Fachhochschule Nordwestschweiz
Hochschule für Wirtschaft

Valo Dokumentation

Gruppenname:

Valo

Projektmitglieder:

Severin Nyffenegger

Kevin Bernet

Dozent:

Lukas Frey

Olten, 31.05.2024

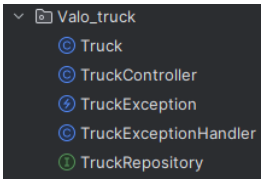
Inhalt

1 Back-End

Als Orientierung für den Back-End Aufbau haben wir uns an unser altes Projekt aus Software-Engineering gehalten. Dies gab uns die Möglichkeit klar und bewusst die Verschiedenen Klassen und die dazugehörigen Unter-Klassen zu erstellen.

- *Um den Server zu starten: Valo_Server/ServerStart.java*

1.1 Aufbau



Das Framework lautete wie folgt, pro Package haben wir eine Klasse erstellt (Tours, Truck, Customer, User, Package) und einen zugehörigen Klassen Controller. In diesem Controller verarbeiten wir allfällige Anfragen vom Server. Dazu gab es für jede Klasse eine Exception Unter-Klasse und einen passenden Handler. Schlussendlich wird noch für jede Klasse ein separates Repository, respektive separaten Table, aufgesetzt. Dies ermöglicht uns die Daten klar und übersichtlich zu speichern. Zudem erfolgt die Datensuche in einem Repository viel effizienter. Bei unserer Spezifikation sind wir davon ausgegangen, dass wir nur zwei Repositories brauchen, für Truck und User. Wir haben jedoch schnell in der Implementation bemerkt, dass dies nicht so funktionieren kann. Somit haben wir uns auf 5 verschiedene Repositories geeinigt. Zudem haben wir ein Package «Valo_Helper» erstellt für Hilfs-Klassen. Zum einen haben wir eine Klasse für das Initialisieren der Datenbank, wobei wir einen User, zwei Customer und zwei Trucks vorladen. Zum anderen gibt es die Token-Klasse, welche einen Token erstellt und allfällige Token überprüft.

Zusätzlich haben wir ein Package erstellt, welches die Tour Berechnung beinhaltet. Dies ist das Package «Valo_tourEngine». Darin enthalten ist die Klasse TourGenerator, welche die erhaltene Tour verarbeitet und in der optimalen Reihenfolge wieder zurückgibt. Wie dies genau funktioniert, wird unten in der Dokumentation beschrieben. Zudem gibt es eine Sub-Package «SearchAlgorithms», welches unser Suchalgorithmus beinhaltet. Dieser wird aus der Klasse TourGenerator aufgerufen. Der Suchalgorithmus basiert auf Nodes (mögliche Abladestationen) und den Edges, welche die Nodes verbindet mit der Distanz in Kilometer. In unserem Suchalgorithmus-Projekt in Software-Engineering Modul haben wir bereits mit dieser Methode gearbeitet, daher übernahmen wir die Vorgehensweise. Auf Geo-Admin konnten wir unser Netz an Abladestationen zeichnen und anschliessend unsere zwei benötigen «endgesTour» und «nodesTour» CSV-Dateien manuell erstellen. Wir haben uns auf mindestens eine Abladestation pro Kanton geeinigt. Diese zwei Dateien werden nun über die Klasse MapData im Package «SearchAlgorithms» eingelesen und können von jeglichen Suchalgorithmen verwendete werden. Die CSV-Dateien sind im Ressourcen Directory abgelegt.

1.1.1 Klassen Aufbau

Beim Aufbau der Klassen gibt es zwischen den Klassen nicht grosse Unterschiede, ausser bei den Klassen Tour und Package. Bei allen Klassen ist die definierende ID ein generierter Wert, ausser bei User, dort ist die ID der Nutzername. Die ID nimmt bei jedem Eintrag um eins zu. Damit können wir auch, nachdem die Werte in die Datenbank gespeichert wurden, diese mit der ID wiederfinden. Ansonsten werden die dazugehörigen Getter und Setter Methoden erstellt, dies ist bei allen Klassen gleich.

Bei den Klassen Tour und Package wurde eine OneToMany/ManyToOne Beziehung hinterlegt. Dies aus dem Grund, dass eine Tour verschiedene Pakete haben kann, jedoch nicht ein Paket verschiedene Touren.

```java
@OneToMany(fetch = FetchType.EAGER, mappedBy = "tours", cascade = CascadeType.ALL)
@JsonManagedReference
private List<Package> packages;
```

Tour Klasse

```java
@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "tourID")
@JsonBackReference
private Tour tours;
```

Package-Klasse

Mit dieser Verbindung wird nun im Package-Repository eine neue Spalte erstellt, welche die zugehörige TourID beinhaltet. Damit können wir einfach und effizient die Pakete aus der Datenbank ziehen und wissen zu welcher Tour diese gehören.

1.2 Tour Berechnung

Wie bereits erwähnt, werden alle Serveranfragen in den jeweiligen Controller Klassen bearbeitet. Bei der Tour Berechnung ist dies die Klasse TourController. Falls ein User eine Tour an den Server schickt, hat er zwei Möglichkeiten. Einerseits kann er die Route nur berechnen lassen mit der URL «/tours/generate», dies gibt dem User die korrekte Reihenfolge der Abladestationen zurück ohne, dass die Tour in der Datenbank gespeichert wird. Anderseits gibt es die Möglichkeit die Tour berechnen zu lassen und in der Datenbank zu speichern mit der URL «/tours/save». Die zwei Methoden sind identisch aufgebaut, nur das Speichern unterscheidet sich.

```java
@PostMapping("/tours/save")
Tour tourSave(@RequestBody Tour tour) {
    if (Token.validate(tour.getToken())) {

        Tour tourIn = new Tour(tour.getTruckID());

        List<Package> packages = new ArrayList<>();
        for (Package packageIn : tour.getPackages()) {
            Package pck = new Package(packageIn.getPackageWeight(), packageIn.getDeliveryAddress(), packageIn.getCustomerID());
            pck.setTours(tourIn);
            packages.add(pck);
        }
        tourIn.setPackages(packages);


        tourIn = tourGenerator.generateTour(tourIn);

        tourRepository.save(tourIn);
        System.out.println("Saved Tour object: " + tourIn);

        return tourIn;
    } else {
        throw new TourException("Invalid token");
    }
}
```

Die Methoden sind beide @PostMapping, welche ein Tour-Objekt als Parameter haben und auch eine Tour zurückgeben. Wie bei allen Serveranfragen wird zuerst überprüft, ob es sich um eine valide Anfrage handelt. Dies machen wir mit dem Überprüfen des Tokens, da falls ein User nicht eingeloggt ist, kann dieser auch das System nicht benutzen. Wir erstellen zuerst mit der TruckID ein neues Tour Objekt. Danach ziehen wir mit der for-Methode die Pakete aus dem alten Tour Objekt. Damit erstellen wir ein neues Package Objekt und setzen dieses neue Tour Objekt als die zugehörige Tour. Sobald dies erfolgte, fügen wir das Package Objekt eine Liste von allen erhaltenen Package Objekte hinzu. Diese Liste wird dem neuen Tour Objekt als Packages gesetzt. Somit haben wir nun die Tour mit den zugehörigen Paketen erstellt. Diese Tour können wir nun der Klasse TourGenerator weitergeben, um die korrekte Reihenfolge, wie auch die Distanz und ungefähre Zeit, zu erhalten.

```java
List<Package> packages = tour.getPackages();
ArrayList<String> destinations = new ArrayList<>();
Map<String, List<Package>> addressToPackageMap = new HashMap<>();

for (Package pck : packages) {
    String address = pck.getDeliveryAddress();
    destinations.add(address);
    addressToPackageMap.computeIfAbsent(address, k -> new ArrayList<>()).add(pck);
}
```

In der Klasse TourGenerator werden zuerst die Abladestationen aus der Tour gefiltert. Zudem werden die Abladestationen mit dem Package Objekt in einer HashMap verlinkt, um die neue optimale Reihenfolge der Abladestationen wieder mit den Package Objekten zu verlinken. Sobald wir die Liste mit den

Abladestationen erstellt haben, geben wir diese mit unserem Startpunkt Olten (Standort des Unternehmens) weiter an unser Suchalgorithmus TSP (Travel salesmen problem).

```
ArrayList<String> orderedStops = findBestTour(start, stops);
double totalDistance = 0;
String currentNode = start;

    for (String stop : orderedStops) {
        Path path = AStar(currentNode, stop);
        if (path == null) {
            System.out.println("No path found from " + currentNode + " to " + stop);
            return result;
        }
        //printPath(path);
        totalDistance += path.totalDistance;
        currentNode = stop;
    }
    result = orderedStops;
    String str = Double.toString(totalDistance);
    result.add(str);

return result;
```

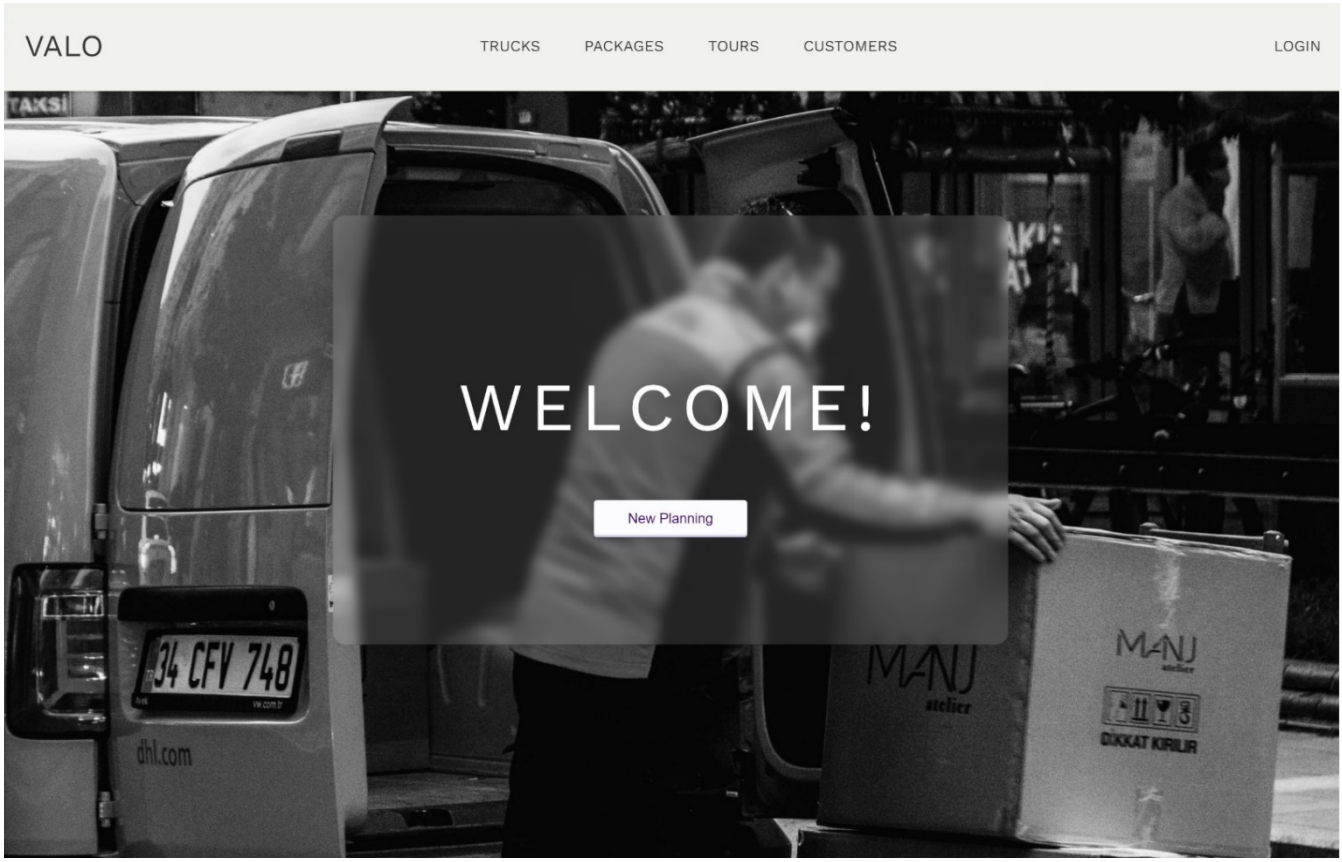
Dieser Algorithmus basiert auf unserem bereits erstellen A* Algorithmus aus unserem Software Engineering Projekt. Zusammen mit Chat-GPT haben wir diesen weiterentwickelt, damit dieser nicht nur die kürzeste Distanz findet zwischen zwei Orten, sondern auch verschiedene Zwischenstopps berücksichtigen kann. Interessanterweise konnten wir den gesamten A* Algorithmus übernehmen für die exakte Distanzberechnung. Jedoch mussten wir zuerst noch die optimale Route finden mit der Methode findBestTour(). In dieser Methode werden alle möglichen Permutationen berechnet und anschliessend die Möglichkeit mit der kürzesten Distanz gewählt. Diese Art ist auch bekannt unter Brut-Force. Sobald wir das Resultat haben, wird es weitergegeben in den for-loop, welcher mit dem Startort beginnt, in unserem Fall ist dies Olten, danach wird Schritt für Schritt die Distanz zwischen den Orten mittels dem bekannten A* Algorithmus berechnet. Zudem wird die Erhaltene Distanz zu einer summierten Distanz dazugezählt. Schlussendlich fügen wir die optimale Route mit der totalen Distanz zusammen und geben diese als ArrayList zurück. Somit haben wir in einer Antwort zwei Antworten gespeichert.

```
System.out.println("Stops: " + destinations);
ArrayList<String> result = TSP.runTSP( start: "Bern", destinations);
String distance = result.get(result.size() - 1);
double totalDistance = Double.parseDouble(distance);
result.remove(distance);

List<Package> optimalRoutePackages = new ArrayList<>();
for (String address : result) {
    List<Package> packageList = addressToPackageMap.get(address);
    if (packageList != null && !packageList.isEmpty()) {
        optimalRoutePackages.add(packageList.remove( index: 0));
    }
}
```

Diese Antwort nehmen wir in der Klasse TourGenerator noch auseinander und Löschen die totale Distanz aus der ArrayList. Mit dieser Liste der korrekten Reihenfolge der Abladestationen können wir nun dank der bereits erstellten HashMap mit den korrekten Package Objekten verbinden. Dies passt uns auch die Paketreihenfolge an und somit können wir schlussendlich die Liste optimalRoutePackage dem Tour Objekt wieder zurückgeben. Die totale Distanz teilen wir noch durch 80, da wir davon ausgehen, dass die Fahrer eine Durchschnittgeschwindigkeit von 80 km/h haben. Diese geschätzte Zeit und die Totale Distanz setzen wir dem Tour Objekt und geben dies dem TourController so zurück. Dieser speichert das Tour Objekt noch in die Datenbank und gibt schlussendlich wieder das neue Tour Objekt zurück. Mit der Rückgabe des Tour Objektes weiss auch das Front-End, was nun die beste Reihenfolge für die Tour ist.

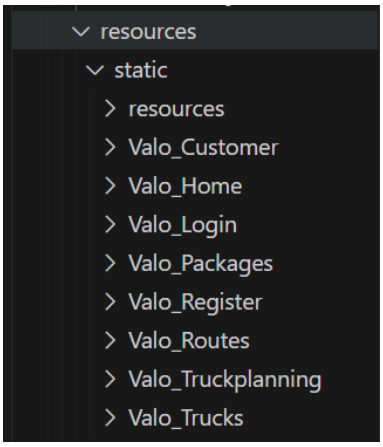
2 Front-End

Für das Front-End haben wir uns für eine Webseite entscheiden, die wir mit HTML, CSS und Javascript programmierten. Im Modul Internettechnologien hatten wir bereits eine Webseite erstellt und wir konnten einige der Funktionen davon übernehmen und anpassen. Das Layout haben wir auch teils aus dem alten Projekt übernommen. Unten ist die Startseite der Webseite zu sehen.



2.1 Aufbau

Der Aufbau ist in der Grafik recht zu sehen. Wir haben für jede Teilseite einen Ordner erstellt mit den dazugehörigen Klassen. Beispielsweise für die Login-Seite gibt es ein login.html sowie ein loginScript.js, so sind alle Ordner aufgebaut. Das CSS-File und die anderen verwendeten Ressourcen sind im Ressourcen Ordner gespeichert. Wir haben nur ein CSS-File erstellt, damit die Styles von beispielsweise eines Knopfs nur einem Codiert werden muss. Das ganze Front-End ist im static-Ordner des Projekts abgelegt, damit kann die Applikation mit der ServerStart.java gestartet werden und dann im Browser unter:



http://localhost:8080/Valo_Home/home.html

aufgerufen werden.

2.2 HTML

Alle Teilseiten sind ähnlich aufgebaut, die Navigationsleiste, das Alert-Fenster und die Fusszeile sind bei allen Seiten genau gleich. Die Fusszeile hat einige Informationen, über Valo, die sind frei erfunden und dienen nur dazu, dass die Webseite professioneller wirkt. Einzig, der Webseitencontent ist von Seite zu Seite anders, aber da haben wir grob zwei unterschiedliche aufbauten.

```
<!-- Website Content -->
<div class="module" id="module-login"></div>
<div class="div-form">
    <h1 class="h1-form">Login</h1>

    <div class="form__group field">
        <input type="input" class="form__field" placeholder="Username" name="username" id='username' required />
        <label for="username" class="form__label">Username</label>

        <input type="input" class="form__field" placeholder="Password" name="password" id='password' required />
        <label for="password" class="form__label">Password</label>

      </div>
      <div class="login_register"><p class="p_login">Don't have an account? </p>
        <a class="a_login" href="../Valo_Register/register.html"> Register</a>
    </div>
    <button class="button-login" onclick="loginButtonClick()">Login</button>
  </div>
```

Ein Aufbau ist die Eingabeform, wo vom User Eingaben getätigt werden müssen, wie im Beispiel oben die Login-Informationen wie der Username und das Passwort. Zusätzlich sind da noch zwei Buttons, einer um in das Register-Fenster zu wechseln und ein Knopf um sich anzumelden. Das funktioniert natürlich nur, wenn die eigegebenen Informationen stimmen und der User bereits registriert ist.

```
<!-- Website Content -->
<div class="module" id="module-trucks"></div>
<div id="toursContainer" class="tours-container">
  <!-- Tour cards will be dynamically added here -->
</div>
  <button class="addTour-button" onclick="changeToTruckplanningWindow()">New Tour</button>
```

Der zweite Aufbau ist am Beispiel der Torus gezeigt. Da ist deutlich weniger Code, denn die Webseite wird dynamisch durch das Skript gefüllt mit den Informationen, die vom Server erhalten werden. Das wird im nächsten Kapitel beschrieben.

2.3 Javascript

Wie bereits im Aufbau beschrieben, haben wir für jede Teilseite ein einzelnes Skript erstellt, um einen Überblick der Funktionen zu haben und eine Struktur zu erhalten. Aus diesem Grund haben wir in jedem Skript einige Funktionen drin, die auf jeder Seite verwendet werden. Das sind die Alert-Funktionen, die dazu verwendet werden, um Alerts zu zeigen, die schöner aussehen, wie die Standart-Alerts vom Browser selbst und die Login- und Logout-funktionen, die auf jeder Seite aufgerufen werden. Die Alert-Funktionen werden an verschidenen Stellen im Code aufgerufen und werden daraus mit der showAlert(message) aufgerufen, die Message, die angezeigt werden soll,

```
//ALERT FUNCTIONS
function showAlert(message) {
    const customAlert = document.getElementById('customAlert');
    const alertMessage = document.getElementById('alertMessage');

    alertMessage.textContent = message;
    customAlert.style.display = 'block';
}

function hideAlert() {
    const customAlert = document.getElementById('customAlert');
    customAlert.style.display = 'none';
}

document.getElementById('closeAlertButton').addEventListener('click', hideAlert);
```

muss als Parameter mitgegeben werden. Wenn im Alert-Fenster der «Close» Knopf gedrückt wird, wird der Alert wieder geschlossen.

```
//LOGIN FUNCTIONS
  document.addEventListener('DOMContentLoaded', function() {
    const loginButton = document.getElementById('loginRegisterButton');
    const loginStatus = localStorage.getItem('loginStatus');

    // Initialize loginStatus if it's not set in localStorage
    if (!loginStatus) {
      localStorage.setItem('loginStatus', 'Login');
      loginStatus = 'Login';
  }

    if (loginStatus === 'Logout') {
        loginButton.textContent = 'Logout';
    } else {
        loginButton.textContent = 'Login';
    }
});
```

Im lokalen Speicher werden drei Informationen gespeichert, einerseits wird beim Login der Token und der Username gespeichert und immer gespeichert ist der Login-Status, der kann entweder «Login» oder «Logout» sein. Beim ersten Öffnen der Webseite ist der Login-Status auf «Login», das heisst, dass der Button oben rechts den Text «Login» hat und der User sich somit zuerst einloggen muss. Der Status wird beim Login und Logout des Users jeweils angepasst. Die anderen beiden Informationen im lokalen Speicher werden bei den Interaktionen mit dem Server benötigt, bei den meisten Anfragen an den Server wird der Token des Users verwendet, um zu verifizieren, dass er Änderungen vornehmen kann. Der Username wird für das Logout verwendet.

2.3.1 Anzeigen von Datenbankeinträgen

```
<!-- Website Content -->
<div class="module" id="module-trucks"></div>
    <div id="customersContainer" class="customers-container">
      <!-- Customer cards will be dynamically added here -->
    </div>
</div>
```

```
//Load customers into the table
document.addEventListener('DOMContentLoaded', function() {
    const customersContainer = document.getElementById('customersContainer');

    function fetchcustomers() {
      fetch('http://localhost:8080/customers')
        .then(response => response.json())
        .then(customers => {
          customersContainer.innerHTML = '';

          customers.forEach(customer => {
            const customerCard = document.createElement('div');
            customerCard.classList.add('customer-card');

            customerCard.innerHTML = `
              <div class="customer-info">
                <h3>Customer ${customer.customerID}</h3>
                <p><strong>Name:</strong> ${customer.customerName}</p>
                <p><strong>Address:</strong> ${customer.addressName}</p>
                <p><strong>City:</strong> ${customer.cityName}</p>
              </div>
              <button class="delete-button" data-customer-id="${customer.customerID}">x</button>
            `;

            customersContainer.appendChild(customerCard);

            const deleteButton = customerCard.querySelector('.delete-button');
            deleteButton.addEventListener('click', handleDeleteButtonClick);
          });
        })
        .catch(error => {
          console.error('Error fetching customers:', error);
        });
    }
```

Die Methode, um die Einträge von der Datenbank auf der Webseite anzuzeigen, wird am Beispiel der Customer gezeigt. Zuerst wird der EventListener ausgeführt, sobald die Seite geladen wird, danach wird auf die «customerContainer» div im HTML code zugegriffen. Danach werden die Customer von dem Server abgerufen und die Inhalte in container gelöscht, damit danach die aktuellen reingeladen werden können. Dann wird für jeden einzelnen Customer eine customer-card erstellt die dann untern mittels innerHTML befüllt werden mit den Informationen aus der Datenbank die per JSON-Format zurückgeschick werden. Danach wird die customer-card dem container hinzugefügt und dem delete-button die handleDeleteButtonClick funtion hinzugefügt, dass mann die einzelnen Customers auch wieder löschen kann. Falls ein Fehler in dem Prozess entsteht, wird der in der Konsole ausgegeben. Die Methoden um die anderen Datenbankeinträge anzuzeigen sind gleich aufgebaut.

2.3.2 Löschen von Datenbankeinträgen

```
function handleDeleteButtonClick(event) {
  const customerId = event.target.dataset.customerId;

  fetch(`http://localhost:8080/customers/delete/${customerId}`, {
    method: 'GET'
  })
    .then(response => response.json())
    .then(data => {
      if (data.Customer === "deleted") {
        showAlert('Customer has been successfully deleted.');
        event.target.closest('.customer-card').remove();
      } else {
        showAlert('Unexpected response from server.');
      }
    })
    .catch(error => {
      console.error('Error:', error);
      showAlert('An error occurred. Please try again.');
    });
}
```

Die Methode, um Datenbankeinträge zu löschen wird ebenfalls am Customer gezeigt. Der «event» im Parameter der Methode kommt vom eventListener, der in der Methode oben definiert wird, somit wird das richtige Objekt ausgewählt. Danach wird aus diesem Objekt die customerID rausgezogen, die man dann der GET-Methode mitschicken muss, damit der Customer gelöscht werden muss. Wenn die Antwort vom Server «deleted» schickt, wurde der Customer korrekt gelöscht und die customer-card kann gelöscht werden. Wenn vom Server eine andere Antwort gesendet wird, wurde der Customer nicht gelöscht und die customer-card wird auch nicht gelöscht.

2.3.3 Hinzufügen von Datenbankeinträgen

```
// Function to handle the Button to get to the add customer page
function changeToAddCustomerPage(){
  var token = localStorage.getItem('token');

  if (token === null || token === 'null') {
      showAlert("You need to login first!");
  } else {
    window.location.href = '../Valo_Customer/addCustomer.html';
  }

}
```

Zur Illustration verwenden wir auch hier die Customer. Auf die Seite um neue Kunden einzutragen, kommt man nur aus der Customer-Seite mit dem Knopf «Add Customer» oben rechts. Wenn man den Knopf drückt, ohne, dass man eingeloggt ist, wird man nicht auf die addCustomer-Seite weitergeleitet. Dies wird mit dem Token im lokalen Speicher überprüft.

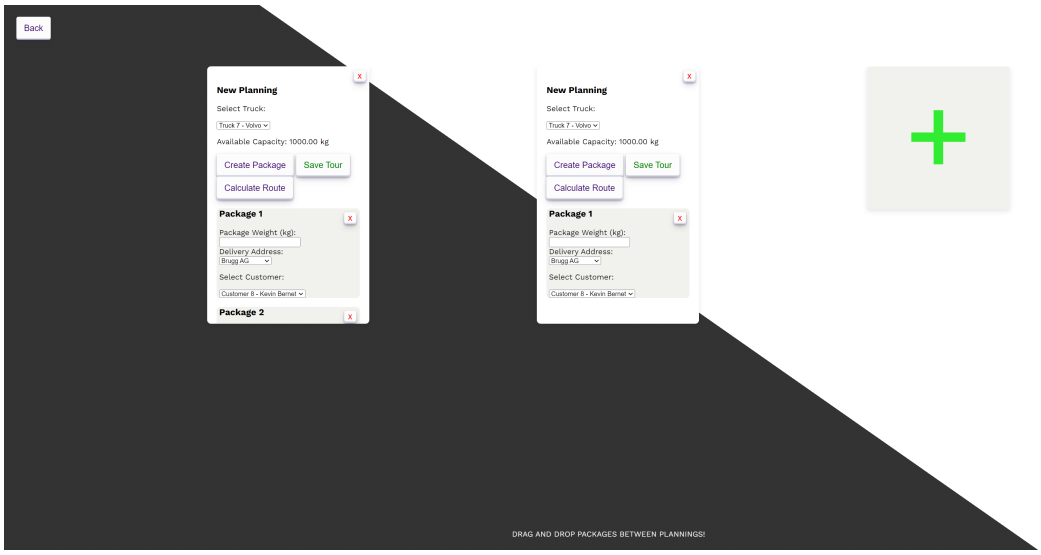
```
// Function to handle newCustomerButton click
function newCustomerButtonClick() {
  var customerName = document.getElementById('name').value;
  var customerAddress = document.getElementById('address').value;
  var customerCity = document.getElementById('city').value;

  if (!customerName || !customerAddress || !customerCity) {
    showAlert('All fields are required. Please fill in all fields.');
    return; // Prevent the request from being sent
}

  var token = localStorage.getItem('token');

  const customerData = {
      customerName: customerName,
      addressName: customerAddress,
      cityName: customerCity,
      token: token
  };

  // Send a POST request to the server to add a new customer
  fetch('http://localhost:8080/customers/new', {
      method: 'POST',
      headers: {
          'Content-Type': 'application/json'
      },
      body: JSON.stringify(customerData)
  })
  .then(response => {
      if (!response.ok) {
          throw new Error('Error: ' + response.status);
      }
      return response.json();
  })
  .then(data => {
      window.location.href = '../Valo_Customer/customers.html';
  })
  .catch(error => {
      console.error('Error:', error);
      showAlert('An error occurred, please try again.');
  });
}
```

Auf der Seite um Kunden hinzuzufügen müssen die drei Felder: Name, Address und City ausgefüllt werden, damit der Customer abgespeichert werden kann, sonst wird eine Fehlermeldung ausgegeben. Sind alle drei Angaben gemacht, wird der Token aus dem lokalen Speicher geholt und die customerData werden mit den Informationen, die aus der Form geholt werden, abgefüllt. Danach wird die customerData mittels einer POST-Methode als JSON-format an den Server geschickt. Wenn das erfolgreich geschieht, wird der User wieder auf die Customer-Seite weitergeleitet, wo der neue Kunde direkt angezeigt wird. Falls ein Fehler geschieht, wird der User nicht weitergeleitet und ein Alert wird angezeigt, dass etwas nicht funktioniert hat.

2.3.4 New Planning

```
function startNewPlanning() {
    const newPlanningCard = document.createElement('div');
    newPlanningCard.classList.add('planning-card');

    newPlanningCard.innerHTML = `
        <div class="card-header">
            <h3>New Planning</h3>
            <p>Select Truck:</p>
            <select id="truckSelect" onchange="updateAvailableCapacity(this.options[this.selectedIndex])"></select>
            <p id="availableCapacityLabel">Available Capacity: 0 kg</p>
            <button onclick="createPackage()">Create Package</button>
            <button onclick="saveTour(event)" data-action="saveTour" class="saveTour-button">Save Tour</button>
            <button onclick="deleteTour(event)" data-action="deleteTour" class="delete-button">x</button>
            <button onclick="calculateRoute(event)" data-action="calculateRoute">Calculate Route</button>
        </div>
        <div class="package-container"></div>
    `;


    const planningContainer = document.getElementById('plannningContainer');

    planningContainer.insertBefore(newPlanningCard, planningContainer.firstChild);

    fetchTrucksForDropdown();

}
```

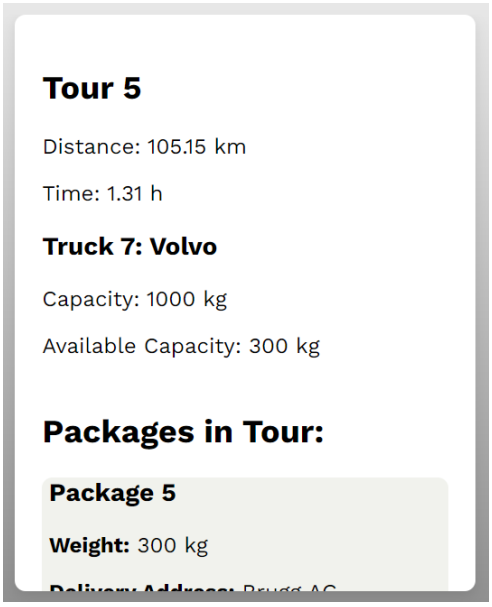
Das grösste Skript ist das, um eine neue Tour zu erstellen. Wenn eine neue Tour gestartet wird, wird eine neue «planning-card» erstellt, die dann mit einigen Objekten befüllt werden. Das Dropdown-Menu für die Trucks zeiht alle verfügbaren Trucks in der Datenbank und speichert diese mit der ID und dem Namen des Trucks in das Menu. Die verfügbare Kapazität wird je nach Truck angepasst. Vier Buttons werden noch erzeugt für jede Tour. Einer erstellt ein neues Package, einer Berechnet die schnellste Route und ordnet die Pakete entsprechend in der Tour an, ein Button speichert die Tour in der Datenbank und der letzte löscht die Tour in der Planung. Es können mehrere Touren gleichzeitig geplant werden, auch mit demselben Truck, die werden dann einfach nacheinander ausgeliefert. Jede neue Tour wird jeweils ganz vorne angezeigt.



Im Beispiel oben sind zwei geplante Touren gestartet mit je einem und zwei Paketen. Die einzlnen Pakete können auch gelöscht werden oder zwischen den verschiedenen Touren mittels Drag-and-Drop verschoben werden, dafür haben wir die Pakete «dragable» gemacht. Damit die beste Route berechnet oder die Tour gespeichert werden kann, müssen alle Angaben eingetragen werden, sonst wird eine Meldung ausgegeben. Ebenfalls kann eine Toru nicht gespeichert werden, wenn das zusammengerechnete Gewicht der Pakete die Kapazität des Trucks übersteigt, dann wird die «Available Capacity» rot markiert. Die beste Route wird auch beim Speichern einer Route berechnet, bevor diese in die Datenbank gespeichert wird. Die «Delivery Adress» von jedem Paket wird aus dem csv-File, mit den nodes gezogen, damit nur mögliche Lieferadressen ausgewählt werden, damit der Algorithmus für die Berechnung der Tour funktioniert. Der customer je Paket

wird auch aus der Datenbank gelesen, damit auch hier nur die ausgewählt werden können, die es auch gibt.

Eine gespeicherte Tour ist in der Abbildung rechts zu sehen. In der Tour wird die berechnete Distanz, die Dauer der Auslieferung, welcher Truck ausgewählt wurde mit der Totalen Kapazität und der noch verfügbaren Kapazität sowie alle Pakete mit den Informationen angezeigt.



3 Test

3.1 Unit Test

Um die Funktionalität des Servers zu testen haben wir für jede Server Anfrage einen Unittest geschrieben. Diese Unittests basieren auf der bereits in SpringBoot enthaltenen Annotation @Test. Mit dieser Annotation können wir einzelne Methoden schreiben, welche wir einzeln oder mehrere pro File testen können. Wir erstellten 5 verschiedene Files, um jede Controller-Klasse der zugehörigen Repositories zu testen. Grundsätzlich sind alle Unittests gleich aufgebaut, nur der Inhalt ändert sich von Methode zu Methode.

3.1.1 Registrierung Test

Nehmen wir an wir wollen die Registrierung eines neuen Benutzers testen, dazu müssen wir zuerst überprüfen, ob ein User bereits mit dem Namen «tester» existiert. Dies passiert mit when(userRepository.findById.. und gibt uns ein leeres Optional zurück, was bedeutet, dass kein Benutzer mit diesem Namen existiert. Danach simulieren wir mit when(userRepository.save.. die Speicherung in der Datenbank, sodass wir das übergebene Benutzerobjekt zurückerhalten. Mit mvc.perform können wir schlussendlich die POST-Anfrage an unseren Endpunkt ("/users/register") durchführen. In der .content Methode definieren wir unseren Body, welchen wir an den Endpunkt schicken. Der Server braucht für diese Anfrage nur den Usernamen und das Passwort. Schlussendlich definieren wir noch mit .andExpect die Antwort, welche wir vom Server erwarten. Mit .isOK() erwarten wir den Statuscode 200, dass alles funktioniert hat. Mit den zwei jsonPath() erwarten wir zum einen, dass im Feld «userName» der Wert «tester» steht und im Feld «password» nichts steht. Falls wir genau das erhalten, was wir definiert haben, wir der Test als korrekt durchlaufen.

```
@Test
public void registerUserTest() throws Exception{
    when(userRepository.findById("tester")).thenReturn( t: Optional.empty());
    when(userRepository.save(any(User.class))).thenAnswer(invocation -> invocation.getArgument( i: 0));

    mvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/users/register")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content("{\"userName\":\"tester\",\"password\":\"tester\"}"))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.userName").value( expectedValue: "tester"))
            .andExpect(jsonPath( expression: "$.password").value( expectedValue: ""));

}
```

3.1.2 Truck hinzufügen Test

Sobald ein User eingeloggt ist, braucht der Server vom Client immer den Token, um sicherzustellen, dass es sich um eine valide Anfrage handelt. Nehmen wir an wir wollen einen Truck hinzufügen, dazu müssen wir zuerst einen User erstellen, damit wir einen Token erstellen können. Dieser Token wird danach verwendet im Body der POST-Anfrage, damit der Server diese Anfrage als eine korrekte weiterverarbeitet.

Zuerst müssend wir das userRepository nachbilden, um den User in die Datenbank zu speichern. Bei der Validierung eines Tokens wird im UserRespository überprüft, ob ein User mit diesem Token existiert. Falls dies nicht der Fall ist, können wir die Anfrage nicht vollenden. Deshalb simulieren wir mit when(userRepository.save.. die Speicherung eines Users in diese Datenbank. Die zwei weiteren Methoden .findById() und .findByToken() simulieren noch, dass wirklich ein User in der Datenbank in der Datenbank gefunden wurde ohne, dass mit der Datenbank interagiert wurde. Mit dem String tourJson legen wir den Body für die Server-Anfrage fest. Hier ist wichtig, dass wir den bereits erstellten «LoginToken» mitgeben. Mit mvc.perform simulieren wir die POST-Anfrage an den Endpunkt ("/trucks/new"). In der .content Methode geben wir unsere JSON Nachricht mit. Schlussendlich definieren wir noch mit .andExpect die Antwort, welche wir vom Server erwarten. Mit .isOK() erwarten wir den Statuscode 200, dass alles funktioniert hat. Mit den zwei jsonPath() erwarten wir zum einen, dass im Feld «BrandName» der Wert «BMW» steht und im Feld «truckCapacity» der Wert «15» steht. Falls wir genau das erhalten, was wir definiert haben, wir der Test als korrekt durchlaufen.

```java
@Test
public void newTruckTest() throws Exception{
    User newUser = new User();
    newUser.setUserName("tester");
    newUser.setPassword("tester");
    String LoginToken = Token.generate();
    newUser.setToken(LoginToken);
    when(userRepository.save(any(User.class))).thenReturn(newUser);
    when(userRepository.findById("tester")).thenReturn(Optional.of(newUser));
    when(userRepository.findByToken(LoginToken)).thenReturn(List.of(newUser));

    String tourJson = "{ " +
            "\"token\":\"" + LoginToken + "\", " +
            "\"truckCapacity\":\"15\", " +
            "\"brandName\":\"BMW\"" +
            "}";

    mvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/trucks/new")
                .contentType(MediaType.APPLICATION_JSON)
                .content(tourJson))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.brandName").value( expectedValue: "BMW"))
            .andExpect(jsonPath( expression: "$.truckCapacity").value( expectedValue: "15"));

}
```

3.2 E2E Test

Für die End-to-End Tests haben wir Playwright benutzt. Um die Tests auszuführen, muss zuerst der Server gestartet werden und danach können die Tests ausgeführt werden. Getestet damit haben wir die Login Funktion und darauf aufbauend das hinzufügen von Trucks und Customer. Dazu fügen wir 10 Trucks/ Customer dem Server hinzu und bleiben am Ende des Tests auf der Seite, wo die Trucks/ Customer angezeigt werden, um noch selbst überprüfen zu können, dass die Trucks/ Customer auch richtig hinzugefügt wurden.

Beispiel hinzufügen von Customers:

```
Playwright playwright = Playwright.create();
Browser browser = playwright.chromium().launch(new BrowserType.LaunchOptions().setHeadless(false));
BrowserContext context = browser.newContext();
Page page = context.newPage();

// Step 1: Login
page.navigate( url: "http://localhost:8080/Valo_Login/login.html");
page.fill( selector: "#username",  value: "test");
page.fill( selector: "#password",  value: "test");
page.waitForNavigation(() -> {
    page.click( selector: "#button-login");
});

// Check for a successful login
boolean loginSuccessful = page.isVisible( selector: "#loginRegisterButton") &&
        "Logout".equals(page.textContent( selector: "#loginRegisterButton"));
if (loginSuccessful) {
    System.out.println("Login was successful!");
} else {
    System.out.println("Login failed.");
    return;
}
```

Der 1. Schritt ist das Login, denn ohne eingeloggt zu sein, können keine Customer hinzugefügt werden. Dazu wird zuerst ein Playwright Objekt erstellt und startet danach einen Chromium Browser im sichtbaren Modus, damit wir sehen, was das Programm macht. Danach öffnet er einen neuen Browserkontext und darin eine neue Seite. Danach navigiert er auf die Login-Seite und gibt die Eingaben ein und drückt dann den Login-Knopf. Die Seite leitet dann selbständig zurück auf die Homepage und da wird dann geprüft, ob der Knopf oben rechts auf «Logout» steht. Wenn das stimmt, wird das in der Konsole ausgegeben und es geht mit Schritt zwei und drei weiter.

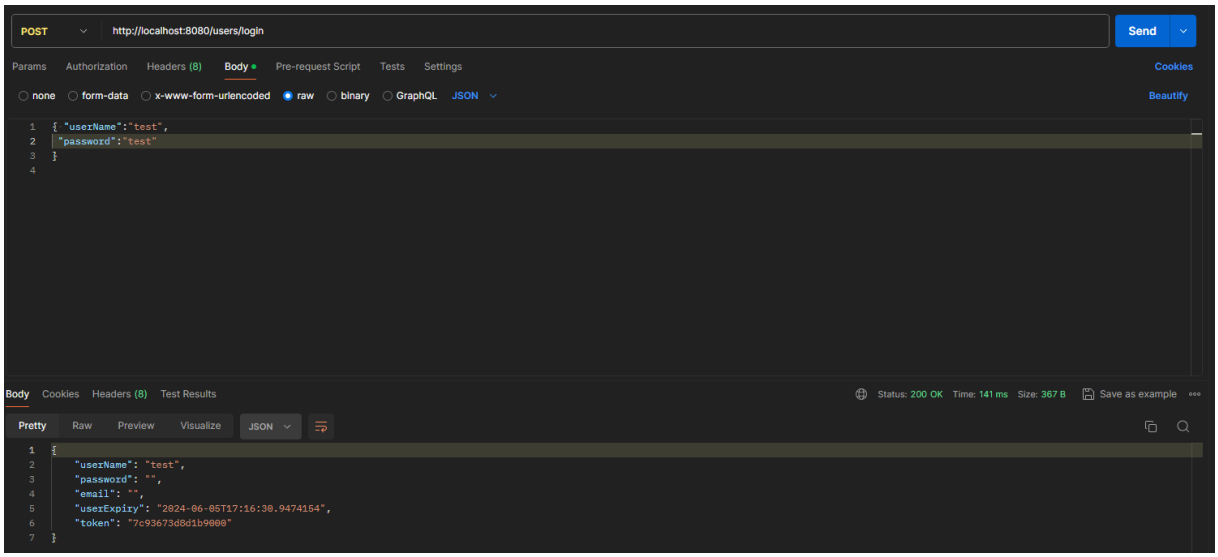
```
// Step 2: Navigate to the customers page
page.navigate( url: "http://localhost:8080/Valo_Customer/customers.html");
page.waitForSelector("#customersContainer");

// Step 3: Add 10 customers
for (int i = 1; i <= 10; i++) {
    // Navigate to the add customer page
    page.navigate( url: "http://localhost:8080/Valo_Customer/addCustomer.html");

    // Fill in the customer details
    String customerName = "Customer " + i;
    String customerAddress = "Address " + i;
    String customerCity = "City " + i;

    page.fill( selector: "#name", customerName);
    page.fill( selector: "#address", customerAddress);
    page.fill( selector: "#city", customerCity);

    page.waitForNavigation(() -> {
        page.click( selector: "#button-newCustomer");
    });
}

System.out.println("Successfully added 10 customers.");
```

Schritt zwei navigiert auf die Customer-Seite, wo alle customer angezeigt werden und wenn das customersContainer-Objekt entdeckt wird, geht es mit Schritt drei weiter.

Schritt drei iteriert zehnmal über die for-Schleife. In der Schleife wechselt das Programm auf die addCustomers-Seite und füllt die drei Felder mit Daten und drückt danach den button-newCustomer-Button und fügt somit den customer zur Datenbank hinzu. Wen diese Schleife zehnmal durchlaufen ist, wird das in der Konsole ausgegeben und das Programm stoppt auf der Customer-Seite, dass die erstellten Daten noch überprüft werden können.

3.3 Manuelle Tests

Während der Umsetzung des Projektes haben wir die erstellten Methoden immer gut ausführlich getestet, um Bugs frühzeitig zu erkennen. Severin hat dazu die Serverseitigen Tests mit Postman getestet, damit die Eingaben vom Server korrekt angenommen werden und auch die Ausgaben vom Server stimmen, die er zurückschickt nach einer Anfrage. Unten ist das am Beispiel vom Login gezeigt.



Im Front-End hat Kevin die neuen Funktionen immer direkt getestet und überprüft, dass die Eingaben richtig am Server ankommen und richtig in die Datenbank gespeichert werden. Er hat auch Fehleingaben getestet, dass diese nicht beim Server ankommen und da Probleme verursachen können. Dazu hat er viel mit console.log() gearbeitet, um die einzelnen Schritte in einer Funktion in der Konsole zu sehen und Fehler zu finden. Weiter hat auch Severin regelmässig das Front-End ausprobiert, um weitere Bugs zu entdecken.