

Projektbericht

Inhaltsverzeichnis

1 . Einleitung.....	3
1.1 Motivation und Themenbeschreibung	3
1.2 Vorstellung der Gruppenmitglieder und Ihre Rollen und Aufgaben im Team	4
1.3 Theoretische Grundlagen	4
2. Hauptteil.....	6
2.1 Konzeptionelle Arbeiten im Backend	6
2.1.1 Visualisierung und Softwareeinstieg	6
2.1.2 Use-Cases	7
2.1.3 User Stories	7
2.1.4 Software-Architektur.....	9
2.1.5 Klassendiagramm	10
2.1.6 Sequenzdiagramm.....	11
2.1.7 ERM – Diagramm.....	12
2.2 Konzeptionelle Arbeiten im Frontend	13
2.2.1 UseCase Diagramm	13
2.2.2 Komponentendiagramm	14
2.2.3 Aktivitätsdiagramme	15
2.3 Code-Ausschnitte zu aussagekräftigen Details im Frontend	17
2.3.1 Konfig	17
2.3.2 “dashboard_control.js” - Zentrale Logik	17
2.3.3 “dashboard_global.js” - Globale Variablen	19
2.3.4 “dashboard_msg.js” - Popup-Fenster und co.	19
2.3.5 “dashboard_smartdata.js” - Backend-Calls.....	20
2.3.6 Visualise-Komponente	20
2.3 Code-Ausschnitte zu aussagekräftigen Details im Backend (Janik)	21
2.4.1 Datenbankschema.....	21
2.4.2 REST-Schnittstellen	24
3. Ausblick & Fazit	29
3.1 Fazit im Team	29
4. Installationshinweise.....	30
4.1 Dashboard Frontend	30
4.2 Dashboard Backend	31

5. Benutzerhandbuch	33
5.1 Login.....	33
5.2 Dashboard Karte erstellen (Admin)	33
5.3 Karten minimieren	36
5.4 Karten Konfiguration (Admin).....	36
5.5 Karten Konfiguration (Benutzer).....	38
5.6 Globaler Zeitraum (Admin)	38
5.7 Globaler Zeitraum (Benutzer)	39
5.8 Ansichten Wechsel Admin/Benutzer	40
5.9 Karten allgemein	40
5.10 Sprache	41
6. Quellenverzeichnis	Fehler! Textmarke nicht definiert.
7. Anlagen - Teamarbeitsplan, SW-Design mit kompl. Use Case Diagrammen, Sequenzdiagrammen und kompl. Klassendiagrammen)	Fehler! Textmarke nicht definiert.

1. Einleitung

1.1 Motivation und Themenbeschreibung

Die Nutzung der Solarenergie hat in den letzten Jahren stark zugenommen und ist mittlerweile eine wichtige Alternative zu fossilen Brennstoffen. Um die Effizienz und Wirtschaftlichkeit von Solaranlagen zu optimieren, ist es von großer Bedeutung, die erzeugten Daten zu analysieren und zu visualisieren. Durch die Verwendung von Dashboards können diese Daten einfach und übersichtlich dargestellt werden.

Die Datenvisualisierung mit Hilfe von Dashboards ermöglicht es, große Mengen an Daten schnell und einfach zu erfassen und zu verstehen. Durch die Verwendung von Diagrammen, Grafiken und Karten können die Daten in einer visuell ansprechenden Form dargestellt werden, was das Verständnis der Daten erleichtert. Insbesondere im Bereich der Solarenergie ist die Datenvisualisierung von großer Bedeutung, da hier oft sehr viele Daten erzeugt werden, die manuell kaum zu verarbeiten sind.

Ein Beispiel für die Verwendung von Datenvisualisierung im Bereich der Solarenergie ist die Darstellung der Leistung von Solaranlagen. Durch die Verwendung von Diagrammen kann man schnell erkennen, wann die Anlage am meisten Strom erzeugt und wann die Leistung am geringsten ist.

Dashboards sind ein wichtiges Instrument zur Datenanalyse. Sie ermöglichen es, die Daten in Echtzeit darzustellen und so schnell auf Veränderungen reagieren zu können. Auf diese Weise kann man die Leistung der Anlagen kontinuierlich überwachen und gegebenenfalls Reparaturen durchführen, um die Effizienz der Anlagen zu optimieren.

Dashboards bieten auch die Möglichkeit, verschiedene Datenquellen zusammenzuführen und in einer einzigen Ansicht darzustellen. Dies erleichtert die Übersicht und ermöglicht es, Zusammenhänge schneller zu erkennen. So kann man beispielsweise die Leistung der Solaranlage mit dem Wetter in der Region vergleichen und so erkennen, ob die Anlage bei bestimmten Witterungsbedingungen besonders gut oder schlecht arbeitet.

Darüber hinaus ermöglicht die Verwendung von Dashboards auch die Möglichkeit, die Daten mit anderen zu teilen und über diese zu diskutieren. Auch die Zusammenarbeit mit anderen Unternehmen und Organisationen wird dadurch erleichtert, da die Daten einfach und übersichtlich dargestellt werden können.

Aus diesen Gründen war es unsere Aufgabe für die Solaranlage, die sich am Campus Minden befindet, ein Dashboard zu entwickeln. Das Dashboard soll es Administratoren ermöglichen, auszuwählen, welche Daten auf dem Dashboard angezeigt werden sollen und wie diese für Nutzer visualisiert werden. Dazu kann der Administrator Karten zum Dashboard hinzufügen. In jeder Karte kann eine Visualisierung-Art für die gewählten Daten ausgewählt werden.

Mögliche Anzeigen sind beispielsweise Wetterdaten, die mit Symbolen für Regen, Sonne etc. dargestellt werden, die Temperatur, die als Thermometer angezeigt wird, die Einstrahlungsstärke, die in einer Art Kreisdiagramm dargestellt wird, sowie den momentanen Stromertrag.

Der Administrator wählt anschließend noch die Datenquelle aus. Er kann auch festlegen, wie ein Nutzer mit der Karte interagieren darf. Diese Karte wird mit seiner Konfiguration dann für alle Nutzer in einer Datenbank gespeichert, was es ermöglicht Nutzern überall dieselbe Erfahrung zu bieten, da ihre Einstellungen geräteübergreifend genutzt werden.

Der Nutzer hat die Möglichkeit, einzelne Anzeigen auszublenden, andere Anzeigen einzublenden und Parameter in den Karten selbst einzustellen. Der Administrator hat dafür die Möglichkeit zum Beispiel den Zeitraum, in dem sich der Nutzer bewegen kann zu beschränken, um mögliche Datenbereiche, die fehlerhafte Daten beinhalten, für Nutzer nicht zugänglich zu machen.

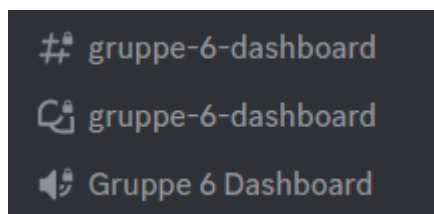
Ein weiteres Element, was der Benutzerfreundlichkeit dienen soll, ist ein leeres Platzhalter-Element. Dieses hat zwar keinen Inhalt und visualisiert keinerlei Daten, kann aber verwendet werden, um eine solaranlagenähnliche Darstellung der Karten zu erreichen.

1.2 Vorstellung der Gruppenmitglieder und Ihre Rollen und Aufgaben im Team

Das Projektteam besteht aus sechs Mitgliedern, die jeweils unterschiedliche Rollen und Aufgaben innehaben. Luca-Miguel Humke übernimmt die Rolle des Projektleiters und ist verantwortlich für die Organisation und Koordination des Projekts. als Projektleiter, hat er die Aufgabe, das Projekt im Zeitplan zu halten. Er ist auch für die Zusammenarbeit mit den Stakeholdern verantwortlich und sorgt dafür, dass alle Anforderungen erfüllt werden.

Das Frontend-Team besteht aus Finn, Justin und Alexey. Sie sind für die Gestaltung und Umsetzung des Dashboards verantwortlich. Dies beinhaltet die Erstellung von Benutzeroberflächen und die Umsetzung von Interaktionen und Animationen. Sie arbeiten eng mit dem Backend-Team zusammen, um sicherzustellen, dass die Anwendung reibungslos funktioniert. Sie sind auch für die Anpassung an verschiedene Geräte und Bildschirmgrößen verantwortlich.

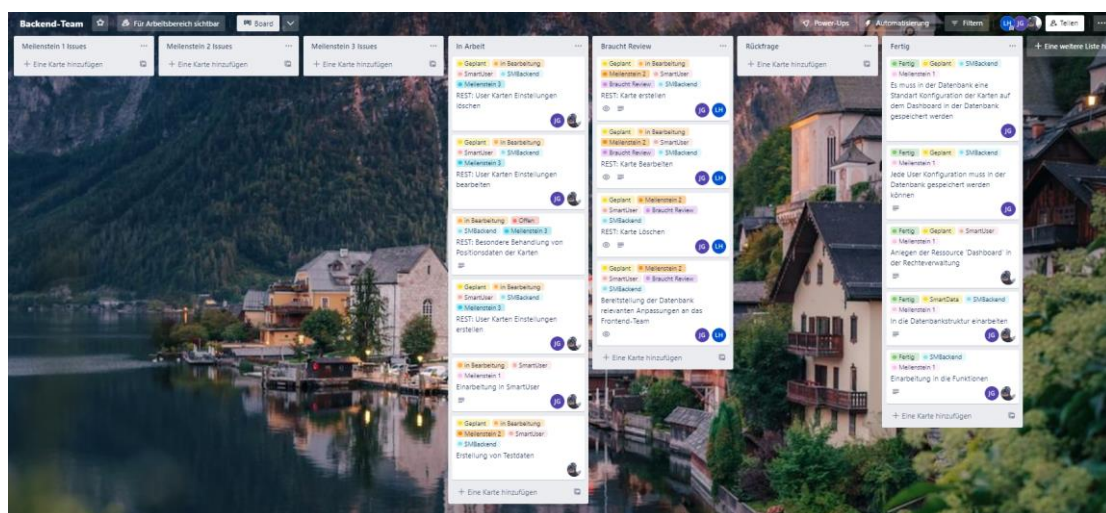
Janik Geist und Paul Merlin Nagel arbeiten eng zusammen, um sicherzustellen, dass das Backend reibungslos funktioniert. Sie arbeiten auch an der Datenbank und stellen damit sicher, dass die Daten für das Backend wie gewünscht persistiert werden. Sie erstellen auch REST-Endpoints, die es ermöglichen, dass das Dashboard Daten für die darzustellenden Karten aus der Datenbank abrufen und speichert.



Insgesamt arbeiten alle Mitglieder des Teams eng zusammen, um sicherzustellen, dass das Projekt erfolgreich abgeschlossen wird und die Anforderungen der Stakeholder erfüllt werden. Um diese Kommunikation möglichst simpel zu halten, haben wir uns für Discord als Hauptkommunikationsweg festgelegt. Durch Discord konnten wir Asynchron über den Text-Chat kommunizieren und waren somit nicht auf die Anwesenheit

jedes Teammitglieds angewiesen. Für unsere wöchentlichen Meetings haben wir den Voice-Chat genutzt. Durch die umfangreichen Funktionen die Discord bietet, konnten wir beispielsweise unsere Bildschirme übertragen, was die Zusammenarbeit stark vereinfacht hat.

Als weiteres Hilfsmittel haben wir Trello genutzt. Mit Hilfe von Trello konnten wir Aufgaben verteilen und den Fortschritt überwachen und beobachten. Außerdem konnten wir Aufgaben priorisieren, damit diese zuerst abgearbeitet werden konnten.



1.3 Theoretische Grundlagen

Um die Software zu planen und um eine Dokumentation zu erstellen haben wir uns auf das Wissen aus den vorangegangenen Semestern bezogen. Die Softwarespezifikation sollte die Anforderungen und Funktionen

einer Anwendung beschreiben, noch bevor die Entwicklung startet. Sie dient dazu, sicherzustellen, dass die Anwendung die erwarteten Anforderungen erfüllt und dass alle Beteiligten ein gemeinsames Verständnis darüber haben, was die Anwendung leisten soll.

Ein wichtiger Aspekt der Software-Spezifikationen ist die Verwendung von Diagrammen und Darstellungen, um die Anforderungen visuell darzustellen. Einige der von uns verwendeten Diagramme und Darstellungen sind:

Use-Case-Diagramme: Sie zeigen die Interaktionen zwischen Benutzern und der Anwendung. Sie helfen dabei, die Anforderungen an die Anwendung zu verstehen und zu dokumentieren.

Aktivitätsdiagramme: Sie zeigen die Abläufe und Prozesse innerhalb der Anwendung. Sie helfen dabei, die Abhängigkeiten zwischen verschiedenen Teilen der Anwendung zu verstehen.

Klassendiagramme: Sie zeigen die Beziehungen zwischen verschiedenen Klassen und Objekten innerhalb der Anwendung. Sie helfen dabei, die Struktur der Anwendung zu verstehen.

Datenflussdiagramme: Sie zeigen, wie Daten durch die Anwendung fließen. Sie helfen dabei, die Abhängigkeiten zwischen verschiedenen Teilen der Anwendung zu verstehen.

Ablaufdiagramme: Sie zeigen die Abläufe und Prozesse innerhalb der Anwendung in einer zeitlichen Abfolge. Sie helfen dabei, die Abläufe innerhalb der Anwendung zu verstehen und zu dokumentieren.

Es ist wichtig zu beachten, dass die Verwendung von Diagrammen und Darstellungen nicht nur für Entwickler wichtig ist, sondern auch für alle Stakeholder. Sie ermöglichen es, die Anforderungen und Funktionen der Anwendung einfacher zu verstehen und zu kommunizieren.

Da wir unsere Planung in mehrere Meilensteine unterteilt hatten und wir uns auch wöchentlich mit unseren Stakeholdern getroffen haben, um unseren Fortschritt und die Anforderungen zu besprechen, kam für uns nur eine agile Arbeitsweise mit Scrum-Elementen in Frage. Mit dieser Arbeitsweise konnten wir auch schon in der Vergangenheit Erfahrungen sammeln und auf diesem Wissen aufbauen.

Durch diese Arbeitsweise konnten wir sicherstellen, dass auf die Sicht der Stakeholder Rücksicht genommen wird und das Ergebnis den Anforderungen entspricht. Man muss auch bereit sein, Veränderungen anzunehmen und flexibel zu sein, falls sich die Anforderungen im Laufe des Projekts ändern.

Scrum ist ein Rahmenwerk, das auf den Prinzipien der agilen Softwareentwicklung basiert und es ermöglicht, Projekte in kleinen Schritten abzuschließen.

Das Team arbeitet in sogenannten "Sprints", die in unserem Fall zwei Wochen dauerten. In jedem Sprint werden die Aufgaben geplant, die abgeschlossen werden mussten. Am Ende des Sprints stellen sie die erreichten Ergebnisse den Stakeholdern vor. Dies ermöglicht eine schnellere Umsetzung und Anpassung an die Anforderungen.

Um unsere Planung umzusetzen konnten wir auf das SmartEnviroSystem zugreifen, was uns einiges an Arbeit abgenommen hat, da z.B. Datenzugriffe bereits über SmartData gelöst ist. Durch den Microservice Ansatz konnten wir SmartData ganz einfach in unsere Anwendung einbinden. Durch diesen Ansatz könnte unsere gesamte Entwicklung und auch das Zurückführen der Anwendung stark vereinfacht werden da, statt einer Anwendung, die alle Funktionalitäten enthält, die Funktionalitäten in kleinere Dienste aufgeteilt werden, die jeweils für eine bestimmte Aufgabe verantwortlich sind. Dies ermöglicht es, dass wir schneller und effizienter arbeiten konnten, da wir uns nur auf einen kleinen Teil der Anwendung konzentrieren mussten.

Ein weiterer Vorteil von Microservices ist die Skalierbarkeit. Da jeder Dienst unabhängig von anderen Diensten ist, kann jeder Dienst einzeln skaliert werden, um die Anforderungen der Anwendung zu erfüllen. Dies bedeutet, dass Sie nur die Ressourcen erhöhen müssen, die benötigt werden, anstatt die gesamte Anwendung zu skalieren.

Microservices erfordert auch eine andere Art der Kommunikation zwischen den Diensten. Statt direkt auf andere Teile der Anwendung zugreifen zu können, müssen die Dienste über eine Art von Schnittstelle miteinander kommunizieren. Dies kann zum Beispiel über eine REST-API erfolgen, die wir auch für unsere Anwendung im SmartMonitoringBackend ergänzen mussten.

Um die ganzen Services erreichbar zu machen, müssen diese auf einem Webserver bereitgestellt werden. In der bereitgestellten Entwicklungsumgebung wurde uns dafür ein Payara Webserver zur Verfügung gestellt.

Diesen kannten wir bereits durch das Modul „Web basierte Anwendungen“ aus dem vorangegangenen Semester.

Der Payara Webserver ist ein Open-Source-Java-EE-Anwendungsserver, was für uns bedeutet, dass es wir ihn kostenlos verwenden können und dieser von einer Gemeinschaft von Entwicklern stetig weiterentwickelt wird.

Der Payara Server basiert auf einer Plattform namens GlassFish und unterstützt die neuesten Java-EE-Spezifikationen. Das bedeutet, dass es mit den neuesten Technologien kompatibel ist und Anwendungen schnell und zuverlässig laufen.

Durch dasselbe Modul hatten wir in der Vergangenheit auch bereits Erfahrungen mit Hibernate gemacht. Hibernate ist eine Bibliothek für, die es uns ermöglicht, Daten aus einer relationalen Datenbank zu lesen und zu schreiben. Es ist ein Framework für die Verwaltung von Datenbank-Verbindungen und dient dazu, die Interaktion zwischen einer Anwendung und der Datenbank zu vereinfachen.

Um das zu erreichen, stellt Hibernate eine Schicht zwischen Ihrer Anwendung und der Datenbank bereit. Diese Schicht, auch als ORM (Object-Relational-Mapping) bekannt, ermöglicht es, Daten in einer objektorientierten Weise zu verwalten, anstatt sich mit SQL-Abfragen und anderen relationalen Datenbank-Konzepten auseinandersetzen zu müssen.

Ein wichtiger Aspekt von Hibernate ist die Verwendung von sogenannten Entitäten. Eine Entität ist eine Java-Klasse, die mit einer Tabelle in der Datenbank verknüpft ist. Jede Instanz dieser Klasse repräsentiert einen Datensatz in der Tabelle. Hibernate stellt Methoden bereit, um diese Entitäten zu erstellen, zu aktualisieren, zu löschen und abzufragen.

Hibernate bietet auch erweiterte Funktionen wie die Möglichkeit, Abfragen zu erstellen und zu optimieren, Caching zu verwenden, um die Leistung zu verbessern. Jedoch mussten wir gar nicht so tief in die Funktionen von Hibernate eintauchen, da das Grundgerüst im SmartMonitoringBackend uns bereits zur Verfügung gestellt wurde.

2. Hauptteil

2.1 Konzeptionelle Arbeiten im Backend

2.1.1 Visualisierung und Softwareeinstieg

Vor dem Erstellen der benötigten Diagramme war es wichtig sich auf eine Visualisierungssoftware für die nötigen Diagramme zu einigen. Wir nutzten hier die Software DrawIO um Diagramme übersichtlich darzustellen. DrawIO kommt mit der „Apache License 2.0“, welches uns das Arbeiten ohne extra Kosten ermöglicht. Die Entscheidung zu der richtigen Softwarelösung ist wichtig, da von dieser Entscheidung viel Zeit und somit auch mögliche Kosten abhängen.

Use-Case Diagramme können hier übersichtlich dargestellt werden und mit den entsprechenden ovalen Kreisen für die Aufgaben die ein Nutzer durchführen möchte, Striche die gerichtet aus einer Nutzerperspektive die Abhängigkeiten bzw. Beziehungen darstellen und die Nutzer selber, welche durch einfache Strich-Visualisierungen dargestellt werden.

Dazukommt die Userstories, welche eine allgemeine Erklärung der Software-Features aus Sicht eines Endnutzers darstellen sollen. Dabei versuchten wir bestmöglich aus der Sicht des Endnutzers, solche Punkte zu formulieren. Die Sicht ist hierbei entscheidend. Der Endnutzer steht bei den Userstories im Mittelpunkt. Verzichtet wird hierbei beispielsweise auf Code-Schnipsel, konkrete Lösungsansätze sowie hohe Fachsprache. Nur wieso das alles? User Stories zählen zu den wichtigsten Bestandteilen der agilen Softwareentwicklung. Durch sie wird festgestellt, was der Nutzer wirklich mit der Software machen will. Sich an dieser zu orientieren, ist ein essentieller Punkt, um ein passendes Backend für die benötigten Lösungen im Frontend zu schaffen. Dadurch wird sichergestellt, dass eine Software benutzerfreundlich erstellt wird.

Zur besseren Visualisierung der Struktur eines Systems haben wir es von Relevanz ein Klassendiagramm zu erstellen. Dieses zeigt gerade für das Backend, aber natürlich auch für das Frontend, übersichtlich visualisiert die strukturellen Abhängigkeiten innerhalb eines Systems. Es zeigt Klassen, Attribute, Vorgänge und Beziehungen durch Linien zwischen einzelnen Objekten.

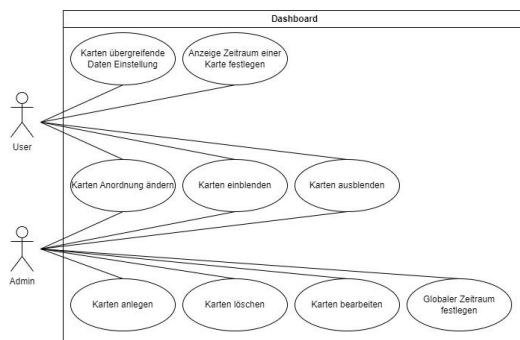
Um Abläufe nach chronologischer Reihenfolge darzustellen, haben wir ein Sequenzdiagramm in UML erstellt. Dieses dient zum besseren Verständnis der eingesetzten Software. Ebenso bietet es einen guten Überblick, um die Software zu erweitern und an vorhandene Abläufe sinnvoll anzuschließen. Hierbei wurde die UML Definierung als Grundlage zum veranschaulichen genutzt.

2.1.2 Use-Cases

Um eine gute Übersichtlichkeit über die Aufgaben, die wir in unserer Software darstellen wollen, zu gewährleisten, war es wichtig die Use-Cases zu entwickeln. Dies hilft speziell im Backend, da wir aus Sicht des Nutzers und des Admins, unsere zwei Entitätsgruppen (ebenso bekannt als Akteure), Aktivitäten darstellen wollen, die diese ausführen können. Dies ist geschrieben aus der Sicht der Akteure und passt inhaltlich gut zu den User Stories, obgleich es die Informationen über das „Wieso“ und der Priorisierung vernachlässigt. Ebenso ist es visuell gebündelter, übersichtlicher und kürzer gefasst für unseren Anwendungsfall: Das Dashboard.

Wir definieren hier die UserCardSettings als Kartenübergreifende Daten Einstellung und die Anzeige eines Zeitraumes einer Karte, sowie inklusive des Festlegens eines solchen Zeitraumes als UseCases die speziell für einen User definiert sind. Übergreifend wollen will ein User die Karten Anordnung ändern. Dies ist besonders relevant im Backend, da wir hier mit Positionen und die dazugehörige Implementation konfrontiert worden sind. Übergreifend für beide Akteure sind ebenso das ein und ausblenden einer Karte, welches in der Datenbank persistiert werden soll.

Ein Karten zur Verfügung stellen und damit anlegen, soll nur ein Admin können, was an unser Backend weitergeleitet werden muss und innerhalb der Karte und nicht der konkreten UserCardSetting gespeichert werden muss.



Eine Karte soll gelöscht werden können und Einstellungen einer Karte sollen bearbeitet werden können. Im Backend wird dies per PUT HTTP-Anfrage realisiert. Wird eine neue Karte mit den Kartendaten als JSON-Body übergeben, und sind die Daten valide, wird in der Datenbank entsprechend aktualisiert. Eine neue aktualisierte Variante der Karte steht dann zum Abruf in der Datenbank bereit. Ein globaler Zeitraum in der die Karte Daten abrufen kann, steht dem Admin auch zur Verfügung. Hierzu kommen im Abschnitt „User Stories“ weitere Anekdoten.

2.1.3 User Stories

Was muss ein User können und was ein Nutzer? Warum muss er dies können und wie gewichten wir diese Punkte?

Fragen die wir uns im Backend, sowie im ganzen Team, gestellt haben.

2.1.3A Admin

Beispielsweise kann der Admin eine Datenquelle auswählen. Hierbei stehen ihm alle aufgezeichneten Wetter und Solardaten aus der Datenbank zur Verfügung. Für diesen Fall haben wir eine Anforderung für den Admin geschrieben. Als Grund hierfür ist klar zu nennen das wir nicht nur Daten aus einer Datenquelle bekommen möchten, sondern aus mehreren. Diese Daten sollten zu bestimmt festgelegten Quellen eingeholt werden. Dafür musste eine Möglichkeit geplant werden um die Datenquelle im Backend sowie in der Datenbank für die jeweilige Karte auf dem Dashboard zu hinterlegen. Dies haben wir mit dem Datenbankeintrag resource in der UserCard realisiert. In diesem varchar wird eine Adresse des Modules hinterlegt. Somit weiß das Backend immer persistent, aus welcher Quelle die Daten geholt werden sollen. Ein Admin kann erstellte Karten speichern, die er angelegt hat. Dies musste ebenfalls als UserStorie definiert werden. Denn ein Admin muss die Karte speichern können. Durch diese Speicherung wird die Karte beim nächsten Abrufen von einem zur Verfügung gestellten User durch Anlegen auf seinem Dashboard oder durch einfaches Aufrufen seines Dashboards erneut aus dem Backend und damit auch aus der Datenbank geholt. Priorität des Abrufen, sowie speichern einer Usercard haben wir als hoch gewertet. Es handelt sich hierbei um integrale Bestandteile unseres Projektes, die nicht vernachlässigt werden dürfen und bei Fehlen eventuell essenzielle Funktionen behindern und das Dashboard möglicherweise nicht mehr bedienen lassen. Mögliche Folgen durch eine unzureichende Sicherstellung der Grundfunktionen können auch das Verschieben eines Meilensteines oder gar die Verschiebung und nicht Einhaltung des Projektzieles sein. Ein Admin soll doch weitere Funktionen zur Verfügung stehen haben. Ein Admin soll beispielsweise ein Limit für den globalen Zeitraum für die Datenerhebung festlegen können. Dieser soll den Nutzer daran hindern, zu große Datenzeiträume auszuwählen und dann durch eine lang andauernde Anfrage gegebenenfalls die Serverstruktur zeitweise zu blockieren. Implementiert wurde dies mit einem maxspan_beginn und maxspan_end als timestamp Werten in der Datenbank. Entsprechende Funktionen zum Abrufen sind durch die Get,Post sowie Update im Backend zur Verfügung gestellt. Auch dies ist ein wichtiges Feature, welches als Userstorie Vorrang hat. Auch kann der Admin Karten für den Nutzer sperren. Dies soll er machen können um den Nutzer von dem

Admin	kann ein Limit für den globalen Zeitraum für Datenerhebung festlegen	Um den Server vor zu großen Abfragen zu schützen	Hoch
Admin	kann Karten für den User sperren	Um diese dem User vor zu enthalten	Hoch
User	kann einzelne Elemente ausblenden	Um sein Dashboard zu personalisieren	Hoch
User	kann die Anordnung der Karten beeinflussen	Um sein Dashboard zu personalisieren	Hoch
User	kann Parameter selbst einstellen	Um sein Dashboard noch feiner zu personalisieren	Hoch

Abbildung 1: Ausschnitt aus UserStories

behindern und das Dashboard möglicherweise nicht mehr bedienen lassen. Mögliche Folgen durch eine unzureichende Sicherstellung der Grundfunktionen können auch das Verschieben eines Meilensteines oder gar die Verschiebung und nicht Einhaltung des Projektzieles sein. Ein Admin soll doch weitere Funktionen zur Verfügung stehen haben. Ein Admin soll beispielsweise ein Limit für den globalen Zeitraum für die Datenerhebung festlegen können. Dieser soll den Nutzer daran hindern, zu große Datenzeiträume auszuwählen und dann durch eine lang andauernde Anfrage gegebenenfalls die Serverstruktur zeitweise zu blockieren. Implementiert wurde dies mit einem maxspan_beginn und maxspan_end als timestamp Werten in der Datenbank. Entsprechende Funktionen zum Abrufen sind durch die Get,Post sowie Update im Backend zur Verfügung gestellt. Auch dies ist ein wichtiges Feature, welches als Userstorie Vorrang hat. Auch kann der Admin Karten für den Nutzer sperren. Dies soll er machen können um den Nutzer von dem

2.1.3B Nutzer

Doch kommen wir zu dem Nutzer. Welche Funktion bedarf es um die Grundfunktionen und allgemeinen Funktionen eines Nutzers zur Verfügung zu stellen? Mit Nutzer meinen wir in diesem Kontext den im System genannten „User“.

Ein User kann einzelne Elemente ausblenden. So sollte die Userstorie geschrieben und verfasst sein. Hat ein Nutzer eine Karte auf seinem Dashboard ausgeblendet, kann dieser sie nicht mehr unter den normalen Karten sehen. Hierbei taucht dann oben ein kleines Fenster auf, indem der Nutzer sehen kann, welche Karten er ausgeblendet hat. Sobald der Nutzer oben wieder auf die Schrift der Karte klickt, bekommt dieser Nutzer die Karte wieder angezeigt. Was dabei im Hintergrund geschieht, ist und soll dem Nutzer nicht gezeigt werden. Da die Abhängigkeit zwischen der tbl_card und tbl_user_card_settings gewährleistet ist, wird in der speziellen Einstellung der Karte ein Wert visible auf true gesetzt. Die Einstellung der Karte die speziell für einen Nutzer und nicht im Allgemeinen für alle Nutzer zur Verfügung gestellt wird, bedarf es eine Speicherung in tbl_user_card_settings. Eine Speicherung einer allgemeinen Karte wird in tbl_card vorgenommen. Dies ist dann auch die Tabelle, in der vereinzelt Werte vom Admin gesetzt werden. Ein Nutzer kann des Weiteren die Anordnung der Karten beeinflussen. Das bedeutet er kann die Reihenfolge dieser ändern. Das ist ein Mittel, welches den Nutzer eine erweiterte Personalisierung seines Dashboards erlaubt. Im Frontend wird dies über eine Drag and Drop Funktion dargestellt, welche zu der Softwareübergabe mit implementiert werden sollte.

User	kann einen globalen Zeitraum für die Datenerhebung festlegen	Um sich einen genauen Zeitpunkt der Daten anzeigen zu lassen	Hoch
User	kann keine fehlerhaften Daten eingeben.	Um nicht lange nach einer Lösung für fehlende/fehlerhafte Anzeigen zu suchen	Hoch

Abbildung 2: Weiterer Ausschnitt aus den Userstories

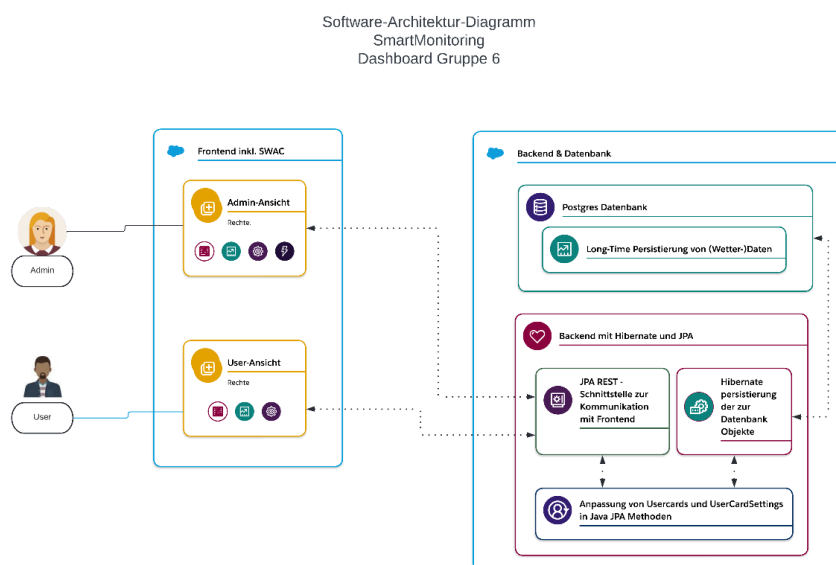
Dies bedeutet das ein Nutzer auf eine Karte klickt, gedrückt hält und diese verschiebt, bis diese an ihrer neuen Position liegt. Im Backend haben wir uns dazu entschlossen dies über eine Linked-List einzubauen. Dies bringt den großen Vorteil, keine großen Rechenaufwände sondern in der Tat eine Laufzeitkomplexität von $O(1)$ darzustellen. Im Vergleich zum üblichen Ändern der Reihenfolge einer Liste, müsste man auch alle Vorgänger oder Nachfolger der Liste und deren Position ändern. Dies wäre eine Laufzeitkomplexität von $O(n)$. Dazukommend handelt es sich hierbei nicht einfach um x Anzahl an Ausführungen von print Befehlen, sondern an x Anzahl Datenbankzugriffen, wodurch die Performance starke Einbuße tragen kann, gerade so bald mehrere Nutzer auf die Schnittstelle zugreifen würden. Fern von der Position einer Karte, soll der Nutzer seine Parameter in den Einstellungen einer Karte selbst einstellen können. Auch dies ist ein integraler Bestandteil und wurde von uns mit einer hohen Priorität angesehen. Er soll die Möglichkeit haben den Zeitraum für seine Wetter-Datenabfrage auswählen können. Dies ermöglicht ihm das Anzeigen unterschiedlicher Zeiträume und vor allem auch das Festlegen von fixen schmalen Zeiten, wie zum Beispiel einem Tag. Möchte der Nutzer beispielsweise den Zeitraum von einer Woche sehen, kann er dies über die gegebenen Datumpicker im Frontend auswählen. Dies wird dann beim Speichern einer UserCard und damit auch das Speichern der der Nutzer zugehörigen User-Card-Setting in den Attributen `span_begin` für den Anfang des Zeitraumes und `span_end` für das Ende des Zeitraumes der User-Card-Setting. Zu guter Letzt war es wichtig aufzunehmen, dass ein User keine fehlerhaften Daten eingeben kann und diese in der Datenbank speichern kann. Dies wird sowohl durch Backend als auch durch das Frontend dargestellt. Gibt der Nutzer beispielsweise einen ungültigen Zeitraum an, kann er nicht auf solche Datenzeiträume zugreifen und bekommt entsprechend keine Daten zurück.

2.1.4 Software-Architektur

So, Software-Architektur-Diagramme sind nützliche Werkzeuge, aber es ist wichtig, sie sorgfältig zu erstellen und zu pflegen, und sie sollten nicht als Ersatz für andere Formen der Kommunikation und Dokumentation verwendet werden. Es ist ein wichtiges Diagramm, um die Struktur und die Interaktionen von Komponenten in einer Anwendung zu verstehen. Das Software-Architektur-Diagramm dient als super Dokumentation für zukünftige Entwickler. Falls an den Komponenten allerdings weitergearbeitet wird, ist es zu beachten, das:

- es schnell veraltet sein kann und nicht die neuen Anforderungen widerspiegelt.
- es nicht alle Aspekte einer Anwendung abbilden kann.
- eine Aktualisierung eines Software-Architektur-Diagrammes Zeit und Ressourcen in Anspruch nehmen kann.

Nichts desto trotz dient es uns als ein sehr hilfreiches Tool für alle Projektbeteiligten. Hier unsere Umsetzung:



2.1.5 Klassendiagramm

Das Klassendiagramm nutzen wir um die Klassen, deren Attribute und Methoden, sowie die Beziehungen zwischen ihnen in einer Anwendung darzustellen. Dies taten wir vor allem aus den folgenden Gründen:

Übersicht: Das Klassendiagramm bietet uns eine visuelle Darstellung der Klassen, deren Attribute und Methoden sowie der Beziehungen zwischen ihnen. Dies erleichtert es uns, die Architektur der Anwendung schnell zu verstehen und zu navigieren.

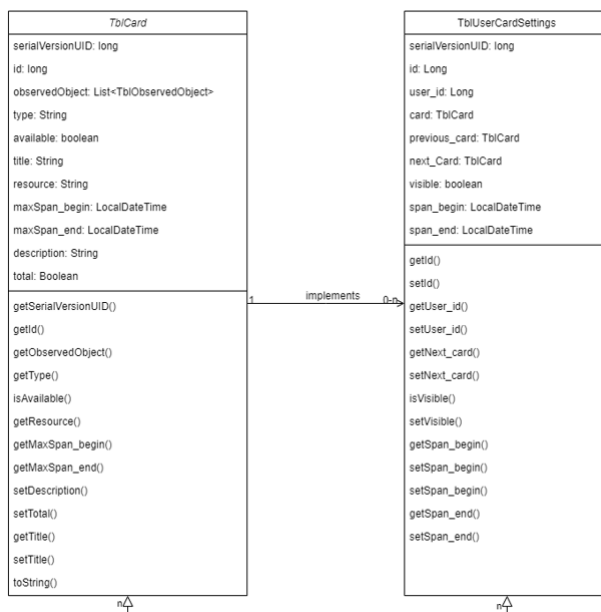
Kommunikation und Dokumentation: Es kann als gemeinsame Sprache zwischen Entwicklern, Architekten und anderen Beteiligten verwendet werden, um die Anforderungen und die geplante Architektur der Anwendung zu kommunizieren und Feedback zu sammeln.

Design-Überprüfung: Ebenso kann es verwendet werden, um das Design der Anwendung auf Probleme wie Überkomplexität, fehlende Kohäsion und Unvollständigkeit zu überprüfen, bevor die Implementierung beginnt.

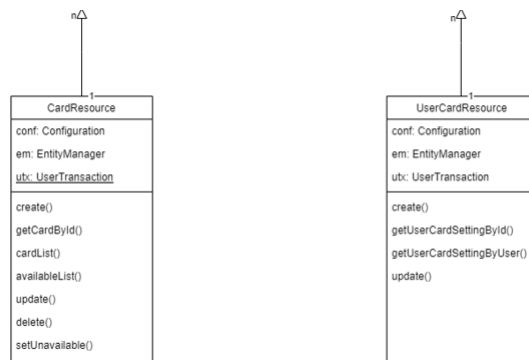
Refactoring: User Klassendiagramm hilft beim Refactoring von Code, da es die Beziehungen zwischen Klassen und ihre Abhängigkeiten zeigt und dadurch die Auswirkungen von Änderungen auf die Anwendung besser abschätzbar werden.

Im Bezug auf das Backend unserer Web-Anwendung, die Hibernate, Java Persistence API (JPA) und eine Postgres-Datenbank verwendet, enthält das Klassendiagramm die folgenden Elemente:

- **Entitätsklassen:** Dies sind Klassen, die die Datenstruktur der Datenbanktabellen darstellen und von JPA verwendet werden, um die Daten in die Datenbank zu persistieren und daraus abzurufen. Diese Klassen haben normalerweise Attribute, die den Spaltennamen der Datenbanktabellen entsprechen, und JPA-Annotationen, die die Beziehungen zu anderen Entitäten definieren.



- **DAO-Klassen (Data Access Object):** Dies sind Klassen, die die CRUD-Operationen (Erstellen, Lesen, Aktualisieren und Löschen) auf den Datenbanktabellen ausführen. Sie verwenden die Methoden von JPA, um die Datenbankabfragen auszuführen und die Ergebnisse zurückzugeben.

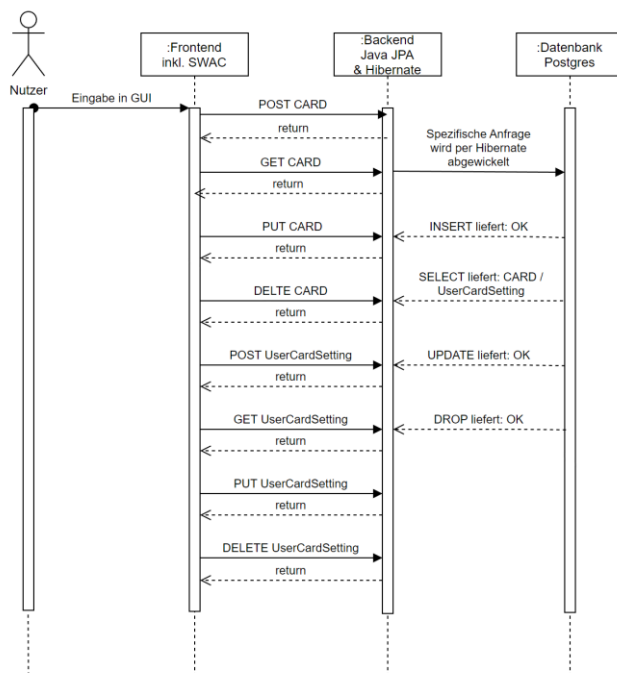


- **Beziehungen:** Das Klassendiagramm zeigt auch die Beziehungen zwischen den Entitäten. Beispielsweise kann eine Entitätsklasse "Kunde" eine Beziehung zu einer Entitätsklasse "Bestellung" haben, die besagt, dass ein Kunde mehrere Bestellungen haben kann.



2.1.6 Sequenzdiagramm

Um eine Interaktion von Objekten und unserer Komponenten im SmartMonitoring-Dashboard-Projekt über Zeit darzustellen haben wir ein Sequenzdiagramm angefertigt. Dieses hilft bei der Visualisierung der Abläufe und



Abhängigkeiten zwischen verschiedenen

Komponenten. Dabei stellen vertikale Pfeile für jedes Objekt eine Nachricht dar. Ebenso stellen horizontale Linien einen zeitlichen Ablauf dar und bewegen sich von einem Objekt zu einem neuen Objekt. Hierbei wird gewechselt. Sobald die Nachricht auf dem horizontalen Pfeil an einem Objekt ankommt, ist dieses Objekt aktiv und ggf. bearbeitet die Nachricht. Dies passiert so lange bis eine Nachricht ihren finalen Ort erreicht hat. In unserem Fall definieren wir das Frontend inklusive SWAC als ein Objekt, das Backend inklusive Java mit JPA und Hibernate als ein weiteres Objekt und die Datenbank, basierend auf Postgres. Es handelte sich in der Entwicklung um eine Testdatenbank mit Testdaten. Sobald wir das Projekt deployed haben, wurde die Produktivdatenbank der FH angesteuert. Beginnend auf der linken Seite des Sequenzdiagramm steht der Nutzer. Dieser stellt die Eingabe in unserem Sequenzdiagramm dar. Er tätigt die Eingabe in das System über mehrere Möglichkeiten aus dem Frontend. Es gibt Möglichkeiten zur textuellen Eingabe sowie zum Auswählen eines Zeitpunktes über einen Datepicker. Es können Daten in unterschiedlichen Typen übermittelt werden. Hat der Nutzer dies getan, wechselt man innerhalb des Sequenzdiagrammes zu dem nächsten Objekt. Dies ist das Frontend inklusive SWAC. Hier werden nun weitere Funktionen durchgeführt, welche in 2.2 Konzeptionelle

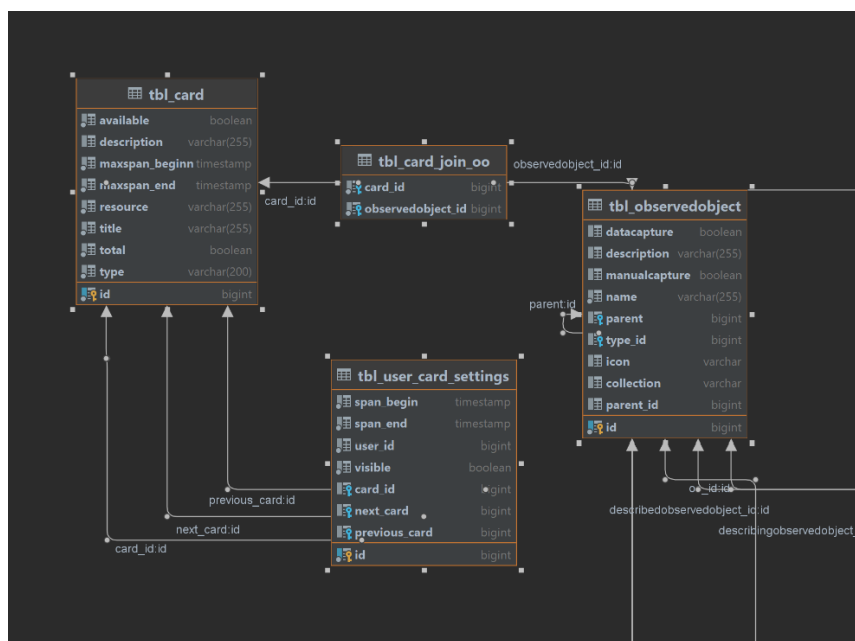
Arbeiten im Frontend und bei den Codeausschnitten im Frontend genauer erklärt werden. Hat das Frontend die Eingabedaten weiter auf eine REST und vor allem Schnittstellenkonforme Struktur gebracht, werden die Daten über die HTTP-Anforderungen wie POST, GET, PUT und DELETE an das Backend weitergeleitet. Dabei steht POST für das Erstellen einer Karte, oder einer UserCardSetting, GET für das Abrufen, PUT für das Aktualisieren und DELETE für das Löschen von Daten. Es gibt auch einige anderen Methoden wie PATCH, HEAD und OPTIONS, auf die wir allerdings verzichtet haben, da sie in unserem Anwendungsfall keinen erheblichen Vorteil gebracht hätten und diese auch nicht sehr oft genutzt werden. Wartbarkeit ist im Backend sowie Frontend wichtig. Dabei ist immer zu berücksichtigen das, falls Erweiterungen benötigt werden, eine sinnvolle Codebasis erschaffen wird, die gut und einfach erweiterbar ist. Sollte die Software also für das nächste Semester zur Verfügung stehen und es wird gewünscht die Software, um einige Funktionen zu erweitern, war es stets unser höchstes Streben ein gut strukturierten, wartbaren Code zu erschaffen.

Ist die HTTP-Anforderung an das Backend gesendet, wechseln wir innerhalb des Sequenzdiagrammes auf das Objekt „Backend Java JPA & Hibernate“ hier werden dann alle REST Request-Daten in passende Java Klassen und Objekte transformiert. Mittels Java JPA werden die Daten dann abhängig von der Eingabe und des Datentyps entsprechend weiterverarbeitet. Das bedeutet, wenn ein Nutzer versuchen sollte anstatt eines true- oder falschen Boolean Wertes einen leeren String zu übergeben, bekommt das Frontend entsprechende Fehlermeldung. In diesem Objekt wird auch das Behandeln von einer verketteten Liste, im konkreten Fall, das Verarbeiten der Positionsdaten einer Karte auf dem Dashboard dargestellt. Das wird benötigt, sobald der Nutzer eine Karte verschiebt, eine Karte löscht, ausblendet oder einfügt. Im Detail gehen wir hierzu nochmal in „Codeausschnitte zu Aussagekräftige Details im Backend“ ein.

Sobald die Verarbeitung im Backend erfolgreich ist, wird über Hibernate die Datenbank angesprochen. Hier werden dann die entsprechenden Befehle ausgeführt, um die Daten auf der Datenbank zu schreiben bzw. zu aktualisieren. In unserem Sequenzdiagramm kann man dann sehr gut erkennen, dass das Objekt Datenbank Postgres aktiv wird. Hier werden die Daten persistiert und stehen für weitere Aktualisierungen sowie Abrufe zur Verfügung. Die Kommunikation innerhalb des Sequenzdiagrammes kann immer von einem Objekt zum nächsten stattfinden. In diesem Fall wechselt der Status des Backends wieder auf aktiv, sobald es von der Datenbank eine Rückmeldung gegeben hat.

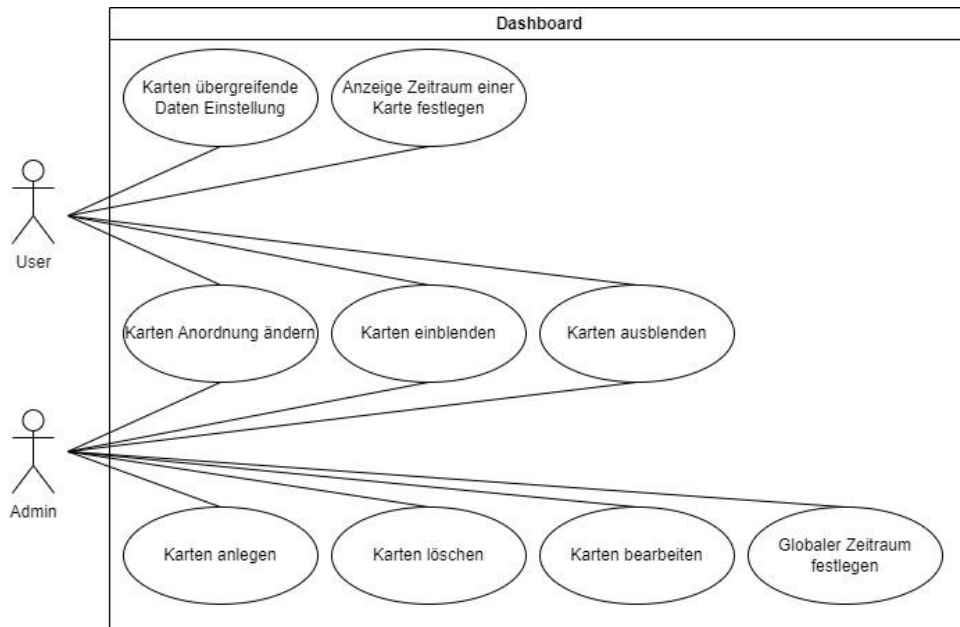
2.1.7 ERM – Diagramm

Zu guter Letzt wollen wir im Backend auf das ERM-Diagramm eingehen. Dieses war relevant, weil es uns unter anderem Modellierungsfehler schnell aufzeigt. Es hilft diese Fehler zu erkennen, wie z.B. Redundanz, Inkonsistenz, und Unvollständigkeit. Die Normalisierungsstufen kann man überprüfen und sicherzustellen, dass die Datenmodellierung den Anforderungen entspricht. Dazu Hand in Hand kann man die Normalisierung der Datenbanktabellen überprüfen. Ebenso gibt es uns eine gute Übersichtlichkeit. Es bietet eine visuelle Darstellung der Entitäten (Tabellen), deren Attribute und Beziehungen zwischen ihnen. Dies erleichtert es Entwicklern, die Datenmodellierung der Anwendung schnell zu verstehen und zu navigieren. In manchen Tools wird es ebenso genutzt, um aus diesem ERM-Diagramm automatisch eine Datenbank zu generieren. Dies wird aktuell zwar nicht benötigt, es könnte allerdings in Zukunft dafür genutzt werden.



2.2 Konzeptionelle Arbeiten im Frontend

2.2.1 UseCase Diagramm

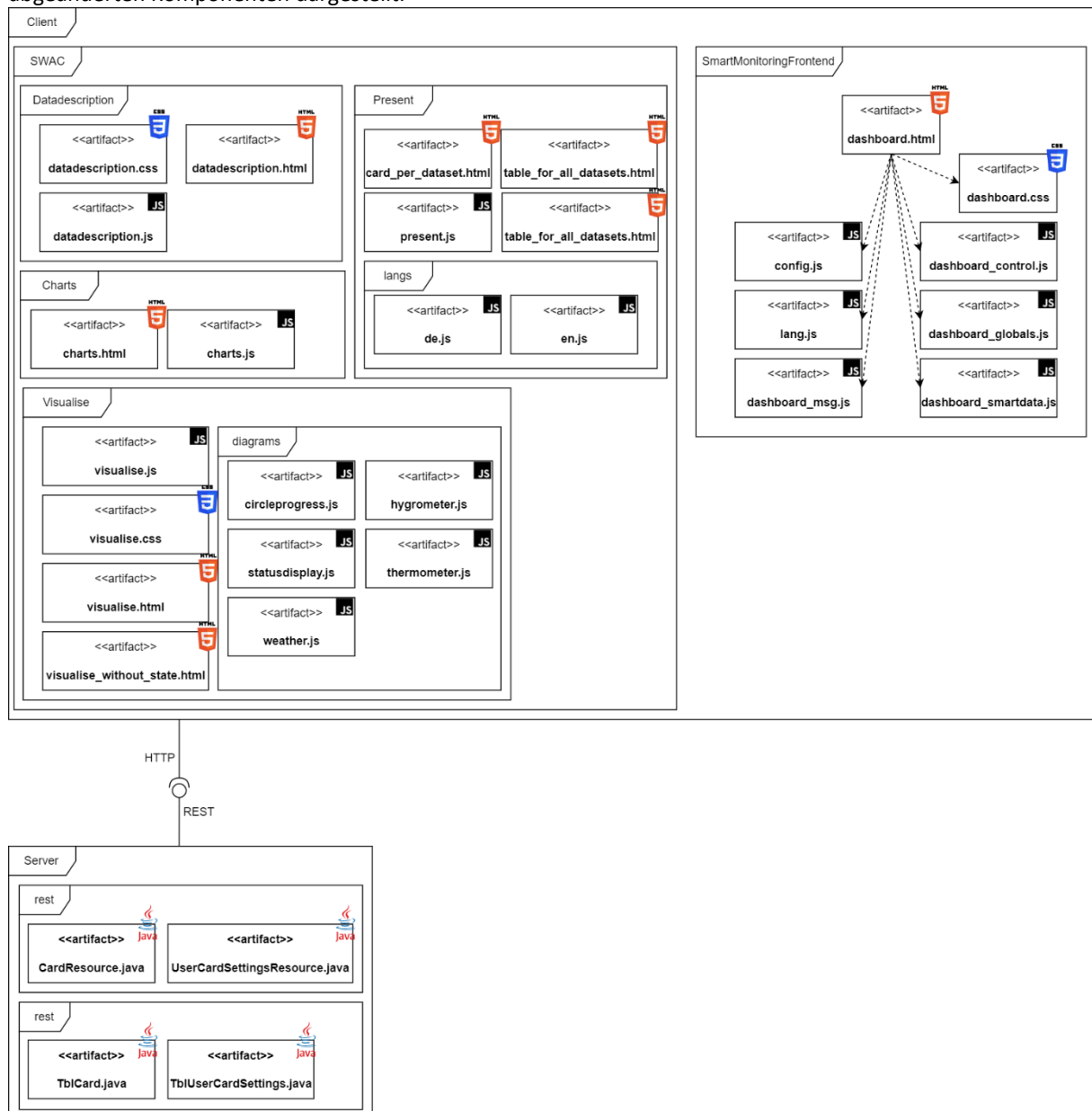


Im oben abgebildeten UseCase Diagramm ist zu sehen, dass ein User auf unsere Dashboard-Seite in der Lage ist seine eigene Kartenanordnung per Drag-and-drop zu ändern. Diese Karten von seinem Dashboard auszublenden und wieder einzublenden. Für jede Karte kann der User dann einen Zeitraum festlegen, welcher bestimmt, welche Daten auf dieser Karte angezeigt werden. Diese Einstellung kann der User auch Global vornehmen, das ausgewählte Datum wird dann für alle Karten auf dem Dashboard gesetzt.

Der Admin hat zu den Funktionen des Users auch noch die Möglichkeit neue Karten anzulegen oder Karten zu löschen. Diese angelegten Karten können dann bearbeitet werden. Zudem kann der Admin einen globalen Zeitraum festlegen, dieser Zeitraum wird dann als „Maximaler“ Wert für alle Karten auf dem Dashboard gesetzt. Somit ist der User nur noch in der Lage einen Zeitraum zwischen diesem Wert zu wählen.

2.2.2 Komponentendiagramm

Ein klassisches Klassendiagramm gibt es bei uns im Frontend nicht, da wir nicht objektorientiert gearbeitet haben, sondern uns für die Prozedurale Programmierung entschieden haben. Stattdessen haben wir ein umfangreiches Komponentendiagramm erstellt, dies beinhaltet auf der Client Seite einen Auszug aus dem SWAC wo aufgelistet wird, welche Komponenten wir bearbeitet haben. Die Charts Komponente bietet grundlegende Diagramme zum Plotten von Daten. Diese haben wir verwendet, um im Dashboard verschiedene Datentypen schön zu visualisieren. Zudem haben wir noch eigene Diagrammarten erstellt z. B.: "circleprogress", "statusdisplay", ... alle Diagramme werden mithilfe der Visualise Komponente vom SWAC erstellt und gerendert. Der Inhalt unseres Dashboards wird iterative über die Present Komponente bereitgestellt. Dazu kommen noch clientseitig die Komponenten aus dem SmartMonitoringFrontend, die aufgelisteten SWAC Komponenten kommen dann hier zum Einsatz. Serverseitig wurden die von uns abgeänderten Komponenten dargestellt.



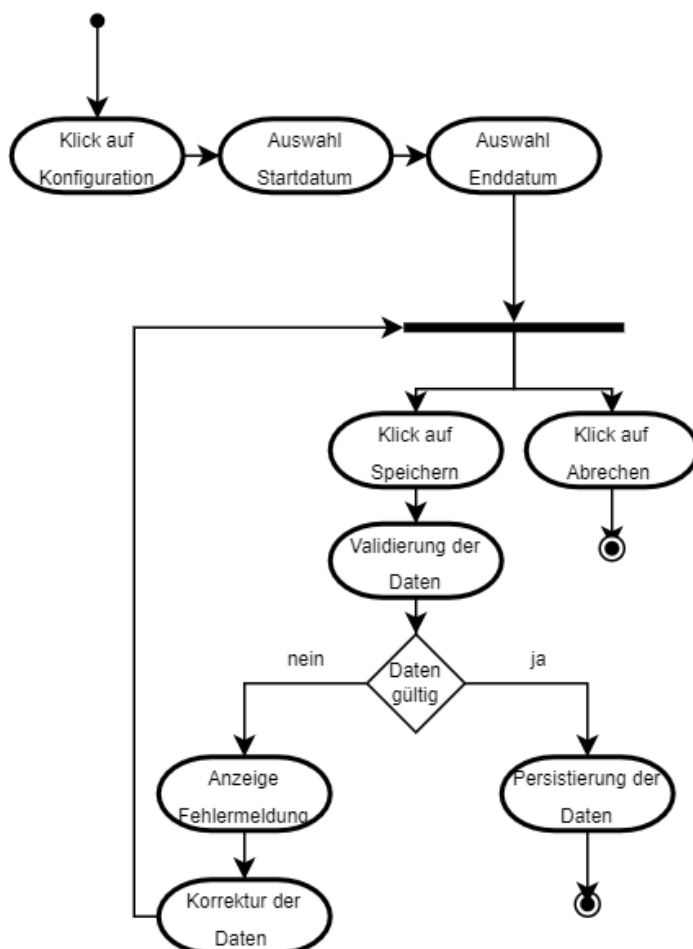
2.2.3 Aktivitätsdiagramme

Da die meisten Aktivitäten auf dem Dashboard jeweils nur mit zwei bis drei Klicks und/oder Aktionen verbunden sind, haben wir uns dazu entschieden, nur die umfangreicheren Abläufe in Form von Aktivitätsdiagrammen abzubilden.

2.2.3.1 - Konfiguration einer bestehenden Karte aus der Benutzerperspektive

Das erste Aktivitätsdiagramm beschreibt den Ablauf der Konfiguration einer bestehenden Karte aus der Benutzerperspektive. Dabei muss der Benutzer zuallererst auf den Konfigurationsbutton einer Kachel auf dem Dashboard klicken. Dabei öffnet sich ein Popup-Fenster, welches im Falle des Benutzers, zwei Eingabefelder für das Start- und Enddatum des Zeitraums der Datenabfrage umfasst. Sobald diese beiden Felder ausgefüllt wurden, hat der Benutzer die Möglichkeit, seine Eingaben über den "Speichern"-Button zu bestätigen. Sollte er dies tun, werden die eingegeben Daten validiert, indem sie auf Richtigkeit und Vollständigkeit geprüft werden und im Falle von korrekt eingegebenen Daten, wird die aktualisierte Kachel mit Hilfe des Backends persistiert. Sollte der Benutzer invalide Daten eingegeben haben, wird er darauf aufmerksam gemacht, woraufhin er seine Eingaben überarbeiten kann und erneut versuchen kann, diese zu speichern. Zusätzlich hat er jederzeit die Möglichkeit, den Bearbeitungsprozess zu beenden, indem er auf "Abbrechen" klickt.

Im Gegensatz zu dem Benutzer hat ein Admin bei der Konfiguration einer bestehenden Kachel ebenfalls die Möglichkeit, die anderen Felder zu überarbeiten, die er auch beim Erstellen einer neuen Kachel befüllt. Außerdem beziehen sich die Datumsfelder im Falle des Admins bei der Konfiguration nicht auf den tatsächlichen Zeitraum einer Karte, sondern auf den maximal verfügbaren Zeitraum, den ein Benutzer auswählen kann.

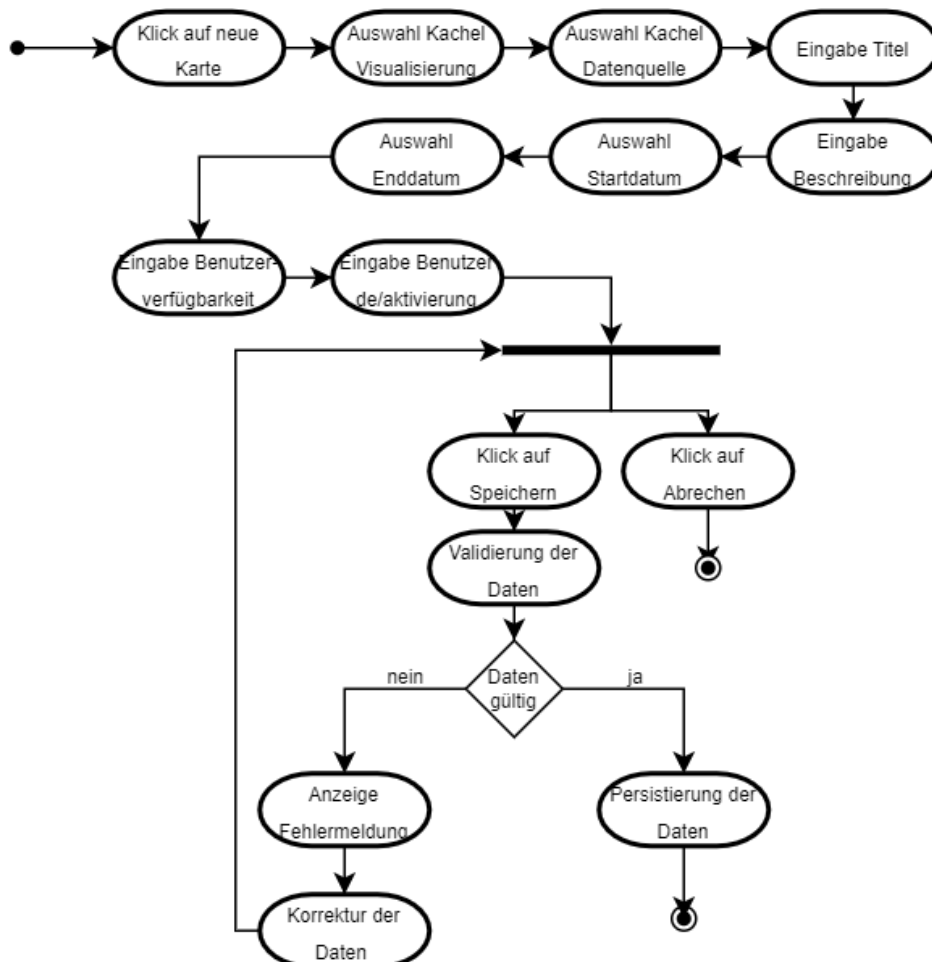


2.2.3.2 - Erstellung einer neuen Kachel aus der Admin-Perspektive

Im Gegensatz zu einem normalen Benutzer, hat ein Admin die Möglichkeit, neue Kacheln zu erstellen.

Dafür muss er zuallererst auf den "Plus"-Button am Ende der Kachelnliste klicken, woraufhin sich ein Popup-Fenster öffnet. In diesem kann er anschließend Kachel-relevante Informationen eingeben, wobei er mit der Auswahl der Visualisierungsart anfängt. Nachdem er sich für den Visualisierungstypen entschieden hat, muss der Admin eine Datenquelle auswählen, welche er mit Hilfe eines Input-Feldes anhand des Namens der Datenquelle filtern kann. Nachdem auch die Datenquelle ausgewählt wurde, kann er nun einen Titel für die Kachel vergeben und eine Beschreibung, sowie ein Start- und Enddatum festlegen. Abschließend kann er noch entscheiden, ob die Kachel für Benutzer verfügbar ist und ob diese in der Lage sind, die Kachel auszublenden.

Nachdem der Admin alle Eingaben getätigt hat, kann er diese über den "Speichern"-Button bestätigen, woraufhin die Eingaben auf ihre Richtigkeit und Vollständigkeit überprüft werden. Sollten diese den Anforderungen entsprechen, wird die neu erstellte Kachel über das Backend in der Datenbank persistiert und auf dem Dashboard angezeigt. Sollten die Daten nicht valide sein, wird eine Fehlermeldung angezeigt und der Admin hat die Möglichkeit, seine Eingaben zu überarbeiten. Zudem hat der Admin jederzeit die Möglichkeit, das Erstellen der Kachel abzubrechen, in dem auf den "Abbrechen"-Button klickt.



2.3 Code-Ausschnitte zu aussagekräftigen Details im Frontend

Im Frontend mussten wir zum einen das SmartMonitoringFrontend erweitern, welches u.a. die verschiedenen Landing-Pages umfasst und die zentrale Frontendarchitektur an sich darstellt, und zum anderen das Frontend-Framework SWAC, welches fertige Komponenten bereitstellt und als Templating-Framework dient, mit dem dynamisch JavaScript Inhalte in HTML eingepflegt werden können. Im SmartMonitoringFrontend haben wir dafür unter `"/SmartMonitoringFrontend/js/object"` einen neuen Ordner "Dashboard" angelegt, welcher insgesamt vier JavaScript Dateien für die Logik, eine Konfigurations-Datei und eine Language-Datei für die mehrsprachige Verfügbarkeit des Systems umfasst. Zudem gibt es als Einstiegspunkt die `"dashboard.html"`-Datei, welche sich unter dem Pfad `"SmartMonitoringFrontend/sites/object"` befindet.

2.3.1 Konfig

In der Konfigurationsdatei gibt es verschiedene Objekte, für die unterschiedlichen Umgebungen. Dabei wird zwischen dem Dev-Environment, dem internen Produktiv-Environment und dem externen Produktiv-Environment unterschieden. Innerhalb der Objekte können die verschiedenen Endpoints, die Umgebung an sich und sonstige umgebungsabhängige Konstanten definiert werden.

```
const dev_config = {
  DOMAIN: "http://localhost:8080/",
  ENDPOINT_GET_CARD_LIST: "smartmonitoringbackend/smartmonitoring/card/list",
  ENDPOINT_GET_CARD_AVAILABLE: "smartmonitoringbackend/smartmonitoring/card/available",
  ENDPOINT_GET_CARD_SETTINGS: "smartmonitoringbackend/smartmonitoring/usercardsettings/getByUser?id=",
  ENDPOINT_POST_CARD_SETTINGS: "/smartmonitoringbackend/smartmonitoring/usercardsettings/create",
  ENDPOINT_UPDATE_CARD_SETTINGS: "/smartmonitoringbackend/smartmonitoring/usercardsettings/update",
  ENDPOINT_GET_CARD_DATA_DB_1: "SmartDataEnvironmental",
  ENDPOINT_GET_CARD_DATA_DB_2: "SmartDataTest",
  ENDPOINT_GET_CARD_DATA_STORAGE_1: "public",
  ENDPOINT_GET_CARD_DATA_STORAGE_2: "smartmonitoring",
  ENDPOINT_POST_CARD: "/smartmonitoringbackend/smartmonitoring/card/create",
  ENDPOINT_DELETE_CARD: "/smartmonitoringbackend/smartmonitoring/card/delete?id=",
  ENDPOINT_UPDATE_CARD: "/smartmonitoringbackend/smartmonitoring/card/update",
  ENDPOINT_UPDATE_CARD_HIDE: "/smartmonitoringbackend/smartmonitoring/card/hide?id="
};
```

2.3.2 `"dashboard_control.js"` - Zentrale Logik

Die erste und zugleich die wichtigste der vier JavaScript Dateien im SmartMonitoringFrontend ist die `"dashboard_control"`-Datei. Diese umfasst die Hauptlogik zur Verwaltung der Karten bezüglich ihres Aufbaus, der Verwaltung dieser und ihrer Anordnung. Zudem wird in der Datei die Funktionalität zum Umschalten zwischen Benutzer und Admin, sowie die Validierung der Benutzerdaten verwaltet. Eines der interessanteren Code-Fragmente bezieht sich auf die Verschiebung der Kacheln, welche im Folgenden genauer beschrieben wird.

```

1  const draggables = document.querySelectorAll(".draggable");
2
3  draggables.forEach(draggable => {
4    draggable.addEventListener("dragstart", e => {
5      draggable.classList.add("dragging");
6    });
7
8    draggable.addEventListener("dragend", e => {
9      load("Neue Position...");
10     draggable.classList.remove("dragging");
11     const afterElement = getDragAfterElement(e.clientY, e.clientX);
12     const tileid = draggable.id?.split("-")[1];
13     const tile = getTileById(tileid);
14
15     if(afterElement.closest === null || (afterElement.closest.nextSibling === null && !afterElement.before)) {
16       container.appendChild(draggable);
17       persistOrder(tile, window.tile_areas.length - 1);
18     } else {
19       let after = afterElement.before ? afterElement.closest : afterElement.closest.nextSibling;
20       container.insertBefore(draggable, after);
21
22       let nextId = after.id?.split("-")[1];
23       let next = getTileById(nextId);
24       persistOrder(tile, tile.index > next.index ? next.index : next.index - 1);
25     }
26   });
27 });

```

Für die Umsetzung der Verschiebung der Karten, musste darauf geachtet werden, dass alle sensiblen Stellen stets auf dem aktuellen Stand gehalten werden. Die Verschiebung an sich haben wir so implementiert, dass alle verschiebbaren Elemente die Klasse `draggable` haben und die Event-Listener `"dragStart"` und `"dragEnd"` registriert werden, welche triggern, sobald ein Element gezogen, bzw. losgelassen wird. Sobald ein `draggable` Element gezogen wird, wird es über die Klasse `"dragging"` markiert. Beim Loslassen des Elements, wird die Markierung wieder entfernt und die UI, sowie die Liste der Kacheln werden aktualisiert. Zudem werden die Änderungen über das Backend persistiert. Somit wird erst sichergestellt, dass die Anordnung für den Benutzer immer richtig aussieht, die interne Anordnung der Karten aber ebenfalls korrekt bleibt.

```

function sort_tiles(tiles_unordered) {
  let tiles_sorted = Array();
  for(i = 0; i < tiles_unordered.length; i++) {
    if(tiles_unordered[i].previous_card === undefined || tiles_unordered[i].previous_card === null) {
      tiles_sorted.push(tiles_unordered[i]);
    }
  }
  if(tiles_unordered.length !== 1) {
    while(tiles_unordered.length !== tiles_sorted.length) {
      tiles_sorted.push(get_next(tiles_unordered, tiles_sorted[tiles_sorted.length-1]));
    }
  }
  for(i = 0; i < tiles_sorted.length; i++) {
    tiles_sorted[i].index = i;
  }
  return tiles_sorted;
};

```

Da wir die Datenbank-interne Speicherung über eine Linked-List mit Next- und Previous-Card vorgenommen haben und die Karten bei der Übergabe an das Frontend nach ID sortiert sind und nicht nach der Logik der Linked-List, musste zudem darauf geachtet werden, dass diese Sortierung beim Aufruf der Seite frontendseitig vorgenommen wird.

2.3.3 "dashboard_global.js" - Globale Variablen

Die zweite der vier JavaScript Dateien ist die "dashboard_global.js", welche sich mit dem Setzen von globalen Variablen ausgehend von bestimmten Inputs befasst. Dazu gehören alle global relevanten Systemkomponenten, wie z.B. die Kachelinformationen. Um diese global verfügbar zu machen, werden die relevanten Werte in dem Window-Objekt gespeichert.

Im Folgenden ist als Beispiel die "tile_global_legend" Funktion zu sehen, welche dazu da ist, die Legenden für die Diagramme, ausgehend von dem Diagrammtyp in dem Window-Objekt zu speichern.

```
// Globale Legenden Variable Options (nur nicht Charts)
function tile_global_legend() {
    for(i = 0; i < window.tile_areas.length; i++) {
        if(window.tile_areas[i].type !== "Charts") {
            window["vis_leg_"+parseInt(window.tile_areas[i].id)+"_options"] = {
                visuAttribute: 'PHK_class',
                showLegend: false
            };
        }
    }
};
```

2.3.4 "dashboard_msg.js" - Popup-Fenster und co.

Im Gegensatz zu den anderen JavaScript-Dateien, geht es bei der "dashboard_msg.js" primär darum, HTML-Inhalte dynamisch zu erstellen. Dies ist in erster Linie wichtig, um auf Button-Klicks zu reagieren und anschließend Popup-Fenster oder Ähnliches anzuzeigen.

Wie in dem Folgenden Beispiel zur Erstellung des "Speichern"-Buttons und der dazugehörigen Animation zu sehen, wird dabei das Element an sich in JavaScript erstellt und alle relevanten Attribute wie Styling-Information in Form von Klassen oder ID's, sowie sonstige Attribute und Informationen gesetzt.

```
// Speichern Box + Animation
function save(msg) {
    conf_box = document.getElementById("config-box");
    // Ist eine Box schon offen => Schließen und diese Box wieder öffnen
    if(conf_box.classList.contains("flex")) {
        msg_off();
        save(msg);
        return;
    }
    save_box = document.createElement("div");
    save_box.setAttribute("class", "save-box");
    save_img = document.createElement("img");
    save_img.setAttribute("class", "save-anim");
    save_img.setAttribute("src", "../../img/dashboard/bestaetigt.png");
    save_img.setAttribute("alt", "Save");
    text_box = document.createElement("span");
    text_box.setAttribute("id", "saveboxtext");
    text_box.innerText = msg;
    save_box.appendChild(save_img);
    save_box.appendChild(text_box);
    conf_box.appendChild(save_box);
    conf_box.classList.add("flex");
};
```

Neben den Darstellungsfunktionen umfasst die "dashboard_msg.js" zudem einige Utility-Funktionen, wie z.B. die "formatData"-Funktion, welche dazu da ist, ein übergebenes Datum in ein Datenbank-kompatibles Format zu bringen, da es Unterschiede zwischen dem Datenbank-Datenformat und dem Frontend-Datenformat gibt.

```
// Datum für die DB Formatieren
function formatDate(date, tp) {
    date = new Date(date);
    if(tp === 0) {
        date.setHours(0, 0, 0, 0);
    } else {
        date.setHours(23, 59, 59, 999);
    }
    let year = date.getFullYear();
    let month = addLeadingZero(date.getMonth() + 1);
    let day = addLeadingZero(date.getDate());
    let hours = addLeadingZero(date.getHours());
    let minutes = addLeadingZero(date.getMinutes());
    let seconds = addLeadingZero(date.getSeconds());
    return `${year}-${month}-${day}T${hours}:${minutes}:${seconds}`;
};
```

2.3.5 "dashboard_smartdata.js" - Backend-Calls

Die letzte der vier JavaScript-Hauptdateien im SmartMonitoringFrontend ist die "dashboard_smartdata.js", welche primär die Backend-Zugriffe und asynchrone Aufrufe im allgemeinen umfasst.

Dazu gehört u.a. das Erstellen, Abfragen und Bearbeiten der Karten und die damit verbundenen Persistierungen über das Backend.

In dem Folgenden ist die Funktion für das Updaten einer Kachel zu sehen, welche ein überarbeitetes Kachel-Objekt entgegennimmt und dieses über die interne fetchData-Funktion mit dem Backend kommuniziert, um über den aus der Konfig importierten Endpoint die besagten Daten zu persistieren.

```
const updateUserSettings = async data => {
    dt = await fetchData(config.ENDPOINT_UPDATE_CARD_SETTINGS, {
        method: 'PUT',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(data)
    });
    return dt;
};
```

2.3.6 Visualise-Komponente

Neben dem SmartMonitoringFrontend mussten auch einige Komponenten im SWAC überarbeitet und erweitert werden, wobei wir uns primär mit der Visualise-Komponente auseinandergesetzt haben, welche dafür da ist, die verschiedenen Grafiktypen innerhalb der Kachel darzustellen. Zu den Grafiktypen gehören u.a. ein Wetter Icon, ein dynamisches Thermometer und ein Kreisdiagramm, welches den prozentualen Fortschritt eines Datensatzes visualisiert. Da dieses Kreisdiagramm der einzige Diagrammtyp ist, der nicht erweitert, sondern von Anfang an implementiert werden musste, wird im Folgenden noch etwas genauer darauf eingegangen.

Zuallererst war es wichtig, von der Diagrammklasse zu erben, da über diese und das Konzept der Polymorphie in der Visualise-Komponente umgesetzt wird, indem ausgehend von der ausgewählten Diagrammart und der gegebenen Parameter der richtige Diagrammtyp (Hier der Kreis) erstellt wird.

```

/**
 * @class
 * Creates the CircleProgress
 */
export default class CircleProgress extends Diagram {
  constructor(unit, name, width, height, datadescription, diagramDef, comp) {
    super(unit, name, width, height, datadescription, diagramDef, comp);
    // Default style values

    this.circleX = 25;
    this.circleY = 115;
    this.circleR = 55;
  }
}

```

Außerdem musste die drawValuediagram-Methode überschrieben werden, da diese der Anlaufpunkt ist, um die für das Diagramm relevanten Inhalte zu erstellen und diese darzustellen. In diesem Fall wurde das Diagramm mit Hilfe eines svg-Elements umgesetzt, wobei das Diagramm an sich aus drei Kreisen (Hintergrund, Fortschritt-Hintergrund und tatsächlicher Fortschritt), sowie einem Textelement besteht.

```

drawValueDiagram(name, value) {
  if (!this.datadescription) {
    Msg.error('CircleProgress', 'There is no datadescription defined.');
```

```

    return;
  }

  if (!this.datadescription.getMinDefinition(this.diagramDef.attr)) {
    Msg.error('CircleProgress', 'There is no >visualise_legend_data< for attribute >' + this.diagramDef.attr + '<');
```

```

    return;
  }

  // Wrapper Div Element
  let wrapper = document.createElement('div');
  wrapper.style.position = 'relative';
  wrapper.style.display = 'flex';
  wrapper.style.justifyContent = 'center';
  wrapper.style.alignItems = 'center';

  // Svg Element
  let svgElement = document.createElementNS("http://www.w3.org/2000/svg", "svg");
  svgElement.setAttribute("viewBox", "0 0 60 240");
  svgElement.setAttribute("width", this.width);
  svgElement.setAttribute("height", this.height);
  svgElement.style.position = 'relative';
}

```

Andere Komponenten, die ebenfalls im Zuge des Projekts erweitert werden mussten, waren die DataDescription-Komponente und die Charts-Komponente.

2.3 Code-Ausschnitte zu aussagekräftigen Details im Backend (Janik)

2.4.1 Datenbankschema

Das Fundament der gesamten Erweiterung des Smart-Monitoring-Backend ist die Erweiterung des Datenbankschemas. Damit sowohl die Dashboard-Karten als auch die Daten für diese an derselben Stelle zu finden sind, verwenden wir das *smartmonitoring*-Schema in der *smartmonitoring_dachanlage*-Datenbank. Hier waren vorher die einzelnen Module der Solaranlage zu finden und nun befinden sich auch Dashboard relevante Daten an dieser Stelle.

Da das Smart-Monitoring-Backend Hibernate nutzt, können wir die benötigten Tabellen mithilfe von Java-Beans erstellen und automatisch beim Start des Backends von Hibernate auf die Datenbank aufspielen lassen. Außerdem ermöglicht uns Hibernate Relationen zwischen den einzelnen Tabellen mithilfe von Annotation zu realisieren, so ist alles an einer Stelle im Code zu finden und leicht nachvollziehbar. Um alle Aspekte des Dashboards in der Datenbank abbilden zu können, benötigen wir drei Tabellen. Die Definitionen dieser

Tabellen sind in dem Package *de.fhbielefeld.smartmonitoring.jpa* zu finden. *TblCard.java* repräsentiert eine Karte auf dem Dashboard, *TblUserCardSettings.java* sind die Einstellungen des Nutzers für eine Karte und *tbl_card_join_oo* wird automatisch von Hibernate erstellt und spiegelt die Beziehung zwischen Karten und Modulen der Solaranlage wieder.

2.4.1.1 TblCard.java

Die TblCard-Tabelle wird genutzt, um einzelne Karten in der Datenbank zu speichern. Sie soll Informationen wie Name, Inhalt und Zeitraum (der Daten) der Karten speichern. Außerdem soll eine TblCard-Entität in Verbindung zu einem Modul der Solaranlage stehen können, um spezifische Daten zu diesem Modul darstellen zu können.

```
1 @Entity
2 @Table(name = "tbl_card", schema = "smartmonitoring")
3 @XmlRootElement
4 @Schema(name="TblCard",description="A card, used to display set information")
5 @NamedQueries({
6     @NamedQuery(name = "TblCard.findAll", query = "SELECT t FROM TblCard t ORDER BY t.id"),
7     @NamedQuery(name = "TblCard.findAvailable", query = "SELECT t FROM TblCard t WHERE available=true")
8 })
9 public class TblCard implements Serializable{
```

Abbildung 3 de.fhbielefeld.jpa.TblCard.java Z. 35 ff. - Bean Annotationen

Abbildung 3 zeigt, wie die TblCard.java mithilfe von Hibernate Annotationen konfiguriert wird. Die *@Entity*-Annotation weist Hibernate darauf hin, dass es sich bei dieser Klasse um eine Entität handelt, welche in der Datenbank persistiert werden soll.

@Table ermöglicht es, die Informationen der Tabelle gezielt zu verändern. In Zeile 2 legen wir hiermit den tatsächlichen Namen („tbl_card“), sowie das zu nutzende Schema („smartmonitoring“) in der Zieldatenbank fest. Die *@XmlRootElement*-Annotation wird von javax.json genutzt, um die Javabeans in JSON zu serialisieren, auf diese Weise können wir das Objekt einfach an eine Response hängen und es wird automatisch in JSON konvertiert. *@Schema* ist eine Swagger-Annotation, sie gibt genauere Informationen für Swagger über dieses Schema, die später in der von Swagger generierten Dokumentation angezeigt werden.

@NamedQueries ist wieder eine Hibernate-Annotation, welche gezielte Datenbankabfragen ermöglicht.

```
1 @Schema(description="Cardtype, setting the way the information is displayed in the frontend")
2 @Column(nullable=false,length=200)
3 private String type;
```

Abbildung 4 de.fhbielefeld.jpa.TblCard.java Z. 60 ff.- Attributbezeichnung

Abbildung 4 zeigt beispielhaft, wie ein Attribut mithilfe von Hibernate definiert und des Weiteren mit Swagger dokumentiert wird. Mithilfe der *@Column*-Annotation können wir die Ausprägung der type-Eigenschaft einer Karte beeinflussen. Der Parameter *nullable* bestimmt, ob die Spalte *type* der Kartentabelle angegeben werden muss oder mit *null* belegt werden kann. In diesem Fall haben wir uns dazu entschieden den Typ einer Karte als Pflichtfeld zu definieren, da wir ansonst nicht ermitteln können, wie die Karte visualisiert werden soll. Der *length* Parameter legt die maximale Länge des Strings fest, der in dieser Spalte gespeichert werden kann. Mit einer Länge von 200 Zeichen sollte jeder mögliche Typ einer Karte beschrieben werden können.

Die *@Schema*-Annotation wird erneut von Swagger genutzt, um in diesem Fall eine Beschreibung für das Attribut des Schemas zu erstellen.



```

1  @ManyToMany(cascade = { CascadeType.DETACH })
2  @JoinTable(
3      name = "smartmonitoring.tbl_card_join_oo",
4      joinColumns = { @JoinColumn(name = "card_id") },
5      inverseJoinColumns = { @JoinColumn(name = "observedobject_id") }
6  )
7  private List<TblObservedObject> observedObjects = new ArrayList<>();

```

Abbildung 5 de.fhbielefeld.jpa.TblCard.java Z. 51 ff. – Beziehung zu Observed-Objects

Damit im Frontend klar ist, welche Daten mithilfe einer Karte angezeigt werden sollen benötigt die Karte einen Verweis auf die Quelle dieser Daten. Dies geschieht im Falle eines Solarmoduls mithilfe der Beziehung zu *Observed-Objects*, welche bereits Teil des Smart-Monitoring-Backend waren. Ein *Observed-Object* beinhaltet einen Verweis auf ein Modul, oder genauer eine *Collection*, in der dann mithilfe von Smart-Data nach Daten gesucht werden kann.

In Abbildung 5 ist zu sehen, wie eine Beziehung zwischen *Observed-Object* und *TblCard* mithilfe von Hibernate-Annotationen definiert werden kann. In diesem Fall handelt es sich um eine Many-To-Many-Beziehung, da eine Karte die Daten von mehreren Modulen zusammenfassen kann und ein Modul auch in mehreren Karten genutzt werden kann.

Mithilfe der *@ManyToMany-Annotation* wird Hibernate darauf hingewiesen, dass hier eine neue Tabelle benötigt wird. Der *cascade*-Parameter gibt an, welche Operationen in diese Tabelle kaskadiert werden sollen, in diesem Fall nur detach-Operationen. *@JoinTable* definiert die Tabelle, mit welcher der ManyToMany-Join stattfinden soll. Im Parameter *name* wird ein Name für die neu erstellte Join-Tabelle angegeben. Entsprechend der Namensgebung im Smart-Monitoring-Backend haben wir uns hier für *tbl_card_join_oo* entschieden, wodurch sofort zu erkennen ist welche Tabellen hier gejoined werden. Die Parameter *joinColumns* und *inverseJoinColumns* geben die Felder an, auf denen die beiden Tabellen gejoined werden sollen, in diesem Fall ihre IDs.

Damit diese Beziehung später einfach nutzbar ist, werden die Observed-Objects in einer Liste gespeichert, sobald eine Karte von der Datenbank abgerufen wird.

Alle weiteren Attribute der Karte werden wie in dem Beispiel der Abbildung 4 mit einer *@Schema* und einer *@Column-Annotation* konfiguriert.

Neben dem Typ und der Beziehung zu ObservedObjects beinhaltet die Karte Informationen zur Sichtbarkeit (*available*), einen Titel (*title*), eine Quelle der Daten (*resource*), einen zulässigen Zeitraum für den Abruf der Daten (*maxSpan_begin*, *maxSpan_end*), sowie eine Beschreibung (*description*).

Wie von Hibernate gefordert, erhält jedes dieser Attribute Getter- und Setter-Methoden, in folgendem Format: `public TypDesAttributes getNameDesAttributes(){...}`.

2.4.1.2 TblUserCardSettings

Jeder Nutzer eines Dashboards hat eigene Ansprüche an dieses und ist vermutlich nicht immer an allgemeinen Daten interessiert, daher muss unser Dashboard, aber auch die Daten, vom User konfigurierbar und personalisierbar sein. Dies ermöglichen wir dem User mithilfe der *TblUserCardSettings*. Hier wird die Anordnung der Karten für einen User gespeichert sowie weitere Anpassungen wie der Zeitraum der Daten. Da die wichtigsten Elemente einer Java-Bean Definition bereits in [2.4.2.1 TblUserCardSettings](#) erwähnt wurden, wird im Folgenden nur auf Besonderheiten der *TblUserCardSettings* eingegangen.

Es gibt mehrere Ansätze die Reihenfolge von Objekten in einer Datenbank zu speichern, jeder dieser Ansätze hat Vor- und Nachteile. Erste Überlegungen waren, die Position einer Karte einfach als Integer zu speichern und diese Zahl dann als Position in einem Array zu interpretieren. Jedoch ist schnell aufgefallen, dass dies sehr schlecht skaliert, denn wenn eine Karte in der Mitte von 500 Karten gelöscht wird, muss nicht nur diese eine Karte, sondern alle 249 Nachfolger ebenfalls aktualisiert werden. Das gleiche gilt für Einfüge- und Verschiebeoperationen.

Damit unsere Applikation besser skaliert werden kann, entschieden wir uns für eine doppelt verkettete Liste. Mithilfe einer solchen Liste können einzelne Karten gelöscht/eingehängt/verschoben werden und nur die direkten Nachbarn dieser Karten müssen aktualisiert werden. Auf diese Weise wird beim Bearbeiten der Position einer Karte immer nur eine feste Anzahl von Karten bearbeitet, unabhängig von der Gesamtanzahl von Karten.

```
1  @Schema(description="Card affected by this setting")
2  @ManyToOne(fetch=FetchType.LAZY)
3  @JoinColumn(name="card_id")
4  private TblCard card;
5
6  @Schema(description="Card to the left of this card, null if no card to the left")
7  @OneToOne()
8  @JoinColumn(name="previous_card", referencedColumnName="id")
9  private TblCard previous_card;
10
11 @Schema(description="Card to the right of this card, null if no card to the right")
12 @OneToOne()
13 @JoinColumn(name="next_card", referencedColumnName="id")
14 private TblCard next_card;
```

Abbildung 6 de.fhbielefeld.jp.a.TblUserCardSettings.java Z. 58 ff. - Relationen zur TblCard als doppelt verkettete Liste

Um eine solche doppelt verkettete Liste in der Datenbank abbilden zu können, müssen die UserSettings einen Verweis auf die einzustellende Karte (Abbildung 6 Z. 4), den linken Nachbarn (Abbildung 6 Z. 9) sowie den rechten Nachbarn (Abbildung 6 Z. 14) haben. Durch diese Anordnung können wir nun über diese Liste von Anfang bis Ende navigieren und für jeden User eine Unterschiedliche Anordnung speichern.

Wie in Abbildung 6 zu sehen, sind die linken und rechten Nachbarn als *OneToOne*-Beziehung modelliert, da eine Karte immer nur an einer Stelle in der Liste stehen kann, die einzustellende Karte kann jedoch mehrere Einstellungen erhalten, weshalb hier eine *ManyToOne*-Beziehung von Nöten ist.

Wie auch schon bei TblCard.java werden hier Swagger-Annotationen genutzt, um das Schema vernünftig zu dokumentieren.

2.4.2 REST-Schnittstellen

Damit das Frontend unsere neu erstellten Tabellen auch voll ausnutzen kann, haben wir uns dazu entschieden die bestehenden REST-Schnittstellen des Smart-Monitoring-Backend zu erweitern. So werden auch Karten und Nutzereinstellungen über diese Schnittstelle gehandhabt. Dies bietet im Gegensatz zu Smart-Data den Vorteil, dass wir im Backend die Parameter überprüfen und mehrere Datensätze in einer Anfrage behandeln können. Somit haben wir auch volle Kontrolle über die Handhabung der doppelt verketteten Liste, wodurch im Frontend keine weitere Arbeit geleistet werden muss.

Die Erweiterung berücksichtigt alle typischen Anfragen, also POST-/PUT-/GET- und DELETE-Anfragen, sowohl für Karten als auch für Nutzereinstellungen.

2.4.2.1 TblCard

Die POST-Methode für eine neue Karte ist ziemlich simpel, wir erhalten das Kartenobjekt dank *javax.json* als Parameter in der Methode und überprüfen einige wichtige Parameter. Sind alle Parameter innerhalb ihrer Vorgaben und das Kartenobjekt vollständig, wird der Entitymanager *em* genutzt, um das Objekt zu persistieren.

```
1 //only add the card if the type is defined
2 if (card.getType() == null || !Arrays.stream(Types).anyMatch(card.getType()::equals)) {
3     rob.setStatus(Response.Status.BAD_REQUEST).addErrorMessage("ERRORMSG");
4     return rob.toResponse();
5 }
6
7 //persist the card
8 try {
9     this.utx.begin();
10    this.em.persist(card);
11    this.utx.commit();
12    //signale that card was created
13    rob.setStatus(Response.Status.CREATED);
14    //return id of created card
15    rob.add("id", card.getId());
16 } catch (/*EXCEPTIONS*/) {
17     rob.setStatus(Response.Status.INTERNAL_SERVER_ERROR).addErrorMessage("EXCEPTIONMSG");
18     return rob.toResponse();
19 }
20 return rob.toResponse();
21 }
```

Abbildung 7 de.fhbielefeld.smartmonitoring.rest.CardResource.java Z. 141 ff. – POST-Route

In Abbildung 7 sehen wir, wie dieser Vorgang im Code gehandhabt wird. In Zeilen 2-4 wird geprüft, ob der Typ der Karte innerhalb der festgelegten Vorgaben liegt. Damit hier kein Fehlercode von der Route zurückgegeben wird, muss der Typ einen der folgenden Werte annehmen:

[Weather, Thermometer, CircleProgress, Charts, StatusDisplay]

Das Frontend nutzt dieses Feld, um zu entscheiden welcher Kartentyp am Ende gerendert werden muss, wird hier ein sinnloser oder nicht vorhandener Wert eingetragen, können Fehler auftreten. Daher achten wir schon hier beim Erzeugen einer neuen Karte darauf, dass die Werte den festgelegten Bereich nicht verlassen.

Ist die Parameterüberprüfung durchgelaufen, kann das neue Kartenobjekt gespeichert werden. Dazu nutzen wir eine UserTransaction *utx* und den Entitymanager *em*. Die Transaction stellt eine Verbindung zur Datenbank her und erzeugt einen Persistenz Kontext, in dem die Änderungen durchgeführt werden, bevor sie auf die Datenbank gepushed werden. Der Entitymanager übernimmt die SQL-Anfragen und ist dank der zuvor erwähnten Annotationen in der Lage das Javaobjekt auf der Datenbank zu persistieren.

Ist während des Speicherns nichts schiefgelaufen, wird die ID der neu erstellten Entity an die Response gehängt und mithilfe des *Response-Object-Builders* an das Frontend zurückgegeben.

Damit das Frontend nicht immer alle Karten anfragen muss die vorhanden sind, gibt es verschiedene GET-Methoden, um verschiedene Rückgaben zu empfangen.

Eine einzelne Karte kann über die *getById*-Route angefragt werden, dazu wird eine ID im Queryparameter übergeben. Findet das Backend eine Karte wird sie zurückgegeben, wird keine Karte gefunden wird ein *404_NOT_FOUND* zurückgegeben.

Die *list*-Route kann genutzt werden um alle verfügbaren Karten als Liste zu erhalten. Hier sind keinerlei Parameter nötig, es werden einfach alle Karten zurückgegeben die vorhanden sind.

Mithilfe der *available*-Route können alle Karten angefragt werden, welche nur für Nutzer sichtbar sind. Ist das *available*-Feld einer Karte *true*, sollen sowohl Admins als auch Nutzer die Karte erhalten können. Ist das Feld *false* sollen nur Admins diese Karten erhalten. Da Smart-User momentan noch nicht zwischen Admins und Nutzern unterscheiden kann, wird diese Unterscheidung vom Frontend übernommen und somit muss eine Route bestehen, die nur von Nutzer sichtbare Karten zurückgibt.

```
1 public Response availableList() {
2     ResponseObjectBuilder rob = new ResponseObjectBuilder();
3     List<TblCard> tblCardObjects = this.em.createNamedQuery("TblCard.findAvailable", TblCard.class).getResultList();
4     rob.add("list", tblCardObjects);
5     rob.setStatus(Response.Status.OK);
6     return rob.toResponse();
7 }
```

Abbildung 8 de.fhbielefeld.smartmonitoring.rest.CardResource.java Z. 294 ff. – GET-Route

Wie in Abbildung 8 zu sehen, wird hier eine *NamedQuery* genutzt, um alle Karten zu filtern, die *available* sind. *NamedQueries* werden an mehreren Stellen genutzt, um die Suchergebnisse schon Datenbank seitig zu filtern.

In Abbildung 9 sind ein paar *NamedQuery*s dargestellt, wie wir sie im Backend nutzen, um Suchanfragen schnell durchführen zu können. Mithilfe dieser *NamedQuery*s können komplexere SQL-Anfragen erstellt und

```

1 @NamedQueries({
2     @NamedQuery(name = "TblCard.findAll", query = "SELECT t FROM TblCard t ORDER BY t.id"),
3     @NamedQuery(name = "TblCard.findAvailable", query = "SELECT t FROM TblCard t WHERE available=true")
4 })
5 public class TblCard implements Serializable{

```

Abbildung 9 de.fhbielefeld.smartmonitoring.jpa.TblCard.java Z. 39 ff. – *NamedQuery*s

dynamisch mit Parametern gefüllt werden. Auch *TblUserCardSettings* verfügt über *NamedQuery*s welche einige Anfragen deutlich simpler und effizienter gestalten.

Ähnlich wie das Erstellen einer Karte funktioniert das Aktualisieren von bereits existierenden Karten, nur wird hier eine PUT-Operation genutzt. Es wird eine Karte im Body übergeben, von *javax.json* in ein Javaobjekt konvertiert und dann mithilfe des Entitymanagers auf die Datenbank gemerged. Als Antwort erhält der Nutzer dann entweder einen Fehlercode oder ein 200 (OK), wenn die Karte aktualisiert wurde.

Muss eine Karte entfernt werden, geschieht dies über die DELETE-Route. Da die Karten in Relation zu den Nutzereinstellungen stehen muss hier darauf geachtet werden, dass nach dem Löschen der Karte alle Nutzereinstellungen konsistent bleiben. Eine gelöschte Karte sollte nicht mehr in der doppelt verketteten Liste existieren und so die Liste unbrauchbar machen. Dazu muss, bevor eine Karte gelöscht wird, geprüft werden, ob diese irgendwo bereits in einer Liste existiert und dort erst ausgehängt werden. Um dieses Problem zu lösen, nutzt die DELETE-Route eine Hilfsfunktion, die sich allein um das Aushängen der Karte aus der Liste kümmert.

```

1 private void onDelete_UserCardSettings(List<TblUserCardSettings> userCardSettings) throws /*EXCEPTIONS*/{...}

```

Abbildung 10 de.fhbielefeld.smartmonitoring.rest.CardResource.java Z. 486 – DELETE-Route

Als Parameter erhält die Funktion eine Liste von Nutzereinstellungen (siehe Abbildung 10), welche diese Karte enthalten. Da das Dashboard von mehreren Nutzern genutzt werden kann und jeder Nutzer eine eigene Konfiguration des Dashboards vornehmen kann, müssen alle verschiedenen Konfigurationen bearbeitet und korrigiert werden, wenn eine Karte gelöscht wird.

```

1 //handle every setting (every user)
2 for(TblUserCardSettings setting : userCardSettings){
3
4     if(setting.getNext_card() != null && setting.getPrevious_card() != null){
5         //card is somewhere in the middle of the list
6         /*...*/
7     }else if(setting.getNext_card() == null && setting.getPrevious_card() == null){
8         //Do nothing since there are no links to be handled
9         /*...*/
10    }else if(setting.getNext_card() == null){
11        //card is head of current list
12        /*...*/
13    }else if(setting.getPrevious_card() == null){
14        //card is tail of current list
15        /*...*/
16    }else{
17        //should never reach this as setting should be somewhere in the list
18        this.utx.rollback();
19        //throw notSupported when reaching this case (something must have gone wrong)
20        throw new NotSupportedException();
21    }
22    //delete the setting we just handled from database
23    this.em.remove(em.contains(setting) ? setting : em.merge(setting));
24 }

```

Abbildung 11 de.fhbielefeld.smartmonitoring.rest.CardResource.java Z. 492 ff. – DELETE-Route

Dann wird für jede Nutzereinstellung geprüft, wo sie sich aktuell in der Liste befindet (siehe Abbildung 11), ob Operationen auf der doppelt verketteten Liste ausgeführt werden müssen und wie diese Operationen aussehen müssen. Abbildung 12 zeigt, wie mit einem Eintrag in der Liste umgegangen wird, wenn er sich am

Head, also vorne in der Liste, befindet. Da der Head nur eine Verbindung in der Liste hat, muss auch nur dieser bearbeitet werden.

```

1 //card is head of current list
2 TblUserCardSettings previous = this.em.createNamedQuery("TblUserCardSettings.findByUser", TblUserCardSettings.class)
3     .setParameter("user_id", setting.getUser_id())
4     .setParameter("card", setting.getPrevious_card())
5     .getResultStream().findFirst().orElse(null);
6 //unlink current setting from existing list --> turn previous into new head
7 previous.setNext_card(null);
8 //persist new state on database
9 this.em.merge(previous);

```

Abbildung 12 de.fhbielefeld.smartmonitoring.rest.CardResource.java Z. 517 ff. – DELETE-Route

Sind alle Relationen zwischen Karte und Nutzereinstellungen gelöscht, kann auch die Karte sicher gelöscht werden und keine Liste ist inkonsistent.

2.4.2.2 UserCardSettingsResource

Wie schon oft erwähnt werden die Nutzereinstellungen extra im Backend gespeichert, um individuelle Kartenkonfigurationen zu unterstützen. Damit die Positionierung der Karten konsistent gespeichert werden kann, werden die Nutzereinstellungen in Form einer doppelt verketteten Liste verwaltet um das Verschieben, Löschen und Einfügen so effizient wie möglich zu gestalten.

Beim Erstellen einer Nutzereinstellung wird diese automatisch vorne in die Liste eingehängt. Dies ist die einfachste Operation, die eine doppelt verkettete Liste bietet und somit ideal für das Hinzufügen von neuen Elementen. Damit die Nutzereinstellung erstellt werden kann wird, wie bei den Karten auch, erst überprüft, ob alle Felder vollständig und richtig belegt sind. Dann wird der aktuelle Head der Liste gesucht und die Karte an dieser Position als *next_card* eingehängt. Existiert noch keine Liste für diesen User wird sie „initialisiert“, indem die zu erstellende Einstellung als neuer Head der Liste gekennzeichnet wird.

Die wichtigste und wohl komplizierteste Route für die Nutzereinstellungen ist die PUT-Route. Da hier mit Einstellungen gearbeitet wird, die bereits existieren und sich die Position dieser Einstellungen jederzeit ändern kann, muss zwischen verschiedenen Operationen unterschieden werden (siehe Abbildung 13). So kann es zum Beispiel sein, dass die Position einer Karte sich gar nicht verändert. In diesem Fall soll die Einstellung nicht unnötig aus der Liste ausgehängt und dann wieder an derselben Stelle eingehängt werden. Also werden einfach die anderen Felder aktualisiert unabhängig von der Position der Einstellung.

```

1 //persist updated card
2 try {
3     if(updated.getNext_card() == null && updated.getPrevious_card() == null){
4         //only the settings need to be updated, no positional update
5     }else if(updated.getNext_card() != null && updated.getPrevious_card() != null){
6         //the card needs to be added somewhere in the middle of the list
7         return inMiddle(current, updated, rob);
8     }else if(updated.getNext_card() != null){
9         //the card needs to be added as tail
10        return asTail(current, updated, rob);
11    }else if(updated.getPrevious_card() != null){
12        // the card needs to be added as tail
13        return asHead(current, updated, rob);
14    }else{
15        //something in this request does not work out, !should never reach this!
16        rob.setStatus(Response.Status.BAD_REQUEST).addErrorMessage("ERRORMSG");
17        return rob.toResponse();
18    }

```

Abbildung 13 de.fhbielefeld.smartmonitoring.rest.UserCardSettingsResource.java Z. 401 ff. – UPDATE-Route

Um diese einzelnen Fälle behandeln zu können, werden drei Hilfsfunktionen eingeführt, die jeweils einen Fall abfangen und gezielt behandeln. Da die aktuelle Nutzereinstellung bereits in der Liste hängt und sich überall in dieser befinden kann, müssen die Hilfsfunktionen unterscheiden, ob sie gerade eine Einstellung von vorne nach hinten schieben oder vielleicht von der Mitte an eine andere Stelle in der Mitte.

inMiddle() nimmt die aktuelle Nutzereinstellung und fügt sie irgendwo in der Mitte der Liste ein, dann prüft sie wo sich die aktuelle Einstellung befindet, und führt entsprechende Operationen aus, um die Einstellung konsistent in der Liste zu verschieben. *asTail()* und *asHead()* gehen ähnlich vor, nur fügen sie die Einstellung jeweils als neuen Tail oder neuen Head wieder in die Liste ein.

Da sich diese Hilfsoperationen über ca. 400 Zeilen Code verteilen, schauen wir uns hier nur ein kleines Beispiel an. Alles andere würde den Rahmen dieser Ausarbeitung sprengen. In dem hier betrachteten Fall soll eine Karte aus der

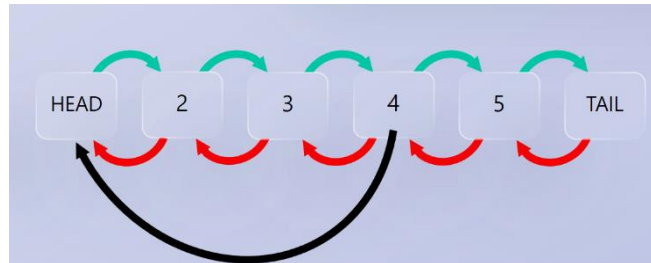


Abbildung 14 Verkettete Liste Beispiel

Mitte der Liste an den Head der Liste verschoben werden (siehe Abbildung 14). Da die Einstellung an den Head der Liste verschoben wird, wird die *asHead()*-Funktion aufgerufen. Die Funktion prüft nun, ob die aktuelle betrachtete Einstellung in der Mitte der Liste hängt, dies ist hier der Fall.

```

1 //current is currently somewhere in the middle of the list
2 //unlink current from list
3 TblUserCardSettings next = this.em.createNamedQuery("TblUserCardSettings.findByUser", TblUserCardSettings.class)
4     .setParameter("user_id", current.getUser_id())
5     .setParameter("card", current.getNext_card())
6     .getResultStream().findFirst().orElse(null);
7 TblUserCardSettings previous = this.em.createNamedQuery("TblUserCardSettings.findByUser", TblUserCardSettings.class)
8     .setParameter("user_id", current.getUser_id())
9     .setParameter("card", current.getPrevious_card())
10    .getResultStream().findFirst().orElse(null);
11 previous.setNext_card(next.getCard());
12 next.setPrevious_card(previous.getCard());
13 //relink current as head of list
14 TblUserCardSettings head = this.em.createNamedQuery("TblUserCardSettings.findByUser", TblUserCardSettings.class)
15     .setParameter("user_id", current.getUser_id())
16     .setParameter("card", updated.getPrevious_card())
17     .getResultStream().findFirst().orElse(null);
18 //If the head is the same as the old next, make sure all values are the same
19 if(Objects.equals(head.getCard().getId(), next.getCard().getId())){
20     next.setNext_card(current.getCard());
21 }
22 head.setNext_card(current.getCard());
23 current.setPrevious_card(head.getCard());
24 current.setNext_card(null);

```

Abbildung 15 de.fhbielefeld.smartmonitoring.rest.UserCardSettingsResource.java Z. 784 ff. - *asHead()*-Funktionsausschnitt

Die Funktion ermittelt nun alle beteiligten Einstellungen mithilfe von *NamedQueries*. Dann werden die Einstellungen neu verlinkt wie in Abbildung 15 Zeile 11 f. zu sehen. Für den Fall, dass der alte Head der Liste dieselbe Einstellung ist, wie die *next_card* Einstellung der aktuell betrachteten Einstellung, wird in Zeile 20 dafür gesorgt, dass die neue Verkettung nicht überschrieben wird. Sind alle Einstellungen neu verkettet, können sie wie immer mithilfe des Entitymanagers auf der Datenbank gespeichert werden.

3. Ausblick & Fazit

3.1 Fazit im Team

Die Entwicklung des Projekts begann mit der Erstellung einer Projektstruktur und der Zusammenstellung des Teams. Wir arbeiteten eng zusammen, um die Anforderungen des Projekts zu definieren und die Designkonzepte zu erarbeiten. Wir hielten regelmäßige Meetings ab, um den Fortschritt des Projekts zu überwachen und Probleme schnell zu lösen.

Eine der Herausforderungen bei der Entwicklung des Dashboards war die Anbindung an die Daten aus der Datenbank, zum einen der bereits vorhandenen aber auch die von uns definierten Daten, die für die Nutzerverwaltung und die Visualisierung der Daten auf dem Dashboard essenziell sind. Wir mussten eine robuste und zuverlässige Schnittstelle entwickeln, um die Daten aus der bereitgestellten Datenbank zu sammeln und auf angemessene Weise zu visualisieren. Dies erforderte eine gründliche Planung und Qualitätssicherung, während der Entwicklung, um sicherzustellen, dass die Daten korrekt und komplett auf dem Dashboard angezeigt werden. Durch das bereitgestellte Grundgerüst konnte mit der Entwicklung der Schnittstellen schnell begonnen werden, ohne sich groß in die verwendeten Technologien einarbeiten zu müssen.

Ein weiteres wichtiges Element des Projekts war die Entwicklung einer intuitiven Benutzeroberfläche. Wir haben uns bemüht, die Navigation und die Anzeige der Daten so einfach wie möglich zu gestalten, um sicherzustellen, dass die Nutzer problemlos auf die Informationen zugreifen und diese verstehen können. Wir haben auch sichergestellt, dass die Website für die Verwendung auf verschiedenen Geräten und Bildschirmgrößen optimiert ist.

Die Aufgabestellung des Projekts, bei dem es darum ging, in einer Gruppe eine Website mit einem Dashboard für Solaranlagen zu entwickeln und implementieren, war von Anfang an sehr umfangreich und komplex. Dies hatte zur Folge, dass der zeitliche Umfang des Projekts bereits durch die Aufgabenstellung selbst bestimmt wurde und es keinen Spielraum für unvorhergesehene Probleme gab. Leider stellte sich jedoch heraus, dass wir im Laufe des Projekts mit zahlreichen Herausforderungen und Schwierigkeiten konfrontiert wurden, die uns nicht nur Zeit, sondern auch Nerven gekostet haben.

Eines der größten Probleme, mit denen wir konfrontiert waren, war die bereitgestellte Entwicklungsumgebung. Zum einen gab es Schwierigkeiten bei der Installation und dem Starten des PostgreSQL-Servers, die uns am Anfang des Projekts bereits hinter den Zeitplan brachten. Darüber hinaus hatten wir im Laufe des Projekts immer wieder Probleme mit der gegebenen IDE, die zu zufälligen Abständen dazu führten, dass Deployments nicht mehr gebaut und auf dem lokalen Payara-Server aufgespielt werden konnten. Dies erforderte oft eine komplette Neuinstallation der Software, was uns weitere Zeit und Ressourcen gekostet hat.

Die größten und zeitaufwendigsten Probleme traten jedoch bei der Frontend-Entwicklung mit dem SWAC-Framework auf. Wir stießen während der Entwicklung auf mehrere Hindernisse, die unsere Weiterentwicklung ausgebremst und uns in Zeitnot gebracht haben. Diese Probleme konnten nur durch ein Update des Frameworks behoben werden. Weitere Probleme entstanden durch eine nicht vollständige und fehlerhafte Dokumentation. Diese Probleme konnten meist nur durch Kommunikation mit Herrn Fehring behoben werden, was allerdings durch die Asynchrone Kommunikation durch Emails, immer einen Entwicklungs-Stop bedeutete. Diese Verzögerungen haben uns in eine schwierige Zeitsituation gebracht und uns gezwungen, uns schnell an die veränderten Umstände anzupassen.

Schlussendlich konnten wir, durch den großen Einsatz unserer Teammitglieder und unserer Umfangreichen Planung und Organisation, unsere Meilensteine erreichen und unser Projekt erfolgreich abschließen.

4. Installationshinweise

4.1 Dashboard Frontend

Im Folgenden Abschnitt werden die Installationshinweise des Dashboard Frontend beschrieben.

Das Dashboard enthält folgende JavaScript-Dateien (SmartMonitoringFrontend\js\object\dashboard\):

- (Neu) config.js
- (Neu) dashboard_msg.js
- (Neu) dashboard_globals.js
- (Neu) dashboard_smartdata.js
- (Neu) dashboard_control.js
- (Neu) lang.js
- (Entfernt) dashboard.js

Das Dashboard enthält folgende HTML-Dateien (SmartMonitoringFrontend\sites\object\):

- (Modifiziert) dashboard.html

Die Einhaltung der Reihenfolge der hinzugefügten JavaScript Dateien in der dashboard.html vor dem „</body>“ Tag ist bindend (Reihenfolge der JavaScript Dateien entspricht der Reihenfolge der Aufzählung im darüberliegenden Abschnitt „Das Dashboard enthält folgende JavaScript Dateien“).

Das Dashboard enthält folgende CSS-Dateien (SmartMonitoringFrontend\css\):

- (Modifiziert) dashboard.css

Das Dashboard enthält folgende neue Bilddateien (SmartMonitoringFrontend\img\dashboard\):

- (Neu) bestaetigt.png
- (Neu) load.png
- (Neu) plus.png

Hinzugefügte und editierte SWAC JavaScript Dateien (SWAC\swac\):

- (Neu) components\Visualise\diagrams\CircleProgress.js
- (Neu) components\Visualise\diagrams>StatusDisplay.js
- (Neu) components\Visualise\langs\en.js
- (Neu) components\Charts\libs\chartjs-plugin-zoom.min.js
- (Neu) components\Charts\langs\en.js
- (Neu) components\Charts\plugins\Barchart\langs\en.js
- (Neu) components\Datadescription\langs\en.js
- (Neu) components\Present\langs\en.js
- (Modifiziert) components\User\User.js

4.2 Dashboard Backend

Im Folgenden Abschnitt werden die Installationshinweise des Dashboard Backend beschrieben.

Das Dashboard enthält folgende Java-Dateien:

(SmartMonitoringBackend\src\java\de\fhbielefeld\smartmonitoring\)

- (Neu) jpa\TblCard.java
- (Neu) jpa\TblLocationJoinOo.java
- (Neu) jpa\TblUserCardSettings.java
- (Neu) rest\CardRessource.java
- (Neu) rest\LocationJoinOoResource.java
- (Neu) rest\UserCardSettingsResource.java

Dashboard Datenbank Tabellen PostgreSQL (smartmonitoring_dachanlage\smartmonitoring\):

- (Neu) tbl_card
- (Neu) tbl_card_join_oo
- (Neu) tbl_user_card_settings
- (Bereits vorhanden) alle Tabellen mit dem Präfix: modul_
- (Bereits vorhanden) irradiation_1
- (Bereits vorhanden) tbl_observedobject

Die Erstellung der neuen Tabellen und Spalten erfolgen mit dem Start des Payara Web Servers automatisch.

Sollte die Erstellung fehlschlagen sind hier im Einzelnen die Spalten und Eigenschaften jeder neuen Tabelle im Detail:

tbl_card					
Spalte	Datentyp	Identität	Kollation	Nicht Null	Standard
id	bigserial	-	-	TRUE	nextval('smartmonitoring.tbl_card_id_seq'::regclass)
available	bool	-	-	TRUE	-
description	varchar(255)	-	default	-	-
max_span_beginn	timestamp	-	-	TRUE	-
max_span_end	timestamp	-	-	TRUE	-
tesource	varchar(255)	-	default	-	-
title	varchar(255)	-	default	TRUE	-
total	bool	-	-	TRUE	-
type	varchar(255)	-	default	TRUE	-

tbl_user_card_settings					
Spalte	Datentyp	Identität	Kollation	Nicht Null	Standard
id	bigserial	-	-	TRUE	nextval('smartmonitoring.tbl_user_card_settings_id_seq'::regclass)
dpan_begin	timestamp	-	-	TRUE	-
dpan_end	timestamp	-	-	TRUE	-
user_id	int8	-	-	TRUE	-
visible	bool	-	-	TRUE	-
card_id	int8	-	-	TRUE	-
next_card	int8	-	-	-	-
previous_card	int8	-	-	-	-

tbl_card_join_oo					
Spalte	Datentyp	Identität	Kollation	Nicht Null	Standard
card_id	int8	-	-	TRUE	-
observedobject_id	int8	-	-	TRUE	-

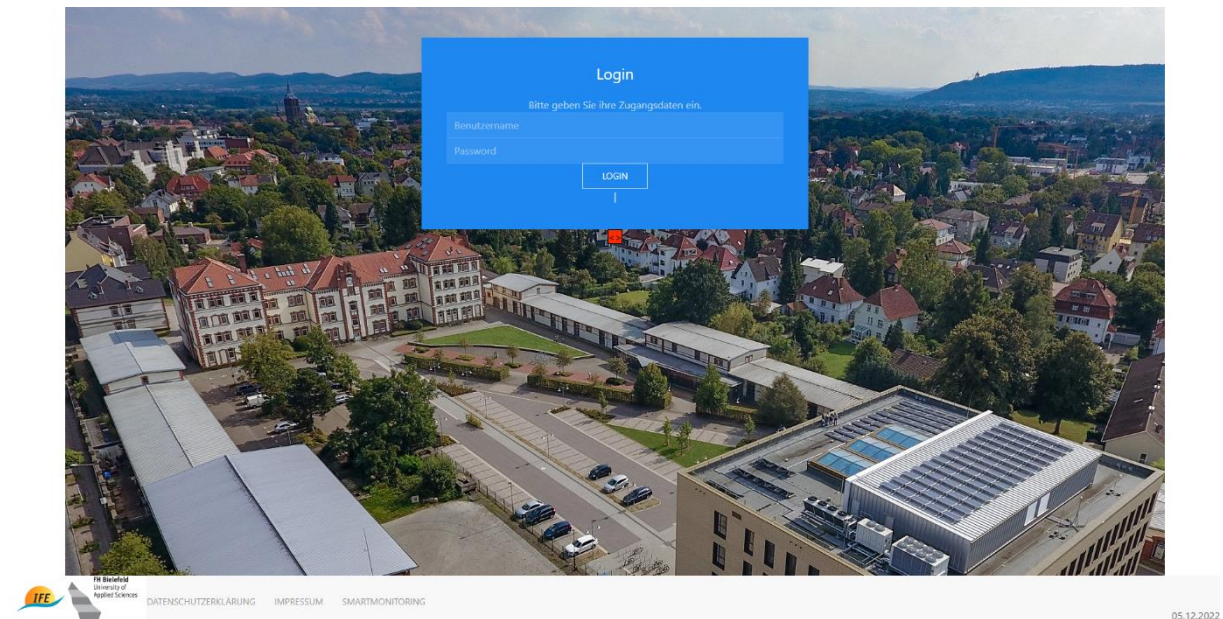
Zusätzliche Zeilen in der Tabelle **tbl_observedobject** für die Wetter und Einstrahlungsstärke Karte:

Wetter (Minden) / Einstrahlungsstärke		
Spalte	Wetter	Einstrahlungsstärke
id	Automatisch	Automatisch
datacapture	FALSE	FALSE
description	NULL	NULL
manualcapture	NULL	NULL
name	Wetter	Einstrahlungsstärke
parent	5	5
type_id	NULL	NULL
icon	NULL	NULL
collection	minden_weather	irradiation_1
parent_id	NULL	NULL

5. Benutzerhandbuch

5.1 Login

Um das Dashboard benutzen zu können müssen Sie sich zunächst anmelden, dazu rufen Sie die Webseite <https://scl.fh-bielefeld.de/SmartMonitoringDachanlage/sites/login.html> auf und geben Ihre Anmeldedaten in die Eingabefelder „**Benutzername**“ und „**Passwort**“ ein, klicken Sie auf den darunter liegenden Button „**Login**“, Sie werden daraufhin automatisch zu der Dashboard Seite weitergeleitet.



(Bild 1: [Für eine größere Darstellung klicken Sie hier.](#))

5.2 Dashboard Karte erstellen (Admin)

Nachdem Sie sich erfolgreich auf der Login Seite angemeldet haben können Sie die Karten, die von anderen Admins erstellt wurden direkt sehen (im Fall von Bild 2 sind noch keine Karten vorhanden). Um eine neue Karte anzulegen, Klicken Sie auf das „**Plus Symbol**“ in der Karte „**Neue Karte**“.

Die Karte „**Neue Karte**“ steht nur Admins zur Verfügung, normale Benutzer haben diese Einstellung nicht.

Dashboard options

Manage dashboard options.

ADMIN

DATUM VON

tt.mm.jjjj

DATUM BIS

tt.mm.jjjj

SPEICHERN

Neue Karte

+

1

(Bild 2: [Für eine größere Darstellung klicken Sie hier.](#))

(Bild 3)

Es öffnet sich ein Einstellungsfenster mit mehreren Eingabe-/Auswahlmöglichkeiten:

- Karten Visualisierung
- Karten Datenquelle
- Karten Titel
- Karten Beschreibung
- Karten Datum von
- Karten Datum bis
- Ist für Benutzer verfügbar
- Kann von Benutzern de/aktiviert werden

(Bild 3, Punkt 1)

Wählen Sie eine Visualisierung für die Karte aus, Ihnen stehen folgende Optionen zur Verfügung:

- Wetter Anzeige
- Temperatur Anzeige
- Kreis Diagramm
- Balken Diagramm (Statusanzeige)
- Balken Diagramm (Datenpunkte)

Für weitere Informationen über die einzelnen Karten Visualisierungen schauen Sie sich den Punkt „[Karten allgemein](#)“ an.

(Bild 3, Punkt 2)

Nachdem Sie eine Visualisierung ausgewählt haben werden die beiden Felder darunter freigegeben, im Eingabefeld können Sie Ihren Suchtext eingeben, dadurch erhalten Sie in der Auswahlbox nur Auswahlmöglichkeiten, die mit Ihrem Suchtext übereinstimmen.

(Bild 3, Punkt 3)

Geben Sie in diesem Eingabefeld Ihren Titel für die Karte ein, der Titel erscheint bei der neu erstellten Karte als Überschrift.

(Bild 3, Punkt 4)

Geben Sie in diesem Eingabefeld eine Beschreibung der Karte ein, dieser Text wird unterhalb der Karten Visualisierung angezeigt.

(Bild 3, Punkt 5)

Geben Sie in diesem Eingabefeld ein Begin Datum ein, ab der die Karte Daten ermitteln soll, ein Tag beginnt immer um 00:00 Uhr.

(Bild 3, Punkt 6)

Geben Sie in diesem Eingabefeld ein Ende Datum ein, die Karte ermittelt die Daten vom Begin Datum bis Ende Datum, das Ende Datum bzw. Tag zählt immer bis 23:59 Uhr.

(Bild 3, Punkt 5 und 6)

Bei der Erstellung einer neuen Karte wird das Begin/Ende Datum automatisch auch für die individuelle Benutzereinstellung übernommen, Admins können für sich nachträglich in den Karten [Einstellungen](#) ein eigenes Datum festlegen, das nur sie selbst sehen, für die normalen Dashboard Benutzer gilt der Globale Zeitraum der Karte.

(Bild 3, Punkt 7)

Ist die Checkbox „**Ist für Benutzer verfügbar**“ aktiv können auch normale Dashboard Benutzer die Karte sehen, ist diese Option deaktiviert sehen nur Admins diese Karte.

(Bild 3, Punkt 8)

Ist die Checkbox „**Kann von Benutzern de/aktiviert werden**“ aktiv können normale Dashboard Benutzer diese Karte nicht mehr ausblenden (Admins ausgeschlossen).

The image shows a web form titled "Neue Kachel" (New Widget) with several input fields and checkboxes. Red boxes and numbers 1 through 8 highlight specific parts of the form:

- 1: "KACHEL VISUALISIERUNG:" dropdown menu with "Auswählen" as the placeholder.
- 2: "KACHEL DATENQUELLE:" dropdown menu with "Auswählen" as the placeholder.
- 3: "TITEL:" text input field.
- 4: "BESCHREIBUNG:" text input field.
- 5: "DATUM VON:" date input field with placeholder "tt.mm.jjjj" and a calendar icon.
- 6: "DATUM BIS:" date input field with placeholder "tt.mm.jjjj" and a calendar icon.
- 7: Checkbox "Ist für Benutzer Verfügbar" (checked).
- 8: Checkbox "Kann von Benutzern de/aktiviert werden" (checked).

At the bottom of the form are two buttons: "SPEICHERN" (Save) and "ABBRECHEN" (Cancel).

(Bild 3: [Für eine größere Darstellung klicken Sie hier.](#))

5.3 Karten minimieren

(Bild 4, Punkt 1)

Minimieren Sie die Karten in dem Sie auf den „-“ Button klicken, es erscheint automatisch ein „Lesezeichen“ oberhalb der Karten.

(Bild 4, Punkt 2)

Expandieren Sie die Karte in dem Sie auf das entsprechende „Lesezeichen“ klicken, die Karte erscheint wieder an ihrem ursprünglichen Platz.

Dashboard Einstellungen

Hier können Sie die für das Dashboard verfügbaren Elemente konfigurieren.

(Bild 4: [Für eine größere Darstellung klicken Sie hier.](#))

5.4 Karten Konfiguration (Admin)

In diesem Abschnitt erfahren Sie wie Sie eine bestehende Karte verändern, alle Optionen bis auf die Daten Eingabe können nur von Admins erfolgen, normale Benutzer sehen nur die Daten Eingabefelder.

(Bild 5, Punkt 1)

Um eine bestehende Karte zu verändern klicken Sie auf den Button „Konfiguration“, es öffnet sich ein Einstellungsfenster in dem Sie folgende Eigenschaften verändern können:

- Karten Titel
- Karten Beschreibung
- Karten Datum von
- Karten Datum bis
- Ist für Benutzer verfügbar
- Kann von Benutzern de/aktiviert werden
- Karte Entfernen

(Bild 5, Punkt 2)

Diese Eigenschaften sind unveränderlich, wenn Sie eine Karte mit einer anderen Darstellung oder für eine andere Daten Quelle haben möchten erstellen Sie eine neue Karte.

(Bild 5, Punkt 3)

Geben Sie in diesem Eingabefeld Ihren neuen Titel für die Karte ein, der Titel erscheint nach dem speichern der Einstellung als Karten Überschrift.

(Bild 5, Punkt 4)

Geben Sie in diesem Eingabefeld eine neue Beschreibung der Karte ein, dieser Text wird unterhalb der Karten Visualisierung angezeigt.

(Bild 5, Punkt 5)

Geben Sie in diesem Eingabefeld ein neues Begin Datum ein, ab der die Karte Daten ermitteln soll, ein Tag beginnt immer um 00:00 Uhr.

(Bild 5, Punkt 6)

Geben Sie in diesem Eingabefeld ein neues Ende Datum ein, die Karte ermittelt die Daten vom Begin Datum bis Ende Datum, das Ende Datum bzw. Tag zählt immer bis 23:59 Uhr.

(Bild 5, Punkt 5 und 6)

Diese Einstellung verändert **nicht** den Globalen Zeitraum für die normalen Benutzer! Um den Globalen Zeitraum zu verändern, benutzen Sie bitte die Funktion unter „[Globaler Zeitraum \(Admin\)](#)“. Beide Daten Eingabefelder verändern nur die eigene Zeiteinstellung (Zeiteinstellung des angemeldeten Nutzers).

(Bild 5, Punkt 7)

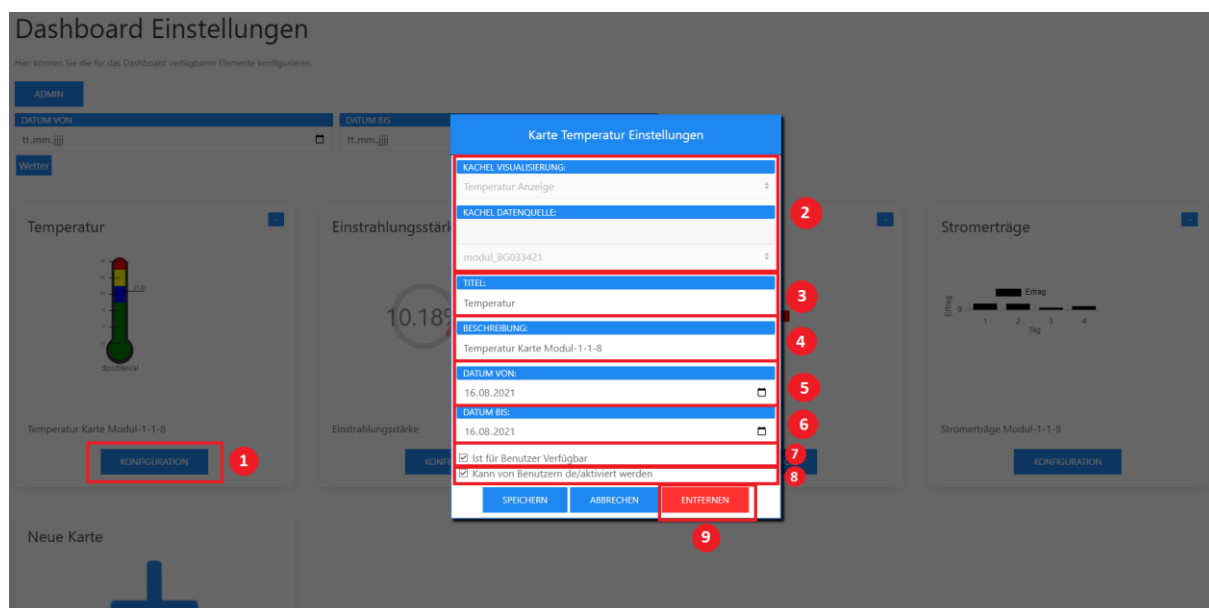
Ist die Checkbox „**Ist für Benutzer verfügbar**“ aktiv können auch normale Dashboard Benutzer die Karte sehen, ist diese Option deaktiviert sehen nur Admins diese Karte. Wenn Sie diese Option aktivieren oder deaktivieren und speichern wird eine Repositionierung aller Karten Einstellungen vorgenommen, dies ist notwendig, um die Reihenfolgen konsistent zu halten. Die veränderte Karte wird als letzte Karte angezeigt.

(Bild 5, Punkt 8)

Ist die Checkbox „**Kann von Benutzern de/aktiviert werden**“ aktiv können normale Dashboard Benutzer diese Karte nicht mehr ausblenden (Admins ausgeschlossen). Hat ein normaler Benutzer die Karte vor der Einstellung schon ausgeblendet wird die Karte trotzdem „maximiert“ angezeigt, sollten Sie die Option wieder zurück ändern ist die Karte bei dem Benutzer wieder „minimiert“ (diese Option überschreibt nicht die bereits Gesetzten Karten Einstellungen).

(Bild 5, Punkt 9)

Der Button „**Entfernen**“ entfernt die Karte für **alle** Benutzer, die Repositionierung der übrigen Karten erfolgt automatisch.



(Bild 5: [Für eine größere Darstellung klicken Sie hier.](#))

5.5 Karten Konfiguration (Benutzer)

In diesem Abschnitt erfahren Sie wie Sie eine bestehende Karte verändern.

(Bild 6, Punkt 1)

Um eine bestehende Karte zu verändern klicken Sie auf den Button „**Konfiguration**“, es öffnet sich ein Einstellungsfenster in dem Sie folgende Eigenschaften verändern können:

- Karten Datum von
- Karten Datum bis

(Bild 6, Punkt 2)

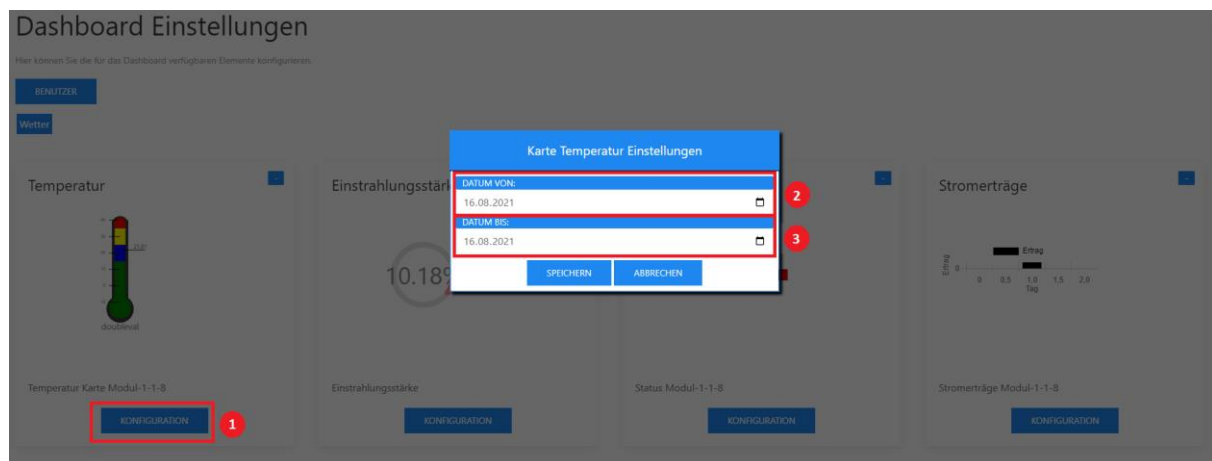
Geben Sie in diesem Eingabefeld ein neues Begin Datum ein, ab der die Karte Daten ermitteln soll, ein Tag beginnt immer um 00:00 Uhr.

(Bild 6, Punkt 3)

Geben Sie in diesem Eingabefeld ein neues Ende Datum ein, die Karte ermittelt die Daten vom Begin Datum bis Ende Datum, das Ende Datum bzw. Tag zählt immer bis 23:59 Uhr.

(Bild 6, Punkt 2 und 3)

Sie können sich nur innerhalb des vom Admin festgelegten Zeitraums der Karte bewegen, sollte Ihre neue Eingabe ungültig oder den festgelegten Zeitraum des Admins verlassen erscheint eine Fehlermeldung, in der Sie sehen können, welche Daten Reichweite Sie nicht über-/unterschreiten dürfen.



(Bild 6: [Für eine größere Darstellung klicken Sie hier.](#))

5.6 Globaler Zeitraum (Admin)

Hier können Sie die Zeitachse **aller** vorhandenen Karten ändern, die individuellen Admin/Benutzer Einstellungen jeder Karte bleiben zunächst unberührt. Admins können sich immer außerhalb der Globalen Karten Zeitachse bewegen, normale Benutzer können nur Daten zwischen den gesetzten Zeiträumen wählen.

(Bild 7, Punkt 1)

Setzen Sie ein Globales Begin Datum in das Eingabefeld (Eingabe/Auswahl).

(Bild 7, Punkt 2)

Setzen Sie ein Globales Ende Datum in das Eingabefeld (Eingabe/Auswahl).

(Bild 7, Punkt 3)

Speichern Sie Ihre neue Zeitachse, ungültige Eingaben z.B. Begin Datum ist größer als Ende Datum sind nicht möglich.

Dashboard Einstellungen

Hier können Sie die für das Dashboard verfügbaren Elemente konfigurieren.

(Bild 7: [Für eine größere Darstellung klicken Sie hier.](#))

5.7 Globaler Zeitraum (Benutzer)

Hier können Sie die Zeitachse **aller** vorhandenen Karten ändern. Es werden nur Ihre individuellen Zeitangaben verändert. Sind für die einzelnen Karten unterschiedliche Globale Zeitachsen von einem Admin gesetzt wird diese Option den Benutzern **nicht** angeboten.

(Bild 8, Punkt 1)

Setzen Sie ein Globales Begin Datum in das Eingabefeld (Eingabe/Auswahl).

(Bild 8, Punkt 2)

Setzen Sie ein Globales Ende Datum in das Eingabefeld (Eingabe/Auswahl).

(Bild 8, Punkt 3)

Speichern Sie Ihre neue Zeitachse, ungültige Eingaben z.B. Begin Datum ist größer als Ende Datum sowie Angaben außerhalb des Globalen Zeitraums der Karten sind nicht möglich.

Dashboard Einstellungen

Hier können Sie die für das Dashboard verfügbaren Elemente konfigurieren.

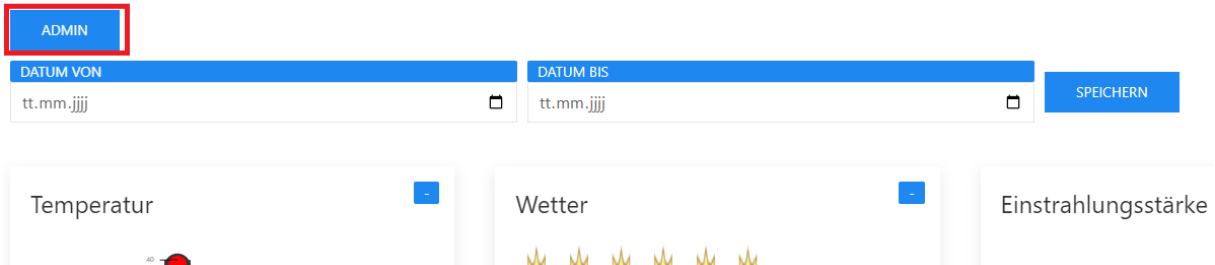
(Bild 8: [Für eine größere Darstellung klicken Sie hier.](#))

5.8 Ansichten Wechsel Admin/Benutzer

Mit dieser Option können Sie zwischen der Admin und Benutzer Ansicht wechseln, dieses Feature wird später wieder entfernt (es existieren noch keine Unterscheidungsmerkmale zwischen Admin und Benutzer => DB & Login).

Dashboard Einstellungen

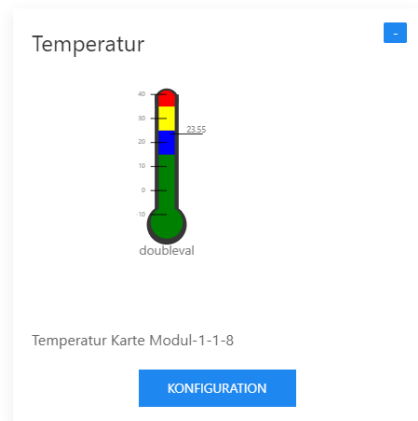
Hier können Sie die für das Dashboard verfügbaren Elemente konfigurieren.



(Bild 9: [Für eine größere Darstellung klicken Sie hier.](#))

5.9 Karten allgemein

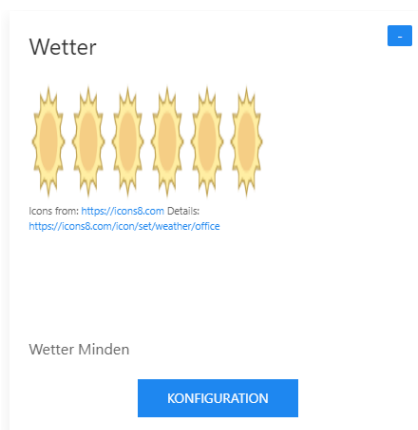
Hier erfahren Sie wie die einzelnen „Visualisierungen“ der „Karten“ aufgebaut sind und was die Angezeigten Werte bedeuten. Alle Karten zeigen Daten ab dem gesetzten Begin Datum 00:00 Uhr bis zum gesetzten Ende Datum 23:59 Uhr an.



Temperatur Karte

Die Temperatur Karte zeigt die durchschnittliche Temperatur eines Solar Panels im gewählten Zeitraum an. Alle Messergebnisse, die sich in diesem Zeitraum befinden werden addiert und durch die Summe der Anzahl der Messungen geteilt.

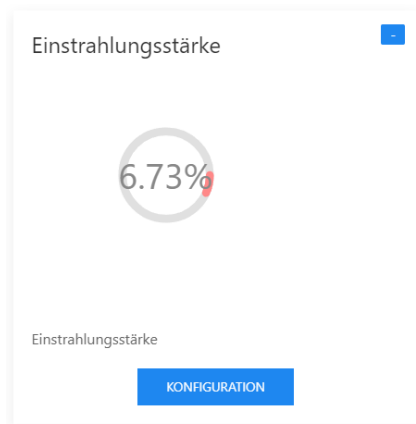
(Bild 10: [Für eine größere Darstellung klicken Sie hier.](#))



Wetter Karte

Die Wetter Karte zeigt das durchschnittliche Wetter des ausgewählten Zeitraums an. Alle Messergebnisse, die sich in diesem Zeitraum befinden werden addiert und durch die Summe der Anzahl der Messungen geteilt.

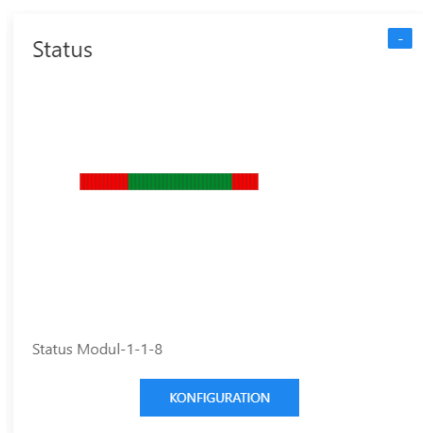
(Bild 11: [Für eine größere Darstellung klicken Sie hier.](#))



Kreisdiagramm Karte

Die Kreisdiagramm Karte zeigt die durchschnittliche Einstrahlungsstärke/Stromertrag eines Solar Panels im gewählten Zeitraum an. Alle Messergebnisse, die sich in diesem Zeitraum befinden werden addiert und durch die Summe der Anzahl der Messungen geteilt. Für Einstrahlungsstärke und Stromertrag sind feste Werte definiert (Einstrahlungsstärke = 1089,4 || Stromertrag = 46).

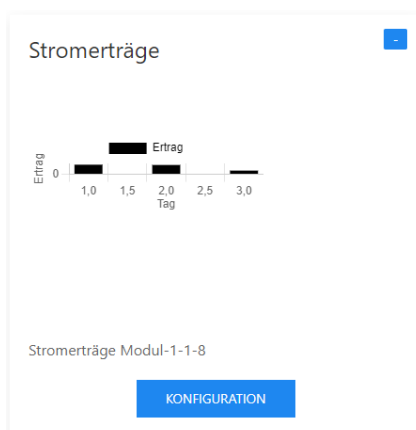
(Bild 12: [Für eine größere Darstellung klicken Sie hier.](#))



Status Anzeige Karte

Die Status Anzeige Karte zeigt den Status eines Solar Panels im gewählten Zeitraum an. Die rot/grünen Werte in der Darstellung richten sich nach der gemessenen Spannung. Alle Werte die gleich Null oder kleiner sind gelten als Ausfall/Nachtzyklus und werden rot dargestellt, alle Positiven Werte sind grün dargestellt.

(Bild 13: [Für eine größere Darstellung klicken Sie hier.](#))



Balkendiagramm Karte

Die Balkendiagramm Karte zeigt die Stromerträge in KW eines Solar Panels im gewählten Zeitraum an. Jeder Tag im gewählten Zeitraum wird als eigener Stromertrags Balken angezeigt. Alle Daten eines Tages (U = Spannung in Volt und I = Strom in Ampere) werden zu einem Tages Durchschnittswert addiert und durch die Summe der Anzahl an Messungen geteilt. Zum Abschluss wird der Durchschnittswert von Spannung und Strom zum Errechnen der Leistung benutzt ($P = (U \cdot I) / 1000$).

(Bild 14: [Für eine größere Darstellung klicken Sie hier.](#))

5.10 Sprache

Das Dashboard unterstützt die Sprachen Englisch und Deutsch. Die Datenfelder, die aus der Datenbank stammen werden **nicht** übersetzt (Titel und Beschreibung der Karte).