

Weather App "weatherFLOSScast"

WP-Seminar "Quelloffene Software in der modernen Informatik" Projekt

Unter Beaufsichtigung von Dr. Chris Zimmermann

Frankfurt University of Applied Sciences

Brychcy, Patryk
`patryk.brychcy@stud.fra-uas.de`

Fluegel, Dwipa
`dwipa.fluegel@stud.fra-uas.de`

Karaman, Deniz
`deniz.karaman@stud.fra-uas.de`

Zimmermann, Alwin
`alwin.zimmermann@stud.fra-uas.de`

Filatoff, Michael
`michael.filatoff@stud.fra-uas.de`

Gavrilov, Sascha
`sascha.gavrilov@stud.fra-uas.de`

Lepore, Dominik
`dominik.lepore@stud.fra-uas.de`

Zundel, Benedikt
`benedikt.zundel@stud.fra-uas.de`

Friday, 2025-06-20

<https://github.com/bzundel/weather-flosscast>

Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Projektmanagement mit Scrum & GitHub Projects (Deniz Karaman) | 5 |
| 1.1 | Darstellung Boards | 5 |
| 2 | Product ownership (Alwin Zimmermann) | 6 |
| 2.1 | Aufgaben des Product Owners | 6 |
| 2.2 | Probleme | 6 |
| 2.3 | Ziel der Entwicklung | 6 |
| 2.4 | Requirements | 7 |
| 2.5 | Kernfunktionen (Hohe Priorität) | 7 |
| 2.5.1 | Automatisches Anzeigen des aktuellen Wetters am Standort (US-001) | 7 |
| 2.5.2 | Städtesuche und -verwaltung (US-002, US-005) | 7 |
| 2.5.3 | Wettervorhersage für zukünftige Tage (US-003, US-004) | 7 |
| 2.6 | Offline-Funktionalität (Mittlere Priorität) | 7 |
| 2.6.1 | Offline-Zugriff auf das letzte Wetter am Standort (US-005) | 7 |
| 2.6.2 | Offline-Zugriff auf das Wetter gespeicherter Städte (US-006) | 7 |
| 2.6.3 | Fehlermeldung bei fehlender Internetverbindung (US-007) | 7 |
| 2.7 | App-Informationen und Hilfe (Niedrige Priorität) | 8 |
| 2.7.1 | Hilfefunktion und Impressum (US-008) | 8 |
| 2.8 | User-Stories | 8 |
| 2.8.1 | Übersicht der User Stories | 8 |
| 2.9 | Detaillierte User Story Beschreibungen | 8 |
| 2.9.1 | User Story: Als Nutzer möchte ich beim Öffnen der App das aktuelle Wetter für meinen Standort sehen. | 8 |
| 2.9.2 | User Story: Als Nutzer möchte ich nach Städten suchen und sie einer Liste hinzufügen können. | 10 |
| 2.9.3 | User Story: Als Nutzer möchte ich das Wetter der zukünftigen Tage an meinem Standort sehen. | 11 |
| 2.9.4 | User Story: Als Nutzer möchte ich das zukünftige Wetter von gespeicherten Städten sehen. | 12 |
| 2.9.5 | User Story: Als Nutzer möchte ich Städte aus meiner Liste löschen können. | 13 |
| 2.9.6 | User Story: Als Nutzer möchte ich auch ohne Internet das letzte aktuelle Wetter an meinem Standort sehen. | 14 |
| 2.9.7 | User Story: Als Nutzer möchte ich auch das Wetter aus zuvor hinzugefügten Städten sehen ohne Internet zu haben. | 15 |
| 2.9.8 | User Story: Als Nutzer möchte ich eine Fehlermeldung sehen, wenn ich kein Internet habe. | 16 |
| 2.9.9 | User Story: Als Nutzer möchte ich eine Hilfefunktion und ein Impressum in der App finden. | 17 |
| 3 | Design (Michael Filatoff) | 17 |
| 3.1 | Ablauf | 17 |
| 3.2 | Software Vorstellung | 17 |
| 3.3 | Planung | 19 |
| 3.4 | Aufbau und Layout | 19 |
| 3.4.1 | Wetter Liste | 19 |
| 3.4.2 | Wetter Vorhersage | 20 |
| 4 | Technische Dokumentation | 21 |
| 4.1 | Frontend (Patrik Brychcy, Sascha Gavrilov) | 21 |
| 4.1.1 | Einleitung | 21 |
| 4.1.2 | Architekturübersicht | 21 |
| 4.1.3 | MainActivity | 22 |
| 4.1.4 | SearchScreen | 22 |
| 4.1.5 | WeatherScreen | 22 |

| | | |
|-------|--|----|
| 4.1.6 | Utils.kt | 23 |
| 4.1.7 | CityList.kt | 23 |
| 4.1.8 | Zusammenfassung | 23 |
| 4.2 | Datenvorbereitung (Benedikt Zundel) | 23 |
| 4.3 | Caching (Benedikt Zundel) | 24 |
| 4.4 | Testing (Dwipa Flügel, Dominik Lepore) | 25 |
| 4.4.1 | Einleitung | 25 |
| 4.5 | Testplan | 25 |
| 4.5.1 | Probleme | 26 |
| 4.5.2 | Verbesserungen | 27 |

Abbildungsverzeichnis

| | | |
|---|--|----|
| 1 | Screenshot der GitHub Projects Ansicht des Backlogs | 5 |
| 2 | Screenshot der GitHub Projects Ansicht der Quality Assurance | 5 |
| 3 | Screenshot der GitHub Ansicht einer Story | 6 |
| 4 | Figma linke Ansicht | 18 |
| 5 | Figma rechte Ansicht | 19 |
| 6 | Wetter Liste | 20 |
| 7 | Wetter Vorhersagen | 21 |
| 8 | Bild der Google Testing Pyramide | 27 |

Listings

| | | |
|---|---------------------------------------|----|
| 1 | Forecast Datenstruktur | 23 |
| 2 | Auszug des Parsingprozesses | 24 |

1 Projektmanagement mit Scrum & GitHub Projects (Deniz Karaman)

Um unseren Entwicklungsprozess agil zu strukturieren, haben wir Scrum mithilfe von GitHub Projects implementiert. Dabei haben wir uns an klassischen Scrum-Prinzipien orientiert und diese wie üblich an unsere Teamgröße und die Projektanforderungen angepasst. Jede Story wird dabei als Issue angelegt.

Zunächst haben wir in unseren Issues sowohl Labels als auch Milestones definiert, um einen schnellen Überblick zu gewährleisten und einzelne Issues entsprechend ihrer Zuordnung hervorzuheben. Da wir unsere Stories im Team abgestimmt und deren Komplexität berücksichtigt haben, gelten die Definition of Done (DoD) sowie die Bewertung als feststehend.

Folgende Labels haben wir definiert: *backend*, *bug*, *documentation*, *enhancement*, *epic*, *frontend*, *story*, *testing*. Zusätzlich zu unseren Labels benötigen wir unterschiedliche Status, für die wir folgende verwenden: *Todo*, *In Progress*, *Needs Review*, *Done*, *Canceled*.

Durch die Nutzung von Custom Fields vom Typ Iteration können wir unsere Sprints zeitlich organisieren und in wöchentlichen Zyklen abbilden. Bei der Sprintplanung in unseren Meetings haben wir zudem die Möglichkeit, sowohl die Story Points als auch die Priorität (*Low*, *Medium*, *High*, *Critical*) über Custom Fields abzubilden.

Nach Abschluss jedes Sprints haben wir dienstags abends wöchentlich erst eine Retrospektive und anschließend die Sprint-Planung abgehalten. Da nicht immer alle teilnehmen konnten, haben wir unsere Entwicklung in den User Stories protokolliert, um sie für alle nachvollziehbar zu machen.

1.1 Darstellung Boards

Abbildung 1: Screenshot der GitHub Projects Ansicht des Backlogs

My Assignments

Backlog

Current Sprint





Next Sprint

Quality Assurance

+ New view

Q

-status:Done -status:Canceled -status:Rejected -status:"Needs review"

| Title | ... | Assignees | ... | Status | ... | Priority | ... |
|--|-----|--|-----|-------------|-----|----------|-----|
| 1 Als Nutzer möchte ich das aktuelle Wetter einer festgelegten Stadt sehen #2 | |  Patss2 and sascha... | | Todo | | High | |
| 2 Als Nutzer moechte ich beim Klicken der Vorhersagen mehr Informationen angezeigt beko... #9 | |  Patss2 and sascha... | | Todo | | High | |
| 3 Als Nutzer Möchte ich beim Öffnen der App das Wetter an meinem Aktuellen Standort seh... #11 | |  saschaG2000 | | | | High | |
| 4 Als Tester möchte ich Frontend und Backend Funktionalitäten validieren #7 | |  Dwippah and sged... | | In Progress | | Medium | |

Um die Übersichtlichkeit weiter zu erhöhen, haben wir für jede Board-Ansicht vordefinierte Filter konfiguriert.

So erhält jede Ansicht genau die Issues, die für den jeweiligen Arbeitsschritt relevant sind, und das Team behält jederzeit den passenden Fokus.

Abbildung 2: Screenshot der GitHub Projects Ansicht der Quality Assurance

My Assignments

Backlog

Current Sprint

Next Sprint

Quality Assurance

+ New view

Q

-assignee:@me status:"Needs review"

| Title | ... | Assignees | ... | Status | ... |
|--|-----|-----------|-----|--------------|-----|
| 1 Als Designer möchte ich ein Mockup in Figma erstellen #8 | | Nennmicha | | Needs review | |

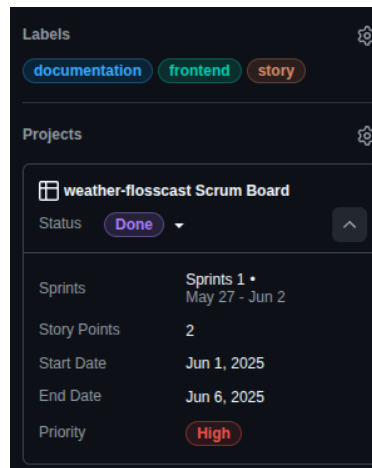
- **Backlog:** Alle geplanten User Stories und Tasks
- **Current Sprint:** Die aktuell bearbeiteten Items
- **Next Sprint:** Vorgemerkte Stories für den nächsten Sprint
- **Quality Assurance:** Abgenommene Features in der Testphase

- **Done:** Abgeschlossene Stories

Jeder Eintrag auf dem Board ist als GitHub Issue angelegt und hat folgende Felder

1. Story Points (zur Aufwandsschätzung in Fibonacci)
2. Priority-Labels (*Low, Medium, High, Critical*)
3. Sprint-Labels (z. B. *Sprint1, Sprint2, etc.*)
4. Status-Labels (*Todo, In Progress, Needs review, Done, Canceled*)

Abbildung 3: Screenshot der GitHub Ansicht einer Story



Durch diese enge Verzahnung von Issue-Tracking, Code-Repositories und agilen Meetings erreichen wir eine hohe Transparenz, schnelle Feedback-Zyklen und eine stetige Verbesserung unseres Workflows.

2 Product ownership (Alwin Zimmermann)¹

2.1 Aufgaben des Product Owners

In der Rolle des Product Owners war es erforderlich, eine präzise Definition des Projekts zu formulieren. Von entscheidender Bedeutung war die Erstellung von Anforderungen sowie die Ausarbeitung der verschiedenen User-Stories. Das Produkt muss den Erwartungen der Stakeholder entsprechen. Die Pflege des Backlogs zählt ebenfalls zu den Aufgaben des Product Owners, um den Fortschritt des Projekts zu dokumentieren.

2.2 Probleme

Eine der größten Herausforderungen im Team bestand in der Organisation regelmäßiger Besprechungen. Obwohl unsere Stundenpläne und täglichen Abläufe variieren, haben wir es geschafft, uns in regelmäßigen Abständen zuverlässig zu treffen. Der Austausch mit dem Scrum Master erwies sich als äußerst nützlich, ebenso wie die Kooperation innerhalb des gesamten Entwicklerteams.

2.3 Ziel der Entwicklung

Das Ziel des Projekts bestand darin, eine Wetteranwendung für das Android-Betriebssystem zu entwickeln. Die Anforderungen bestanden darin, dass die Anwendung bestimmte Vorgaben erfüllt und den Erwartungen der Stakeholder gerecht wird. Im Folgenden werde ich auf die von uns erstellten Requirements eingehen und die daraus resultierenden User-Stories erläutern.

¹Für die Bearbeitung meiner Aufgabenteile wurde die Unterstützung von Gemini, einem KI-gestützten Sprachmodell von Google, sowie dem Duden Mentor (einschließlich seines KI-Assistenten für Umformulierungen) in Anspruch genommen. Die finale Prüfung und Bearbeitung sämtlicher Inhalte erfolgte eigenverantwortlich.

2.4 Requirements

In diesem Teil gehe ich auf die Requirements ein. Diese beziehen sich auf die unten stehenden User-Stories. Die Kern-Requirements dienen als erste Anhaltspunkte, was der Product Owner verlangt und welche Kern-funktionalitäten wichtig sind.

2.5 Kernfunktionen (Hohe Priorität)

2.5.1 Automatisches Anzeigen des aktuellen Wetters am Standort (US-001)

- Beim Start der App wird das aktuelle Wetter für den geografischen Standort des Nutzers automatisch und ohne manuelle Interaktion angezeigt.
- Ziel ist es, dem Nutzer sofortige und präzise Informationen über seine unmittelbare Umgebung zu liefern.

2.5.2 Städt suche und -verwaltung (US-002, US-005)

- Nutzer müssen in der Lage sein, Städte zu suchen (z.B. über eine Suchleiste).
- Gefundene Städte können einer personalisierten Favoritenliste hinzugefügt werden.
- Das Wetter der in der Liste gespeicherten Städte muss abrufbar sein.
- Nutzer müssen die Möglichkeit haben, Städte aus ihrer Favoritenliste zu entfernen, um die Ansicht zu optimieren.

2.5.3 Wettervorhersage für zukünftige Tage (US-003, US-004)

- Die App muss eine Wettervorhersage für die kommenden Tage am aktuellen Standort des Nutzers anzeigen können.
- Die App muss eine Wettervorhersage für die kommenden Tage der in der Favoritenliste gespeicherten Städte anzeigen können.
- Ziel ist es, den Nutzer umfassend über zukünftige Wettersituationen zu informieren.

2.6 Offline-Funktionalität (Mittlere Priorität)

2.6.1 Offline-Zugriff auf das letzte Wetter am Standort (US-005)

- Wenn keine Internetverbindung besteht, soll die App die zuletzt abgerufenen Wetterdaten für den aktuellen Standort anzeigen.
- Diese Funktion stellt sicher, dass der Nutzer auch ohne Internetverbindung grundlegende Informationen erhält.

2.6.2 Offline-Zugriff auf das Wetter gespeicherter Städte (US-006)

- Die App muss die zuletzt abgerufenen Wetterdaten für die in der Favoritenliste gespeicherten Städte auch im Offline-Modus anzeigen können.
- Die angezeigten Offline-Daten sollten nicht älter als das Ende des letzten verfügbaren Forecasts sein, um eine gewisse Relevanz zu gewährleisten.

2.6.3 Fehlermeldung bei fehlender Internetverbindung (US-007)

- Bei fehlender oder unterbrochener Internetverbindung muss eine eindeutige Fehlermeldung angezeigt werden, die den Nutzer über den Offline-Status informiert.

2.7 App-Informationen und Hilfe (Niedrige Priorität)

2.7.1 Helfefunktion und Impressum (US-008)

- Die App soll eine Helfefunktion bereitstellen, die dem Nutzer die Bedienung der App erklärt.
- Dieser Bereich muss auch ein Impressum enthalten, das Informationen über den Entwickler und die Lizenz der App bereitstellt.

2.8 User-Stories

In diesem teil sammle und beschreibe ich die User Stories für das Projekt **Weather-flosscast**. Jede User Story repräsentiert eine Anforderung aus der Perspektive eines Benutzers und dient als Grundlage für die Entwicklung und Schätzung.

2.8.1 Übersicht der User Stories

Um eine klare Übersicht zu gewährleisten, sind die wichtigsten Metadaten der User Stories in der folgenden Tabelle zusammengefasst. Detaillierte Beschreibungen und Akzeptanzkriterien finden sich in den nachfolgenden Abschnitten.

| Beschreibung | US | Priorität |
|---|--------|-----------|
| Als Nutzer möchte ich beim Öffnen der App das aktuelle Wetter für meinen Standort sehen. | US-001 | Hoch |
| Als Nutzer möchte ich nach Städten suchen und sie einer Liste hinzufügen können. | US-002 | Hoch |
| Als Nutzer möchte ich das Wetter der zukünftigen Tage an meinem Standort sehen. | US-003 | Hoch |
| Als Nutzer möchte ich das zukünftige Wetter von gespeicherten Städten sehen. | US-004 | Hoch |
| Als Nutzer möchte ich Städte aus meiner Liste löschen können. | US-005 | Hoch |
| Als Nutzer möchte ich auch ohne Internet das letzte aktuelle Wetter an meinem Standort sehen. | US-006 | Mittel |
| Als Nutzer möchte ich auch ohne Internet das Wetter aus zuvor hinzugefügten Städten sehen. | US-007 | Mittel |
| Als Nutzer möchte ich eine Fehlermeldung sehen, wenn ich kein Internet habe. | US-008 | Mittel |
| Als Nutzer möchte ich eine Helfefunktion und ein Impressum in der App finden. | US-009 | Niedrig |

2.9 Detaillierte User Story Beschreibungen

In diesem Abschnitt werden die einzelnen User Stories ausführlich beschrieben, inklusive ihrer Akzeptanzkriterien.

2.9.1 User Story: Als Nutzer möchte ich beim Öffnen der App das aktuelle Wetter für meinen Standort sehen.

ID: US-001

Priorität: Hoch

Status: Neu

Verantwortlicher: Siehe GitHub

Geschätzte Story Points: 8

Als Nutzer
möchte ich die App öffnen und ohne mein Zutun das aktuelle Wetter für meinen Standort angezeigt bekommen,
damit ich genaue Infos über meinen Standort habe.

2.9.1.1 Details und Beschreibung

Diese User Story beschreibt die Kernfunktionalität der App, bei der der Nutzer sofort nach dem Start die relevantesten Wetterdaten für seinen aktuellen geografischen Standort sieht. Dies erfordert Zugriff auf die Standortdienste des Geräts und die Abfrage einer Wetter-API.

2.9.1.2 Akzeptanzkriterien

- Die App startet erfolgreich und fordert ggf. die Standortberechtigung an.
 - Nach Erteilung der Berechtigung wird das aktuelle Wetter (Temperatur, Wetterlage, ggf. Wind, Luftfeuchtigkeit) für den aktuellen Standort angezeigt.
 - Bei fehlender Standortberechtigung wird eine informative Meldung angezeigt und die Möglichkeit geboten, die Berechtigung zu aktivieren oder manuell eine Stadt einzugeben (siehe US-002).
 - Die angezeigten Daten sind aktuell und stammen von einer zuverlässigen Wetter-API.
-

2.9.1.3 Abhängigkeiten

* Benötigt eine Wetter-API-Integration. * Benötigt Zugriff auf Standortdienste des Geräts.

2.9.1.4 Anmerkungen/Diskussion

* 05.06.2025: Klärung, welche spezifischen Wetterdaten (z.B. Regenwahrscheinlichkeit, UV-Index) standardmäßig angezeigt werden sollen.

2.9.2 User Story: Als Nutzer möchte ich nach Städten suchen und sie einer Liste hinzufügen können.

ID: US-002

Priorität: Hoch

Status: Neu

Verantwortlicher: Sihe GitHub

Geschätzte Story Points: 6

Als *Nutzer*
möchte ich *nach Städten suchen und sie einer Liste hinzufügen können,*
damit *ich nachher auch das Wetter in diesen Städten anschauen kann.*

2.9.2.1 Details und Beschreibung

Diese User Story ermöglicht es dem Nutzer, Städte manuell hinzuzufügen, um deren Wetter zu verfolgen. Dies beinhaltet eine Suchfunktion und die Möglichkeit, ausgewählte Städte in einer persistenten Liste zu speichern.

2.9.2.2 Akzeptanzkriterien

- Es gibt ein Suchfeld, über das der Nutzer Städte eingeben kann.
 - Die Suchergebnisse zeigen relevante Städte basierend auf der Eingabe an (z.B. mit Autovervollständigung).
 - Der Nutzer kann eine Stadt aus den Suchergebnissen auswählen und diese zu einer Liste hinzufügen.
 - Die hinzugefügten Städte bleiben auch nach dem Schließen der App in der Liste gespeichert.
 - Die Liste der Städte ist übersichtlich dargestellt und leicht zugänglich.
-

2.9.2.3 Abhängigkeiten

* Benötigt eine Stadt-Geocoding-API oder ähnliche Datenbank. * Benötigt eine persistente Speicherung der Städte.

2.9.2.4 Anmerkungen/Diskussion

* 05.06.2025: Überlegen, ob es eine Begrenzung für die Anzahl der speicherbaren Städte geben soll.

2.9.3 User Story: Als Nutzer möchte ich das Wetter der zukünftigen Tage an meinem Standort sehen.

ID: US-003

Priorität: Hoch

Status: Neu

Verantwortlicher: Siehe GitHub

Geschätzte Story Points: 8

*Als Nutzer
möchte ich das Wetter der zukünftigen Tage an meinem Standort sehen,
damit ich genau über die Wettersituation informiert bin.*

2.9.3.1 Details und Beschreibung

Diese User Story erweitert die Standortanzeige um eine Vorhersage für die kommenden Tage (z.B. 3-5 Tage). Der Nutzer soll auf einen Blick sehen können, wie sich das Wetter entwickelt.

2.9.3.2 Akzeptanzkriterien

- Neben dem aktuellen Wetter werden Vorhersagen für die nächsten 3/5 Tage für den aktuellen Standort angezeigt.
 - Jede Tagesvorhersage zeigt mindestens die Höchst- und Tiefsttemperatur und die Wetterlage an (z.B. Sonnensymbol, Regensymbol).
 - Die Vorhersage ist klar und übersichtlich dargestellt.
 - Die Daten stammen von der integrierten Wetter-API.
-

2.9.3.3 Abhängigkeiten

* Benötigt eine Wetter-API, die Vorhersagedaten bereitstellt. * Abhängig von US-001 (Standortbestimmung).

2.9.3.4 Anmerkungen/Diskussion

* 05.06.2025: Festlegen, wie viele Tage die Vorhersage umfassen soll (z.B. 3 oder 5 Tage).

2.9.4 User Story: Als Nutzer möchte ich das zukünftige Wetter von gespeicherten Städten sehen.

ID: US-004

Priorität: Hoch

Status: Neu

Verantwortlicher: Siehe GitHub

Geschätzte Story Points: 8

Als Nutzer
möchte ich das zukünftige Wetter einer der zuvor festgelegten Städte sehen,
damit ich über die Situation an einem anderen Ort Bescheid weiß.

2.9.4.1 Details und Beschreibung

Diese User Story ermöglicht es dem Nutzer, die Wettervorhersage für die Städte einzusehen, die er zu seiner Liste hinzugefügt hat.

2.9.4.2 Akzeptanzkriterien

- Der Nutzer kann eine gespeicherte Stadt aus der Liste auswählen.
 - Nach Auswahl der Stadt werden die aktuellen Wetterdaten und die Vorhersage für die nächsten [X] Tage für diese Stadt angezeigt.
 - Die Anzeige der Vorhersage ist konsistent mit der Anzeige für den eigenen Standort (US-003).
-

2.9.4.3 Abhängigkeiten

* Abhängig von US-002 (Städte hinzufügen und speichern). * Benötigt Wetter-API für Vorhersagedaten für beliebige Orte.

2.9.4.4 Anmerkungen/Diskussion

* 05.06.2025: Wie wird zwischen den Städten gewechselt? (z.B. Swipe-Gesten, Dropdown-Menü).

2.9.5 User Story: Als Nutzer möchte ich Städte aus meiner Liste löschen können.

ID: US-005

Priorität: Hoch

Status: Neu

Verantwortlicher: Siehe gitHub

Geschätzte Story Points: 3

Als Nutzer
möchte ich Städte aus der Liste löschen können,
damit ich meine Ansicht verbessern und vereinfachen kann.

2.9.5.1 Details und Beschreibung

Diese User Story bietet die Möglichkeit, nicht mehr benötigte Städte aus der Favoritenliste des Nutzers zu entfernen.

2.9.5.2 Akzeptanzkriterien

- Der Nutzer kann eine Option zum Löschen von Städten in der Liste finden (z.B. Bearbeiten-Button, Swipe-Geste).
 - Nach Bestätigung wird die Stadt aus der Liste entfernt.
 - Die Änderungen werden dauerhaft gespeichert.
 - Der Nutzer erhält eine visuelle Rückmeldung über das Löschen der Stadt.
-

2.9.5.3 Abhängigkeiten

* Abhängig von US-002 (Städte hinzufügen und speichern).

2.9.5.4 Anmerkungen/Diskussion

* 05.06.2025: Bestätigungsdiallog für das Löschen einer Stadt implementieren, um versehentliches Löschen zu verhindern.

2.9.6 User Story: Als Nutzer möchte ich auch ohne Internet das letzte aktuelle Wetter an meinem Standort sehen.

ID: US-006

Priorität: Mittel

Status: Neu

Verantwortlicher: Siehe GitHub

Geschätzte Story Points: 5

Als *Nutzer*
möchte ich *auch ohne Internet das letzte aktuelle Wetter an meinem Standort sehen,*
damit *ich auch ohne Internet die letzten zuverlässigen Wetterzustände sehen kann.*

2.9.6.1 Details und Beschreibung

Diese User Story beschreibt die Offline-Fähigkeit der App für den aktuellen Standort. Die zuletzt geladenen Wetterdaten sollen gespeichert und bei fehlender Internetverbindung angezeigt werden.

2.9.6.2 Akzeptanzkriterien

- Bei bestehender Internetverbindung werden die Wetterdaten des Standorts lokal gespeichert.
 - Wenn die Internetverbindung unterbrochen ist, werden die zuletzt gespeicherten Wetterdaten für den aktuellen Standort angezeigt.
 - Es wird ein Hinweis angezeigt, dass die Daten nicht aktuell sind und wann sie zuletzt aktualisiert wurden.
 - Der Wert sollte nicht älter als das Ende des Forecasts sein. (Die Daten sollten innerhalb des Gültigkeitszeitraums der letzten Vorhersage liegen).
-

2.9.6.3 Abhängigkeiten

* Abhängig von US-001 (Aktuelles Wetter am Standort). * Benötigt lokale Speichermechanismen.

2.9.6.4 Anmerkungen/Diskussion

* 05.06.2025: Festlegen, wie lange die gecachten Daten als "zuverlässig" gelten (z.B. maximal 1-2 Stunden für aktuelle Daten, bis zum Ende des Forecasts für Vorhersagen).

2.9.7 User Story: Als Nutzer möchte ich auch das Wetter aus zuvor hinzugefügten Städten sehen ohne Internet zu haben.

ID: US-007

Priorität: Mittel

Status: Neu

Verantwortlicher: Siehe GitHub

Geschätzte Story Points: 5

Als Nutzer

möchte ich auch das Wetter aus zuvor hinzugefügten Städten sehen, ohne Internet zu haben.

Der Wert sollte nicht älter als das Ende des Forecasts sein.

2.9.7.1 Details und Beschreibung

Diese User Story erweitert die Offline-Funktionalität auf die vom Nutzer gespeicherten Städte. Die zuletzt abgerufenen Wetterdaten und Vorhersagen für diese Städte sollen ebenfalls gecacht und bei fehlender Internetverbindung angezeigt werden.

2.9.7.2 Akzeptanzkriterien

- Bei bestehender Internetverbindung werden die Wetterdaten der gespeicherten Städte lokal gespeichert.
 - Wenn die Internetverbindung unterbrochen ist, werden die zuletzt gespeicherten Wetterdaten und Vorhersagen für die hinzugefügten Städte angezeigt.
 - Es wird ein Hinweis angezeigt, dass die Daten nicht aktuell sind und wann sie zuletzt aktualisiert wurden.
 - Die angezeigten Wetterdaten und Vorhersagen sollten nicht älter sein als das Ende des letzten abgerufenen Forecasts.
-

2.9.7.3 Abhängigkeiten

* Abhängig von US-002 (Städte hinzufügen), US-004 (Zukünftiges Wetter gespeicherter Städte). * Benötigt lokale Speichermechanismen.

2.9.7.4 Anmerkungen/Diskussion

* 05.06.2025: Speicherkapazität der App für Offline-Daten berücksichtigen, insbesondere bei vielen gespeicherten Städten.

2.9.8 User Story: Als Nutzer möchte ich eine Fehlermeldung sehen, wenn ich kein Internet habe.

ID: US-008

Priorität: Mittel

Status: Neu

Verantwortlicher: Siehe GitHub

Geschätzte Story Points: 1

Als *Nutzer*
möchte ich *dass es eine Fehlermeldung gibt,*
damit *ich weiß, dass ich aktuell kein Internet habe.*

2.9.8.1 Details und Beschreibung

Diese User Story stellt sicher, dass der Nutzer klar über den Verbindungsstatus der App informiert wird, insbesondere wenn keine Internetverbindung verfügbar ist und daher keine aktuellen Wetterdaten abgerufen werden können.

2.9.8.2 Akzeptanzkriterien

- Wenn die App versucht, Wetterdaten abzurufen und keine Internetverbindung besteht, wird eine sichtbare Fehlermeldung angezeigt (z.B. ein Banner oder Toast-Nachricht).
 - Die Fehlermeldung informiert den Nutzer klar darüber, dass keine Verbindung besteht.
 - Die Fehlermeldung verschwindet automatisch, sobald die Verbindung wiederhergestellt ist, oder bietet die Möglichkeit, sie manuell zu schließen.
 - Die Fehlermeldung sollte die primäre Anzeige nicht überdecken.
-

2.9.8.3 Abhängigkeiten

* Benötigt eine Netzwerkstatusprüfung.

2.9.8.4 Anmerkungen/Diskussion

* 05.06.2025: Formulierung der Fehlermeldung abstimmen. Soll sie auch auf die Offline-Funktionalität hinweisen?

2.9.9 User Story: Als Nutzer möchte ich eine Helfefunktion und ein Impressum in der App finden.

ID: US-009

Priorität: Niedrig

Status: Neu

Verantwortlicher: Siehe GitHub

Geschätzte Story Points: 2

Als Nutzer
möchte ich eine Helfefunktion anklicken können,
damit ich die App erklärt bekomme und sehen kann, von wem sie kommt und mit welcher Lizenz sie ist (Impressum und Hilfe).

2.9.9.1 Details und Beschreibung

Diese User Story umfasst die Bereitstellung von Informationen über die App selbst, ihre Nutzung und rechtliche Hinweise. Dies ist wichtig für die Benutzerfreundlichkeit und die Einhaltung gesetzlicher Vorschriften (Impressumspflicht in Deutschland).

2.9.9.2 Akzeptanzkriterien

- Es gibt einen zugänglichen Menüpunkt oder Button, der zur Helfefunktion/zum Impressum führt.
 - Die Helfefunktion enthält eine kurze Einführung in die App und deren Hauptfunktionen.
 - Das Impressum enthält die gesetzlich vorgeschriebenen Informationen (Name, Adresse, ggf. Rechtsform, Vertretungsberechtigter).
 - Informationen zur Lizenzierung der verwendeten Bibliotheken/APIs sind verfügbar.
 - Die Inhalte sind lesbar und gut formatiert.
-

2.9.9.3 Abhängigkeiten

* Keine direkten technischen Abhängigkeiten; Fokus liegt auf Content-Erstellung.

2.9.9.4 Anmerkungen/Diskussion

* 05.06.2025: Inhalte für Hilfe und Impressum finalisieren. Rechtliche Prüfung des Impressums erforderlich.

3 Design (Michael Filatoff)

3.1 Ablauf

Nachdem die Software-Requirements und User-Stories für die Wetter-App vom Product Owner und Scrum Master final definiert wurden, konnte man daraufhin mit der Software 'Figma', die eine Designsoftware ist, mehrere Mockup-Designs erstellen. Nachdem die Designs bereit waren, wurden sie dem Product Owner vorgestellt, um das Go zu erhalten. Ein paar Verbesserungen wurden ergänzt, und die finalen Designs wurden dem Entwicklungsteam übermittelt.

3.2 Software Vorstellung

Die folgenden Mockup-Designs wurden anhand der Requirements und mit der Hilfe der Software "Figma" erstellt. Daher ist es sinnvoll, die Software vorher zu präsentieren.

Figma ist eine kollaborative Software zum Erstellen von Prototypen und Designs im Bereich UX/UI-Design. Sie eignet sich hervorragend zum Gestalten, Teilen und Testen von Designs für Webseiten und

Applikationen. Sobald ein Projekt angelegt wurde, hat man die Möglichkeit, sich für einen Prototypen zu entscheiden, der die Größe bestimmt, die das Frame besitzen soll. Das Frame stellt beispielsweise bei einer Applikation das Display dar, worauf einzelne Elemente gebaut werden können und daraus ein Design entsteht. Auf der linken Seite befindet sich das Projektverzeichnis, welches die Pages, also einzelne Seiten, besitzt, und Layers, die die einzelnen Frames beinhalten.

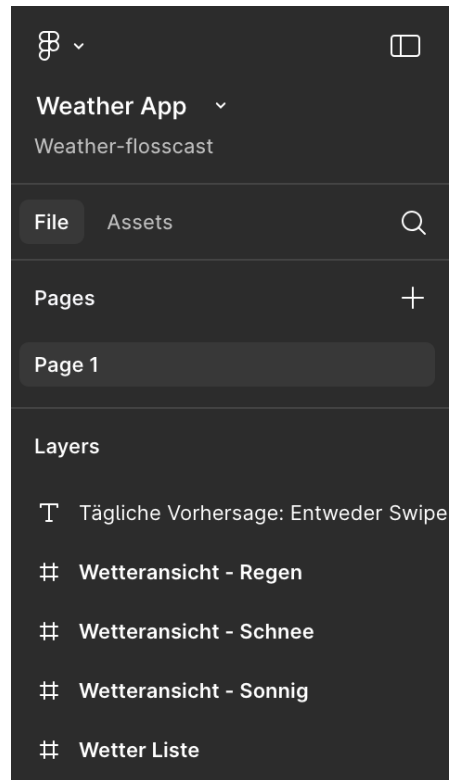


Abbildung 4: Figma linke Ansicht

Auf der rechten Seite befinden sich die Einstellungen für die einzelnen Frames und deren Elemente, um diese zu bearbeiten und zu designen. Außerdem hat man die Option, die Größe des Prototyps jederzeit anzupassen und das Projekt in einer Vorschau vorzustellen bzw. zu teilen oder zu exportieren.

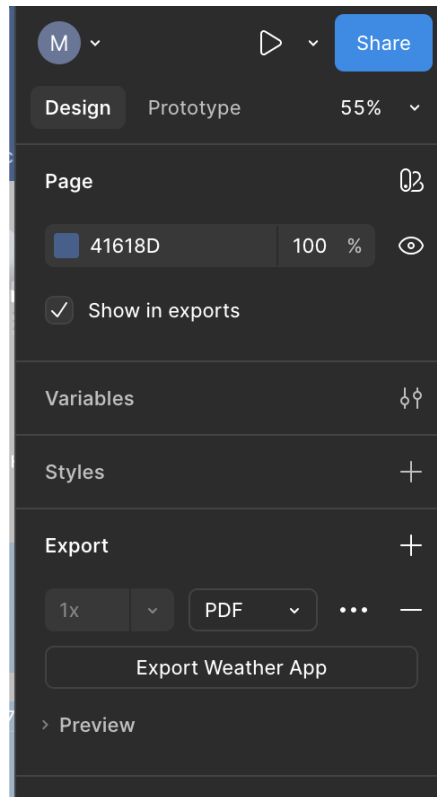


Abbildung 5: Figma rechte Ansicht

3.3 Planung

Die Applikation sollte schlicht, übersichtlich, benutzerfreundlich gestaltet sein und aus zwei Seiten bestehen. Die erste Seite hat die Funktion nach Städten bzw. Flughafen suchen zu können und diese mit den wichtigsten Informationen in der Liste darunter anzeigen zu lassen. Sobald man auf eine der Städte gedrückt hat, wird man auf die zweite Seite navigiert, die wiederum mehr Informationen bezüglich des Wetters von heute und die nächsten 7 Tage beinhaltet und bildlich darstellt, wie unter anderem der UV-Index oder die Niederschlagsmenge.

3.4 Aufbau und Layout

3.4.1 Wetter Liste

Der Hintergrund der Liste hat die Farbe schwarz und die Schriftfarbe ist in weiß. Beginnend von oben, ist links die Überschrift Wetter und rechts ein Dropdown-Menü, welches die Möglichkeit bietet, die Grad zahl in Celsius oder Fahrenheit darzustellen. Darunter gibt es eine Suchfunktion, um eine Stadt oder Flughafen suchen zu können. Daraufhin kann der gesuchte Ort gespeichert werden, um diesen in der Liste immer anzeigen lassen zu können. Die Liste besteht aus gespeicherten Orten, die die wichtigsten Informationen bezüglich des Wetters von heute darstellen. Sobald man auf einen der Orte drückt, expandiert sich das Fenster und eine vergrößerte Ansicht mit Informationen des Wetters, von heute und in den nächsten 7 Tage, wird angezeigt.



Abbildung 6: Wetter Liste

3.4.2 Wetter Vorhersage

Sobald man sich für ein Ort entschieden hat, entsteht eine vergrößerte Ansicht des Ortes mit passendem Hintergrund und Bild, welche abhängig von der Wetterlage ist. Es gibt drei Ansichten. Die erste stellt ein sonniges und die zweite ein regnerisches Wetter da. Die dritte stellt das Wetter beim Schneefall dar. Somit wird das derzeitige Wetterverhältnis bereits mit dem Bild ganz oben dargestellt mit zusätzlichen Informationen, wie der Ortsname, Ortsgrad, Ortswetter und die Höchst- bzw. Tiefst- Temperatur. Darunter befindet sich eine Box, die die Wettervorhersage stündlich für den heutigen Tag mit Bild und Grad ausgibt. Ähnlich ist es bei der zweiten Box, die wiederum das Wetter für die gesamte Woche mit Bild und Höchst- bzw. Tiefst- Temperatur darstellt. Für zusätzliche Informationen scrollt man nach unten und findet den UV-Index, die Windstärke bzw. Richtung und falls es Niederschlag gibt, die mm Anzahl.

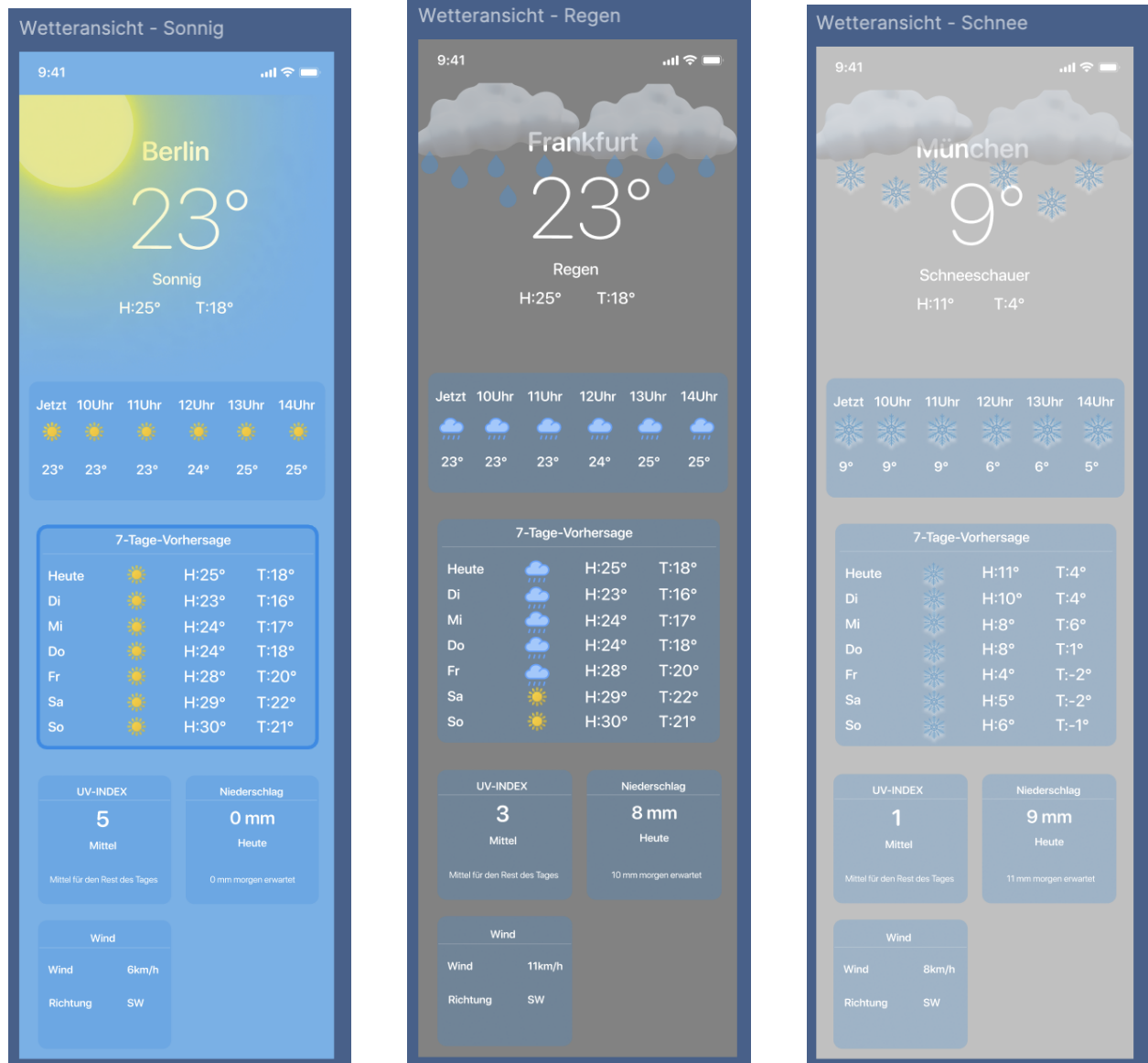


Abbildung 7: Wetter Vorhersagen

4 Technische Dokumentation

4.1 Frontend (Patryk Brychcy, Sascha Gavrilov)

4.1.1 Einleitung

Unsere Wetter-App besteht aus zwei Hauptbereichen: dem „SearchScreen“ zum Anlegen und Verwalten von Städten und dem „WeatherScreen“ zur Anzeige der aktuellen Wetterdaten und der Prognose für eine ausgewählte Stadt. Sobald die App gestartet wird, entscheidet MainActivity anhand der gespeicherten Städte, welcher Bildschirm zuerst angezeigt wird: entweder der Suchbildschirm, falls noch keine Städte hinterlegt sind, oder direkt die Wetteransicht für die zuletzt verwendeten Stadt.

4.1.2 Architekturübersicht

Die App ist nach dem klassischen Drei-Schichten-Prinzip aufgebaut.

In der Präsentationsschicht stellen Composable-Funktionen in SearchScreen.kt und WeatherScreen.kt die Benutzeroberfläche dar, reagieren auf Eingaben und visualisieren die aktuellen Wetterdaten.

Die Logik- und Aufbereitungsschicht in Utils.kt enthält alle Erweiterungsfunktionen, die die unbearbeiteten Forecast-Daten in gebrauchsfertige Teildaten umwandeln – etwa die momentane Temperatur, stündliche und tägliche Prognosewerte sowie die passenden Icons und Farben.

In der Datenhaltung schließlich übernimmt CityList.kt die Persistenz, indem es bis zu 30 Städte als Base64-kodiertes JSON in den SharedPreferences verwaltet.

So sind Darstellung, Geschäftslogik und Datenspeicherung sauber voneinander getrennt, was Wartung und Erweiterung der App erheblich erleichtert.

4.1.3 MainActivity

Die Navigation wird über einen NavHostController realisiert, der zwei Routen besitzt : „search“ für SearchScreen und „weather/cityName“ für WeatherScreen.

Beim Start liest MainActivity über CityList.getCities(context) die gespeicherten Städte ein. Bei einer leeren Liste navigiert die App automatisch zum Suchbildschirm und zeigt eine kurze Info „Keine Städte gespeichert“. Bei einer gefüllten Liste leitet die App direkt weiter zur Detailansicht der zuletzt genutzten Stadt (WeatherScreen(cityName)).

4.1.4 SearchScreen

SearchScreen ist der zentrale Bildschirm, auf dem Nutzer neue Städte suchen und ihre bereits gespeicherten Orte verwalten können. Oben befindet sich eine Suchleiste, in die man per Textfeld den Namen einer Stadt eingibt. Sobald man die Suche auslöst, leert die Liste zunächst alle vorherigen Suchergebnisse und zeigt dann passende Treffer in Form von NewCityCards an. Jede Karte repräsentiert eine Stadt mit Name, Bundesland und Land und lädt bei einem Tipp im Hintergrund über eine Coroutine den Forecast.

Gelingt die Abfrage, wird die Stadt in der Liste gespeichert (Duplikate anhand von Name, Bundesland und Land werden automatisch vermieden) und man springt direkt zur Detailansicht.

Gelingt die Abfrage nicht, erhält man lediglich einen kurzen Toast-Hinweis auf den Fehler.

Unterhalb der Suche zeigt CityListView alle bereits gespeicherten Städte als CityCards an. Jede CityCard enthält das aktuelle Wetter (Icon und Beschreibung), die momentane Temperatur, Minima und Maxima des Tages sowie einen Zeitstempel der letzten Aktualisierung.

Ein kurzer Tap auf eine CityCard öffnet die Detailseite, ein langer Druck ruft ein Bestätigungsdialog auf, mit dem man die Stadt aus der Liste entfernen kann. Beim ersten Laden der gespeicherten Städte oder beim Pull-to-Refresh blendet die App einen animierten Shimmer-Placeholder ein.

4.1.5 WeatherScreen

WeatherScreen zeigt alle relevanten Wetterinformationen für eine ausgewählte Stadt. Direkt beim Betreten lädt die App den Forecast über getForecastFromCacheOrDownload, entweder aus dem lokalen Cache oder frisch von der API. Über einen SwipeRefresh-Wrapper können Nutzer jederzeit nach unten wischen, um die Daten neu zu laden.

Im oberen Bereich, dem WeatherHeader, werden mithilfe der Funktion getWmoCodeAndIsNight() der WMO-Wettercode und der aktuelle Tag oder Nacht-Status ermittelt. Damit wählt die App automatisch die passende Lottie-Animation sowie die Hintergrundfarbe aus und zeigt in einer Spalte den Stadtnamen, die aktuelle Temperatur und einen kurzen Zustands-Text an.

Darunter folgt eine horizontale Stundenübersicht „HourlyForecastRow“, die in einer LazyRow 24 Stunden anzeigt. Jede HourlyItem-Karte besteht aus der Uhrzeit, Temperatur und dem entsprechenden Icon und liegt in einem leicht abgedunkelten Box-Hintergrund, um stets gut lesbar zu bleiben. Anschließend bietet der SevenDayForecastBlock eine Sieben-Tage-Prognose in Form von sieben Spalten: Jeder DailyItem zeigt den Tagesnamen („Heute“ oder Wochentag), die Regenwahrscheinlichkeit in Prozent, ein Icon sowie die Tageshöchst- und Tiefsttemperatur.

Ganz am Ende befindet sich die InfoBoxesSection mit vier Infoboxen: aktuelle Luftfeuchtigkeit, erwarteter Niederschlag für heute und morgen sowie Sonnenauf- und -untergangszeiten. Die Boxen liegen jeweils paarweise in zwei Zeilen, jede in einem verdunkeltem Hintergrund-Rahmen.

4.1.6 Utils.kt

In `Utils.kt` befinden sich Erweiterungsfunktionen, die die unbearbeiteten Forecast-Daten in übersichtliche Teildaten umwandeln. So liefert `getCurrentTemperature()` die Temperatur der aktuellen Stunde, `getDailyMinTemp()` und `getDailyMaxTemp()` ermitteln das Minimum und Maximum aus den 24 Stunden eines Tages. Die Funktion `getHourlyData(offsetHours: Int)` baut daraus ein `HourlyData`-Objekt mit Zeit, Temperatur, Zustand und Nacht-Flag, während `getDailyData(day: Int)` ein `DailyData` mit Label, Regenwahrscheinlichkeit, Min/Max und Icon-Code erstellt. Über Helper-Funktionen wie `getLottieResForWmoCode`, `getIconForWmoCode`, `getConditionForWmoCode` und `colorForWmoCode` wird jeweils basierend auf dem WMO-Code und dem Nacht-Status das richtige Icon, die passende Animation, der Beschreibungstext und die Hintergrundfarbe ausgewählt [3][1]. Eine Parallelisierungsfunktion `loadForecastsForCities(context, cities)` lädt Forecasts für mehrere Städte gleichzeitig und ermöglicht dadurch ein zügiges Nachladen im `SearchScreen`.

4.1.7 CityList.kt

`CityList` ist ein Singleton-Objekt, das die Persistenz aller Städte in den `Android-SharedPreferences` übernimmt. Es legt einen PREF-Datensatz („`CityListPref`“) an und speichert bis zu 30 Städte als Base64-kodiertes JSON. Mit `getCities(context)` liest es die aktuelle Liste aus, `addCity(context, newCity)` fügt eine Stadt vorne hinzu (und ersetzt dabei ein bestehendes Duplikat), `removeCity(context, city)` entfernt einen Eintrag, und `clearCities(context)` leert die gesamte Liste – nützlich zum Debuggen oder Reset. Intern sorgt eine einfache Duplikatprüfung dafür, dass Namen, Bundesland und Land übereinstimmen, bevor eine Stadt wirklich hinzugefügt wird, und bei einem Überschreiten der Listenlänge wird automatisch der älteste Eintrag entfernt.

4.1.8 Zusammenfassung

Dank der klaren Trennung zwischen den Composables für die Bildschirme, den Hilfsfunktionen in `Utils.kt` und der Persistenz in `CityList.kt` ist das Frontend genau strukturiert und leicht wartbar. Neue Features wie Favoriten oder erweiterte Details lassen sich durch zusätzliche Module ergänzen, ohne bestehende Teile anzupassen.

4.2 Datenvorbereitung (Benedikt Zundel)

Als Schnittstelle für die Wetterdaten haben wir uns für die `open-meteo` [2] API entschieden, da diese alle Daten liefert die wir anzeigen wollten und über weitere Funktionalitäten wie geocoding verfügt. Um die Daten zu konsumieren, definierten wir zunächst ein Datenstruktur (vgl. listing 1) um die abgeholten Daten intern zu repräsentieren. Die Idee war es, dem frontend für den Konsum nur ein einzelnes Objekt zu übergeben, über das sinnvoll iteriert werden kann.

Listing 1: Forecast Datenstruktur

```
1 data class Forecast(  
2     val timestamp: LocalDateTime,  
3     val days: List<DailyForecast>,  
4     val units: Units  
5 )  
6  
7 data class DailyForecast(  
8     val date: LocalDate,  
9     val hourlyValues: List<Hourly>,  
10    val sunrise: LocalDateTime,  
11    val sunset: LocalDateTime  
12 )  
13  
14 data class Hourly(  
15     val dateTime: LocalDateTime,  
16     val temperature: Double,  
17     val relativeHumidity: Int,  
18     val precipitationProbability: Int,  
19     val rain: Double,  
20     val showers: Double,  
21     val snowfall: Double,  
22     val weatherCode: Int
```

```

23 )
24
25 data class Units(
26     val temperature: String,
27     val humidity: String,
28     val precipitationProbability: String,
29     val rain: String,
30     val showers: String,
31     val snow: String,
32 )

```

Mittels Herangehensweisen aus der funktionalen Programmierung wie `map` lässt sich das von der API gelieferte JSON-Objekt mit wenig Aufwand, lesbar und modifizierbar in unsere Datenstruktur parsen (vgl. listing 2).

Listing 2: Auszug des Parsingprozesses

```

1 val hourly: JsonObject = json["hourly"]!!.jsonObject
2 val hourlyTime =
3     hourly.getOrThrow("time").jsonArray.map { LocalDateTime.parse(it.jsonPrimitive.content)
4     }
5 val hourlyTemperature =
6     hourly.getOrThrow("temperature_2m").jsonArray.map { it.jsonPrimitive.content.toDouble()
7     }
8 ...
9 val dailyForecasts: List<DailyForecast> = hourlyValues.groupBy { it.dateTime.date }
10    .map { (date, measurements) -> DailyForecast(date
11        , measurements
12        , sunrise.first { it.date == date
13        }.toInstant(TimeZone.UTC).toLocalDateTime(currentTimezone)
14        , sunset.first { it.date == date
15        }.toInstant(TimeZone.UTC).toLocalDateTime(currentTimezone)) }

```

Die Abfrageadresse ergibt sich aus Koordinaten die von Nutzer*innen als ausgewählte Stadt oder aktuellem Standort übergeben werden. Zur Realisierung der Städtewahl wird die selbe API unter dem geocoding Endpunkt angesprochen. Dieser liefert auf Basis eines (eventuell unvollständigem) Städtenamens eine Liste an bekannten Städten als Vorschläge, die dann in einem ähnlichen Prozess von JSON in eine von uns definierte Datenstruktur geparsed werden und dem frontend zum Anzeigen weitergegeben wird.

In dem gesamten Abfrageprozess wird hoher Wert auf die Richtigkeit der Daten und eine erfolgreiche Abwicklung der Anfragen gelegt. Viele der Schritte werden während der Verarbeitung der Daten gegen Erwartungswerte geprüft und unerwartete Situationen werden mittels Exceptions unmittelbar an das frontend kommuniziert. In Fehlersituation, sowie beim normalen Ablauf, wird eine Nachvollziehbarkeit des Prozesses mittels intensiven Loggings erreicht.

4.3 Caching (Benedikt Zundel)

Unnötige Abfragen werden zum Schutze des Verbrauchs der Nutzer*innen, sowie zur Entlastung der API unterbunden. Ein weiterer Nutzen ist die Verfügbarkeit der aktuellen Daten ohne Internetverbindung. Um das zu realisieren, implementieren wir ein dateibasiertes Caching-System. Die unterliegende Datei enthält die von der API abgeholten Wetterdaten nach unserer definierten Datenstruktur serialisiert in einem JSON-Objekt, welches die Daten via gekürztem Koordinatensatz indiziert. Durch das Kürzen der Koordinaten auf eine Nachkommastelle erreichen wir einen klein gehaltenen Datensatz der gecached wird, da nicht jede minimale Veränderung im Standort als neuer individuelle Standort angesehen wird. Der Standort wird durch die Kürzung auf circa $12km \times 12km$ Kacheln abgebildet, was, durch die kleine Fläche, zu keinem signifikanten Genauigkeitsverlust führt. Bei einem Aufruf der vom backend offengelegten Schnittstelle wird zunächst der Cache geladen und geprüft ob für die angefragten Koordinaten bereits ein Eintrag existiert. Sollte dies nicht der Fall sein, wird eine Abfrage auf die API gemacht und die Daten werden im Cache gespeichert. Wenn ein Eintrag gefunden wird, wird das Alter der Daten über ein Zeitstempel Feld geprüft. Ist ein Eintrag älter als eine Stunde, gilt dieser als veraltet und wird aktualisiert. Sonst wird der Eintrag aus dem Cache geladen und es geschieht keine Abfrage an die API.

4.4 Testing (Dwipa Flügel, Dominik Lepore)

4.4.1 Einleitung

Das Testing in Android wird unterschieden in Unit Tests, welche lokal auf der Java Virtual Machine ausgeführt wird (JVM), Component Tests und Feature Tests, welche sowohl lokal als auch emuliert wird und letztlich Application und Release Candidate Tests, welche nur emuliert werden oder auf echten Geräten laufen. Wir werden uns in unsere Applikation nur um Unit, Component, Feature und Application Tests beschäftigen, da Release Candidate Tests tendenzierend auf echten Geräten getestet werden sollten. Unit Tests prüfen eine einzelne Logikeinheit, unabhängig vom Android Framework. Component Tests prüfen einzelne Komponenten unabhängig von anderen Komponenten, jedoch weniger isolierter als nur Methoden und Klassen. Feature Tests prüfen die Interaktion zwischen mindestens zwei unabhängigen Komponenten, als Beispiel könnte man hier die Interaktion zwischen UI und abgefragten Daten nennen. Application Tests prüfen zuletzt die ganze Funktionalität der Applikation, zum Beispiel indem sie einen User Flow imitieren und prüfen.

4.5 Testplan

| Subject under Test | Description | Category | Example Test |
|-------------------------------------|---|-------------------|---|
| Search Bar UI behavior | Shows recommendation based on autocorrect | Component tests | UI Behavior test running on Espresso/Jetpack Compose/Roboelectric |
| Citylist UI behavior | A form with a list that is only enabled when at least one city has been added | Component tests | UI Behavior Test on Jetpack Compose |
| Citylist City selection | A screen showing the weather forecast when a city is selected | Application tests | UI Behavior Test on Jetpack Compose |
| Search result list UI behavior | A form with a list that is only shown after the user has entered an input into the search bar | Component tests | UI Behavior Test on Jetpack Compose |
| Search bar city search | A screen showing the search results when a user has searched for a city | Application tests | UI Behavior Test on Jetpack Compose |
| Search result list adding city | A screen showing the updated city list after a user has selected a city to add | Application tests | UI Behavior Test on Jetpack Compose |
| Current location UI behavior | A form with a forecast of the current location after opening the app | Component tests | UI Behavior Test on Jetpack Compose |
| Retention of Citylist after closing | A complete app opening, closing and opening flow | Application tests | UI Behavior Test on Jetpack Compose |
| WeatherScreen Hourly Item UI | Displays the temperature and a weather icon for an specified hour of an specified day | Component tests | Compose preview running on Jetpack Compose using mock values |

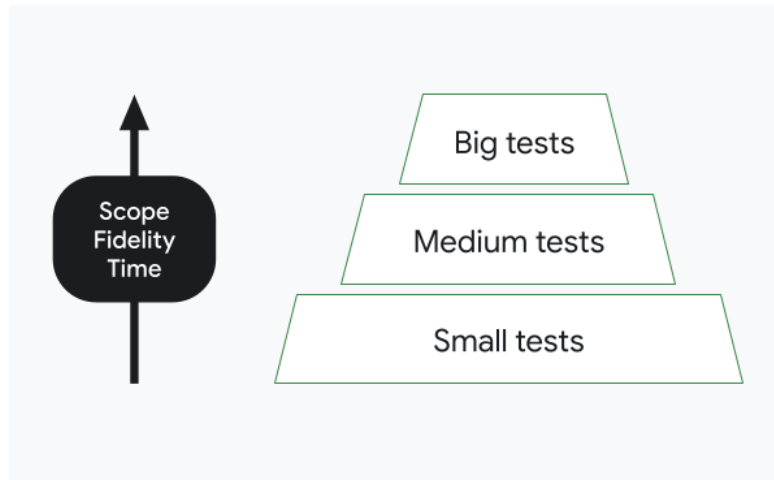
| | | | |
|--|--|----------------|--|
| Backend DataHelper Forecast Caching | Caches retrieved forecast data, updates expired forecast data and retrieves data from the external API | Component test | Using Test context and mock Caches different caching scenarios are tested. E.g. an expired Cache has to be updated |
| City List encoding and decoding | Encodes the City Data-type to a json String and decodes it in the opposite direction | Unit test | Simple validating if the encoding and decoding keeps the right values |
| Utils Background color calculation | Calculates the background color depending on the time of day and the weather | Unit test | Validating the right colors are returned on all possible scenarios |
| Utils helper functions | Defined Utils helper function to easily retrieve needed forecast data | Unit test | Validating that the right forecast data is returned, based on mocked forecast data |
| Utils LottieRes Icons | Returns LottieRes Icons depending on the time of day and the current weather | Unit test | Checking if the right icons are calculated for all possible scenarios |
| Utils Weather Code Handling | Convert Weather Codes to the equivalent german words for the weather | Unit test | Validating that the right german description is returned on every possible weather code |

4.5.1 Probleme

Der Prozess des Testings hatte sich äußerst schwer ergeben. Wir beide haben leider keine Erfahrung in der Entwicklung mit Android als auch Kotlin und empfanden das Testing auch als sehr überwältigend, trotz der vorherigen Beteiligung des Testings im letzten Projekt. Wir haben das reine Ausmaß an Testing im Vergleich zum letzten Projekt einfach zu sehr unterschätzt, zudem hatte sich Kotlin als weitaus anspruchsvollere Sprache herausgestellt als Python. Ein unerwartetes Problem ist überraschenderweise jedoch die Testability unserer Wetter-App. Laut Google sollte eine Testing-Strategie verfolgt werden, in der integrierte Tests auf Isolierten aufbauen. Grafisch sieht das folgendermaßen aus:

Das heißt, ein Großteil des Testings sollte unit-basiert sein, was sich in unserem Falle jedoch als etwas schwierig herausgestellt hat. Das Backend des Projektes beschäftigt sich mit den Geo-Daten, die wir aus OpenStreetMap und den Wetter-Daten, die wir aus Open-Meteo beziehen. Hierfür stehen uns jedoch nur wenige Funktionen zur Verfügung, da ein Großteil der Methoden auf private gestellt ist in unserer DataHelper-Klasse. Eine weitere Herausforderung des Unit Testing des Backends war die CityList-Klasse, welche nur benutzt werden kann, nachdem ein Singleton-Objekt erstellt wurde. Zudem verwenden die Methoden des Objektes eine sharedPreferences-Datei, welche Teil des Android-Framework ist und somit unter die Integration Tests fallen. Unit Testing im Frontend ist eher unüblich, jedoch in Situationen möglich, indem unsere UI-Elemente kein Android-Framework benutzen. Die UI unserer App wird jedoch durch eine MainActivity gestartet, besitzt einen NavHost und damit einen NavController, welche die Sequenzen unserer angezeigten Seiten steuert, zudem benutzen einen Großteil unserer Composables auch noch den NavController oder einen Context, eine weitere Funktion des Android-Frameworks. Es stellt sich also als eher schwierig diese in Isolierung zu testen, der Großteil des UI-Testings sind also Instrumented-Tests, welche den NavController mocken, der gebraucht wird um die Composables überhaupt zu starten. Das Hauptproblem, welches das UI-Testing Volumen ist, basiert auf der Dynamik des Projektes. Eine Wetter-App zieht prinzipiell nur Daten aus dem Internet, speichert sie Lokal und stellt diese Daten dynamisch dar für jede mögliche Kombination der Daten. Da das Aussehen unserer App dynamisch ist und sich an den Daten anpasst, welche alle

Abbildung 8: Bild der Google Testing Pyramide



variieren, sind Screenshot-Test leider sehr unbrauchbar. Die UI besitzt eine zu große Menge an Entropie, welche den Aufwand an korrekten Screenshots drastisch in die Höhe schießt, zudem erhöht sich dies mit der Menge and möglichen Kombinationen von Android Setups. Dadurch wurde entschieden, dass das UI Testing großteils auf Behavior Tests basieren sollte.

4.5.2 Verbesserungen

Es gibt einen weitaus größere Menge an Libraries, die beim Testen benutzt werden könnten. Die Unit-Tests der UI hätte mittels Roboelectric tatsächlich ohne Emulator laufen können, Roboelectric führt diese Tests dann beim Host in der JVM aus, dadurch würde man sich die Zeit des emulieren sparen. Auch wenn die Wetter-App etwas unpassend für Screenshot-Tests ist durch ihre Entropie, gibt es jedoch ein paar Fälle, bei dem das Sinn ergeben hätte, nämlich dem Erststart der App und der Stadtsuche ohne gespeicherte Städte. Zudem würde sich ein paar mehr Integration-Tests für gewisse Use-Cases doch noch eignen, wie das benutzen des Android-Frameworks wie das drehen und swipen auf dem Bildschirm, würden aber zu viele Ressourcen kosten für das Ausmaß der Workload. Außerdem wurden im letzten Projekt ein Monkey-Test erwähnt, den wir noch gerne hätten benutzen wollen, jedoch verblieb uns wenig Zeit durch andere Verpflichtungen. Nichtsdestotrotz sind wir beide zufrieden mit der Menge, die wir gelernt haben, welche durch den Stress des Umfangs und der Zeit entstanden ist.

Literatur

- [1] Lottiefiles: Download free lightweight animations for websites & apps. <https://lottiefiles.com/>. (Accessed: 2025-06-17).
- [2] Open-meteo: Free weather api. <https://open-meteo.com/>. (Accessed: 2025-06-14).
- [3] Svg repo - free svg vectors and icons. <https://www.svgrepo.com>. (Accessed: 2025-06-17).