University of Applied Sciences
Rapperswil

Bachelor Thesis

SensorFlow

Kyburz, Cyril *Scheiwiller, Sandro*

supervised by
Mirko Stocker

industrial partner
Bernhard Zindel, Live Track AG
Remo Liebi, Live Track AG

co-examiner
Daniel Politze

expert
Leo Büttiker

June 14, 2019

Abstract

Our industry partner *LiveTrack* installs sensors to monitor the filling level of containers. The collected data is used to optimize certain attributes of the collection of the container content, such as the time of departure and the route of the truck. In order to map sensors to containers, the technician must inform *LiveTrack* each time a sensor is installed or removed from a container. To solve this problem, we have developed a progressive web application that allows a technician to track his work. The app is already live and enables *LiveTrack* to easily maintain sensors and containers. Through its first productive use, *LiveTrack* has already gained new ideas to expand the capabilities of the app.

Management Summary

Motivation

In the morning after a party, empty glass bottles and beverage cans abound, waiting to be recycled at the next recycling point. There they can be thrown into recycling containers, which store the contents until they are collected. Emptying the containers at the right time is a challenge: Up until now, the municipality has to monitor their containers manually and, if they are nearly full, call for a truck to empty them.

Our industry partner, *LiveTrack*, is going to optimize this process on multiple levels. Sensors will be used to measure the current fill level of the containers, digitizing the data and removing the need for manual monitoring. Using this data, the timing and the routes of truck operations will be optimized.

Problem Statement

The sensors will be installed into the containers by a technician and start sending the fill level. Since the sensor does not know in which container it has been installed, the technician must inform *LiveTrack* of its whereabouts, preferably in a more practical way than displayed in Figure 1. Only then the received sensor data can be mapped to its corresponding container.

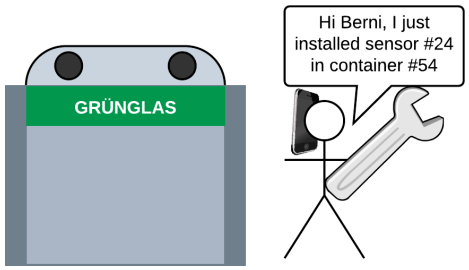


Figure 1: *Can you spot the problem?*

Method

Based on our own experiences, we know that tracking work digitally is often difficult and can prevent one from focusing on the actual work. Therefore our application should be as convenient and easy to use as possible. *And what could be more convenient than using your own smartphone?* Right, nothing, so it was decided to develop a progressive web application (PWA) that runs in the browser, but still feels like a native app.

Result

We created a mobile application called *SensorFlow*. All planned functionality has been implemented and tested thoroughly. For each action that the techni-

cian can perform, a workflow has been provided to make tracking his work as smooth as possible. Because of this, *LiveTrack* can now map sensors to containers and the filling level of containers can be determined. The app (a screen is shown in Figure 2) has been published and is being used in production, very much to the satisfaction of *LiveTrack*.

Outlook

The app has been built highly modular and with certain extensions in mind. *LiveTrack* already has plans for further development of the app.

Figure 2: A screen of the final result. On this screen, the technician is about to select the container using the QR code glued to the container. This is the third step of the workflow "Install Sensor".

Declaration of Authorship

We hereby declare that this thesis and the work presented in it is our own, original work. All the sources we consulted and cited are clearly attributed. We have acknowledged all main sources of help.

Location, Date: Rapperswil, 14. June 2019

Cyril Kyburz .....

Sandro Scheiwiller .....

Contents

1 Analysis **1**
- 1.1 Initial Situation and Goal . . . 1
- 1.2 Use-Cases . . . 3
- 1.3 Domain Model . . . 6
- 1.4 Functional Requirements . . . 8
- 1.5 Non-Functional Requirements . . . 8

2 Architecture **14**
- 2.1 Context and Scope . . . 14
- 2.2 Deployment View . . . 17
- 2.3 Class Diagram . . . 19
- 2.4 Evaluation of Technology . . . 22

3 User Interface and User Experience **24**
- 3.1 Wireframes . . . 24
- 3.2 Mockups . . . 26
- 3.3 Prototype . . . 34
- 3.4 Internationalization . . . 34
- 3.5 Usability Test . . . 35
- 3.6 Result . . . 37

4 Implementation **38**
- 4.1 Building Block View . . . 38
- 4.2 Runtime View . . . 45

5 Testing **49**
- 5.1 Unit Testing . . . 49
- 5.2 Integration Testing . . . 49
- 5.3 End to End Testing . . . 49
- 5.4 Functional Testing . . . 53

6 Results **54**
- 6.1 Achieved Goals . . . 54
- 6.2 Possible Extensions . . . 55
- 6.3 Outlook . . . 56
- 6.4 Acknowledgments . . . 56

Appendices **59**

A Glossary **60**

B Development Concepts **61**
B.1 Documentation 61
B.2 Project Management Workflow 61
B.3 Code Control System 62
B.4 Development Workflow 62
B.5 Development Mindset 63
B.6 Continuous Integration 63

C Milestones **65**

D Time Evaluation **66**

E Personal Reports **68**
E.1 Cyril Kyburz 68
E.2 Sandro Scheiwiller 68

F Task Definition **69**

G Copyright and Usage Agreement **72**

H Protocols **73**

1 Analysis

This chapter explains the starting situation and the goal of this thesis. In addition, the requirements for achieving the objectives are specified.

1.1 Initial Situation and Goal

Our industry partner, *LiveTrack*, wants to monitor the filling level of recycling containers using sensors, which regularly transmit the current filling level to the cloud. These sensors are installed and removed by a technician performing one of the following steps:

- **INSTALL** sensors into containers.
- **REPLACE** sensors with new ones (e.g. if the battery dies).
- **REMOVE** sensors (e.g. if a sensor test is over).

In order to map the filling level sent by the sensor to the corresponding container, *LiveTrack* needs to know which sensor is installed in which container. We decided to implement a workflow for each procedure. This allowed us to make the processes more user-friendly than would otherwise have been possible.

From the technician's procedures we arrived at the following workflows *(p* stands for the physical work)*:

- **Workflow #1: Install Sensor**
  1. Select the sensor that is about to be installed.
  2. Confirm sensor selection.
  3. Select the container in which the sensor is to be installed.
  4. Confirm container selection.

  p* *Install the sensor.*

  5. Check the sensor and container selection, add notes (and/or images) and confirm the installation.
- **Workflow #2: Replace Sensor**

  p* *Remove the old sensor*

  1. Select the sensor that has been removed.
  2. Confirm first sensor selection.
  3. Select the new sensor, which is about to be installed.
  4. Confirm second sensor selection.

  p* *Install the new sensor.*

  5. Check both sensor selections, add notes (a required comment and optionally images) and confirm the replacement.

- **Workflow #3: Remove Sensor**

  p* *Remove the sensor.*

  1. Select the sensor that has been removed.
  2. Check sensor selection, add notes (a required comment and optionally images) and confirm the removal.

The decision was made to enable the selection of sensors and containers with QR codes. Sensors and containers must be tagged as a prerequisite before sensors can be installed. *LiveTrack* decided to tag the sensors upon their arrival and have the technician tag the containers. The containers should also be tagged before any sensors are installed, a process designed to reduce the potential for errors. In order to cover this additional requirement, we created another workflow:

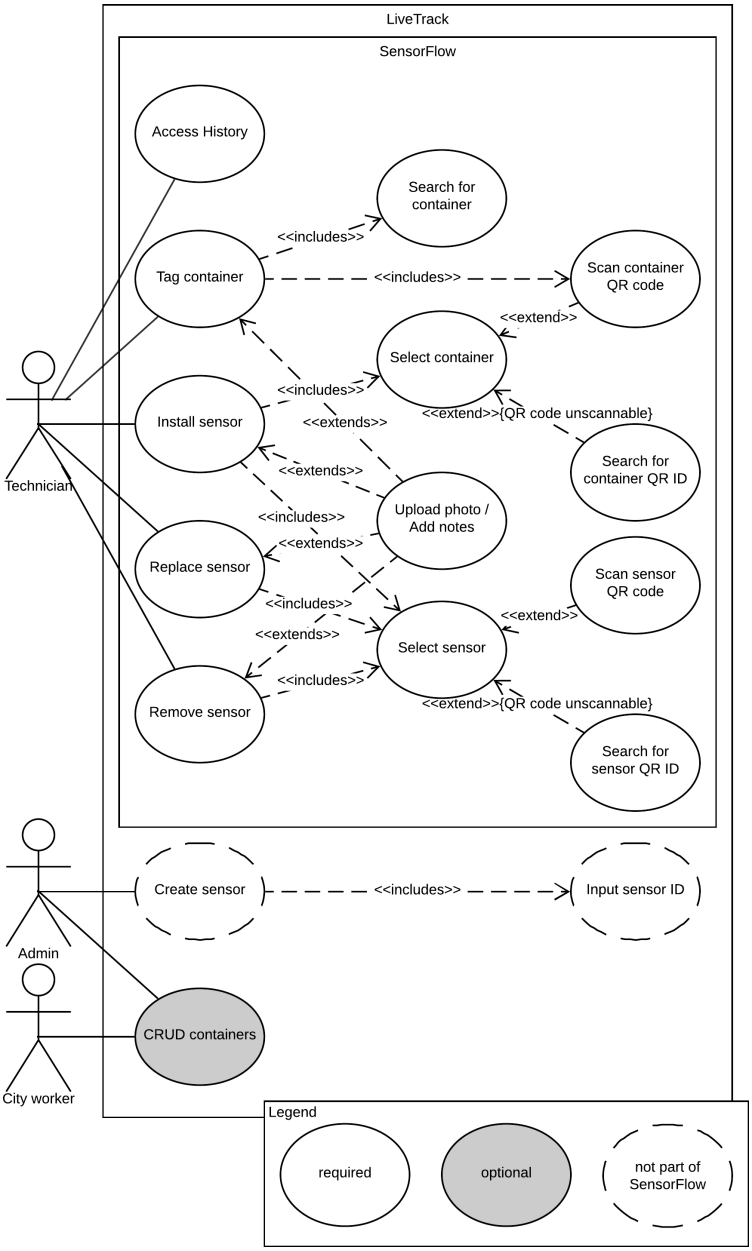
- **Workflow #4: Tag Container**

  p* *Clean area on container and tag container using a new QR code.*

  1. Scan the QR code that has just been stuck to the container.
  2. Select the container group where the container is at (containers are grouped by place).
  3. Select the container of that group (using the description, which might be something like "northernmost green container").
  4. Check the QR code and the container selection, add notes (a comment and/or images) and confirm the tagging of the container.

1.2 Use-Cases

Use cases list the possible interaction between systems and users. In the diagram below use cases are used to clarify the actors, their possible actions and which actions are part of another action.



1.2.1 Actors

1. **Technician**
   The technician is sent by *LiveTrack* to install, replace or remove sensors, and tracks this work using the previously defined workflows. In addition, the technician can view the history of recent actions that have been performed.

2. **Admin**
   The *LiveTrack* admin will tag the sensors upon their arrival and add a record to the system for each of them.

3. **City worker**
   The city worker is responsible for recording the containers of the respective city.

1.2.2 Use Cases Brief

1. **CRUD containers**
   The admin and the city worker may create, read, update and delete (CRUD) container entities. The city worker's access is limited to the containers of his municipality.

2. **Create sensor**
   The admin creates a sensor entity for each real sensor and links the sensor (identified with its ID) with the QR code that was put onto it.

3. **Input sensor ID**
   For each sensor entity which is created by the admin the sensor ID is also inputted.

4. **Search for container**
   The technician identifies the container using only the data provided by the city worker. This will only be used while tagging the container. Afterwards the container will be identified using the QR code.

5. **Scan container QR code**
   The technician can scan the QR code (added during *Tag container*) to identify a container. The QR code is also scanned in the process of tagging the container.

6. **Tag container**
   The technician will, when visiting a container for the first time, add the QR code and link it to the container entity created by the city worker.

7. **Search for container QR ID**
   If the QR code is unreadable or the camera doesn't work, the technician can also enter the container QR ID, which is printed next to the QR code, to identify the container.

8. **Select container**
   A container can be selected either by scanning the QR code or by inputting the QR ID.

9. **Scan sensor QR code**
   The technician can scan the QR code (added by the admin during *Create sensor*) to identify a sensor.

10. **Search for sensor QR ID**
    If the QR code is unreadable or the camera doesn't work, the technician can also enter the sensor QR ID, which is printed next to the QR code, to identify the sensor.

11. **Select sensor**
    A sensor can be selected either by scanning the QR code or by inputting the QR ID.

12. **Upload photo / Add notes**
    At the end of each task (*Install sensor*, *Replace sensor* and *Remove sensor*), a description and photos can be added as documentation.

13. **Install sensor**
    When installing a sensor, the technician selects the sensor and the container in which it is to be installed.

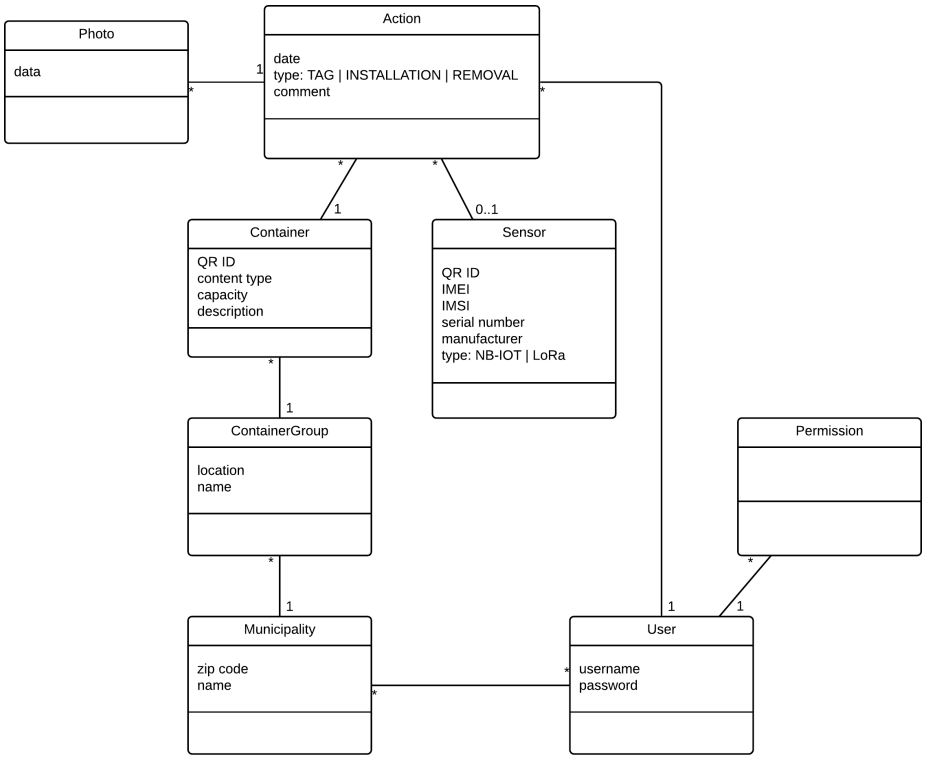
14. **Replace sensor**
    When replacing a sensor, the technician selects the sensor being replaced and the sensor it's being replaced with (old and new sensor).

15. **Remove sensor**
    When removing a sensor, the technician selects the sensor being removed.

16. **Access History**
    The most recently performed actions of a technician are displayed in the history.

1.3 Domain Model

We use the domain model to get an overview of the entire domain, its vocabulary, objects and associations. Additionally it can be used to communicate with non-technical stakeholders.



1.3.1 Objects

Container The container could in theory be any vessel that can be filled and monitored. In practice, however, *LiveTrack* has so far limited itself to recycling containers, which is very evident in our work, e.g. the tests and images are all based on recycling containers. Because the app should later support any type of container nothing has been made recycling-specific.

Sensor A sensor that measures the distance from the contents to itself, as seen in Figure 3. This measurement is then sent to the *LiveTrack* system, where the the filling level of the container is calculated using its total height and the received measurement.

Container Group Most containers are grouped with other containers located at the same location, resulting in a container group.

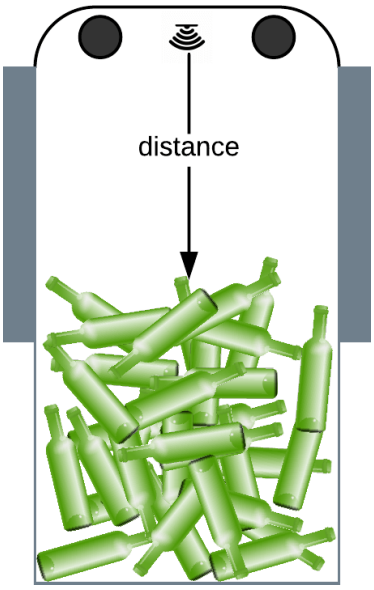


Figure 3: An open view inside the container: The sensor measures the distance from the top of the container to the contents.

Municipality A municipality represents a city, a town or a village. Each municipality will populate the *LiveTrack* system with its containers. The municipality is also used to determine access rights.

User A user is a person who uses *SensorFlow*. Most users will be technicians. In addition, *LiveTrack* administrators and developers who develop and test the app are also users.

Permissions Permissions are used to authorize certain actions for users.

Action An action represents a completed procedure with a container, often including a sensor. There are three actions:

1. **TAG**: Putting a QR code on a container
   Used for the *Tag Container* procedure
2. **INSTALLATION** Installing a sensor into a container
   Used for the *Install Sensor* and *Replace Sensor* procedures
3. **REMOVAL** Removing a sensor from a container
   Used for the *Replace Sensor* and *Remove Sensor* procedures

Because every completed procedure is stored, the whole history, instead of just the current status, is saved.

The user can add a comment and photos at the end of each procedure, documenting anything that might seem noteworthy.

The *Replace Sensor* procedure is the only procedure not directly represented by just one action. Instead, it is modelled as a *REMOVAL* and an *INSTALLATION* action. If the user adds a comment or photos during the *Replace Sensor* procedure, they are added to the *INSTALLATION* action.

Photo Photos can be added at the end of each procedure as additionial documentation.

1.3.2 Entity Creating Processes

Preconditions for SensorFlow

- *LiveTrack* adds the **municipalities** which want to enable tracking capabilities for their containers.
- *LiveTrack* adds the **users** of SensorFlow (i.e. creates accounts).
- *LiveTrack* adds the **sensors**.
- *A municipality* adds its **containers** along with their descriptions, grouped into **container groups**.

SensorFlow

- *A technician* tags a container, creating an **action of type TAG**.
- *A technician* installs a sensor, creating an **action of type INSTALL**.
- *A technician* replaces a sensor with another sensor, creating an **action of type INSTALL** and an **action of type REMOVAL**.
- *A technician* removes a sensor, creating an **action of type REMOVAL**.

1.4 Functional Requirements

The functional requirements are as follows:

- **Required:**
  An application that implements all use cases defined as "required" in the use case diagram in chapter 1.2.
- **Optional:**
  An application that implements the use case defined as "optional" in the use case diagram in chapter 1.2.

1.5 Non-Functional Requirements

The following non-functional requirements were agreed upon.

1.5.1 Scalability

The application should be scalable. A web application is usually only as scalable as its weakest link. For example, imagine if the login functionality of a web application were not scalable. As the user base grows, more and more of them want to log in until the server that hosts the login functionality is overloaded. Suddenly some users wont be able to log in, leading to frustration on the user's side. So in order to make a web application scalable we need to ensure that every single component of it is scalable. Our web app breaks down into the following components:

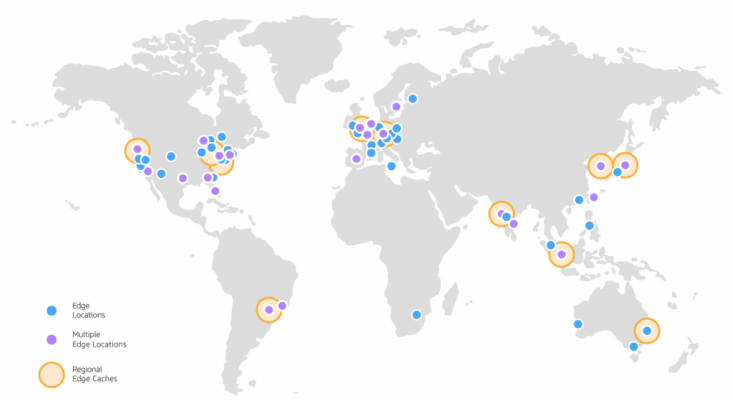
- **Delivery**
  The delivery of the static files of the web application, including HTML, CSS, JavaScript and images.
- **Authentication**
  The API used to authenticate users.
- ***LiveTrack* API**
  The API by *LiveTrack*.

Delivery We use *Amazon S3* to store and *Amazon CloudFront* to deliver the *SensorFlow* web application. According to their product page [1], this pairing is a perfect match:

> Storing your static content with S3 provides a lot of advantages. But to help optimize your application's performance and security while effectively managing cost, we recommend that you also set up Amazon CloudFront to work with your S3 bucket to serve and protect the content. CloudFront is a content delivery network (CDN) service that delivers static and dynamic web content, video streams, and APIs around the world, securely and at scale. By design, delivering data out of CloudFront can be more cost effective than delivering it from S3 directly to your users.

Delivering static assets is very scalable because the web server which handles the request does not have to have any internal state. Therefore these servers can be distributed anywhere in the world and deliver the same assets without having to be synchronized with each other. This is exactly the premise of *Amazon CloudFront* [1]:

> CloudFront serves content through a worldwide network of data centers called Edge Locations. Using edge servers to cache and serve content improves performance by providing content closer to where viewers are located. CloudFront has edge servers in locations all around the world, as you can see from the following map:



> When a user requests content that you serve with CloudFront, their request is routed to a nearby Edge Location. If CloudFront has a cached copy of the requested file, CloudFront delivers it to the user, providing a fast (low-latency) response. If the file they've requested isn't yet cached, CloudFront retrieves it from your origin – for example, the S3 bucket where you've stored your content. Then, for the next local request for the same content, it's already cached nearby and can be served immediately.

Authentication *SensorFlow*'s authentication step is managed by the *Amazon Cognito* service. *Amazon Cognito* is also advertised as scalable [2]:

> **Secure and scalable user directory**
>
> Amazon Cognito User Pools provide a secure user directory that scales to hundreds of millions of users. As a fully managed service, User Pools are easy to set up without any worries about standing up server infrastructure.

***LiveTrack* API** The *LiveTrack* API is based on *Pimcore*, which executes the API functions and handles data storage. The current *Pimcore* instance is running on one *Amazon EC2* server instance. Since the processing power of the instance can be increased, the API is horizontally scalable. It would also be possible to make the API vertically scalable as well, but so far this is not planned, as one instance will suffice according to *LiveTrack*.

1.5.2 Security

The entire application should be secure from attackers. Again, the whole application is only as secure as its weakest part. We have paid particular attention to security in the following areas:

- **Communication**
- **Authentication**
- **XSS Attack Threat**

We will now go into more detail on each of these points.

Communication Every communication channel used by *SensorFlow*, as seen in chapter 2.2, is secured using SSL.

Authentication Using *Amazon Cognito*, we achieve high authentication standards. As Amazon states on *Amazon Cognito*'s product page [2]:

> **Standards-based authentication**
>
> Amazon Cognito User Pools is a standards-based Identity Provider and supports identity and access management standards, such as Oauth 2.0, SAML 2.0, and OpenID Connect.
>
> *[...]*
>
> **Security for your apps and users**
>
> Amazon Cognito supports multi-factor authentication and encryption of data-at-rest and in-transit. Amazon Cognito is HIPAA eligible and PCI DSS, SOC, ISO/IEC 27001, ISO/IEC 27017, ISO/IEC 27018, and ISO 9001 compliant.

XSS Attack Threat One specific threat when developing web apps is Cross-Site Scripting (XSS). The definition of OWASP reads [3]:

> Cross-Site Scripting (XSS) attacks occur when:
>
> 1. Data enters a Web application through an untrusted source, most frequently a web request.
> 2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.
>
> The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information,

to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

To avoid vulnerabilities that could enable a XSS attack, we only trust validated data and took the following security measures:

- **Writing to the DOM**
  Since the DOM is rendered with *React*, most variables that are rendered are escaped. Rendering unescaped HTML is almost only possible using "dangerouslySetInnerHTML" [4]:

  > dangerouslySetInnerHTML is React's replacement for using innerHTML in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out dangerouslySetInnerHTML and pass an object with a __html key, to remind yourself that it's dangerous. For example:

```
1 function createMarkup() {
2     return {__html: 'First &middot; Second'};
3 }
4
5 function MyComponent() {
6   return <div dangerouslySetInnerHTML={createMarkup()}
        />;
7 }
```

- **Reading the URL**
  Most routing is done using React Router [5] and the *SensorFlow* specific code of the app never reads the URL location. Only on one exception access to the lower level History API is needed: To make the back button work while inside the workflow. There, the pushState method is used to save the current location inside the state object [6]:

  > **state object** — The state object is a JavaScript object which is associated with the new history entry created by pushState(). Whenever the user navigates to the new state, a popstate event is fired, and the state property of the event contains a copy of the history entry's state object.

  Since the state object is never written to nor read from the URL it can not be set by an attacker, making this part of the code resilient to XSS attacks.

1.5.3 Browser Compatibility

Supported are the current versions of Chrome and Safari, since we expect the users to access the application on *iOS* or *Android* smartphones. At that time these were version 73 for Chrome and version 12.1 for Safari.

1.5.4 Partial Offline Support

Since the app should feel just like a native app, it should always be possible access it, something that would not be possible with a web site. However, actually *using* the app offline is not a requirement because of the additional complexity that would entail as explained in the following scenario, in which we would implement the full functionality to be available offline.

Scenario: Full Offline Functionality Whenever an action is completed, a check would have to be done whether the device is online. If the device were online, the action could be sent to the API right away, completing this action, but if the device were offline, the action would have to be cached until the device would be online again. This could happen multiple times while the device is offline. Since the data on the device would need to be up to date, the actions that happen on the server need to be handled locally as well. On reconnect, all cached actions would have to be sent again, in the right order. Now if any of these actions fail for any reason, all other actions might be based on the wrong state, resulting in additional failures. The technician would then need a screen to deal with these failures and re-apply the other updates, essentially a merge tool for the committed actions.

As a result of this complexity, the user experience would suffer enormously: While before the technician got instant feedback after confirming an action, that feedback could now appear a few hours later. Since merging functionality is usually very complex it is very hard to design for a mobile phone, where the available display area is limited and would probably introduce a lot of bugs. Also, since we would assume that the device is usually online, this would be a special case, further limiting the usefulness of such a feature.

Current Solution: Partial Offline Support The code of the *SensorFlow* application is cached using a service worker. This enables the app to be started whenever, just like a native app. Local data is cached for eight hours, but synced with the *LiveTrack* system after any action workflow is finished. When confirming an action (e.g. Install Sensor) the client pushes to the server immediately, instead of waiting for an online connection, giving the user instant feedback about the action. In the future, this base offline functionality could also be extended.

2 Architecture

This chapter describes the architecture of *SensorFlow* and provides an overview of the entire *LiveTrack* system.

2.1 Context and Scope

This chapter explains the scope and responsibilities of the *SensorFlow* application and *LiveTrack*. In addition, the interface of the *LiveTrack* API is defined so that the communication can work properly. Figure 4 shows the system diagram of the application.

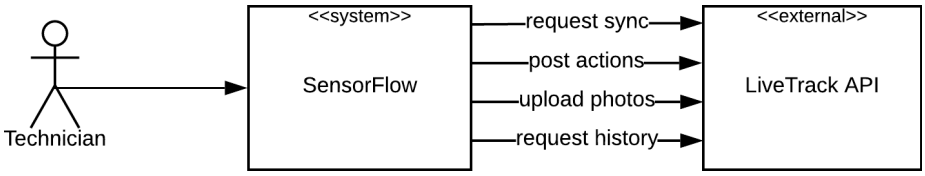


Figure 4: This system diagram shows how the technician uses the *SensorFlow* application. *SensorFlow* uses the *LiveTrack* API to sync the server data to its local storage, post actions (e.g. *Install Sensor*), upload photos and request the history.

2.1.1 Parties Involved

SensorFlow An application providing the workflows which are described in chapter 1.1. The app is the user interface for the *LiveTrack* API.

***LiveTrack* API** *LiveTrack* plans to have multiple applications, *SensorFlow* being one of them, powered by the same data. Therefore, they are planning a central database capable of representing the domain model (Figure 3). Direct access to a database from multiple systems is a bad idea, so instead an API was developed together. The *SensorFlow* team specifies the API definition required for our application and *LiveTrack* implements the API accordingly.

2.1.2 API Definition

We created a definition with *Swagger* [7] following the OpenAPI [8] specification version 3. Based on the *Swagger* file it is possible to simulate API requests and validate requests and responses as seen in Figure 5.

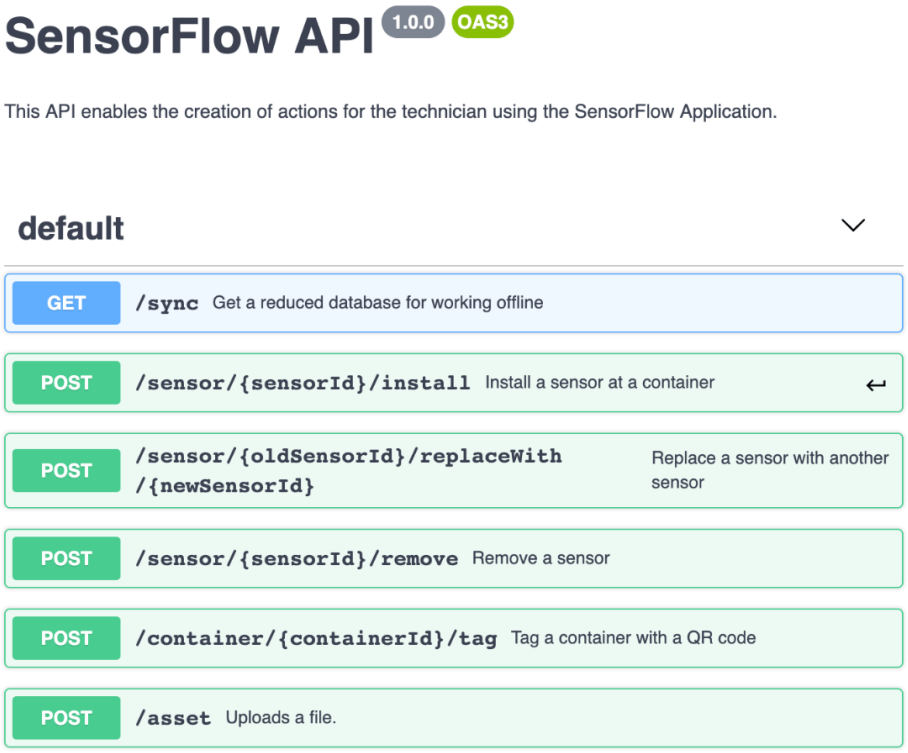


Figure 5: Screenshot of the *Swagger* interface based on the definition displaying all endpoints.

Based on our proposed API definition, *LiveTrack* implemented a very similar API. Some changes were necessary due to the technologies they use. The final result can be seen in Figure 6 or on the web: https://www.live-track.ch/api-doc.

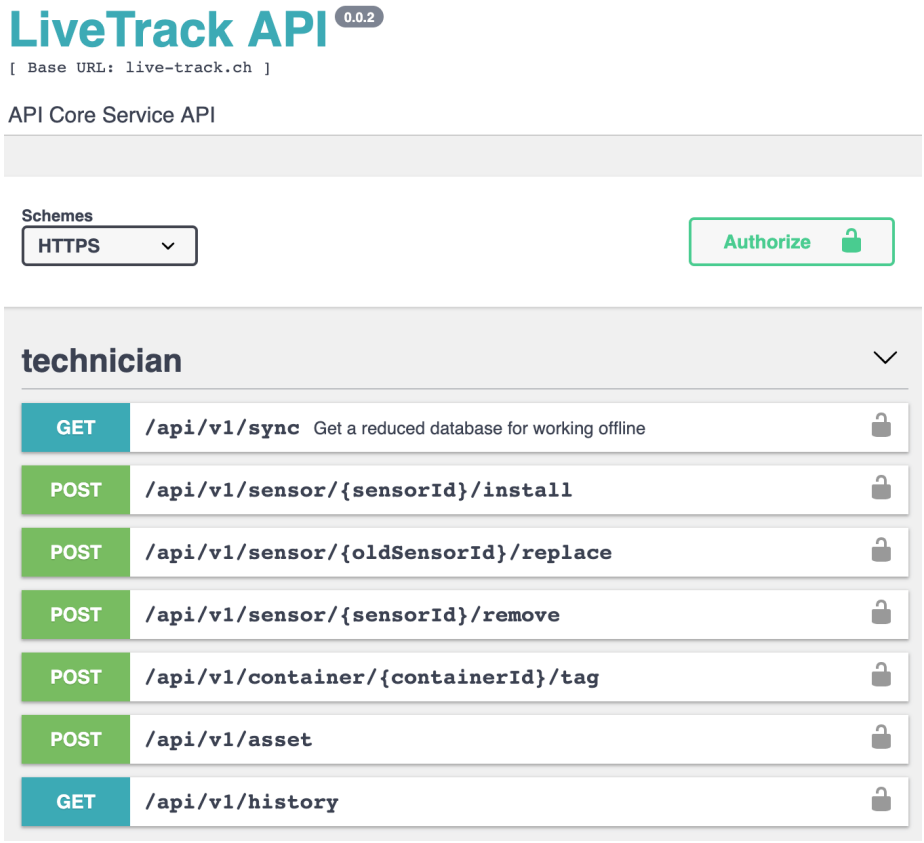


Figure 6: Screenshot of the *LiveTrack* API definition based on our definition, generated with NelmioApiDocBundle[9].

2.2 Deployment View

The system shown in Figure 7 contains four parts: The app (*SensorFlow*), a backend (*LiveTrack* API), the *Amazon Cognito* API and the *Amazon S3* photo storage. In the following chapters they are explained in detail.

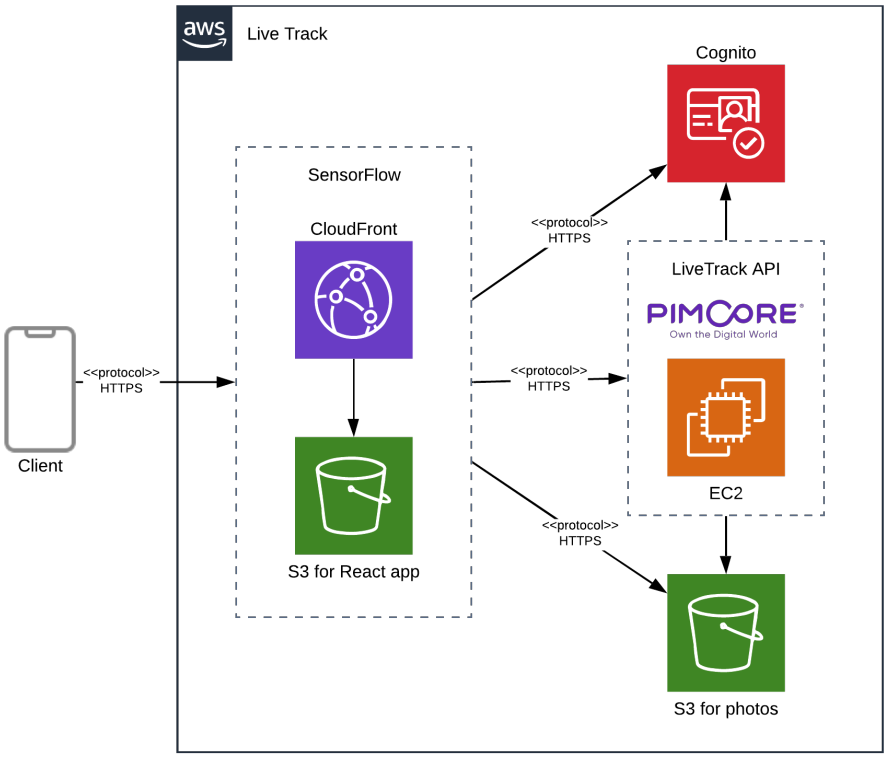


Figure 7: The deployment diagram presents a clear view of the entire infrastructure. Because *LiveTrack* mostly relies on IaaS and PaaS to host its applications, the hardware is abstracted using *Amazon Web Services* (AWS). The *LiveTrack* API is a *Pimcore* instance, which is running on an *Amazon EC2* instance. The static files that make up *SensorFlow* are deployed to an *Amazon S3* bucket and delivered using *Amazon CloudFront*. The communication protocol to and from *SensorFlow* is always HTTPS.

2.2.1 SensorFlow

SensorFlow has two deployed instances, one for developing and one for production. The app is deployed to an *Amazon S3* bucket and delivered using *Amazon CloudFront*.

Amazon S3 Amazon's *Simple Storage Service* is an object storage service to which the app gets deployed to. For every deployment the *S3* bucket contents are overwritten by the new version. The version history functionality of the bucket is not activated because every release is stored in *GitHub*.

Amazon CloudFront *CloudFront* is a content delivery network (CDN) that delivers the app with low latency and high transfer speeds from all over the world. It also allows us to use our own domains with SSL certificates, which were created with the *AWS Certificate Manager*. Our *CloudFront* distribution origin is set to the *S3* bucket where the app is deployed to.

2.2.2 LiveTrack API

The backend of the app is provided by *LiveTrack* as a HTTP API. The API is not RESTful but heavily inspired by REST and uses JSON as data exchange format. The API is part of the *Pimcore* service used by *LiveTrack*, which is also deployed in the *AWS Cloud*, namely an *Amazon EC2* instance.

2.2.3 Cognito

This instance of *Amazon Cognito* is used to authenticate users in multiple *LiveTrack* applications, eliminating the need to create accounts for each application.

2.2.4 Amazon S3 for Photos

The *LiveTrack* API stores the images uploaded by the app on an *Amazon S3* bucket. The app then requests *S3* to deliver the uploaded images.

2.3 Class Diagram

The class diagram is strongly influenced by the domain model (shown in chapter 1.3), but limited to the subset of the domain model that was required and implemented in the application. If necessary, the domain model can be used to explain the names of the different entities. Regarding the type information, the common TypeScript types were used. The biggest differences between the class diagram and the domain model are as follows:

- Because actions are created on the server after a workflow action has been submitted successfully they are not implemented in *SensorFlow*.
- The permissions are also not required from the app's point of view, the *LiveTrack* API implicitly returns only the sensors and the container the user has access to.
- The attributes of the municipality are inlined directly into the container group.
- Sensors and containers are linked after loading the data, resulting in a structure in which objects are linked by reference and ID (e.g. sensor stores a containerId and a reference to a container).

A sync object The sync object returned from the *LiveTrack* API provides all remote data used by the covered classes in the class diagram. A shortened object can be seen in Figure 9.

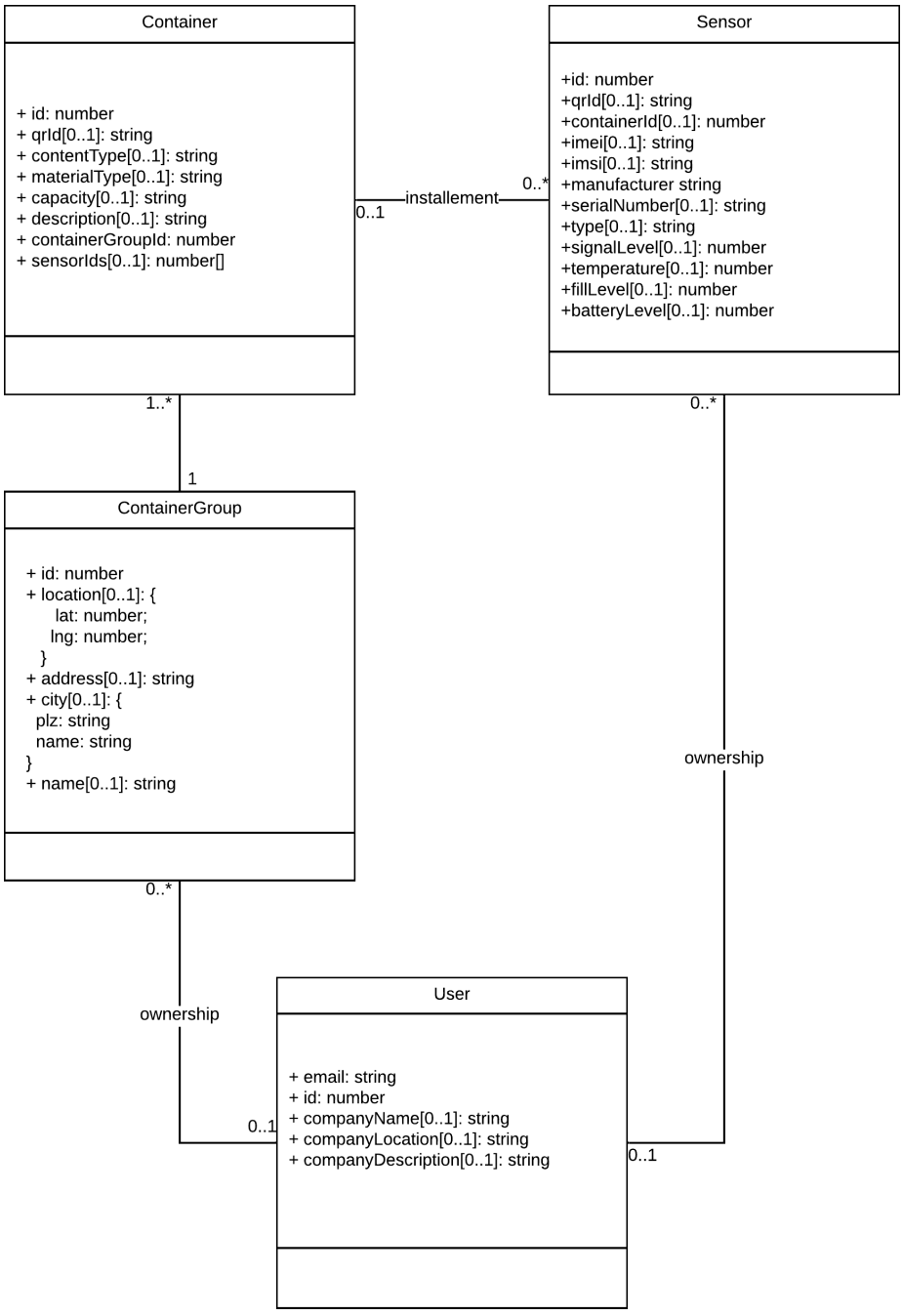


Figure 8: A class diagram showing the TypeScript classes of *SensorFlow*, their connections to each other, and their attributes with type information. The "+" in front of the attributes indicates a public attribute, and the "[0..1]" after the name of some attributes indicate that these attributes are optional.

```
{
  "success": true,
  "data": {
    "containerGroups": [
      {
        "containers": [
          {
            "key": "green-1",
            "id": 110,
            "ownerId": 37,
            "qrId": "c395ba9a2a3b0f6b2bfd83ad41d63717",
            "name": "Green 1",
            "contentType": "Gr\u00fcnglas",
            "materialType": 46,
            "capacity": "3.15375m\u00b3",
            "description": "Erster Gr\u00fcnglas Container",
            "containerGroup": 90,
            "containerType": 50,
            "sensors": [ 166 ],
            "fillLevel": 0.25
          }
        ],
        "key": "gondiswil-gondiswil-genaue-position-unklar",
        "id": 90,
        "ownerId": 37,
        "location": {
          "lat": 47.147281985731,
          "lng": 7.872210284695
        },
        "address": "Hinterdorf",
        "plz": "4955",
        "city": "Gondiswil",
        "containerIds": [ 110 ],
        "name": "Gondiswil Hinterdorf"
      }
    ],
    "sensors": [
      {
        "key": "nb000052-000522",
        "id": 195,
        "ownerId": 37,
        "qrId": "12201e6dac7822533f483e1e237a01e3",
        "imei": "357518080337082",
        "imsi": null,
        "serialNumber": "NB52-522",
        "container": 118,
        "type": null,
        "manufacturer": "netBin",
        "hid": "02365dc7cd689058",
        "signalLevel": "-97",
        "temperature": "28",
        "fillLevel": 0.014,
        "batteryLevel": "94"
      }
    ],
    "current_user": {
      "email": "ckyburz@hsr.ch",
      "id": 35,
      "companyName": null,
      "companyLocation": null,
      "companyDescription": null
    },
    "other_users": [...]
  }
}
```


Figure 9: A shortened JSON response of the *LiveTrack API* sync endpoint. Each response is an object containing a "success" attribute, which specifies whether the call was successful. If that is the case, a data attribute containing the sync data is added. in this example it has been shorted to one container group (id 90) with one container (id 110) and one sensor (id 195).

2.4 Evaluation of Technology

2.4.1 *Amazon S3* and *Amazon CloudFront*

LiveTrack already uses *AWS* infrastructure. Furthermore, an *Amazon S3* and *Amazon CloudFront* setup has the following benefits:

- no provisioning or managing servers.
- includes state of the art security with custom SSL certificate.
- code is delivered from the nearest server.
- scales automatically.

2.4.2 TypeScript

TypeScript is a strict syntactical superset of JavaScript, and adds optional static typing to the language. To run, it is first compiled to JavaScript. During the compilation the compiler type checks the program.

- *In connection with* the programming language
- *considering* past experience
- *we have decided to* use TypeScript
- *and against* the use of any other language
- *to* be able to start and develop quickly
- *for this we accept as consequence* that we have to define typings for libraries which do not support TypeScript from scratch.

2.4.3 React

React is a JavaScript library for creating user interfaces. It is an open source project from Facebook and is maintained in collaboration with the developer community.

- *In connection with* building the user interface
- *considering* our expertise, the interactive UI and native features
- *we made a decision for* React
- *and against* two native applications for iOS and Android
- *to* be able to have the best of both worlds (regular web page and mobile application)
- *because of this we accept as consequence* that progressive web applications are still in their infancy and there may be some challenges.

2.4.4 React Context

React context[10] is described on their page as follows:

> Context provides a way to pass data through the component tree without having to pass props down manually at every level.

- *In connection with* providing the *LiveTrack* API data to every component
- *considering* state of the art data handling
- *we made a decision for* React context
- *and against* unstated[11]
- *to* use a well maintained built-in API and have one dependency less
- *because of this we accept as consequence* to work with a less intuitive API.

2.4.5 Atlaskit

Atlaskit[12] is *Atlassian*'s official UI library built on top of React. It is a frontend framework providing pre-styled components like buttons, navigations, progress bars, and so on.

- *In connection with* a component library
- *considering* ease of use, performance and design
- *we made a decision for* Atlaskit
- *and against* Material-UI[13]
- *to* have a better performance
- *because of this we accept as consequence* to define our own typings for the library.

3 User Interface and User Experience

The goal was to develop an app that minimizes the error rate by maximizing usability. In order to achieve that, we invested in an extensive elaboration with multiple iterations to fine tune the user experience, using wireframes, mock-ups, a prototype and usability testing. Right from the start, we made sure to streamline the processes and design.

3.1 Wireframes

The basis for the wireframes are the workflows from chapter 1.1. In the first iteration wireframes were created to get a general overview. In the second iteration (shown in Figure 10) a first process through the app was proposed.

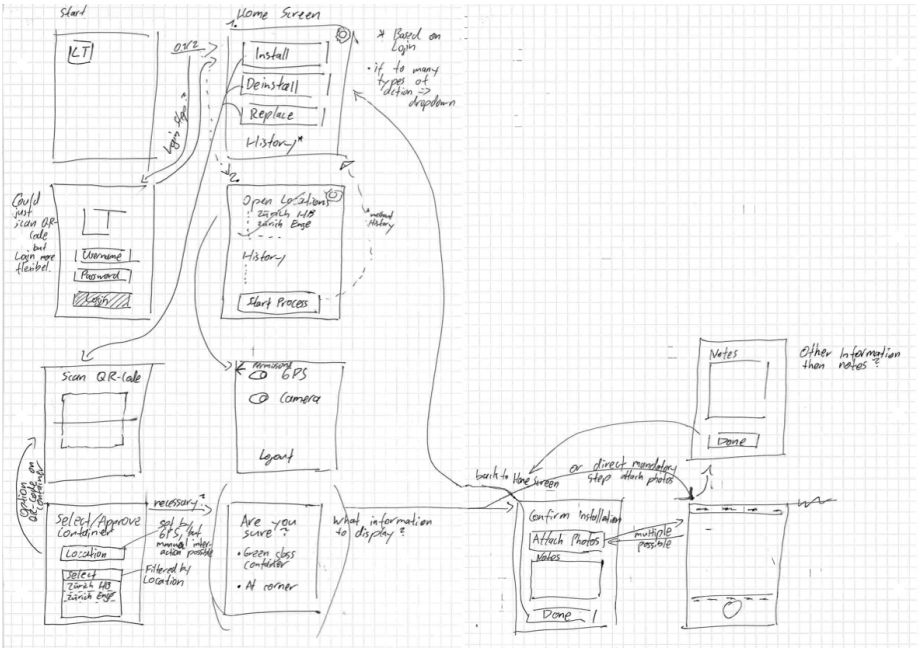


Figure 10: The second iteration of wireframes were created with a focus on the flow through the app.

3.1.1 Special Aspects

Communication The app communicates with notification provided via toasts (also called flags) and modals. Toasts are used for general information without the need for user interaction and do not block other interactions. Toasts appear at the bottom of the screen, as seen on the left side of Figure 11. They can be dismissed using a swipe gesture or disappear automatically after 8 seconds. When user feedback is needed, modals, which block any other interaction, are

used. The modal will also appear at the bottom of the screen, providing a mobile experience that is a consistent and easy to interact with. A modal is shown on the right side of Figure 11.

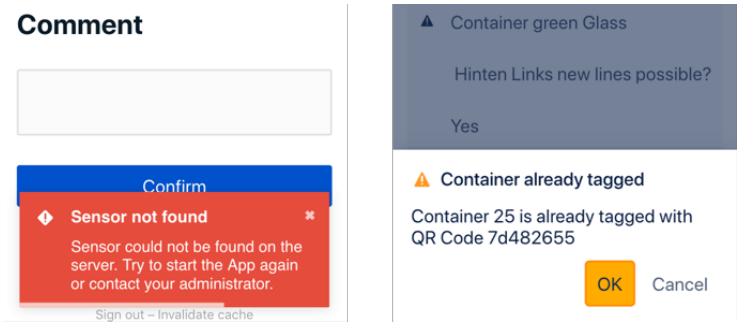


Figure 11: Left: A toast message, notifying the user that the sensor which was selected could not be found on the server.
Right: A modal, warning the user that the container has already been tagged. The user now needs to confirm his decision, all other interactions are blocked.

Login The entry point of the app is a login screen, preventing unauthorized access. It is deliberately not possible to create an account, as each technician is given an account in advance. Other features such as resetting a password in case one forgot it and email address confirmation are available.

Start Page / Home Page After logging on, a user lands on the start or home page, from which navigation to all workflows and other actions is possible. After each successful action (e.g. after successfully installing a sensor) the user is redirected back to this page.

Workflow At the beginning of each workflow, the user can scan a QR code. That way, the user always has the same entry point. As soon as the app recognizes a QR code, it continues if the validation is successful, otherwise the user is informed about the problem via toast. After each selection, a confirmation page is displayed to let the user confirm the chosen sensor or container. As final step of the workflow, a review screen summarizes the work done. If a mistake is detected, the user can jump back to a previous step and change the input. The progress will then be reset to the current step.

3.2 Mockups

Based on the wireframes we created mockups. Unlike most mockups, these are colorless, since *LiveTrack* has not yet defined a style guide. In Figure 12 an overview of all screens and the corresponding navigation between them is shown.

3.2.1 Navigation

To allow the user to navigate, links and automatic redirections that happen after successful actions should be used. The application should behave like a SPA (single page application), so the page should never be completely reloaded.

Browser Backward and Forward Navigation The app should support the current iOS and Android standards for back and forward navigation. The only exception should be when navigating back in a workflow and editing a step. This should result in all subsequent steps becoming invalid and the possibility of going forward being removed.

3.2.2 Communication

Every user interaction should be validated and the app should respond accordingly by providing feedback as early as possible. This will significantly reduce the error rate and the amount of cases where the user would get frustrated by reaching the end of a workflow without being able to finish it. In the event of an invalid user interaction, the feedback should enable the user to understand the situation and solve the problem.

3.2.3 General

In Figure 13 screens that are not related to any workflow are shown. After a successful login, the user should land on the home screen. From there, the user will be able to start an action, e.g. navigate to the history page where his last actions will be listed sorted by date. Another general screen could be the settings screen which would allow to change the permissions, if possible.

3.2.4 Workflow 1: Install Sensor

Figure 14 shows the detailed process of installing a sensor. Starting by scanning the QR code for a sensor or manually entering a QR code in case the reader is not able to scan it. After a successful scan, detailed information about the sensor is displayed to allow the technician to verify the selection. After the confirmation, the technician can scan and verify the container analogously. In the last step, the technician can do another review, comment (optional) or attach photos (optional) before finishing the workflow.

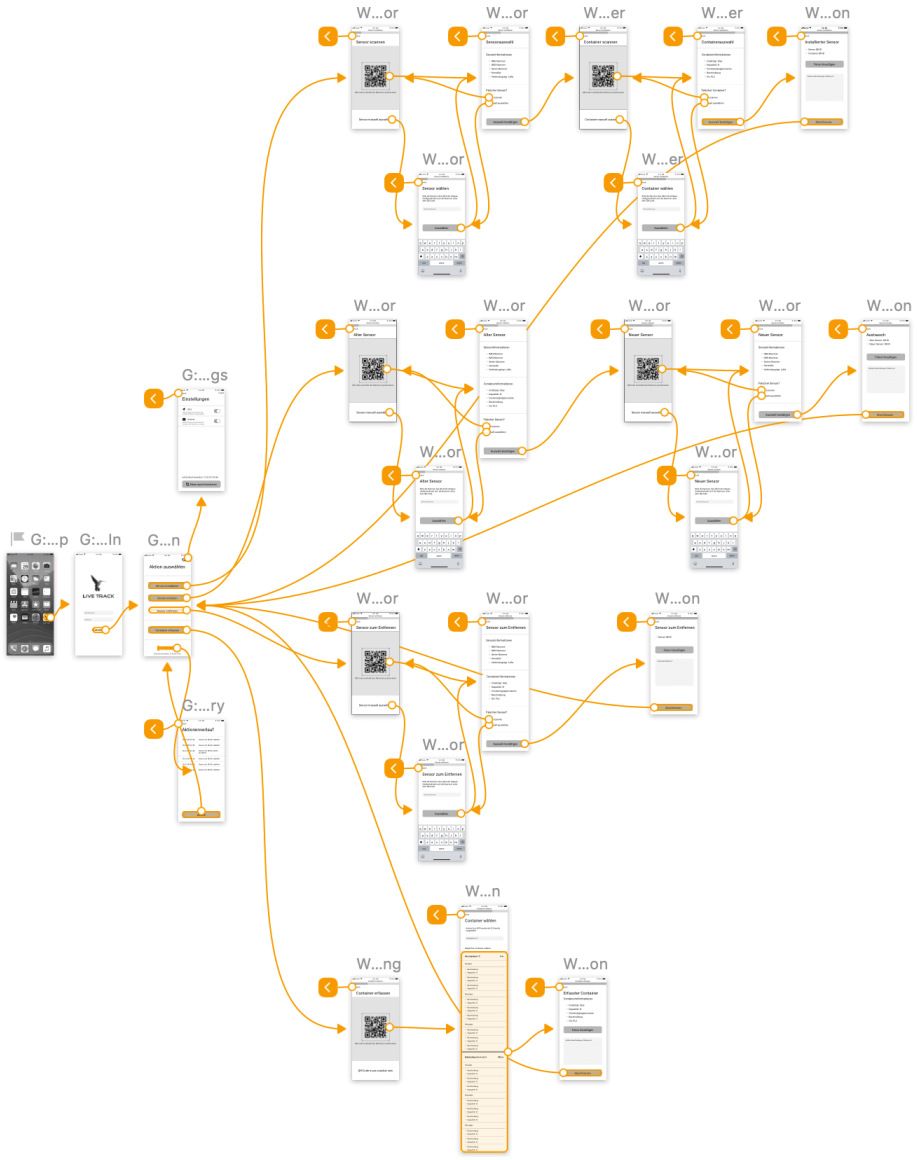


Figure 12: Overview of all screens including the navigation between them.

3.2.5 Workflow 2: Replace Sensor

This workflow in Figure 15 is almost identical to the first workflow. Instead of one container and one sensor, two sensors are selected, starting with the old sensor (must be installed) and moving on to the new sensor (must not be installed). Finally, a comment is required to ensure that the reason for the replacement is described.

3.2.6 Workflow 3: Remove Sensor

The last sensor related workflow deals with removing the sensor, as illustrated in Figure 16. This workflow is the shortest of all skipping the confirm screen and directly moving on to the last screen, since the verification refers to one sensor only. Just like in the replace workflow a comment is required.

3.2.7 Workflow 4: Tag Container

To be able to perform the workflows 1 – 3 the containers need to be tagged first, this process can be found in Figure 17. Like all the other workflows, this workflow starts with scanning, but with one big difference. The client side validation can not check whether the scanned QR code has been used already, since that would require the user to have access to all containers and sensors. If this rare case occurs, the user has to start again. After scanning a QR code, all available container groups are shown, sorted by distance to the technician using the GeoLocation API[14]. After selecting a container group, a user can select a container of that container group. In the last step the work can again be reviewed and commented on, just like in workflow #1.

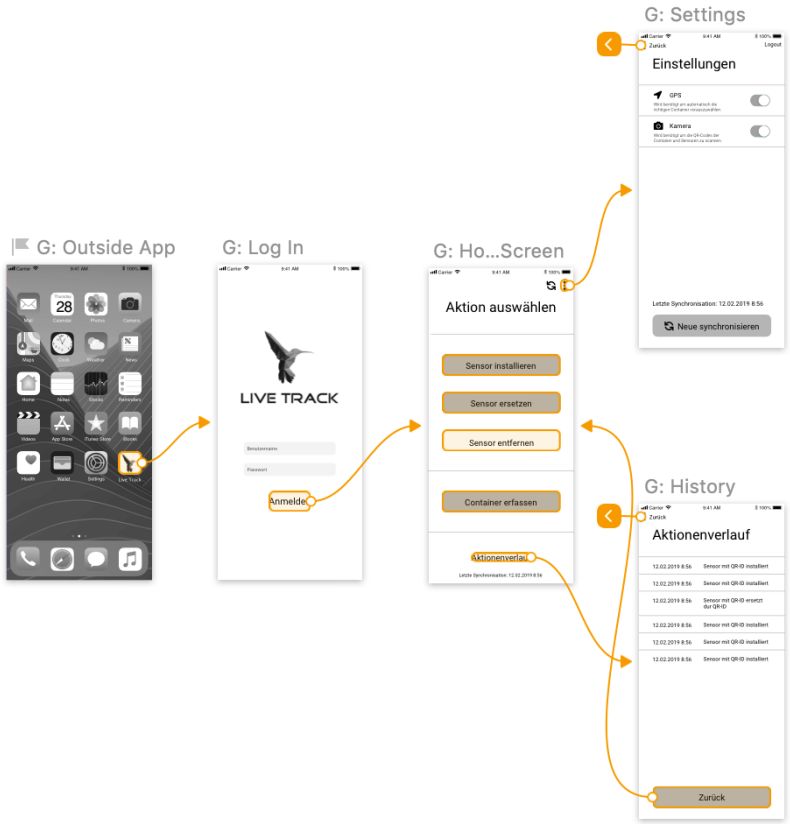


Figure 13: Entry of the app with the login and start screen followed by the settings and history screen.

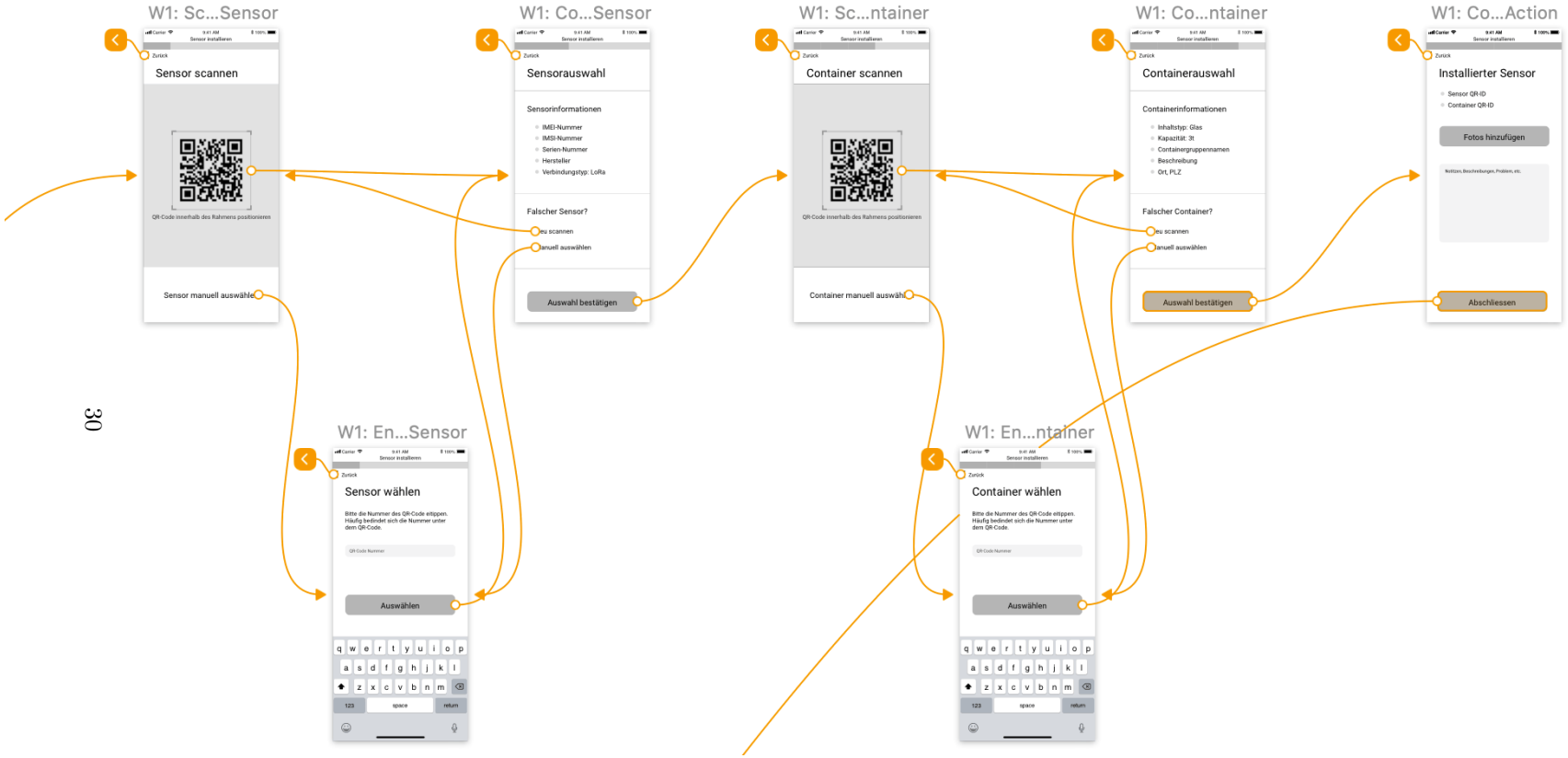


Figure 14: *Workflow 1: Install Sensor* consists of 3 parts, first the sensor is scanned or manually entered, then the same procedure with the container and finally the inputs can be checked and supplemented with photos or notes.

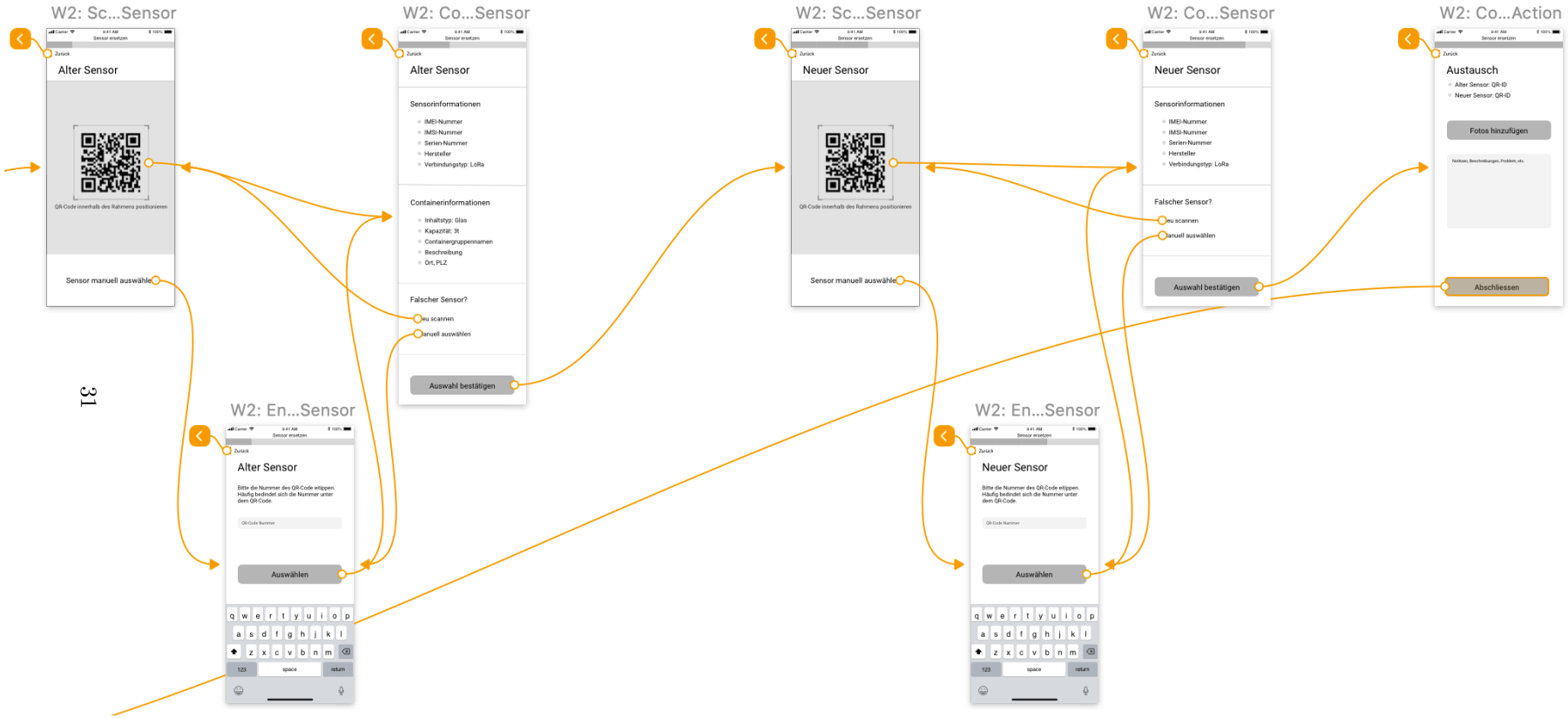


Figure 15: *Workflow 2: Replace Sensor* starts with scanning the old sensor and then follows the same procedure as installing a sensor, except that scanning a container is skipped because we already have the container information based on the old sensor.

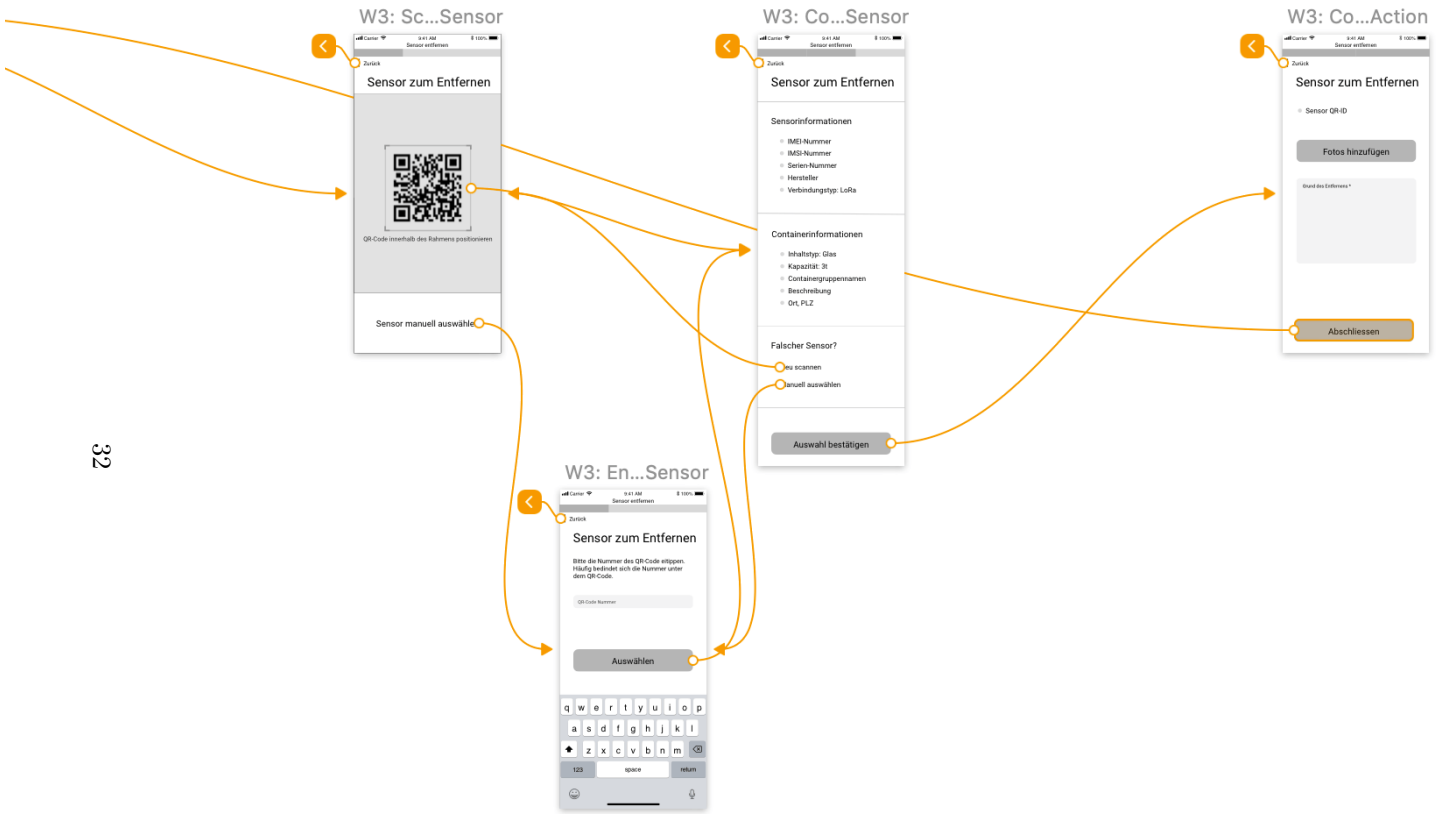


Figure 16: *Workflow 3: Remove Sensor* is the easiest workflow, the technician selects and confirms the target sensor, after that the action can already be reviewed.

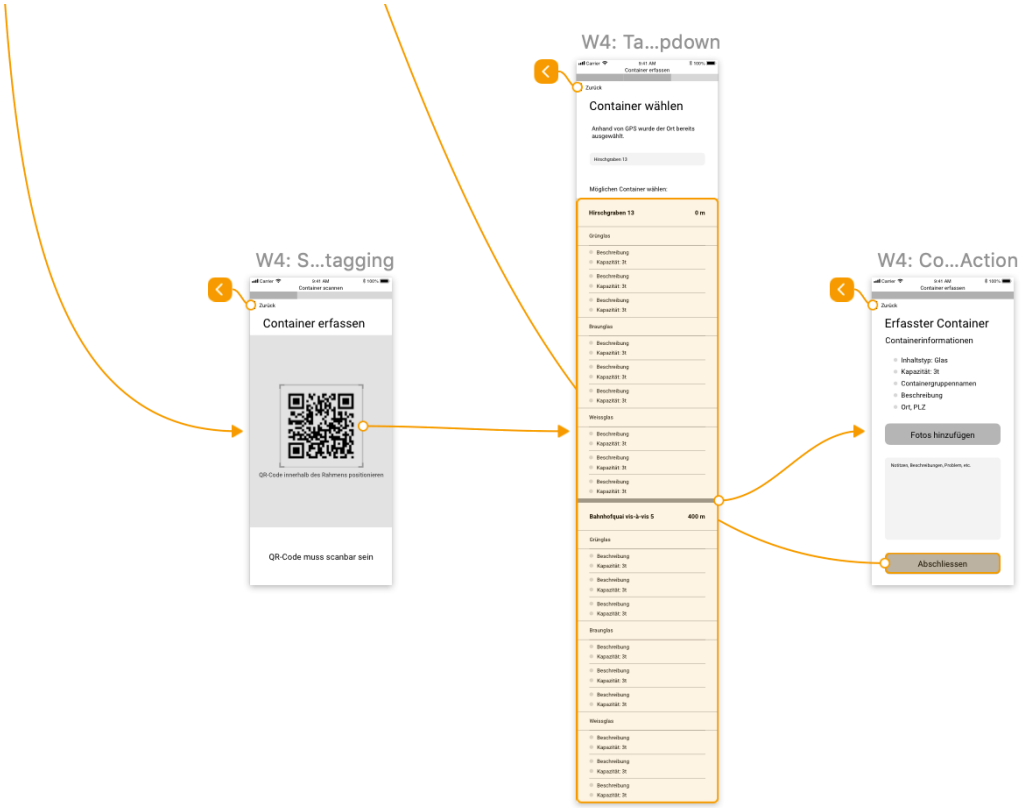


Figure 17: *Workflow 4: Tag Container* is the precondition for the installation of a sensor. At the beginning the technician scans the QR code followed by selecting the corresponding container. To select the container, the technician first selects the container group and then the container. Finally, the last step is the same as for the *Install Sensor* workflow.

3.3 Prototype

With the Mockups we realised a clickable prototype to test if the designed workflows were feeling smooth and to optimize them further. Our prototype can be found and tried on the web: https://sketch.cloud/s/ZdWG3/a/oJd9ej/play.

3.4 Internationalization

Since the app is developed in English, but a German translation is required, we decided to support both languages. If the browser indicates a preference for German or English, the app uses it, otherwise it falls back to German. The translations are located in a JSON file, an excerpt of which can be seen in Figure 18.

```
{
  "en": {
    "translation": {
      "installSensor": {
        "title": "Install Sensor",
        "successMessage": {
          "title": "Sensor installed",
          "description": "The sensor has been installed successfully"
        }
      }
    }
  },
  "de": {
    "translation": {
      "installSensor": {
        "title": "Installiere Sensor",
        "successMessage": {
          "title": "Sensor installiert",
          "description": "Der Sensor wurde erfolgreich installiert"
        }
      }
    }
  }
}
```

Figure 18: An excerpt of our JSON translation file showcasing the two supported languages. The system works by mapping keys to translated strings. This structure allows us to support any other language in the future without any code changes.

3.5 Usability Test

We did usability tests in Emmental with real containers and QR codes. Our industrial partner *LiveTrack* was surprised how fast and easy container tagging worked. The Figure 19 shows a usability test in full action. No big issues were discovered, because tests had already been made with the prototype.

Figure 19: Scanning the QR code of a freshly marked container.

3.5.1 Usability Tasks

Table 1 lists all usability tasks performed in Emmental. Not all of them could be performed because the sensors had a delivery delay. But all completed usability tasks had satisfactory results. The received feedback led to further optimizations of the app, specifically issues 2a, 2c and 3b were solved in the app and 2b is planned to be fixed on the server side. The remaining issues 3a and 3c will be addressed with new features that are not part of this thesis. They are described in the chapter 6.2 Possible Features.

Nr	Usability Task	Issues	Satisfied
1	Open the app and log in with the received account.	None	Yes
2	Choose a QR code and stick it on the container, then tag it. (workflow 4)	a Location is freezing (no UX problem)	Yes
		b Uploaded images are displayed with the wrong rotation	
		c Tagged and untagged containers are indistinguishable	
3	Take a sensor and install the sensor into a container. (workflow 1)	a When is the right time to finish the workflow, before or after the physical work	Yes
		b When a sensor is installed there is no light in the container and the QR reader does not work	
		c Stuck when the QR code gets removed by cleaning personnel	
4	Replace a sensor with another. (workflow 2)		Not performed
5	Remove a sensor. (workflow 3)		Not performed

Table 1: The performed usability tasks and their result.

3.6 Result

The result of our UX and UI investments are easy-to-use workflows. Most workflows can be completed with only three touch interactions. As an example the necessary touch interactions of the workflow *Tag Container* are shown in Figure 20.

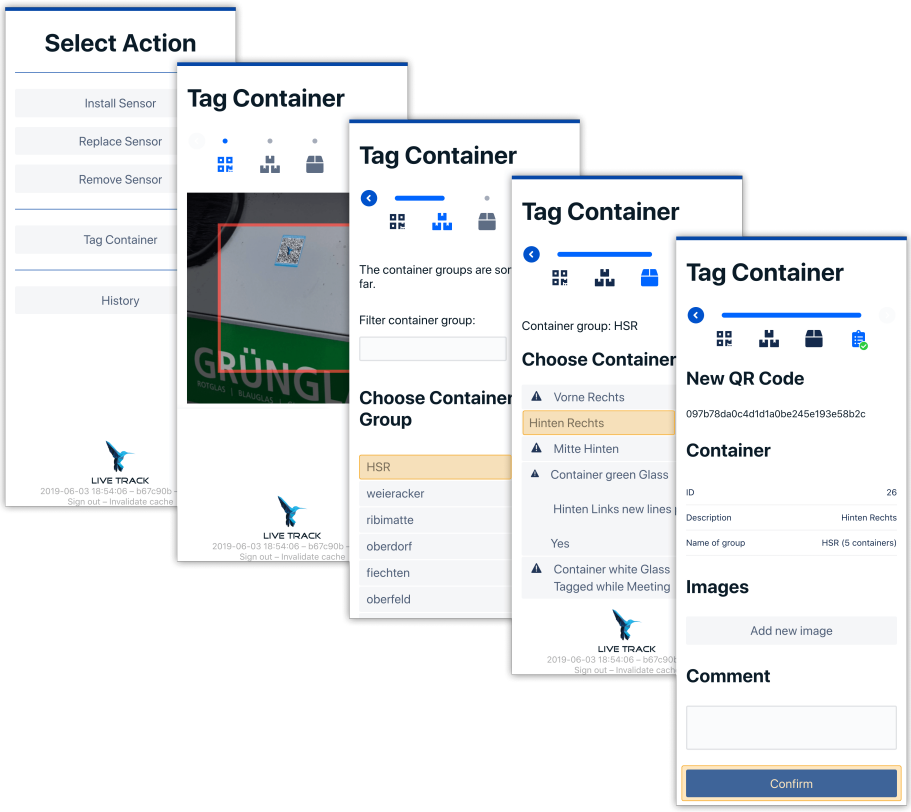


Figure 20: The touch interactions needed to complete the *Tag Container* workflow are marked with an orange box. After starting the workflow, the app scans the QR code automatically, then the user can select the container group (first touch interaction). After that he can select the container (second touch interaction). And finally submit the workflow (third touch interaction).

4 Implementation

In this chapter the code base of SensorFlow is analyzed with different tools:

Building Block View provides an overview of the source code by breaking down the system into building blocks and their dependencies.

Runtime View describe the concrete behavior of these building blocks with scenarios.

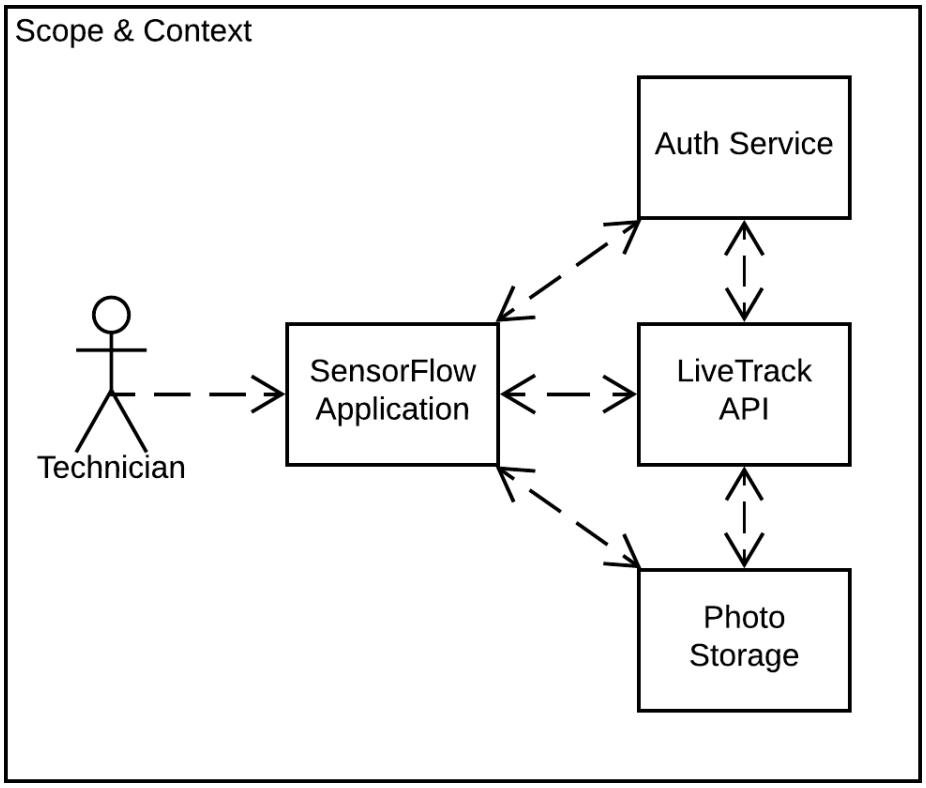
4.1 Building Block View

arc42 describes the building block view in chapter 5 of their documentation[15]:

> The building block view shows the static decomposition of the system into building blocks (modules, components, subsystems, classes, interfaces, packages, libraries, frameworks, layers, partitions, tiers, functions, macros, operations, datas structures, ...) as well as their dependencies (relationships, associations, ...)
>
> This view is mandatory for every architecture documentation. In analogy to a house this is the *floor plan*.

4.1.1 Scope & Context

Before diving deeper into building blocks of the application a scope & context diagram is used to specify external interfaces by delimiting it from all systems and users it communicates with.



Technician A technician working for *LiveTrack*.

SensorFlow App A technician uses the *SensorFlow* app, logging in using the Authentication Service. The app requests and receives a reduced database to work with from the *LiveTrack* API. When finishing an action, the technician can upload photos, which are sent to the *LiveTrack* API. To submit an action, the app sends a post request to the *LiveTrack* API.

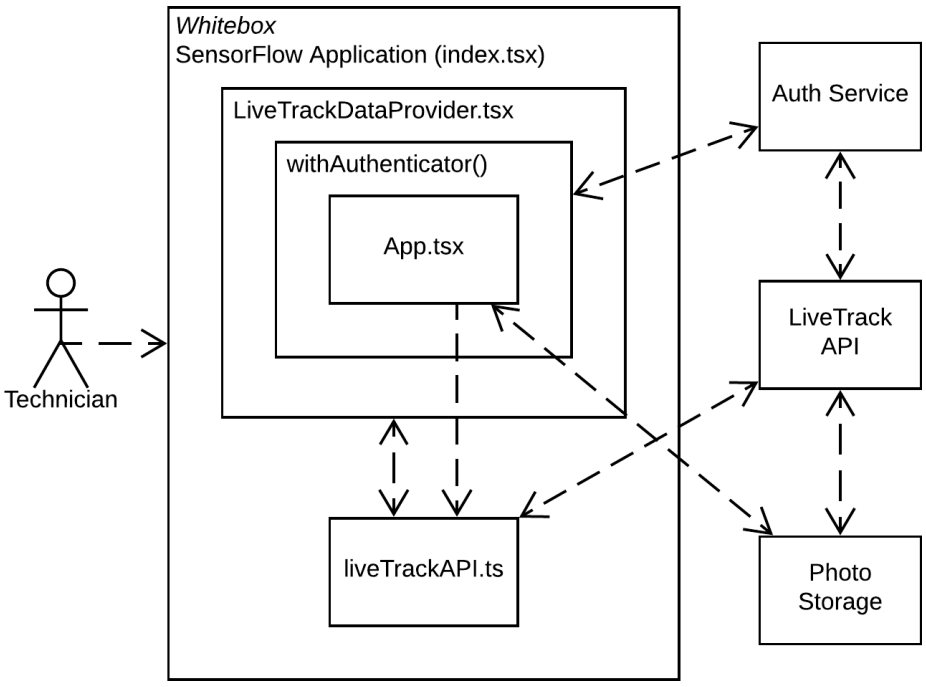
***LiveTrack* API** The *LiveTrack* API provides the API endpoints required to get the sync data and execute all workflows. The API is provided by *LiveTrack* and is documented: https://www.live-track.ch/api-doc. All uploaded photos are stored in the "Photo Storage".

Authentication Service *Amazon Cognito* is an *AWS* service that handles the authentication. This user directory is used in multiple *LiveTrack* applications.

Photo Storage *Amazon S3* is the storage service of *AWS*, here an instance of it is used to store photos. When the technician uploads a photo it is first posted to the *LiveTrack* API which then stores it in the Photo Storage, from where it is directly available via URL. The URL is stored by the *LiveTrack* API and sent to applications from there.

4.1.2 Building Blocks – Level 1

***SensorFlow* Application** *SensorFlow* is a React Application and therefore based on components. The root component is mounted to the DOM in "index.tsx" and renders the React context [10] providers and the App component. "withAuthenticator()" ensures that the app component is only shown if the user is logged in. The LiveTrackDataProvider component gets its data from the liveTrackAPI service.



***SensorFlow* Application (index.tsx)** Renders the React context [10] providers that contain the App component as root component, as shown in the following excerpt:

```
ReactDOM.render(
  <Providers>
    <App />
  </Providers>,
  document.getElementById('root')
);
```

LiveTrackDataProvider.tsx Uses React context [10] to pass the *LiveTrack* data down to any component that needs to have access to it. Renders the authentication guarded App component as child.

withAuthenticator() HOC "withAuthenticator(App, ...)" is a higher-order component[16] (HOC), which means that it is a function that takes a component as an argument and creates a new component. In this case the component passed to the HOC is the App component. The new component created by the "withAuthenticator" function renders either the App component if the user is logged in, or the login component if not. The login component performs the login via the authentication service. The usage of "withAuthenticator" looks as follows:

```
1 withAuthenticator(
2   App, // the app component
3   ...  // additional options for the user login flow
4 );
```

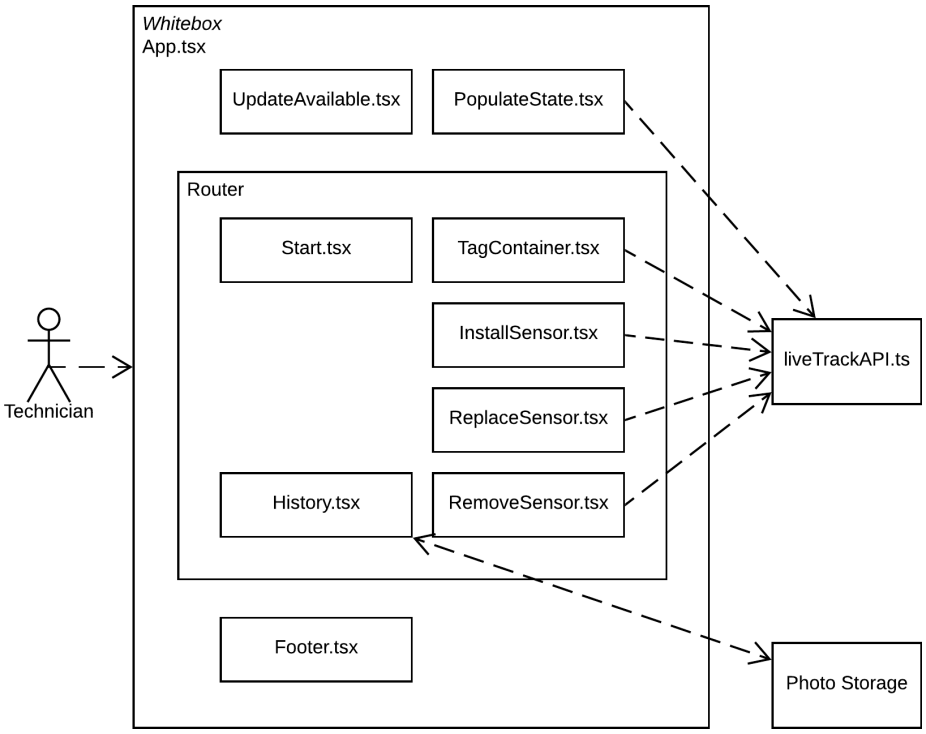
"withAuthenticator" is part of the Amplify Framework[17]. The Amplify Framework is provided by *Amazon* to simplify the use of cloud services when developing mobile & web apps.

App.tsx This is the App component, which is first shown to the user after logging in.

liveTrackAPI.ts This service manages all requests to the *LiveTrack* API.

4.1.3 Building Blocks – Level 2

App.tsx The App component is the heart of *SensorFlow* and delivers all functionality using components.



UpgradeAvailable.tsx Displays a modal that informs the user that the app has an update. The user can click the "Update" button to update *SensorFlow*.

PopulateState.tsx Checks whether the *LiveTrack* data has been loaded successfully. If the data could not be loaded, a modal is shown and the user has to sync again.

Router.tsx The router is used to render components based on the current page address:

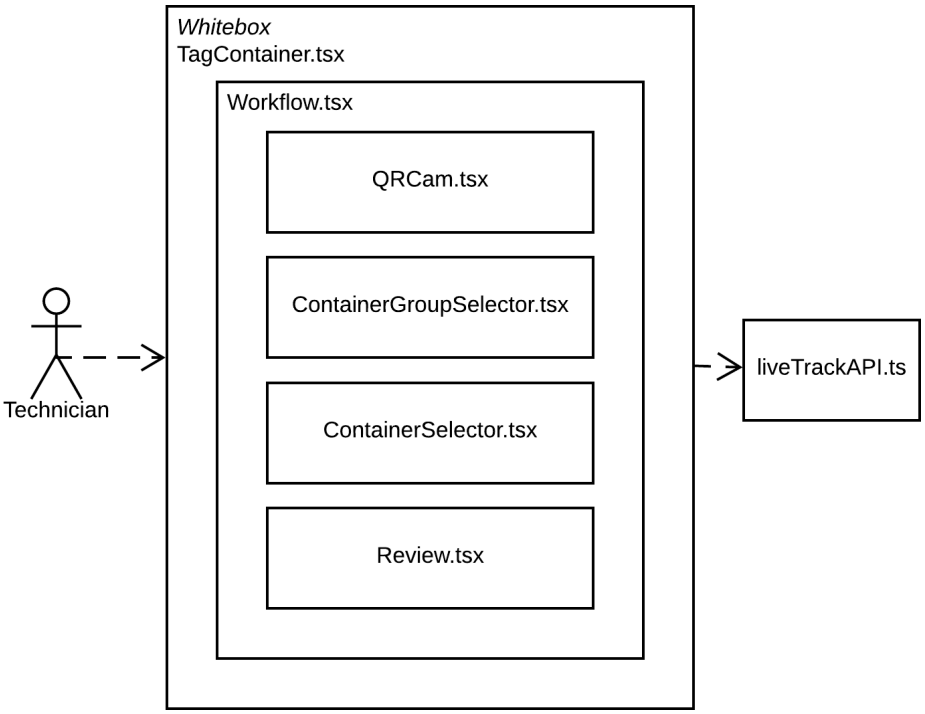
- **"/"** → **Start.tsx** – shows all possible actions the user has.
- **"/tagContainer"** → **TagContainer.tsx** – shows workflow *Tag Container*.
- **"/installSensor"** → **InstallSensor.tsx** – shows workflow *Install Sensor*.
- **"/replaceSensor"** → **ReplaceSensor.tsx** – shows workflow *Replace Sensor*.

- **"/removeSensor" → RemoveSensor.tsx** – shows workflow *Remove Sensor*.
- **"/history" → History.tsx** – shows the history of actions taken by the user.

Footer.tsx Unlike the components which are shown and hidden by the router, the footer of the app is always rendered by the App component. It contains a logout and a sync button and shows the build number, the commit hash and creation date of the delivered build.

4.1.4 Building Blocks – Level 3

TagContainer.tsx If the user navigates to "/tagContainer" the router displays the TagContainer component that renders the entire *Tag Container* workflow.



Workflow.tsx The workflow component is used to render every workflow. It is used by providing a list of workflow items as prop. Each item contains an id, an icon and a render function. The latter is called from the workflow component, providing the following values as arguments:

- **readonly**
  a boolean indicating whether the user has already completed this item.

- **valueStorage**
  in which the data of this workflow is stored.
- **updateValueStorage(partialValueStorage)**
  a function to update the *valueStorage*, works like setState does. If readonly is true, which means that the user already completed this item before, the user is asked to confirm or cancel the action since that would reset the progress to step.
- **moveToNext()**
  a function to move forward to the next item
- **tryUpdateAndMove(partialValueStorage)**
  a function that tries to update the valueStorage and moves to the next item if the update succeeded, combining the two previous functions.

The following example shows the Workflow component as it is being used by the TagContainer component. Only the first item, the one which renders the QRCam component, is shown. Its render function uses the containerQRcode attribute of the valueStorage and the updateValueStorage and tryUpdateAndMove functions. When the QRCam component reports that a QR code has been read, tryUpdateAndMove is called. As explained above, this will usually save the containerQRcode into the valueStorage and move to the next item.

```
<Workflow<TagContainerValueStorage>
  title={t('workFlows.tagContainer.title')}
  initialValueStorage={{ comment: '', images: [] }}
  workflowItems={[
    {
      id: 'camContainerQRcode',
      icon: { name: 'qrcode' },
      render: ({ valueStorage: { containerQRcode }, updateValueStorage,
          tryUpdateAndMove }) => (
        <QRCam
          onCodeRead={containerQRlink => {
            const containerQRcode = parseQR(containerQRlink);
            ...
            tryUpdateAndMove({ containerQRcode });
          }}
          onCodeReset={() => updateValueStorage({ containerQRcode: undefined })}
          code={containerQRcode}
        />
      )
    },...
```

4.2 Runtime View

This chapter describes concrete behavior and interactions of the system with scenarios.

4.2.1 Scenario: From Starting the App to Submitting a Workflow

To explain the runtime procedure of *SensorFlow* a sequence diagram, shown in Figure 21, was created. The diagram illustrates the procedure of the workflow *Tag Container*. We could have chosen any other workflow and the process would still be the same as they all share the identical skeleton, only the contained building blocks differ as explained in chapter 4.1.4.

4.2.2 Scenario: Caching of the API

This scenario describes the process of caching the *LiveTrack* API to achieve partial offline support as explained in chapter 1.5.4 and can be found in Figure 22.

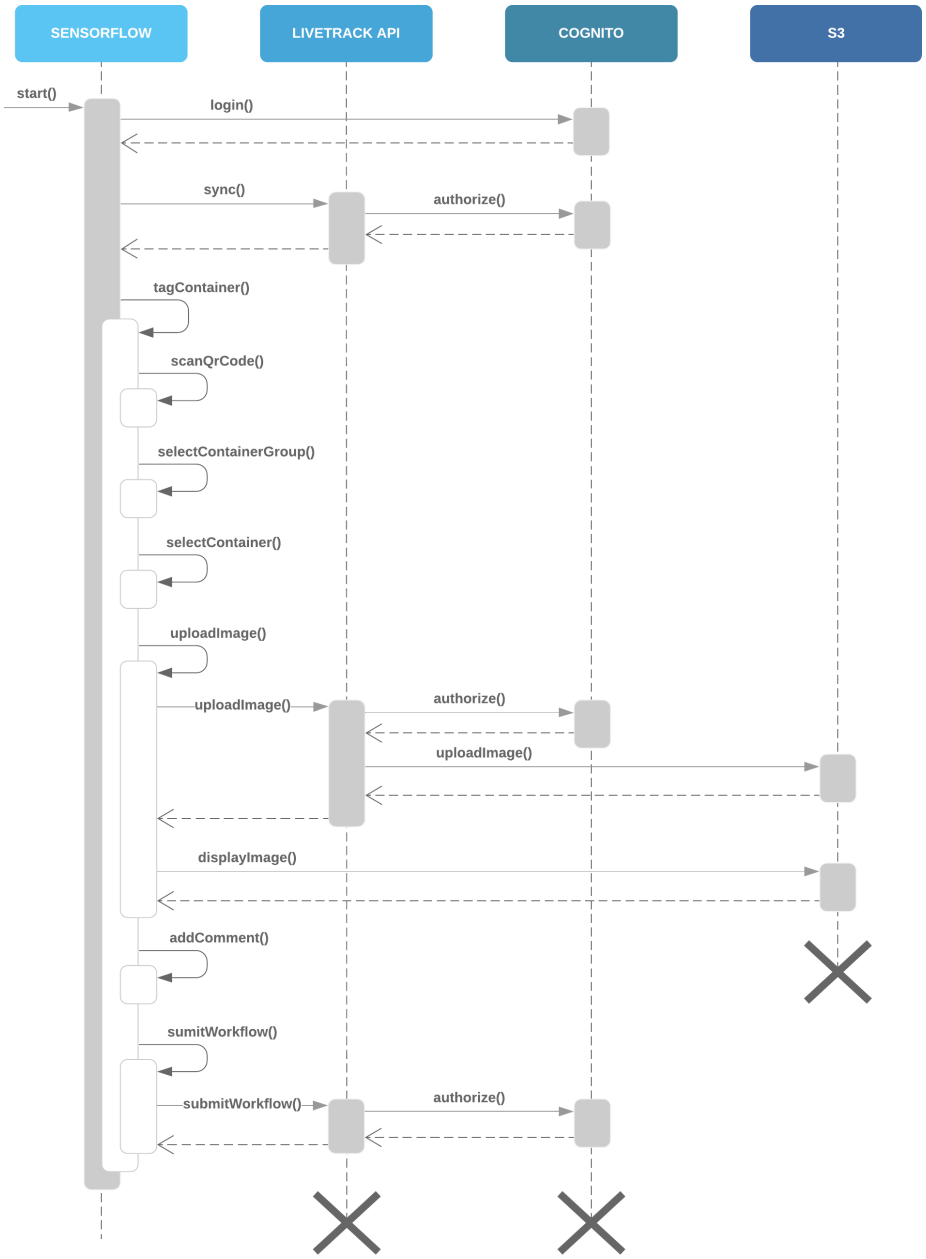


Figure 21: When a technician starts the app and tags a container the following procedure is executed. First, the credentials are checked by the *Amazon Cognito* service, which returns a token if successful. This token is then used to authorize the client against the *LiveTrack* API, as we can see with the `sync()` request. Following a successful sync, the technician executes all the building blocks of the workflow *Tag Container*. The workflow is complete when the last building block is executed, making *SensorFlow* ready for the next action.

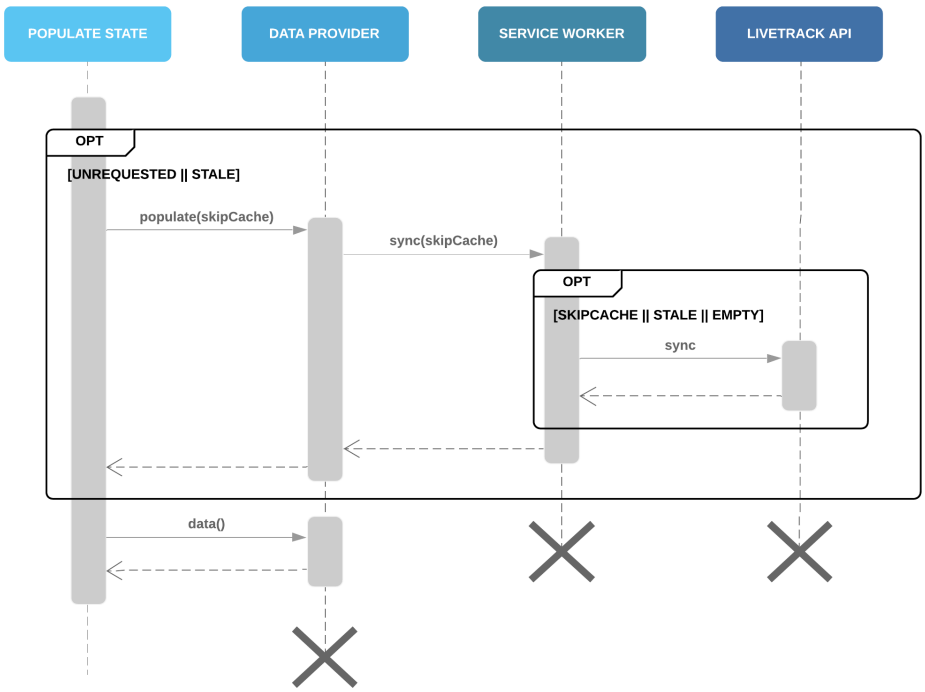


Figure 22: Every time the App component is rendered, the PopulateState component checks whether the data has not been requested yet or is older than 1 hour. In either of these cases, it executes the populate function (explained in detail in Figure 23) of the data provider. As a result, an API synchronization request is executed, but intercepted by the service worker [18]. The service worker checks whether the request should bypass the cache; if so, the request is executed, the cache is updated, and the response is returned. The request is also executed if the cache is empty or the cached request is older than 8 hours. With the caching of the service worker *SensorFlow* is able to offer partial offline support to execute the workflows even with a bad internet connection.

```
public async populate(skipCache: boolean = false) {
  this.setState({ populationState: PopulationState.requested });

  const result = await LiveTrackAPI.sync(skipCache);

  if (result.success === true) {
    const { containerGroups, containers, sensors } = build(result);
    this.setState({
      containerGroups,
      containers,
      sensors,
      populationState: PopulationState.resolved
    });
  } else {
    this.setState({
      populationState: PopulationState.rejected
    });
  }
}
```

Figure 23: An excerpt of the data provider displaying the populate function shown in Figure 22. The populate function is responsible for calling the *LiveTrack* API, handling the progress state (requested, resolved, rejected) and transforming the response (shown in Figure 9) into instances of classes.

5 Testing

We developed *SensorFlow* with *Fault Tolerance* [19] in mind, therefore we know that tests can not show the absence of faults. Nevertheless, testing is important, as tests provide documentation about what the code should do and checks whether it actually does it. We have written unit and integration tests to ensure that the functional and non-functional requirements are met. In addition, we conducted usability tests, which are documented in chapter 3.5 Usability Tests. Last but not least, the application was also subjected to functional testing.

5.1 Unit Testing

Unit tests are written with Jest [20], an open source test framework from *Facebook*. We focused on achieving 100% test coverage for critical parts of the system. In particular, the API connection service has been extensively tested. User interaction heavy parts, on the other hand, were excluded and left to be tested with integration tests. For this reason we only achieve a unit test coverage of 30% – 35%, as we can see in Figure 24.

5.2 Integration Testing

Integration tests simulate the behaviour of a technician using the application. The test can not interfere with the code like a unit test can. Instead it can only interact like a user. This is really valuable to test whether the application works as a whole, when all parts are interacting with each other. We have written seven tests that check critical paths through the application. They are shown in Figure 25. To write the tests we used *Cypress* [21], an end to end testing framework developed by *Cypress.io* and the open source community.

Issues During the test we had problems with the native functions used in the app. First, we had to bypass the authorization prompt for the camera and location. Thereafter, both APIs provided useless results, which led to the need to mimic their behaviour. This worked quite easily for the GeoLocation API, but was much more difficult for the camera. With Chrome, it is possible to feed a fake media stream that is returned instead of the camera input, but unfortunately only one video stream can be provided. Since we need to scan different QR codes, there is currently no way to do this with our setup. To be still able to test all workflows, we decided to use a workaround as explained in Figure 26.

5.3 End to End Testing

For the authentication part, we wrote E2E tests that interact with the *Amazon Cognito* API, unfortunately we could not do the same for the *LiveTrack* API as we can see in Figure 27.

```
-----------------------------------|----------|----------|----------|----------|-------------------|
File                               |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-----------------------------------|----------|----------|----------|----------|-------------------|
All files                          |     35.2 |    28.33 |    24.42 |    35.83 |                   |
 src/components/Workflow           |       82 |    77.42 |    73.08 |    84.78 |                   |
  IconButton.tsx                   |      100 |    66.67 |      100 |      100 |                12 |
  ProgressBar.tsx                  |       80 |       90 |       60 |       80 |             38,57 |
  Workflow.tsx                     |    82.05 |    72.22 |       75 |    85.71 |  35,56,89,169,170 |
 src/helpers                       |      100 |      100 |      100 |      100 |                   |
  geoUtils.ts                      |      100 |      100 |      100 |      100 |                   |
  icons.tsx                        |      100 |      100 |      100 |      100 |                   |
  parseQRlink.ts                   |      100 |      100 |      100 |      100 |                   |
 src/models                        |      100 |      100 |      100 |      100 |                   |
  container.ts                     |      100 |      100 |      100 |      100 |                   |
  containerGroup.ts                |      100 |      100 |      100 |      100 |                   |
  sensor.ts                        |      100 |      100 |      100 |      100 |                   |
 src/services/api                  |     82.5 |    63.64 |    88.89 |    80.56 |                   |
  LiveTrackDataProvider.tsx        |      100 |      100 |      100 |      100 |                   |
  liveTrackAPI.ts                  |      100 |      100 |      100 |      100 |                   |
  liveTrackBuilder.ts              |      100 |      100 |      100 |      100 |                   |
  liveTrackContext.ts              |      100 |      100 |      100 |      100 |                   |
  liveTrackRoute.tsx               |        0 |        0 |        0 |        0 |7,9,13,14,15,16,18 |
 src/services/api/liveTrack        |      100 |      100 |      100 |      100 |                   |
  containerAPI.ts                  |      100 |      100 |      100 |      100 |                   |
  request.ts                       |      100 |      100 |      100 |      100 |                   |
  sensorAPI.ts                     |      100 |      100 |      100 |      100 |                   |
 src/services/toast                |      100 |      100 |      100 |      100 |                   |
  toastConfig.ts                   |      100 |      100 |      100 |      100 |                   |
  toastContext.ts                  |      100 |      100 |      100 |      100 |                   |
  toaster.tsx                      |      100 |      100 |      100 |      100 |                   |
-----------------------------------|----------|----------|----------|----------|-------------------|

Test Suites: 11 passed, 11 total
Tests:       40 passed, 40 total
Snapshots:   4 passed, 4 total
Time:        5.725s
Ran all test suites.
✨ Done in 7.68s.
```

Figure 24: 40 unit tests running in 5.72s covering all services, helpers and the workflow component of our application. The parts of the user interaction, on the other hand, are not unit tested.

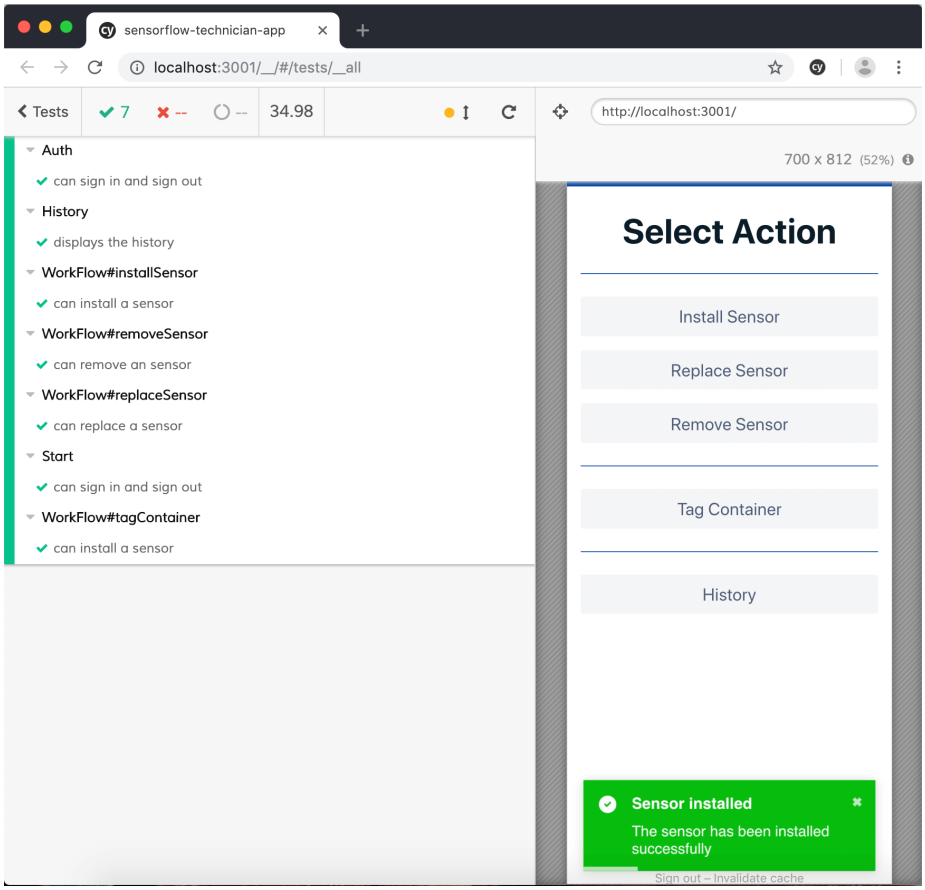


Figure 25: Integration tests executing 7 critical paths including all workflows. Running all the tests takes about a minute and a half.

```
interface QRCamProps {
  onCodeRead: (code?: string) => void;
  onCodeReset?: () => void;
  code?: string;
}

export default class QRCam extends Component<QRCamProps> {
  render() {
    const { code } = this.props;
    window['SET_QR_CODE_PROGRAMMATICALLY_FOR_TESTING'] = this.props.onCodeRead;
```

Figure 26: The *onCodeRead* function is a callback that is executed when the scanner can read a QR code. During testing it is not possible to have a custom camera media stream for each test, so we need to manually simulate scanning the code and call *onCodeRead* so that all tests can run through and test the corner cases.

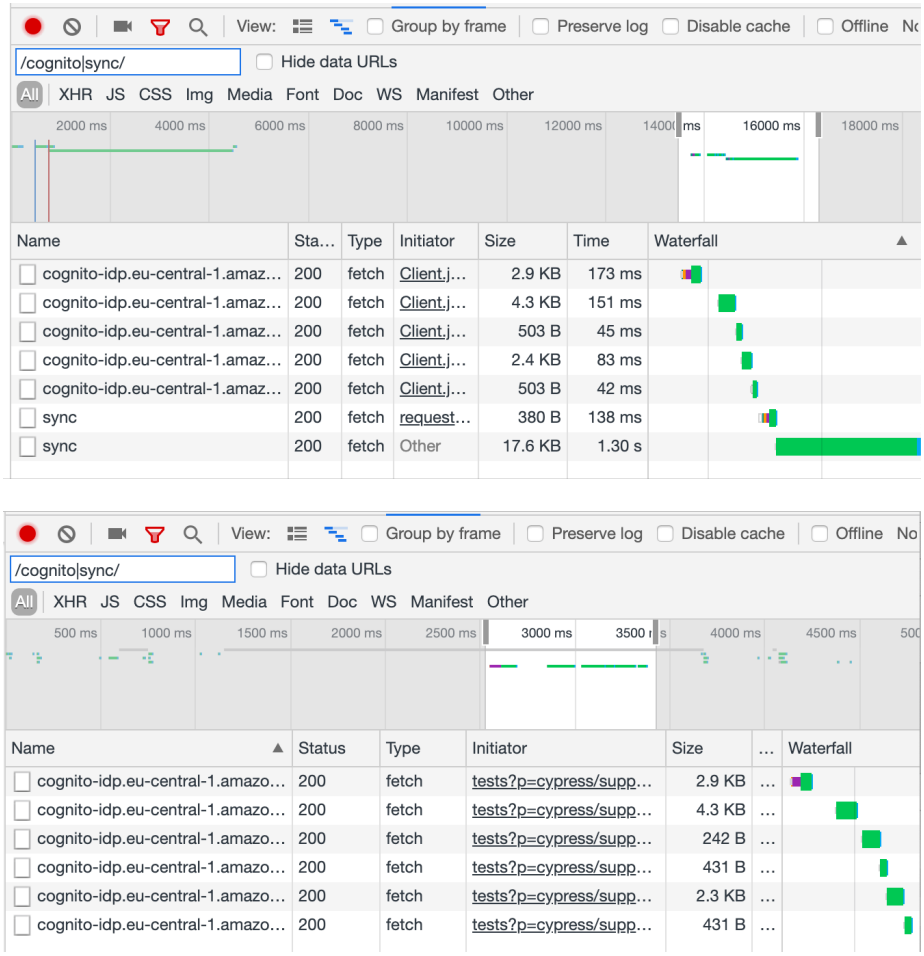


Figure 27: On top we can see how requests are made to the *Amazon Cognito* API and the *LiveTrack* API. This is a screenshot taken from the development environment after a log in. The screenshot below was taken while the E2E tests were running after a log in. We can clearly see that the *LiveTrack* API calls have disappeared because they are being mocked.

Open To take the integration tests to the next level, we would have to stop mocking the *LiveTrack* API, which would lead to real E2E tests running the entire interaction from start to finish. To start writing these tests, we would need to be able to run the *LiveTrack* API locally or have a dedicated testing server. One solution could be a docker image that contains all the requirements for running the API and is started with `docker-compose up`.

5.4 Functional Testing

In order to complete our test setup, we performed functional tests with a focus on native features because we could not properly test them automatically. All implemented use cases from subsection 1.2 were tested:

Nr	Name	Satisfied
1	UC: Access History	Yes
2	UC: Tag container	Yes
3	UC: Install sensor	Yes
4	UC: Replace sensor	Yes
5	UC: Remove sensor	Yes

6 Results

This chapter concludes the thesis by first comparing the results achieved with those planned. Afterwards, possible expansion options are outlined and finally an outlook for *SensorFlow* is given.

6.1 Achieved Goals

With *SensorFlow* we have developed a progressive web application that almost feels like a native app and has a state of the art user interface. All planned non-functional and functional requirements are implemented, resulting in a released and deployed application in successful productive use. The result can be seen in Figure 28. A closer look reveals some differences in the design between the mockups and the finished result. These changes were necessary to further simplify ease of use. Nevertheless, there is always room for improvement and further features are described in the next chapter. In conclusion, we have achieved our goals to the fullest satisfaction.

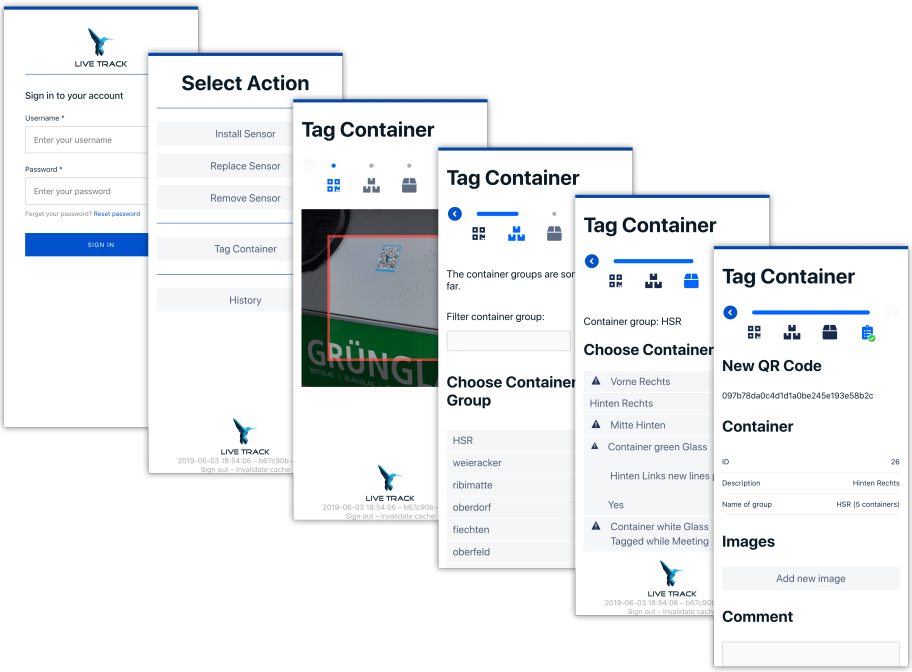


Figure 28: All screens from starting the app to submitting the *Tag Container* workflow.

6.2 Possible Extensions

This chapter lists possible extensions for the application.

6.2.1 Offline Support

The partial offline support could be extended to full offline support as described in chapter 1.5.4.

6.2.2 Extra Safety When Removing Sensors

The workflows *Replace Sensor* and *Remove Sensor* could be extended with a step where the technician scans the QR code of the container first. The technician would be able to check whether the currently installed sensor is the one that should be removed. It would therefore never happen that the technician removes the sensor from the container and determines that it is the wrong sensor.

6.2.3 Split Installation Process

The workflows *Install Sensor* and *Replace Sensor* could be divided into two parts, one before and one after the physical work. This would give the technician an even clearer structure and always keep the real world and its digital twin in sync. This would solve the problem found in the usability tests (chapter 3.5) that the technician has to decide whether he wants to finish the workflow before or after installing the sensor. The former prevents the technician from adding images to the finished installation, the latter sometimes leaves him feeling uncomfortable because he is not sure if the progress will be saved for as long.

6.2.4 Overview Page for Tagged Containers

A new screen similar to the container selection process from the workflow *Tag Container* that displays all tagged containers. With this new screen, a technician could easily check whether the containers are already marked, even if the QR code is removed, as occurred during our usability tests (chapter 3.5).

6.2.5 Expand History Page

The history page could be extended by one traffic light per history entry. The traffic system would work like this: green light if everything is OK, red light if something is broken and yellow light if the action is still pending. This would allow the technician to quickly get an overview of all his work without having to check everything manually.

6.2.6 Permissions Page

A page where the user can provide the necessary permissions to operate the app without having to search for the right permission in the browser or smartphone

settings. Unfortunately the browsers are not ready yet, the required *Permission API* [22] is not yet finalized. This possible feature is blocked until then.

6.3 Outlook

With *SensorFlow* playing a crucial role in tagging containers and maintaining sensors we have just seen the beginning of *SensorFlow*'s life and are glad to see that *SensorFlow* is already growing. *LiveTrack* plans to implement all possible features from above except the first two. In this sense we wish *SensorFlow* a long and happy life.

6.4 Acknowledgments

We thank our supervisor, Mirko Stocker, for his continued support throughout the thesis. The weekly meetings with him were invaluable.

Our thanks also go to our industrial partners Bernhard Zindel and Remo Liebi, who trusted us to implement their vision. They were always available to answer questions and discuss new ideas.

References

[1] Amazon. *Amazon S3 + Amazon CloudFront: A Match Made in the Cloud*. 2018. URL: https://aws.amazon.com/blogs/networking-and-content-delivery/amazon-s3-amazon-cloudfront-a-match-made-in-the-cloud/.

[2] Amazon. *Amazon Cognito. Simple and Secure User Sign-Up, Sign-In, and Access Control*. URL: https://aws.amazon.com/cognito/.

[3] OWASP™ Foundation. *Cross-Site Scripting (XSS)*. 2018. URL: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[4] Facebook. *DOM Elements*. 2018. URL: https://reactjs.org/docs/dom-elements.html.

[5] React Training and contributors. *React Router*. URL: https://reacttraining.com/react-router/.

[6] MDN Web Docs. *Manipulating the browser history. The DOM window object provides access to the browser's session history (not to be confused for WebExtensions history) through the history object. It exposes useful methods and properties that let you move back and forth through the user's history, as well as – starting with HTML5 – manipulate the contents of the history stack.* 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API.

[7] SmartBear. *Swagger aides in development across the entire API lifecycle, from design and documentation, to test and deployment.* 2019. URL: https://swagger.io/.

[8] OpenAPI Initiative. *OpenAPI is a specification for machine-readable interface files.* 2019. URL: https://www.openapis.org/.

[9] Nelmio. *NelmioApiDocBundle generates documentation for REST API from annotations.* 2019. URL: https://github.com/nelmio/NelmioApiDocBundle.

[10] Facebook. *Context*. 2018. URL: https://reactjs.org/docs/context.html.

[11] jamiebuilds. *Unstated*. URL: https://github.com/jamiebuilds/unstated.

[12] Atlassian. *Atlassian's official UI library, built according to the Atlassian Design Guidelines.* 2019. URL: https://atlaskit.atlassian.com/.

[13] Material-UI core team and open source community. *A frontend framework for React with material design styled components for faster and easier web development.* 2019. URL: https://material-ui.com/.

[14] MDN Web Docs. *Geolocation API. The Geolocation API allows the user to provide their location to web applications if they so desire. For privacy reasons, the user is asked for permission to report location information.* 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API.

[15] arc42. *arc42 Documentation*. URL: https://docs.arc42.org/home/.

[16] Facebook. *Higher-Order Components*. 2018. URL: https://reactjs.org/docs/higher-order-components.html.

[17] Amazon. *Amplify Framework*. URL: https://aws-amplify.github.io/docs/.

[18] MDN Web Docs. *Service worker. Service workers essentially act as proxy servers that sit between web applications, the browser, and the network (when available). They are intended, among other things, to enable the creation of effective offline experiences, intercept network requests and take appropriate action based on whether the network is available, and update assets residing on the server. They will also allow access to push notifications and background sync APIs*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[19] Robert S Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[20] Facebook. *JavaScript testing framework*. URL: https://jestjs.io/.

[21] Cypress.io. *JavaScript E2E testing framework*. URL: https://cypress.io/.

[22] MDN Web Docs. *Permissions API. The Permissions API provides a consistent programmatic way to query the status of API permissions attributed to the current context — for example whether permission to use the API is granted or denied, and whether the user will be asked whether permission should be granted for an API*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API.

[23] Overleaf. *Overleaf, Online LaTeX Editor*. 2019. URL: https://www.overleaf.com.

[24] Zube. *Zube — Agile project management with a seamless GitHub integration*. 2019. URL: https://zube.io/.

[25] Git community. *Git is a free and open source distributed version control system*. 2019. URL: https://git-scm.com/.

[26] Atlassian. *GitFlow is a branching model for Git*. 2019. URL: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[27] Tom Preston-Werner. *Versioning scheme for software updates*. 2019. URL: https://semver.org/.

Appendices

We have included the following documents as attachments:

Glossary Common vocabulary of this thesis

Development Concepts Development concepts used during this thesis

Milestones The planned project milestones

Time Evaluation Our time evaluations of the thesis

Personal Reports Personal reports of the thesis authors

Task Definition The task definition we got at the start of the thesis

Copyright and Usage Agreement The agreement to give our industry partner the right to use our code

Protocols The protocols of all our meetings including important decisions