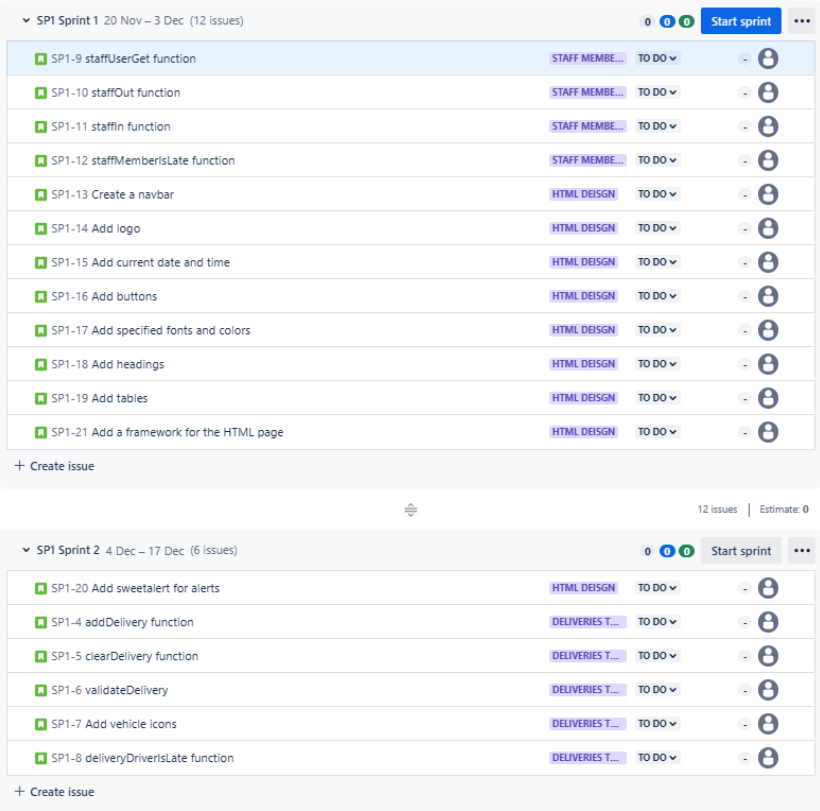
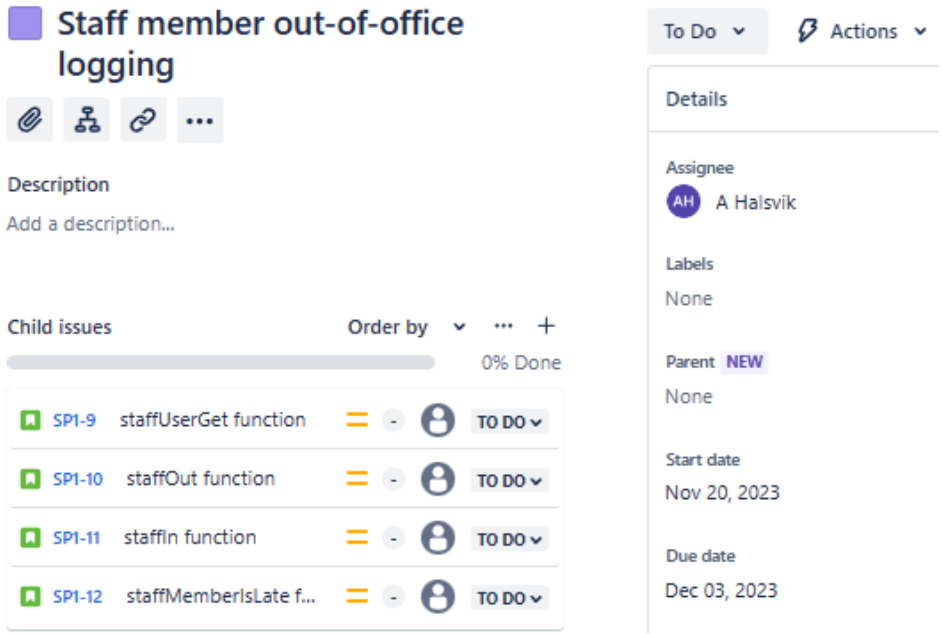
Reflection Report

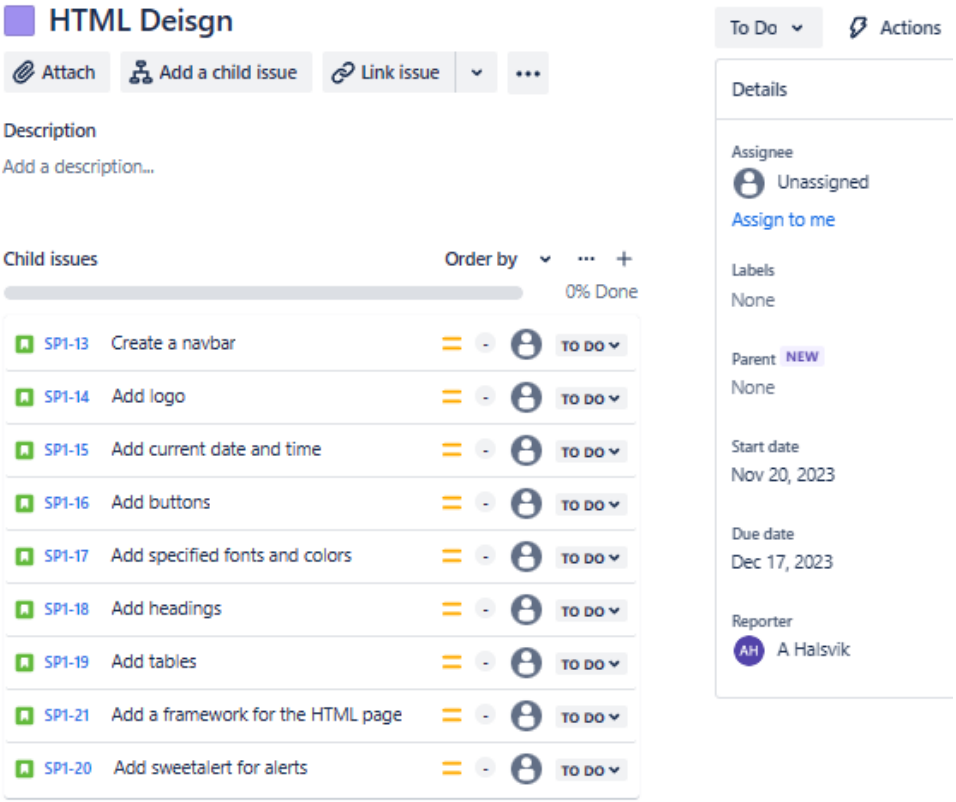
Sprints

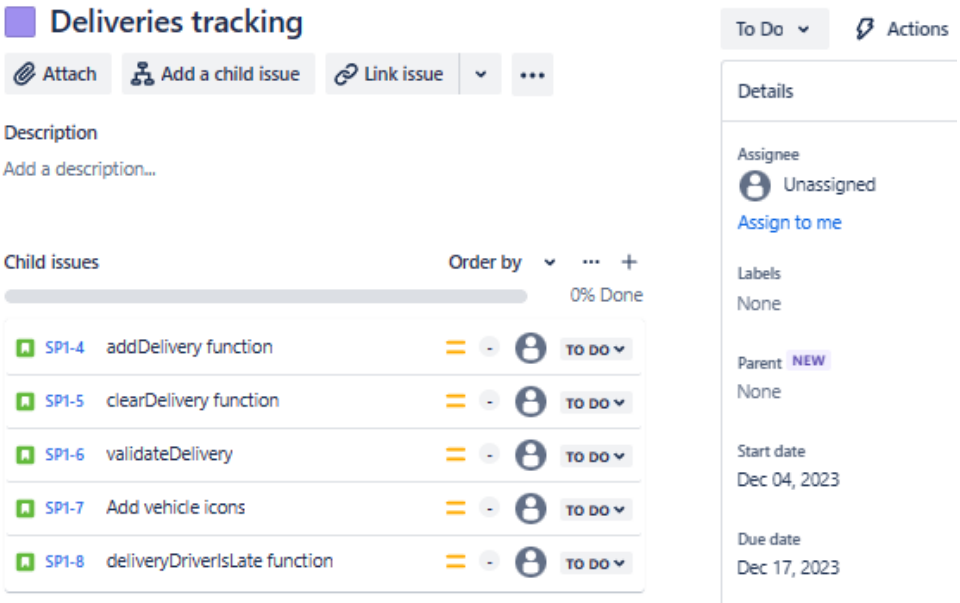


I chose two sprints due to the two primary components of this task: Staff Member out-of-office and Deliveries tracking. As the duration for this project was 4 weeks, I chose 2 weeks per sprint.

Epics and issues



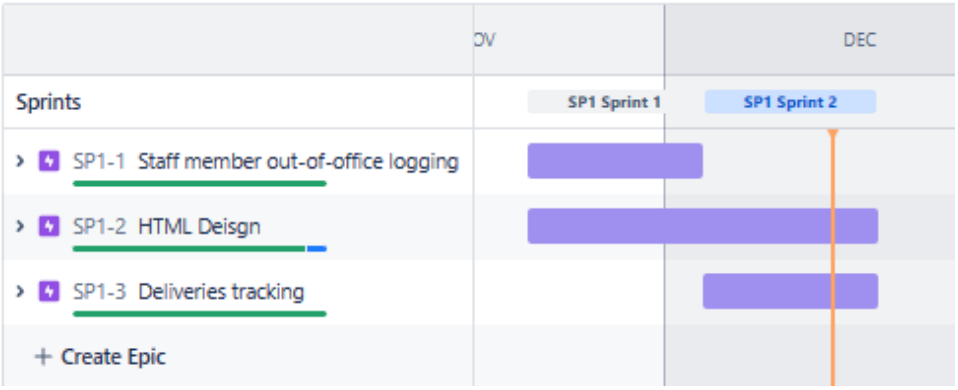




The Epics chosen in this project are based on the two primary components of the task, as well as the HTML design. The choice of issues within each Epic are based on the customers' requirements as well as the “Take note of the following” section of the task. In HTML Design, I chose issues that were essential HTML parts for meeting the requirements. Additionally, I included the date and time task in this Epic, as it did not match the theme of the other two Epics.

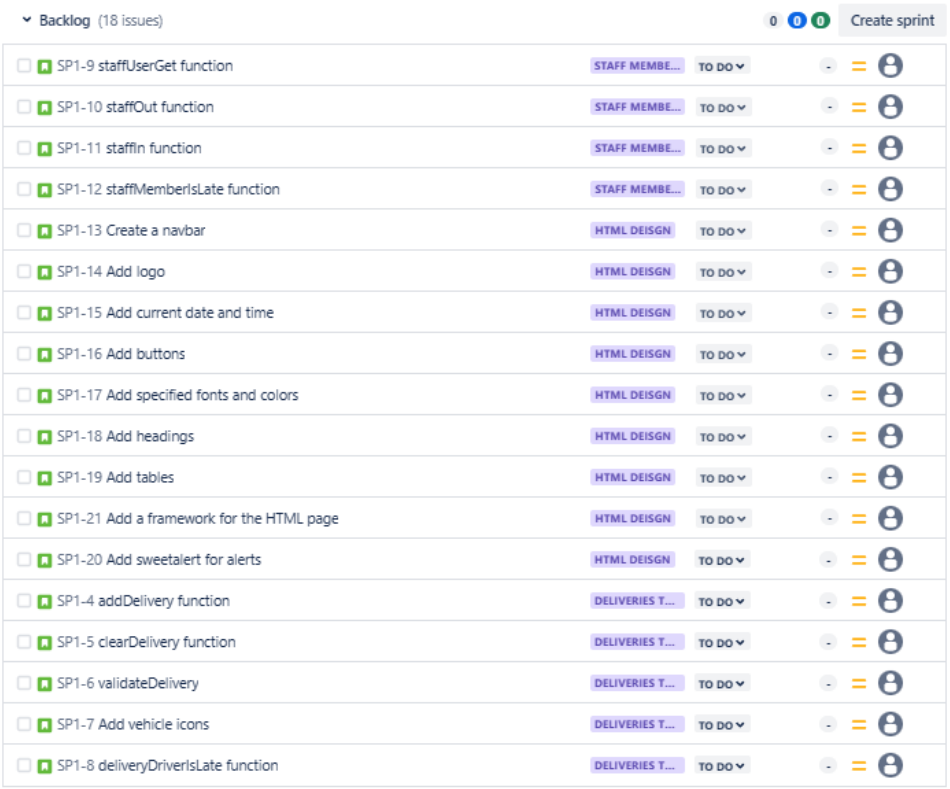
The HTML Design Epic spans over both sprints, as it needed more styling to better user experience when all functions were in place. Sweetalert alerts were added to the second sprint as it was planned to be included after necessary functions and HTML setup were in place. I intended to create new issues based on need along the way.

Project Roadmap



Backlog

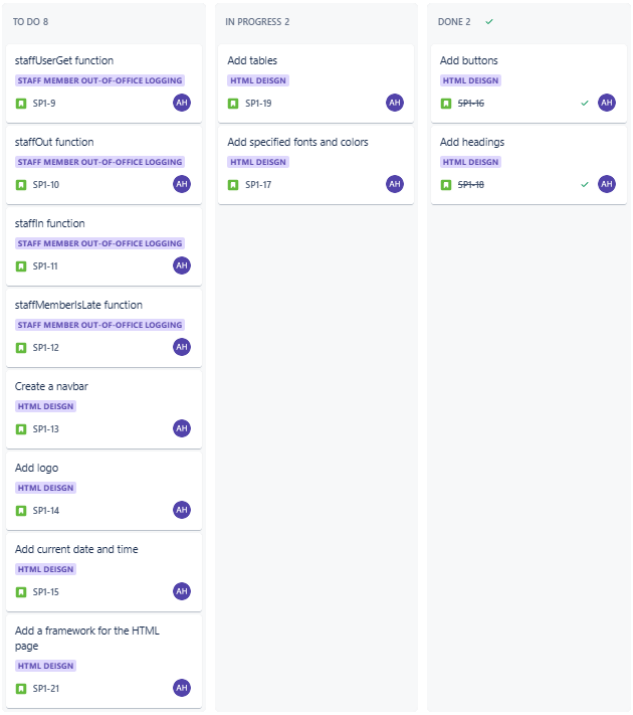
The first backlog before the sprints were created.



Board

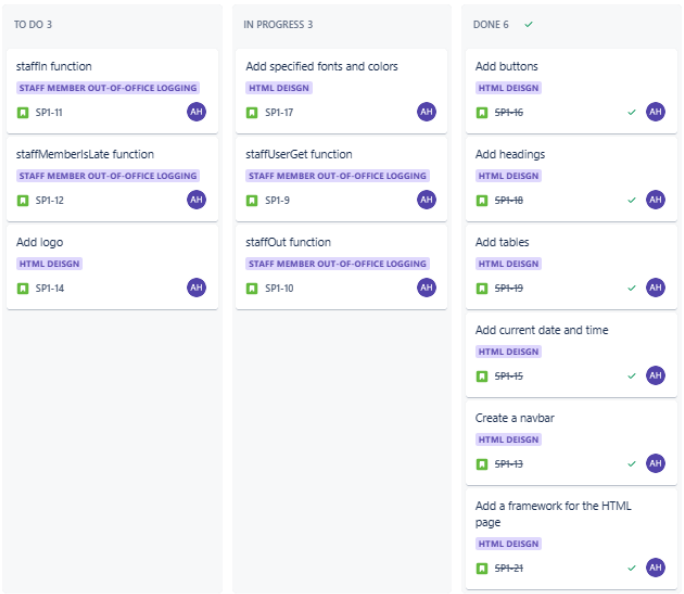
Sprint 1

In the early stages, I started off the project by working on the HTML design. This approach helped me adopt the right mindset before delving into more challenging tasks, such as creating functions. I developed a basic HTML setup and began adding Bootstrap design.

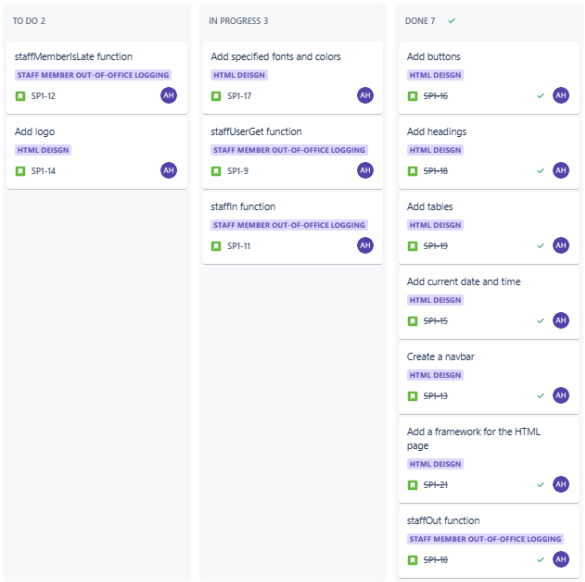


The specified fonts and colors remained in the "in progress" category throughout the entire sprint. My primary focus was on developing the functions after I had delved into them, and as I hadn't completed styling the HTML at that point, they stayed in the same category until Sprint 2 when the HTML was mostly finalized.

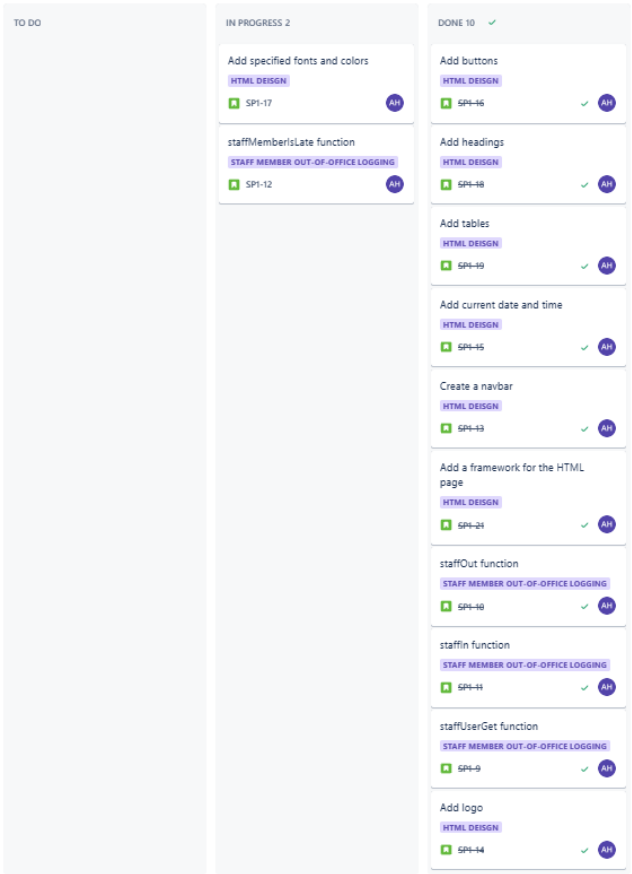
The first function I implemented for the project was the digitalClock function.



When I first delved into the staffUserGet function, I encountered challenges in generating more than one result. After spending hours without making any progress, I decided to set it aside temporarily and concentrate on the other functions.

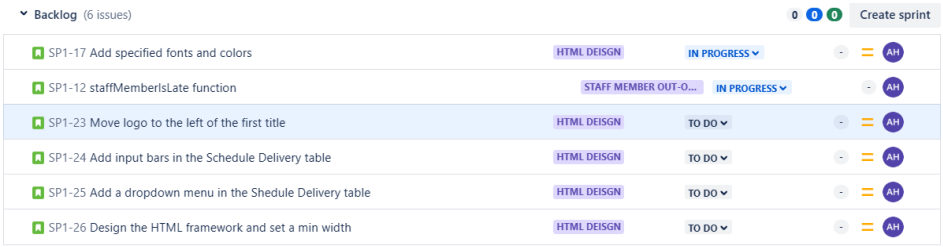


I completed the staffOut function before I moved on to the staffIn function. When I revisited the staffUserGet function again, the previous break turned out to be helpful as my subconscious had processed the issues, and I was now able to create five different results in the table. staffMemberIsLate is another function I struggled a lot with. In the end, I had two issues still “in progress” at the end of the sprint.

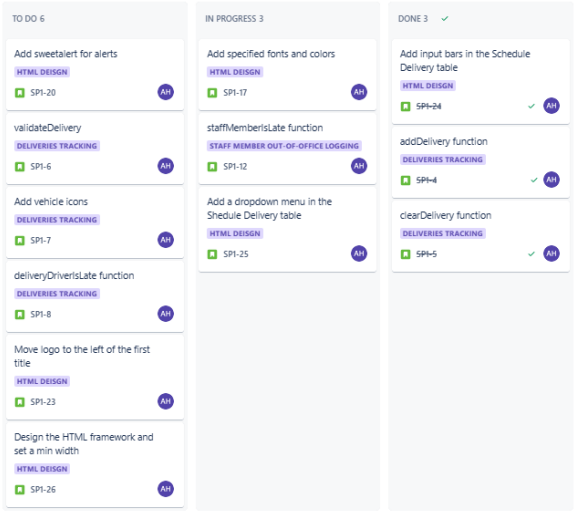


Sprint 2

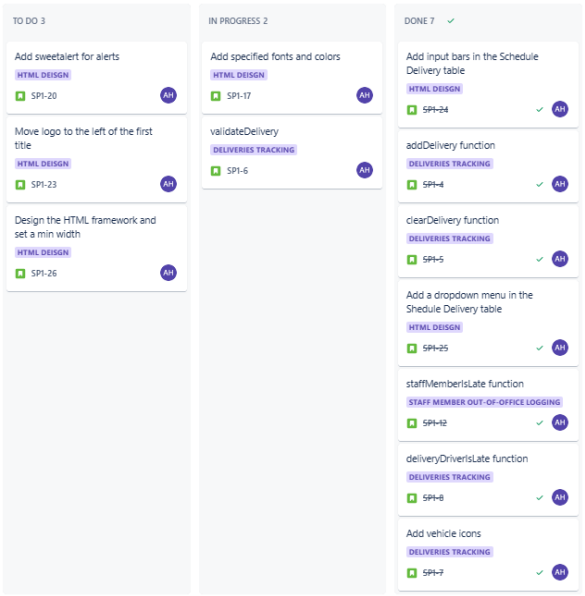
This is the backlog at det start of sprint 2, and these were all incorporated into the sprint.



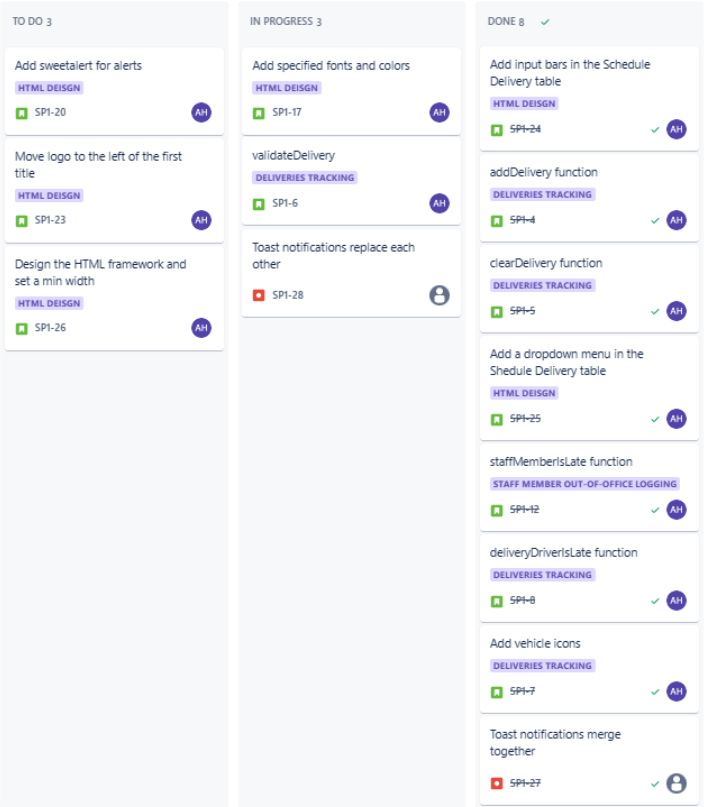
I began the sprint by adding input bars in the Schedule Delivery table. I then created an addDelivery function that utilized the input data to insert a new row into the Delivery Drivers table with said data. I also implemented the clearDelivery function. Then I implemented a dropdown menu in the Schedule Delivery table, so that the receptionist could have different vehicle options.



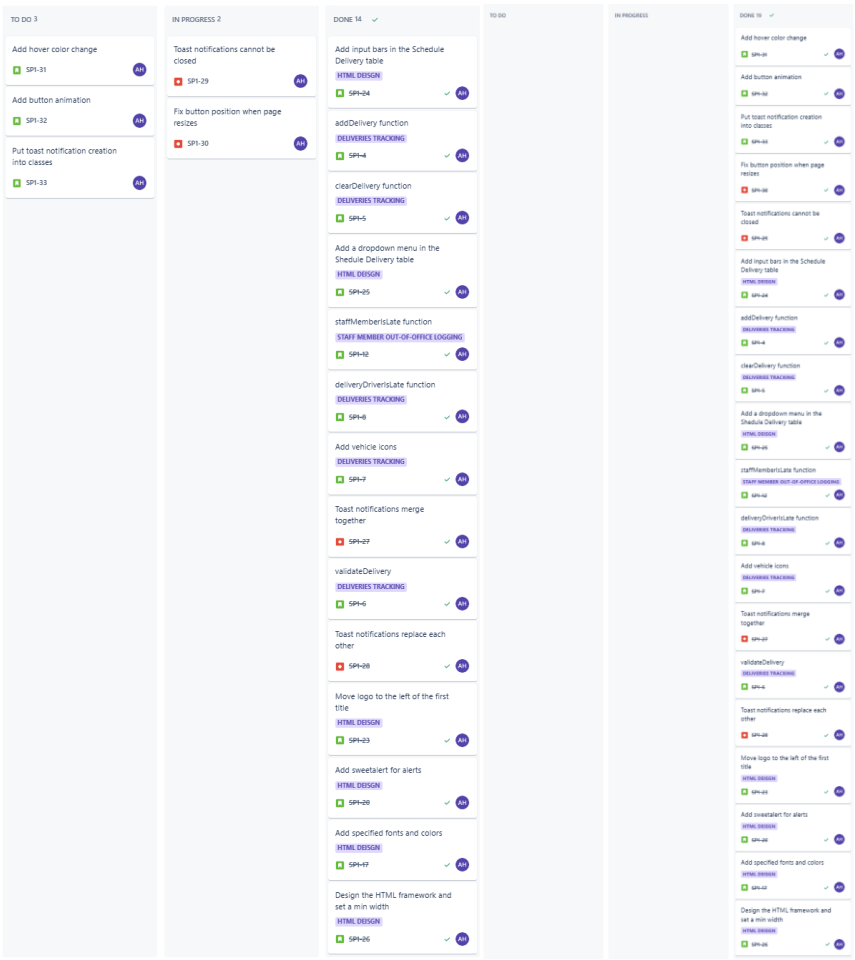
I finally managed to get the toast notification in the staffMemberIsLate function to work and replicated this functionality in the deliveryDriverIsLate function. Vehicle icons were added to the table.



I noticed a bug with the toast notifications during the sprint, where they appeared to merge together. I created a bug issue to address the problem in the sprint. I later realized this was because I was using .append(). After solving that issue, I noticed a new issue where the toast notifications were now replacing each other, instead of staying on the screen until the receptionist closed them.



I solved those bugs, finished the validateDelivery function and continued with HTML design. During this phase, I noticed a new bug where the toast notifications were not closable, and another bug related to the HTML design of buttons. These were added to the board. Additionally, I created three more issues, two of which were for HTML design purposes, while the third involved creating a class system for the toast creation as it wasn't working properly. I managed to implement object-oriented programming more at the end of this sprint, resulting in toast notifications that now operate as intended.


TO DO 3
Add hover color change
SPI-31
Add button animation
SPI-32
Put toast notification creation into classes
SPI-33
IN PROGRESS 2
Toast notifications cannot be closed
SPI-29
Fix button position when page resizes
SPI-30
DONE 14
Add input bars in the Schedule Delivery table
HTML DEISGN
SPI-24
addDelivery function
DELIVERIES TRACKING
SPI-4
clearDelivery function
DELIVERIES TRACKING
SPI-5
Add a dropdown menu in the Shedule Delivery table
HTML DEISGN
SPI-25
staffMemberIsLate function
STAFF MEMBER OUT-OF-OFFICE LOGGING
SPI-12
deliveryDriverIsLate function
DELIVERIES TRACKING
SPI-8
Add vehicle icons
DELIVERIES TRACKING
SPI-7
Toast notifications merge together
SPI-27
validateDelivery
DELIVERIES TRACKING
SPI-6
Toast notifications replace each other
SPI-28
Move logo to the left of the first title
HTML DEISGN
SPI-23
Add sweetalert for alerts
HTML DEISGN
SPI-20
Add specified fonts and colors
HTML DEISGN
SPI-17
Design the HTML framework and set a min width
HTML DEISGN
SPI-26
TO DO
IN PROGRESS
DONE 19

Summary

Developing this web application has brought a bunch of challenges, pushing me to make critical decisions that significantly influenced the project's outcome. This report dives into some of these choices and why they were made.

API Integration and Table Structures

One of the initial struggles of this project revolved around the API board. I didn't know how to create more than one result, nor did I know how to create multiple new table rows with unique data. To address this, I initially created individual table rows for each staff member with unique IDs, resulting in a cluttered HTML code (picture below on the left). I did manage to create more than one result, so this worked. However, I realized later that I needed a cleaner solution, so I revisited the staffUserGet function at the end of Sprint 2. I changed it so that it would create new table rows and implement the received data from the API into the row, creating 5 unique rows. This approach helped me clean up the HTML code, as well as mirroring the approach in the addDeliveryDriver function, giving my code a consistent look (picture below on the right).

```
<h1>Reception Management Dashboard</h1>
<h2>Staff</h2>
<table class="table table-bordered" id="staffTable">
    <thead>
        <tr>
            <th scope="col">Picture</th>
            <th scope="col">Name</th>
            <th scope="col">Surname</th>
            <th scope="col">Email address</th>
            <th scope="col">Status</th>
            <th scope="col">Out Time</th>
            <th scope="col">Duration</th>
            <th scope="col">Expected Return Time</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td id="picture1"></td>
            <td id="name1"></td>
            <td id="surname1"></td>
            <td id="emailAddress1"></td>
            <td id="status1"></td>
            <td id="outTime1"></td>
            <td id="duration1"></td>
            <td id="returnTime1"></td>
        </tr>
        <tr>
            <td id="picture2"></td>
            <td id="name2"></td>
            <td id="surname2"></td>
            <td id="emailAddress2"></td>
            <td id="status2"></td>
            <td id="outTime2"></td>
            <td id="duration2"></td>
            <td id="returnTime2"></td>
        </tr>
        <tr>
            <td id="picture3"></td>
            <td id="name3"></td>
            <td id="surname3"></td>
            <td id="emailAddress3"></td>
            <td id="status3"></td>
            <td id="outTime3"></td>
            <td id="duration3"></td>
            <td id="returnTime3"></td>
        </tr>
        <tr>
            <td id="picture4"></td>
            <td id="name4"></td>
            <td id="surname4"></td>
            <td id="emailAddress4"></td>
            <td id="status4"></td>
            <td id="outTime4"></td>
            <td id="duration4"></td>
            <td id="returnTime4"></td>
        </tr>
        <tr>
            <td id="picture5"></td>
            <td id="name5"></td>
            <td id="surname5"></td>
            <td id="emailAddress5"></td>
            <td id="status5"></td>
            <td id="outTime5"></td>
            <td id="duration5"></td>
            <td id="returnTime5"></td>
        </tr>
    </tbody>
</table>
```

```
<!-- Staff Table -->
<h2 class="display-5">Staff</h2>
<table class="table table-bordered" id="staffTable">
    <thead>
        <tr>
            <th scope="col">Picture</th>
            <th scope="col">Name</th>
            <th scope="col">Surname</th>
            <th scope="col">Email address</th>
            <th scope="col">Status</th>
            <th scope="col">Out Time</th>
            <th scope="col">Duration</th>
            <th scope="col">Expected Return Time</th>
        </tr>
    </thead>
    <tbody>
    </tbody>
</table>
```

Object-Oriented Programming

Early in the project, my understanding of object-oriented programming was limited, leading to less-than-optimal attempts, such as the initial toast implementation within the staffMemberIsLate and deliveryDriverIsLate functions. I had primarily only used the Employee, StaffMember and DeliveryDriver classes for object-oriented programming in this project as you can see below.

```
// Classes
class Employee {
    constructor(jsObject) {
        this.name = jsObject.name.first;
        this.surname = jsObject.name.last;
    };
};

class StaffMember extends Employee {
    constructor(jsObject) {
        super(jsObject);
        this.picture = jsObject.picture.medium;
        this.email = jsObject.email;
        this.status = "In";
    };
};

class DeliveryDriver extends Employee {
    constructor(jsObject) {
        super(jsObject);
        this.vehicle = jsObject.vehicle;
        this.telephone = jsObject.telephone;
        this.address = jsObject.address;
        this.returnTime = jsObject.returnTime;
    }
}
```

Realizing I needed to implement this more as it was a criterion and could help my toast creation, I began trying to implement it more, which had a big impact on how my code was structured.

Towards the end of Sprint 2, I began by trying to create a class system with inheritance for the toast creation. This took some time as I struggled to understand how to use object-oriented programming, but as time went on, I understood it more. This helped me implement this way of programming to other parts of my code. I additionally moved staffMemberIsLate and deliveryDriverIsLate functions in their respective classes, compared to leaving them as their own functions outside of these classes, as I realized that would be more object-oriented. I also added the rest of the properties provided by the customer's diagram to the classes.

```
// Classes
class Employee {
    constructor(jsObject) {
        this.name = jsObject.name.first;
        this.surname = jsObject.name.last;
    }
}

class StaffMember extends Employee {
    constructor(jsObject) {
        super(jsObject);
        this.picture = jsObject.picture.medium;
        this.email = jsObject.email;
        this.status = "In";
        this.outTime = jsObject.outTime;
        this.duration = jsObject.duration;
        this.returnTime = jsObject.returnTime;
    }

    staffMemberIsLate() {
        this.isLate = false;

        if (!intervalStaff) {
            intervalStaff = setInterval(() => {
                const now = new Date();

                if (staffMembers.every(member => member.status === "In") && intervalStaff) {
                    clearInterval(intervalStaff);
                    intervalStaff = null;
                }

                staffMembers.forEach((member) => {
                    if (member.returnTime < now && !member.isLate) {
                        const staffMembersToast = new StaffMemberToast(member);
                        staffMembersToast.createToast();
                        staffMembersToast.show();

                        member.isLate = true;
                    }
                });
            }, 1000);
        }
    }
}
```

```
class DeliveryDriver extends Employee {
    constructor(jsObject) {
        super(jsObject);
        this.vehicle = jsObject.vehicle;
        this.telephone = jsObject.telephone;
        this.deliverAddress = jsObject.deliverAddress;
        this.returnTime = jsObject.returnTime;
    }

    deliveryDriverIsLate() {
        this.isLate = false;

        if (!intervalDriver) {
            intervalDriver = setInterval(() => {
                const now = nowTime();

                if (deliveryDrivers.length === 0 && intervalDriver) {
                    clearInterval(intervalDriver);
                    intervalDriver = null;
                }

                deliveryDrivers.forEach((driver) => {
                    if (driver.returnTime <= now && !driver.isLate) {
                        const deliveryDriverToast = new DeliveryDriverToast(driver);
                        deliveryDriverToast.createToast();
                        deliveryDriverToast.show();

                        driver.isLate = true;
                    }
                });
            }, 1000);
        }
    }
}
```

In addition, I introduced arrays to handle the data better, one for Staff Members and one for Delivery Drivers. Understanding object-oriented programming more led to the implementation of a new property called isLate within the object. This property helped massively to avoid continuous creation of toast notifications.

```
// Arrays for objects
const staffMembers = [];
const deliveryDrivers = [];
```

Toast Notifications

The creation of toast notifications was another big challenge for me. I spent a lot of time on this specifically as it wouldn't always do what I wanted. I started by creating them directly from the staffMemberIsLate and deliveryDriverIsLate functions, but that didn't work well. I used a setTimer function to show these toasts after the specified time the employees should've been back. This caused problems with other parts of the code, such as the staffIn function and being able to close the toast. If I had more than one toast in line to be shown, then I had to wait until the last toast notification so show before the function was done, making the other functions work like normally again. You can see the code in the two pictures below.

```html
<!-- Toast Element Late Staff Member -->
<div id="lateStaffMember" class="toast position-absolute top-50 end-0 p-3" role="alert" aria-live="assertive"
aria-atomic="true" data-bs-autohide="false">
    <div class="toast-header">
        <strong class="me-auto text-danger">Staff Delay Alert!</strong>
        <button type="button" class="btn-close" id="close" data-bs-dismiss="toast" aria-label="Close"></button>
    </div>
    <div class="toast-body">
        <div id="imageToast"></div>
        <div id="staffMemberToast"></div>
        <strong>Time out-of-office: 0hr : 1 min</strong>
    </div>
</div>
```

```javascript
function staffMemberIsLate(name, surname, picture, minutes) {
    let toastDelay = minutes * 60 * 1000 + 1000;
    let newToast = $("#lateStaffMember").clone();

    newToast.find("#imageToast").html(`<img src="${picture}" alt="Staff Member">`);
    newToast.find("#staffMemberToast").html(`${name} ${surname} is delayed.`);

    $("body").append(newToast);

    setTimeout(function() {
        newToast.toast("show");
    }, toastDelay);
};
```

To solve this problem, I tried, as mentioned above, to create a class system for creating toasts. This solved all the problems I had with the previous toast creation, as well as organizing the code better.

```javascript
// Toast notification
class Toast {
    constructor() {
        this.newToast = $("#lateToast").clone();
    }

    createToast() {
        $("body").append(this.newToast);
    }

    show() {
        this.newToast.toast("show");
    }
}

class StaffMemberToast extends Toast {
    constructor(jsObject) {
        super();
        this.name = jsObject.name;
        this.surname = jsObject.surname;
        this.picture = jsObject.picture;
        this.duration = jsObject.duration;
    }

    createToast() {
        super.createToast();

        const textContent = `
            <div class="d-flex align-items-center">
                <img src="${this.picture}" alt="Staff Member">
                <div class="ms-2">Name: ${this.name} ${this.surname} is delayed.</div>
            </div>
            <strong class="mt-2">Time out-of-office: ${this.duration}</strong>
        `;

        this.newToast.find(".toast-body").append(textContent);
        this.newToast.find("#toastTitle").append("Staff Delay Alert!");
    }
}
```

```javascript
class DeliveryDriverToast extends Toast {
    constructor(jsObject) {
        super();
        this.name = jsObject.name;
        this.surname = jsObject.surname;
        this.address = jsObject.deliverAddress;
        this.telephone = jsObject.telephone;
        this.returnTime = jsObject.returnTime;
    }

    createToast() {
        super.createToast();

        const textContent = `
            <div>Name: ${this.name} ${this.surname} is delayed.</div>
            <div>Address: ${this.address}</div>
            <div>Telephone: ${this.telephone}</div>
            <strong>Estimated return time: ${this.returnTime}</strong>
        `;

        this.newToast.find(".toast-body").append(textContent);
        this.newToast.find("#toastTitle").append("Delivery Driver Delay Alert!");
    }
}
```

Conclusion

In summary, this project involved multiple challenges, specifically toast notifications and object-oriented programming. As I learned more about object-oriented programming, it improved the code structure and efficiency.