]

Accessibility and Inclusivity Considerations:

* In Future Plans we consider Adding Accessibility Options to be able to help a wider range of customers. Those options like Text To speech options, High Contrast Mode, Color Filters

Details of Components to Be Reused/Refactored:

* I have created multiple reusable Components like the features card, the items card and Analysis Component. Each of these components is used quite frequently, for example the features component is used 4 times in the same page, and the items component is going to be used as long as there are items in the database.
  + The Analysis Component Also Uses a 3rd party component library called MUI.
* There is also the Calendar Component that uses a 3rd party component library called react-big-calendar.
* Some of these components literally are the backbone of entire pages in the full stack solution.
* There is the navigation-links component also, and it’s used across the different home pages created.
* There is the ways-of-diagnoses and the hardware-fixes components and both of them are used in the knowledge base page.
* Lastly there is the reviews component and it’s used in the home page as a testimonial of Dern-support credibility.

Review and Refinement:

* Based on the review of a few of my colleagues, some of them questioned some of the design choices, and most of those questions are based on Alignment choices like the alignment of the Navbar components. Those will be solved with the final product.
* Based on some other reviews, A search function in the spare parts Inventory Page would be appreciated, and thus it will be included in future plans to do that.
* Most other reviews are about minor bugs that will be mentioned in the bug testing.

Justification for Design Decisions:

* React.js is used because it clearly Aligns with the goal and purpose of the full stack solution along with being up to date and easier to maintain than other options like native HTML/CSS/JS combo.
* MongoDB is used because it’s convenient and easy to use and it allows for NoSQL Database solution.
* Node.js is used for the downloading and managing node packages using the NPM.
* Express.js is used for the sole purpose of creating the backend servers and maintaining the backend.
* The approach of absolute modularity was used because this is a larger scale web application, so having a modular app is essential for easier maintenance and to lessen the complexity of the application.