**D424 – Software Engineering**

**Task 3**

|  |  |
| --- | --- |
| **Capstone Proposal Project Name:** | http://www.idevnews.com/views/images/uploads/general/wgu_logo.png  StatusDeck |
| **Student Name:** | Zachary Pfizenmaier |

**Table of Contents**

Table of Contents

[Table of Contents 2](#_Toc203234905)

[Task 3 Design Document 4](#_Toc203234906)

[Application Design and Testing 4](#_Toc203234907)

[Class Design 4](#_Toc203234908)

[System Design 5](#_Toc203234909)

[UI Design 5](#_Toc203234910)

[Unit Test Plan 11](#_Toc203234911)

[Introduction 11](#_Toc203234912)

[Purpose 11](#_Toc203234913)

[Overview 11](#_Toc203234914)

[Test Plan 12](#_Toc203234915)

[Items 12](#_Toc203234916)

[Features 12](#_Toc203234917)

[Deliverables 12](#_Toc203234918)

[Tasks 12](#_Toc203234919)

[Needs 13](#_Toc203234920)

[Pass/Fail Criteria 13](#_Toc203234921)

[Specifications 15](#_Toc203234922)

[Procedures 16](#_Toc203234923)

[Results 17](#_Toc203234924)

[Hosted Web Application 18](#_Toc203234925)

[Hosted Web Application Link 18](#_Toc203234926)

[GitLab Repository & Branch History 18](#_Toc203234927)

[GitLab Repository Link 18](#_Toc203234928)

[GitLab Branch History 18](#_Toc203234929)

[User Guide for Maintenance and Setup 18](#_Toc203234930)

[Local Development Setup 19](#_Toc203234931)

[Core Maintenance Procedures 19](#_Toc203234932)

[Deployment 20](#_Toc203234933)

[User Guide for Running the Application from User Perspective 20](#_Toc203234934)

[Introduction 20](#_Toc203234935)

[Accessing the Application 20](#_Toc203234936)

[Login and Signup 21](#_Toc203234937)

[Websites 23](#_Toc203234938)

[Add a New Website 23](#_Toc203234939)

[Modify a Website 26](#_Toc203234940)

[Filter Website Data 27](#_Toc203234941)

[Reports 28](#_Toc203234942)

[Profile and Sign Out 29](#_Toc203234943)

**Task 3 Design Document**

# Application Design and Testing

## Class Design

A screenshot of a computer

AI-generated content may be incorrect.

The class diagram for StatusDeck models the core entities and relationships and focuses on the data attributes of each class rather than methods. This is because the application’s business logic resides in separate components, such as API routes triggered by scheduled cron jobs, frontend hooks, and in standalone utility functions rather than within the data models themselves. The design is composed of three primary classes: User, Website, and StatusCheck. The structure is centered on a one-to-many relationship where a single User can monitor multiple Website entities. In turn, each Website is associated with many StatusCheck records that store its performance history. While the User class is shown for conceptual clarity, its implementation is managed by the Supabase Auth service. A screenshot of a computer

AI-generated content may be incorrect.

## System Design

The sequence diagram below illustrates the primary workflows of the StatusDeck application. It begins with a user interacting with the Next.js frontend to sign up or log in via the Supabase Auth service. Once authenticated, the user can add a new website, which calls a serverless function to write the site’s information into the Supabase database. The diagram then shows the automated monitoring process, where a Vercel Cron Job periodically invokes a serverless function that retrieves a list of websites, performs the status checks, and saves the results back to the database. Finally, the sequence closes with the user viewing the reporting pages, where the frontend fetches the historical status check data from the database and displays it in charts, cards, and tables.

A diagram with arrows and text

AI-generated content may be incorrect.

## UI Design

The User Interface (UI) design of StatusDeck is focused on clarity and providing information quickly at a glance, utilizing line charts and specific colors. The UI begins with the login screen, which simply provides input fields for email and password, with clear navigation to the Create Account screen for users without an account.

A screenshot of a login screen

AI-generated content may be incorrect.

Once logged in, the user views the home screen, which provides a user-friendly dashboard of all tracked sites and their status over time. Each dot on the chart can be hovered over for additional details about that specific status check.

A screenshot of a graph

AI-generated content may be incorrect.

Search and filter features are included to help users quickly find the websites they care about most.

A screenshot of a computer

AI-generated content may be incorrect.

Clicking on a website card navigates to a detailed page to view site-specific information and historical data. Clicking the “+ Add Site” button brings up a modal to add a new site so the user can easily add more websites to track.

A screenshot of a computer

AI-generated content may be incorrect.

The site-specific page provides a full-screen chart with an adjustable time axis and advanced statistics based on the data. Below these statistics is a Performance Analysis, which provides actionable insights based on the historical status check data.

A graph on a white background

AI-generated content may be incorrect.

A screenshot of a graph

AI-generated content may be incorrect.

The site name and URL can be adjusted through the settings icon, which brings up a simple modal for input. The site can also be deleted through this modal, which brings up a confirmation to prevent accidental deletion.

A screenshot of a website settings

AI-generated content may be incorrect.

To view the data in a tabular format, the user can navigate to the Report page through the “Generate Report” button. This Report page shows the history of status checks for the site in a clear, tabular format.

A screenshot of a web page

AI-generated content may be incorrect.

A profile button is available in the top-right of the application, which provides account information and the ability to sign out.

A screenshot of a login form

AI-generated content may be incorrect.

# Unit Test Plan

## Introduction

### Purpose

Unit testing for the StatusDeck application was conducted using the Vitest testing framework to validate critical, non-UI business logic. The primary methods involved testing a data structure class and a utility function to ensure data integrity and calculation accuracy. The results of the test were positive, with all tests passing successfully. A key part of this process, however, was code refactoring that had to take place before testing could begin. The logic for calculating performance statistics was originally embedded directly within a UI component, making it difficult to test in isolation and impossible to reuse elsewhere. This business logic was moved into its own independent utility function. This change decoupled the calculation from the UI, which enabled targeted unit testing and improved the overall maintainability of the codebase.

### Overview

Two specific unit tests were created to provide validation of the application’s core functionality. Rather than testing UI components, the focus was placed on business logic to ensure reliable data integrity and calculations. The first test targeted the SuccessResult class, which is a key part in handling various outcomes of a website status check. Testing this class ensures that the ideal case of a functioning website, which occurs most often, is handled correctly. The second test targeted the calculatePerformanceStats utility function, which is responsible for calculating key metrics shown on the user’s dashboard. This logic was originally embedded within a page component but was extracted into a pure standalone function to make it testable and reusable. By focusing on isolated business logic, we can ensure the application’s data processing is sound.

## Test Plan

### Items

A standard development environment with Node.js installed and the application’s complete source code repository was required for testing.

### Features

The unit tests focused on two critical backend features that directly impact the accuracy of the user-facing dashboard. The first test targets the status check result processing by validating the SuccessResult class. This class is crucial because after the application’s cron job successfully pings a website, it creates a SuccessResult object to hold the outcome before saving it to the database. This test ensures the integrity of this data object, confirming that the status code, response time, and “Online” status are stored correctly. The second test targets the dashboard’s statistics calculation by validating the calculatePerformanceStats utility function. In the application, the main dashboard fetches a site’s specific history of checks and passes it to this function. The test guarantees the accuracy of this statistical processing, so that the Uptime Percentage, Total Checks, and Incidents displayed to the user are correctly representing the historical data.

### Deliverables

The testing process produced two unit test script files, “src/lib/status-results/status-results.test.ts” and “src/lib/utils.test.ts”, along with terminal output showing the results of executing the test suite. This written report serves as the formal documentation of the test plan and its outcome.

### Tasks

Several tasks were required to complete the testing process.

1. It began with the initial setup, where the Vitest framework and its dependencies were installed and configured within the Next.js project.
2. This was followed by a refactoring task, where the performance statistics logic was moved from the “[siteName]/page.tsx” component into the calculatePerformanceStats utility function.
3. Next was the test creation, where two unit test scripts were written using the “Arrange-Act-Assert” pattern to ensure each test was clear and focused. This involved Arranging preconditions and inputs like mock test data, Acting by invoking the function under test, and Asserting that the result matched the expected outcome.
4. Once the tests were created, the test suite was executed, and the results were analyzed.
5. The final task was the formal documentation of the process in this report.

### Needs

To perform the two specific unit tests, a minimal set of dependencies was required. The runtime environment needed to be Node.js (v18 or later). The required package manager was npm (v9 or later). The core of the testing stack was the Vitest framework (v3.2.4). This also required several supporting libraries, including “vite” as a peer dependency and the “vite-tsconfig-paths” plugin, which was needed to allow the test runner to resolve the application’s path aliases as defined in the tsconfig.json file.

### Pass/Fail Criteria

A test was considered passed if all expect assertions within the script completed without error, which Vitest reports as passed with a "✓" mark in the terminal output. This indicates the function’s actual output matched the expected outcome. A test would fail if any assertion failed, meaning the actual output did not match the expected output. Vitest reports this with an “x” mark and provides a detailed error message as shown in the example image.

A screenshot of a computer program

AI-generated content may be incorrect.

If a test had failed, the error message would be reviewed, and the highlighted line of code examined to identify the cause. An issue would be created and added to the Kanban board. The code would then be corrected, the unit test re-run to confirm the fix, and the updated code committed to version control. At this point, the issue could be updated to Closed status. For the two specific tests, the conditions were:

* **SuccessResult Test:** This test would pass only if the created SuccessResult object correctly stored all properties (statusCode, responseTime) and if its methods returned the expected values (isHealthy() as true and getStatusText() as “Online”). It would fail if any property was stored incorrectly or if a method returned the wrong value.
* **Performance Statistics Test:** Given a mock dataset of 9 successful checks and 1 failure, this test would pass only if the calculatePerformanceStats function returned an object where the uptime property was exactly 90, indicating 90% uptime. It would fail if the calculation were inaccurate (e.g., returning 89.9 or 91).

## Specifications

The testing code was implemented in two separate script files. The SuccessResult test is in “src/lib/status-results/status-results.test.ts”:

A screen shot of a computer code

AI-generated content may be incorrect.

The Performance Statistics test script is in “src/lib/utils.test.ts”:

A computer screen shot of a program code

AI-generated content may be incorrect.

## Procedures

The testing process was completed using a detailed series of steps. First, during framework installation, the Vitest testing framework and its dependencies were installed using a single npm command. Next, a “vitest.config.ts” file was created at the project root for configuration. The project code was refactored for testability, where the logic for calculating performance statistics was extracted from the site’s dashboard component into a new, separate function named calculatePerformanceStats. The component was then updated ot use this new utility function. Then the test scripts were created using the “Arrange, Act, Assert” pattern. The test suite was then executed from the terminal using the npm test command with the verbose setting applied for greater visibility. During result verification, the terminal output was observed, and both tests were reported as passing. To confirm the tests were not passing by default, the input conditions were intentionally altered to trigger a failure, verifying that the test logic correctly detected incorrect behavior. The test plan documentation was updated to reflect the testing outcomes.

## Results

The unit tests were executed, and both passed successfully. In the first test case, an instance of the SuccessResult class was created, and the test verified that its properties correctly stored the statusCode, responseTime, and url as expected, resulting in a pass. In the second test case, the calculatePerformanceStats function was tested by providing a list of nine successful checks and one failure. The function correctly returned an object containing a total checks at 10 checks, incidents at 1 incident, and an uptime value of 90, matching the expected result and passing the test. While the testing process did not lead to direct code changes for bug fixes in this case, the preparation led to a code structure and maintainability. The test results confirm the proper function of the respective parts of the codebase.

A screenshot of a computer program

AI-generated content may be incorrect.

# Hosted Web Application

Hosted Web Application Link**:**

<https://statusdeck.vercel.app/>

# GitLab Repository & Branch History

GitLab Repository Link**:**

<https://gitlab.com/wgu-gitlab-environment/student-repos/zpfize3/d424-software-engineering-capstone/-/tree/Working?ref_type=heads>

GitLab Branch History**:**

A screenshot of a computer

AI-generated content may be incorrect.

# User Guide for Maintenance and Setup

To set up, run, and maintain the StatusDeck application in a local environment, ensure you have Node.js (v18+), npm, Git, the Supabase CLI, and accounts for both Supabase and Vercel.

## Local Development Setup

1. To run the application locally, first clone the repository from GitLab using “git clone” and install the required dependencies with “npm install”.
2. Next, initialize the local backend by logging into the Supabase CLI (“supabase login”) and running “supabase start”. This command will provide local API keys and a database URL—including the public anon key, the URL, and the secret SUPABASE\_SERVICE\_ROLE\_KEY—which can be copied into a new “.env.local” file at the project’s root.
3. In this same file, you must also add a CRON\_SECRET variable with a self-created random string of at least 16 characters to secure the cron job endpoint.
4. The “supabase start” command automatically applies the database schema from the “supabase/migrations” folder.
5. With the setup complete, run the development server with “npm run dev”, and the application will be available at “http://localhost:3000”.

## Core Maintenance Procedures

For routine maintenance, several key tasks are important.

1. To modify the database schema, generate a new migration file with “supabase migration new”, add your SQL changes, and then reset the local database with “supabase db reset” to apply them.
2. The project’s unit tests can be run at any time with the “npm test” command.
3. The automated status checks are powered by Vercel Cron Jobs, where the schedule is defined in the “vercel.json” file, which periodically calls the protected API route located at “api/cron/status-check”. Although the Vercel scheduler doesn’t run on your local machine, you can test the API endpoint’s logic before deploying. To do this, first run the local development server (“npm run dev”), then use a tool like curl to manually send a GET request to “http://localhost:3000/api/cron/status-check”. This request must include an Authorization header containing your CRON\_SECRET to simulate a legitimate request from Vercel and verify that the endpoint is working correctly.

## Deployment

The application is deployed to Vercel, with new deployments automatically triggered by a “git push” to the configured production branch. For a deployment to succeed, the production environment variables (NEXT\_PUBLIC\_SUPABASE\_URL, NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY, SUPABASE\_SERVICE\_ROLE\_KEY, and CRON\_SECRET) must be configured in the Vercel project settings. The cron jobs defined in vercel.json will only become active in the live production environment on Vercel.

# User Guide for Running the Application from User Perspective

## Introduction

This guide will include how to access, log into, sign up for, and use all the functions of the StatusDeck application.

## Accessing the Application

You can access the application by visiting its URL (<https://statusdeck.vercel.app/>) through your browser of choice. The application is designed for use on a desktop device.

## Login and Signup

1. You will be automatically routed to the Login page. Select “Create Account” to create your account.

A screenshot of a login form

AI-generated content may be incorrect.

1. Enter your email and password and click the “Create Account” button. The password requires at least 6 characters, with no specific required characters.

A screenshot of a login form

AI-generated content may be incorrect.

1. A confirmation email is sent to the email you signed up with. View the email and click the “Confirm your mail” link to confirm your account creation. This will redirect you back to the Login page.

A screenshot of a email

AI-generated content may be incorrect.

A screenshot of a sign up form

AI-generated content may be incorrect.

1. Now enter your login details and click “Sign in” to enter the application.

A screenshot of a login form

AI-generated content may be incorrect.

## Websites

### Add a New Website

1. Once logged in, click on the button at the top labeled “+ Add Site”. This will allow you to add a new website to track on your dashboard.

A screenshot of a computer

AI-generated content may be incorrect.

1. Enter the site name and URL of the site you want to monitor. Click “+ Add Site” on the modal to add the site to your dashboard.

A screenshot of a computer

AI-generated content may be incorrect.

1. You will see the new site added, with no monitoring data yet. The website status check will occur every 60 seconds, so data will be available shortly. You can refresh the page to see the new data.

A screenshot of a web page

AI-generated content may be incorrect.

1. Hover over dots on the chart to see more details about specific status checks.

A graph of a graph

AI-generated content may be incorrect.

1. Click on the website card to navigate to the site-specific details page. Here you will find a full-screen chart of the status checks, performance statistics, and performance analytics.

A graph with a line and numbers

AI-generated content may be incorrect.

1. Scroll down to view the Performance Analysis section.

A screenshot of a website

AI-generated content may be incorrect.

### Modify a Website

1. On the site-specific details page, select the gear icon located top left near the site name.

A graph with a line and green dots

AI-generated content may be incorrect.

1. Through this modal, you may update the site name or URL and click the “Save Changes” button to save the updates.

A screenshot of a computer

AI-generated content may be incorrect.

1. To delete the website, select “Delete Website” and confirm the deletion.

A screenshot of a computer error

AI-generated content may be incorrect.

### Filter Website Data

1. Locate the search box and filter button on the home page. You may search site names to dynamically display only matching sites.

A white background with black dots

AI-generated content may be incorrect.

1. Click on the “Filter” button to open a modal with different status codes to select. Click on the given status to only show websites with that status as the most recent check.

A screenshot of a computer

AI-generated content may be incorrect.

1. A similar “Filter” button is found on the site-specific page as well. This filter updates the chart to only show status checks that match the selected status.

A graph with a line and green dots

AI-generated content may be incorrect.

## Reports

1. To access the reporting feature, from the site-specific page, click on “Generate Report” near the top right of the page.

A graph with a line and a blue line

AI-generated content may be incorrect.

1. All status checks over the last 7 days are displayed in a tabular format.

A screenshot of a computer

AI-generated content may be incorrect.

## Profile and Sign Out

1. To view your user profile details, click on the profile icon in the top-right of the application.

A screenshot of a phone

AI-generated content may be incorrect.

1. Click on the “Sign Out” button to log out of your account.

A screenshot of a account

AI-generated content may be incorrect.