GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Developer Manual

Design Project - Runamic Ghent

Group 16

Depauw Hendrik
van Herwaarden Lorenz
Aelterman Nick
Cammaert Olivier
Deweirdt Maxim
Dox Gerwin
Neuville Simon
Uyttersprot Stiaan

Contents

1 Client - General Architecture

Before we dive into the details of the components, we first give an overview of the architecture of the smartphone client and the wearable.

1.1 Client

The architecture of the client application is based on message passing. One central entity, the *eventbroker*, collects messages (also called *events*) and passes these to the appropriate listeners. These messages consist of (1) event type and (2) the message itself (payload). Figure 1 shows the messages passed for the mobile application.
There are three kinds of actors in this architecture: the eventbroker itself, eventlisteners and eventpublishers.

EventPublisher EventPublishers are objects that need to publish messages that are consumed by other objects. Every class that will act as an eventpublisher has to implement the `EventPublisher` interface.
Components can publish messages themselves by calling the `.addEvent(String type, Object message, EventPublisher source)` method.

EventListener Components can be subscribed to be notified of messages of a certain type, by calling the `addEventListener(String type, Eventlistener component)` method. To be able to subscribe a component to a message, it should implement the *EventListener* interface.

EventBroker The *eventbroker* is a singleton, so there is one `public static final` instance, and at most one eventbroker is instantiated at any time. The eventbroker instance can be accessed using the public getter: `EventBroker.getInstance()`.
Internally, the eventbroker keeps a list of all eventlisteners and a queue of messages that need to be processed. As long as the queue is not empty, it will process messages in the queue in a *first-in-first-out* fashion. For each message being processed, the eventbroker will check which listeners are subscribed to the respective type of messages and pass them along. The listeners' `.handleEvent(String type, Object message)` method will be called to do so.

1.2 Wearable

The architecture of the wearable closely resembles that of the mobile client. It is also based on message passing. It also employs the same type of eventbroker. It is noted that this is not the same eventbroker as used by the mobile device, these are two separate eventbrokers.
For communication between the mobile device and the wearable, two components are of importance, the *AndroidWearProvider* as component of the mobile application and the *WearComm* as component of the wearable application. These two components communicate with each other over bluetooth, via message passing. An overview of messages passed that concern the wearable (both bluetooth messages and messages over the eventbrokers) is shown in Figure 2.

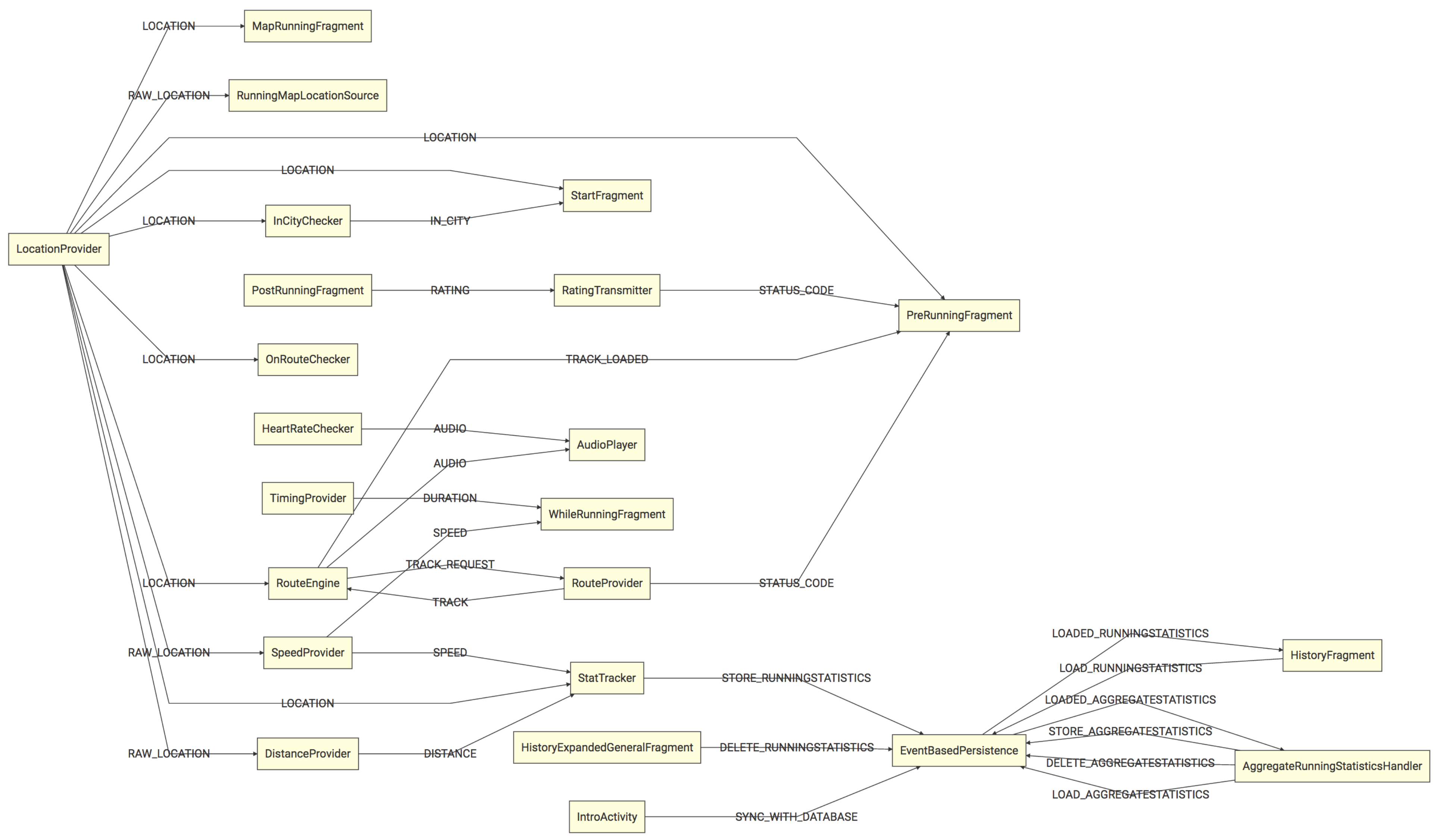

Figure 1: Origin and target of messages passing through the eventbroker for mobile application.

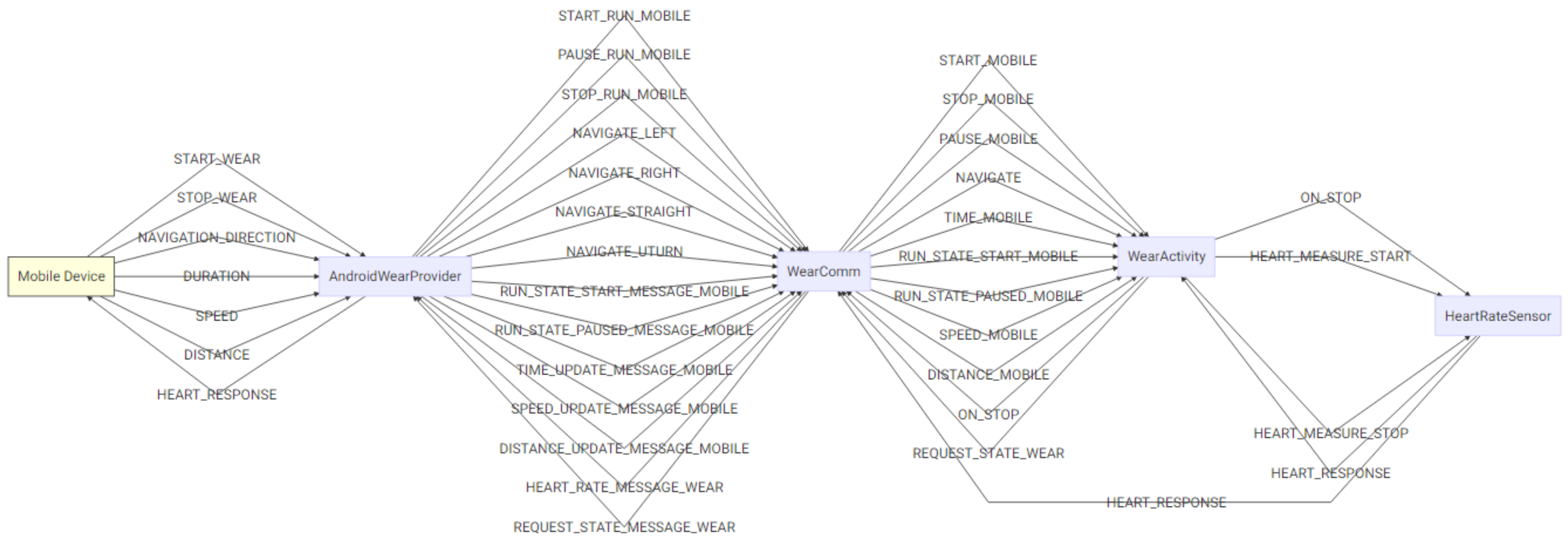

Figure 2: Origin and target of message passing for wearable.

2 Client - Components

The client consists of a lot of different components that are fairly independent. They communicate by listening for certain messages with the EventBroker and sending their responses back.

2.1 Routing Engine

The route engine starts whenever a user wants to start a route. This engine is responsible for managing the route instruction for the user. A route instruction will be handled whenever the user approaches an instruction point (30 metres in advance). The handling of the route instruction happens by publishing an audio event to the event broker with the run direction as content. New route instructions come in whenever a new track is requested. These instructions are added to the attribute "routeList". The route engine first deletes the common points of the new instruction list with the old instruction list. This is necessary to determine the split point of the two routes. The user can decide on the split point which route he/she wants to follow. The route engine will check whether the user is following the new or old route after the split point is reached. The right route instruction list will be selected if the difference in distance between the two routes is more than 30 metres. This 30 meter radius is necessary to overcome the inaccuracy of the GPS.

2.2 Persistence

Description The persistence component takes care of saving runs and aggregate statistics to disk and a database on the server. The local version is changed immediately. The server version is updated whenever internet is available. Communication with this component should happen through the `EventBasedPersistence` class.

Messages acted on This component listens to events related to storage. These events are requests to store, load or delete certain objects. The exact events are `STORE_RUNNINGSTATISTICS`, `STORE_AGGREGATESTATISTICS`, `LOAD_RUNNINGSTATISTICS`, `LOAD_AGGREGATESTATISTICS`, `DELETE_RUNNINGSTATISTICS`, and `DELETE_AGGREGATESTATISTICS`. There is one additional event, `SYNC_WITH_DATABASE`, that is emitted when the application wants to explicitly synchronize with the database, e.g., during start up.

Messages produced The Persistence component produces 2 events, `LOADED_RUNNINGSTATISTICS` and `LOADED_AGGREGATESTATISTICS`, both of them in response to a corresponding load request.

Implementation details The public interface is `EventBasedPersistence`. This class handles all communication with the EventBroker, but contains no logic. The main logic can be found in `PersistenceController` and its helper class `ServerSynchronization`. These classes act on `LocalStorage` and `ServerStorage`, which are responsible for the local disk and the Mongo database respectively.

Related documents

- http://mongodb.github.io/mongo-java-driver/

2.3 Audio

Description The audio component is used to provide instructions or information to the user via audio. Components that want to use audio provide the AudioPlayer with a string. This string is then converted to audio using *Text-to-Speech*. Text-to-speech is implemented in the Android app by using the Android text-to-speech engine. The AudioPlayer component is created and started in the splash screen. If the user disabled audio in the settings, the AudioPlayer will still be constructed but will ignore all incoming messages.

Messages acted on This components acts on `AUDIO` messages. These contain a RunAudio object as payload.

Messages produced This component does not produce any messages.

Implementation details Whenever an `AUDIO` message is passed to this component, SharedPreferences are checked to see whether sound should be played.
Before playing any sound, audio-"focus" should be requested from Android's *Audio Service* through an `AudioManager` instance. The request itself should contain: (1) which kind of behaviour we expect from other applications (2) which kind of audio focus is required (notification, music, ...). Since the audio directions we play are rather short, we ask `AudioManager.STREAM_MUSIC` focus and request other applications to react to the `AudioManager.AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK)`. In our case, other applications receiving this message should dim music that is playing.
Only if the request is granted, we are allowed to play our audio. Once the audio direction has finished playing, the audio focus MUST be released. Because the speak method from Text-To-Speech returns immediatly, the audio focus can not just be released after this method. For this reason a UtteranceProgressListener is used. The onDone() method from this listener is called when Text-To-Speech is finished. When multiple sentences are played after each other, it is possible that the focus is released and again acquired in a very short period of time. This is unpleasant for the user and thus a counter is used that counts how many sentences are still in the queue. Only when there are no more sentences, the audio focus is released.

Related documents

- https://developer.android.com/guide/topics/media-apps/volume-and-earphones.html

2.4 Settings

Description The `Settings` extend `PreferenceActivity`. The `preferences.xml` file defines both layout and keys for preferences.
It contains following preference categories:

- Units (km/h or min/km)
- Dynamic Routing
- Audio
- Reset introduction Tooltips
- About
- Debug (for development)

External Libraries Used For the dynamic routing preference category a range bar is needed. As the Android SDK does not have this type of preference built in, an external library was used. The range bar from the external library can be customised to show the value in the pin above the range bar. This is used to specify an interval in between which the heart rate may lie during a run.

Implementation Details Except for filling textfields, such as version number, no extra code is needed in the SettingsActivity next to the xml file. Next to settings, *licences* and *contact information* are also included. They are loaded as separate activities. To load these activities, a preference containing `<indent android:action="path.to.activity"/>` suffices.

Accessing preferences throughout the application Preferences can be accessed through a `SharedPreferenceManager`. More information.

2.5 Login

The login is based on the services of Google Firebase where user data is managed. Users can subscribe and login with email address and password or with Facebook. If the Facebook application is available on the phone, then this app is used. Otherwise the Facebook login is accessible through the browser. Facebook is implemented by using the Facebook widget login activity and button[1]. The fragments login and sign up are based on Firebase login and sign up. The login through mail/Facebook is stored in Firebase[2]. If admin rights in Firebase are necessary to manage logins, mail to maxim.deweirdt@ugent.be.

2.6 Profile

The profile view uses the `AggregateStatisticsHandler`. By accessing this handler every running statistic about the user can be requested. Profile information is stored in the shared-preferences under the strings "client name", "client email" and "photo url".

2.7 DataProviders

DataProviders generally, as the name implies, provide data for a Fragment/Activity in the app. However, this does not mean that it is not possible for a DataProvider to receive data from a Fragment/Activity and send it somewhere.
All DataProviders should implement the `DataProvider` interface. This interface defines `stop()`, `start()`, `pause()` and `resume()` methods. This is done to make lifecycle management more efficient in fragments.

2.7.1 InCityChecker

The app is designed for specific cities. We can not provide the user with a route if he is not in Ghent. This DataProvider checks whether the user is within a certain bounding box.

Messages acted on This DataProvider acts on `LOCATION` events. These events are published when the location of the user is retrieved from the phone.

[1](For more info about the Facebook login go to https://developers.facebook.com/docs/facebook-login/android)
[2]To access Firebase see https://firebase.google.com/docs/auth/android/start/

Messages produced Once the provider has checked if the user is in Ghent, an `IN_CITY` event is published with a boolean as payload, indicating whether the user is in Ghent or not.

2.7.2 LocationProvider

This class processes location updates and publishes them as `RAW_LOCATION` and `LOCATION` events. An Activity should be given to the constructor. This is necessary to make a connection with the Google API. The Google API is used to process location data from GPS, WiFi and the Mobile Network from the smartphone for more accurate location updates. A LocationRequest is built with some configuration possibilities like fastest update interval, average update interval and priority. The parameters that were used in our app can be found in Table 1. This class also creates a dialog to enable location setting if it is disabled on the user's smartphone. Finally, this class also publishes `LOCATION_ACCURATE` events with a boolean as payload. These indicate if the location of the user has been accurately defined.

parameter	value
priority	PRIORITY_HIGH_ACCURACY
fastest update interval	1000 ms
average update interval	2000 ms

Table 1: LocationRequest Parameters

Messages produced This DataProvider produces both `RAW_LOCATION`, `LOCATION` and `LOCATION_ACCURATE` events.

Messages acted on This DataProvider only listens to location updates from Google Api.

Implementation details The LocationProvider connects to a Google API Client. Once the connection with the API client is made, a LocationRequest is created as previously mentioned. Using this LocationRequest, a LocationSettingsRequests is built that will check if the location setting is enabled on the smartphone and if not, a dialog will be created. If the user accepts to enable the location setting, location updates will be started by using the FusedLocationApi. This API will use the location information of the smartphone from the GPS signal, Mobile network and WiFi signals and send this information to the Google API client. The Google API client in turn will provide the location updates to the LocationProvider in a callback.
Before publishing the `LOCATION` event, the raw location is passed through a Kalman filter. This is done to get a better estimate of the user's actual trajectory. The Kalman filter is based on Apache's Kalman example. The `LOCATION` events are LatLng objects and the `RAW_LOCATION` events are Location objects that have not gone through a Kalman filter.
As previously mentioned, if it is requested in the constructor, the LocationProvider can also publish `LOCATION_ACCURATE` events. The LocationProvider keeps a counter that signifies how accurate the location updates are that it has received. It increments if the location accuracy is below a certain threshold and decrements if the location accuracy is above a certain threshold. The counter starts at 0 and if it hits a certain COUNTER_MAX the boolean will be true. If the counter hits 0 on the other hand, the boolean will be false.

Related documents

- `https://developer.android.com/training/location/change-location-settings.html`

- https://developer.android.com/training/location/receive-location-updates.html
- https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi

2.7.3 DistanceProvider

Description The DistanceProvider will calculate the distance from the point where the DistanceProvider is started up to the current location. Every time a new location arrives at the DistanceProvider, an event will be published containing the total distance up to that point. This is used by the WhileRunningFragment to show the user his total distance. These distances are also saved in the RunningStatistics so the user can reference them later on.

Messages produced The DistanceProvider produces `DISTANCE` events. The payload is a RunDistance object containing the total distance from the starting point up to the current location.

Messages acted on The DistanceProvider listens to `RAW_LOCATION` events.

Implementation details The DistanceProvider works by just calculating the distance between the current location and the previous location. This distance is then added to a class variable that contains the total distance covered from the starting point up to the current location. The inaccurate GPS coordinates cause the DistanceProvider to think that the user covered distance while he is actually standing still. To avoid this problem we only update the distance when the distance between two points is big enough compared to the accuracy of the GPS coordinates. More concretely in our implementation this means the distance is only updated if it is at least five times bigger than the accuracy of the GPS coordinates.

2.7.4 RatingTransmitter

Description RatingTransmitter is a DataProvider that is not really a DataProvider for any Fragment/Activity. It sends the rating that the user assigns to a finished route to the server.

Messages produced If requested in the constructor, the RatingTransmitter can send out `STATUS_CODE` events that contain the status code of the response of the server.

Messages acted on RatingTransmitter listens to `RATING` events that contain a RunRating object with the tag and rating of the run.

Implementation details The rating is sent by using a HttpUrlConnection. This is tried maximally 3 times.

2.7.5 RouteProvider

Description The RouteProvider focusses on fetching routes from the server. These routes can be new routes or dynamic adjustments to routes. After getting the requested route from the server, an event is published with the new route as payload.

Messages produced The RouteProvider produces `TRACK` events. The payload of these events is a TrackResponse object. A second event that can be published is a `STATUS_CODE` event.

Messages acted on The RouteProvider is subscribed to `TRACK_REQUEST` events. These events should include a TrackRequest object.

Implementation details The TrackRequest object in the `TRACK_REQUEST` event tells what type of route that should be requested from the server and what parameters should be used. First thing that is done is construct the correct URL based on this information. The RouteProvider will then try to fetch this route from the server. In case this fails, the URL is tried 2 more times before a `STATUS_CODE` event is published. In that case the user knows something went wrong and he should try again. This event contains the error that the server returned. If the server succesfully returned the requested route, it is converted to a JSONObject. This JSONObject is then put in a TrackResponse object along with some other parameters. This TrackResponse object is then included as a payload in a `TRACK` event that is published.

2.7.6 SpeedProvider

Description The SpeedProvider, as the name implies, calculates the current speed of the user and publishes it using the eventbroker. It is used in the WhileRunningFragment to display to the user, and also saved in the RunningStatistics.

Messages produced The SpeedProvider published `SPEED` events to the eventbroker. As payload these messages have a RunSpeed object containing the current speed of the user.

Messages acted on The SpeedProvider listens to `RAW_LOCATION` events.

Implementation details The `RAW_LOCATION` events contain a speed attribute. However this value is not very smooth due to the imperfections in the GPS coordinates. To solve this issue a very simpel Kalman Filter is used to smooth these values out. This avoids sudden changes in the speed that is displayed to the user.

2.7.7 TimingProvider

Description The TimingProvider is used to measure the duration of a run. When the TimingProvider is started it starts counting. Every second an event is published. These events are used to display a timer to the user in the WhileRunningFragment.

Messages produced The TimingProvider publishes `DURATION` events. These events containt a RunDuration as payload with the time that has passes since the TimingProvider is started.

Messages acted on The TimingProvider is not subscribed to any type of events from the eventbroker.

Implementation details The timing is provided by the Java Timer class. It is set to go off every second. When the timer goes off, the timervalue is increased and a `DURATION` event is published.

2.8 Intro Activity

The Intro Activity contains 3 fragments which can be navigated by using a bottom navigation bar. This Activity also contains a LocationProvider object that is used in the Start Fragment. Additionally, this Activity will check if the user has already given explicit permission for the app to access user location data.

2.8.1 Start Fragment

This Fragment exists of a Google Map showing the user where he is at the moment and 3 buttons. A Floating Action Button in the upper right corner is used to snap to the user's location, a "no route" button is used to let the user run with a route and a "generate route" button which is placed next to a number picker. The number picker can be scrolled to choose the desired distance of the run and the generate route button will lead the user to a screen where he can view the generated route. This Fragment also checks whether an Internet connection is available as this is necessary to generate a route.

Google Map The Google Play Services are used to display a map of Google. Some preferences like minimum and maximum zoom level, enabled compass and disabled tilt have been set. Every time a location update is received by the Location Provider, the map is animated to center on that location. However, this does not happen if the user starts moving the map with gestures. The snap to location then gets temporarily disabled. These gestures can be detected by using an onCameraMoveStartedListener. Once the user pressed the Floating Action button, it turns green and the snap to location function is enabled again.

Related documents

- https://developers.google.com/android/reference/com/google/android/gms/maps/MapView

2.8.2 History Fragment

This Fragment is used to display a short summary of all the user's runs in a list. These summaries each consist of a "Card". To efficiently display all these cards in a list, a RecyclerView is used. All the cards can be either long pressed or clicked. A click opens a new Fragment with a map containing the route and two tabs, General and Details. The General tab has information like average speed, average heart rate, distance and duration. The Details tab has momentarily information. There is seekbar which can be used to scroll through the run and see statistics for each specific moment.
Finally, if a long press is performed on a card, a map pops with the route shown on it. Once the user's finger is lifted back up this map disappears. For this functionality, an external library called PeekAndPop was used to replicate the "3D Touch" behavior on iOS.

Related documents

- https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html
- https://developer.android.com/training/material/lists-cards.html
- https://github.com/shalskar/PeekAndPop

2.8.3 Profile Fragment

This Fragment is used to display basic user information and aggregated statistics and like total distance run and average speed over all runs. If the user is logged in with Facebook, his profile picture will appear. If the user is not logged in with Facebook, it is possible user to change user information like your e-mail address, name and profile picture.

2.9 Run Data Classes

Run Data classes are used to send messages in events through the eventbroker with some extra functionality. In this section they will all be briefly discussed

RunAudio The RunAudio class represents the text that needs to be spoken by the text-to-speach. This can be custom text, but can also depend on directions.

RunDirection The RunDirection class represents the directions of the route the user runs. This class offers some extra functionality like string conversion.

RunDistance The RunDistance class represents the distance run. This class is serializable and offers some extra functionality like adding distance, string conversion and formatting.

RunDuration The RunDuration represents the amount of seconds a user has run. This class is serializable and offers some extra functionality like string conversion.

RunHeartRate The RunHeartRate class represents a heart rate value. This class is serializable and offers string conversion functionality.

RunRating The RunRating class represents a rating that is associated with the tag of a route.

RunRoute The RunRoute class contains a complete route and is constructed from a JSON object. The route consists of an ArrayList of RunRoutePoints. This class offers some extra functionality like conversion from JSON to an ArrayList of RunRoutePoints, calculate total distance, get instructions and get coordinates.

RunRoutePoint The RunRoutePoint class represents 1 point on a route. Each point contains a location and a RunDirection.

RunSpeed The RunSpeed class represents a speed value. This class is serializable and offers some extra functionality like string conversion and speed to pace conversion.

2.10 Tutorial

Description The first time the app is started, the user is guided through the app by means of a tutorial. Upon opening each individual screen, different functionality and UI elements are highlighted with some explanation.

External Libraries used To implement this tutorial, the MaterialShowcaseView by Dean Wild is used. Example code and applications can be found in the link.

Implementation details In the application, it is preferred to have a sequence of "feature-highlights" instead of just one. Moreover it is also preferred to be able to proceed to the next highlight by clicking *anywhere* on the screen rather than clicking a smaller button. This is done by setting `.setDismissOnTouch(true)` for each MaterialShowcaseView.
It is important to specify a `sequenceID` when creating a new sequence. Once the sequence has played, the sharedpreference `material_showcaseview_prefs` will be edited so that `status_ + sequenceID` is set to `uk.co.deanwild.materialshowcaseview.PrefsManager.SEQUENCE_FINISHED`. If this `sequenceID` is not set, the tutorial will play everytime the code is called.

2.11 Wearable

This section will cover the most important components for working with the wearable application

2.11.1 AndroidWearProvider

Description The `AndroidWearProvider` is a DataProvider of the mobile application. It provides heart rate data, read from the wearable device, to the eventbroker. Next to being a heartrate dataprovider, it is primarily responsible for managing the wearable application through bluetooth communication with message passing. It communicates with its counterpart in the wearable device, the `WearComm`.

Messages produced HEART_RESPONSE containing the current heartrate measured in the wearable.

Messages acted on

- START_WEAR, STOP_WEAR: events that notify the component a run has started or ended
- NAVIGATION_DIRECTION: navigation instructions that should be propagated to the wearable
- DURATION, SPEED, DISTANCE: read statistics that are to be shown on the wearable.

Bluetooth Messages produced

- START_RUN_MOBILE, STOP_RUN_MOBILE, PAUSE_RUN_MOBILE: Messages that notify the wearable of a running state change.
- NAVIGATE_LEFT, NAVIGATE_RIGHT, NAVIGATE_STRAIGHT, NAVIGATE_UTURN: Messages for giving navigation instruction via the wearable.
- RUN_STATE_START_MESAGE_MOBILE, RUN_STATE_PAUSED_MESAGE_MOBILE: When the wearable requests the state of the run, these reply with its current state (used when f.e. the wearable app was closed and later started again).
- TIME_UPDATE_MESSAGE_MOBILE, SPEED_UPDATE_MESSAGE_MOBILE, DISTANCE_UPDATE_MESSAGE_MOBILE: Keeping statistics shown in wearable up to date with those of the mobile application.

Bluetooth Messages acted on

- HEART_RATE_MESSAGE_WEAR: Update of heart rate from wearable
- REQUEST_STATE_MESSAGE_WEAR: Wearable requests the current state of the run in the mobile application

Implementation details The NodeApi and MessageApi, contained in the Wearable API provided by Google are used for finding the wearable device and sending messages between wearable and mobile device respectively.
The most important aspect in messaging between the devices that has to be paid attention to, is that when a new message event is added, this is firstly added in Constants.java/ ConstantsWatch.java in the WearMessageTypes class. Subsequently, the messagetype string has to be added to both manifests of the wearable and mobile device, in the intent-filter of the AndroidwearProvider/ WearComm service. If this is not done, messages will not be succesfully passed via bluetooth between both devices.

2.11.2 WearComm

Description The WearComm component is the wearable application counterpart of the AndroidWearProvider and manages all communication between the mobile and wearable from the wearable side. It informs the activity with the current state of the run and statistics received from the mobile while sending heartrate data to the mobile device.

Messages produced

- START_MOBILE, STOP_MOBILE, PAUSE_MOBILE, RUN_STATE_START_MOBILE, RUN_STATE_PAUSED_MOBILE: Informs the `WearActivity` about the current running state
- NAVIGATION: Notifies the `WearActivity` to show a navigation arrow (carries a value for the navigation as message)
- TIME_MOBILE, SPEED_MOBILE, DISTANCE_MOBILE: Updates the current stats in the `WearActivity`

Messages acted on

- ON_STOP: Message for when the application is being shut down.
- REQUEST_STATE_WEAR: Message sent when the application is started to request the current state from the mobile device.

Bluetooth Messages produced See Bluetooth Messages acted on in AndroidWearProvider.

Bluetooth Messages acted on See Bluetooth Messages produced in AndroidWearProvider.

Implementation details This component uses the same APIs as the AndroidWearProvider. The same remarks as for the AndroidWearProvider should be taken into account concerning messages.

2.11.3 HeartRateSensor

Description the HeartRateSensor contains all logic for measuring heart rate in the wearable device. The measurements it gathers are sent to other modules, like the WearCom who then sends it to the mobile device.

Messages produced

- HEART_RESPONSE: a message informing subscribers of the newly measured heartrate

Messages acted on

- HEART_MEASURE_START, HEART_MEASURE_STOP: control by the WearActivity to start and stop the heart rate sensor.
- ON_STOP: Message for when the application is being shut down.

Implementation details For measurements, the sensormanager provided by the Google wearable API is used. Using this sensormanager the default sensor for heartrate measurement is used, if the wearable possesses a heartrate sensor. When the sensor is started, onSensorChanged and onAccuracyChanged events will take place when, respectively a new measurement is available and the accuracy of the measurements changed. The onAccuracyChanged method has not been implemented as we have no use for this method in the current application.

3 Routing Server

The application also contains a backend, namely a server that currently uses Python and Django to generate routes and respond to route requests. For development, consider the server as three different applications: The graph preprocessing, which is run once before starting the server; the server itself, and the database, which the server connects to.

3.1 Graph preprocessing

The routing server does not use the raw Open Street Maps (OSM) data as input, but some datasets that have been derived from OSM.

3.1.1 Node and way selection

First, a rectangle between two coordinates is taken from the OSM graph. This is done with *Osmosis*, a tool developed by the OSM project. All ways that contain references to nodes that are not in the rectangle are removed. From the remaining ways, only the ones that have the "highway" tag are kept. This corresponds to all ways that can be accessed by foot. Only the nodes that lay on these ways are kept. The selected nodes and ways can be found in the array 'elements' in the 'nodes.json' and 'ways.json' files respectively. The attributes of nodes and ways can be found in tables 2 and 3.

attribute	description
id	integer
lat	latitude
lon	longitude
water	integer, amount of water 'close' to node
park	integer, amount of parks 'close' to node
tags	(optional) key value pairs

Table 2: Node attributes

attribute	description
id	integer
nodes	list of node id's
tags	(optional) key value pairs with the key 'highway' present

Table 3: Way attributes

The OSM data we consider parks (green area in general) and water is displayed in tables 4 and 5 respectively.

3.2 Server structure

The server consists of many components, all of which are sorted hierarchically: no two components depend on each other. Originally, the entire server is written in python, however, python turned out to be too slow. To fix that, a small but growing part of the server has been replaced with a Rust equivalent.
See also figure 3 for an outline of the basic dependencies between the different components.
The different components are:

key	value
leisure	park
leisure	common
leisure	nature_reserve
landuse	farmland
landuse	forest
landuse	grass
landuse	meadow
landuse	orchard
landuse	recreation_ground
landuse	village_green
natural	wood
natural	grassland

Table 4: OSM data considered park

key	value
waterway	*
natural	water
natural	bay
natural	beach

Table 5: OSM data considered water

Python components

interface:	contains the necessary functionality to process incoming requests. Every single URL request activates a function from this component.
logic/routing:	contains high-level functions to compute a route.
static:	a bunch of global variables (Yes, I know) each containing a collection or data structure with information about the information the server is mapping. Most of them are loaded from jsons.
logic/grid:	contains support for a spatial bucket data structure, meant for quick storage and access of nodes and edges close to a certain coordinate.
logic/city:	contains functions to load the static data.
logic/projection:	contains functionality to transform lat-lon coordinates in x-y coordinates.
logic/distance:	contains various lat-lon/x-y distance metrics and functions.
logic/graph:	contained a graph data structure with low-level operations. Now contains hooks to the Rust library.
logic/database:	contains hooks to communicate with the database.

Rust components

lib/interface:	contains a list of functionality supported by the library and exported by the .so file.
lib/graph:	contains the graph data structure with low-level functionality and simple routing primitives (like Dijkstra's algorithm).
lib/newtypes:	used to contain a few newtype patterns for use in other crates. Now only contains the Km newtype.
lib/util:	contains a bunch of utilities for use in other crates (e.g. inverted heaps).

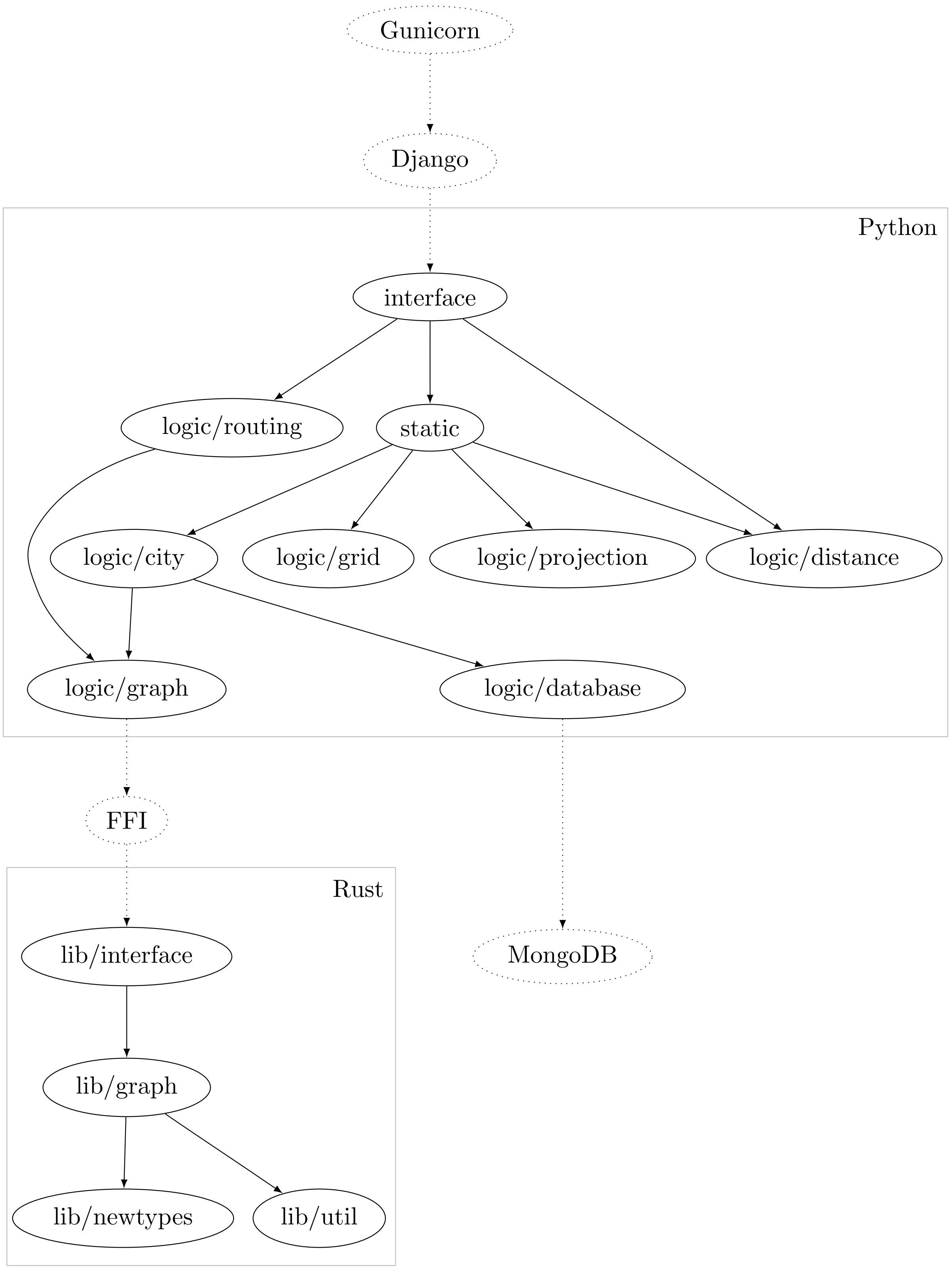

Figure 3: Structure of the server

3.2.1 Component overview

interface
contains the necessary functionality to process incoming requests.

This folder contains multiple files, each either providing an interface, or supporting other files.
closest.py, **nodes.py**, **readme.py** and **routes.py** each provide an interface for requests.
geojson.py provides support for the `type` parameter in requests: `geojson(request, path, markers)` will emit a json-encodable structure that holds the data in indices, coordinates, or geojson format.
util.py contains utilities, supporting the interface.

logic/routing
contains high-level functions to compute a route.

This folder contains most of the routing, poisoning and rating logic. Odds are, if you need to implement a new routing algorithm based on previous primitives, this is where you need to be. However, if your algorithm is completely different and can't use the aforementioned primitives, check out paragraph 3.2.1.
routing.py and **poison.py** contain the current implementation of a routing algorithm.
compress.py contains a function to serialize a route into a string and back.
ratings.py contains functionality to rate a route.
config.py contains a struct determining the parameters of the routing algorithm.
util.py contains utilities, supporting the route generation.

static
Global variables determining the server state.

This folder only contains one useful file: **city.py**. This file contains a definition of `GRAPH`, which contains the data loaded, converted, and processed from two `json` files.

logic/grid
contains data structures for quick location querying.

This folder contains two structures: grid and interval. Both are dependent on each other.
interval.py contains the interval, which is actually a two-dimensional interval (or a rectangle in human terms), spanning a certain size, and supporting simple operations like creation and addition, which returns the smallest interval spanning both arguments.
grid.py is an implementation of a spatial bucket map, with the ability to store intervals and query all intervals close to a given point.

logic/city
contains functionality to load a json and to emit a map.

That's it. Really. JSON goes in, graph data structure comes out.

logic/projection
contains projection functionality

In the original implementations, finding the distance between a point and a line in spherical coordinates proved extremely difficult. Therefore, the coordinates need to be transformed to euclidean coordinates first, by projecting all the points on a plane that minimizes the distortion. This all is implemented by the **util.py** file, which provides, as you guessed, utilities for that.

logic/distance
contains various distance metrics

Basically a collection of utilities that implement functions to calculate the distance from a point to a line, the closest edge to a point, etc.
This all is implemented by the **util.py** file.

logic/graph
contains hooks to a graph class. See paragraph 3.2.1 for the Rust equivalent.

logic/database
contains hooks to the MongoDB database

lib/interface
contains the interface exported in the .so file

lib/graph
contains a graph structure with low level functionality, and some simple algorithms

This crate contains most of the useful code in the shared object file: graphs and routing primitives. For the actual routing algorithms, check paragraph 3.2.1.
graph.rs, **iter.rs** and **graphtrait.rs** contain a low-level graph structure and interface to graph functionality.
poison.rs contains a poisoned version, where some edges have been replaced with poisoned versions.
heapdata.rs contains functionality to work with inverted heaps.
dijkstra.rs contains an implementation of Dijkstra's algorithm.

lib/newtypes
uses the newtype pattern to create a fixed point distance metric

If there is one crate that shouldn't acquire new functionality, it's this one.

lib/util
collection of utilities for graphing algorithms.

This folder contains a few utilities for use in the graph algorithms.
comp.rs contains a Comp struct, used in a heap to make Dijkstra's algorithm work.
distance.rs contains a simple distance metric.
selectors.rs contains tools to select a single random element from an iterator.

4 Database

Some parts of the application use a database. Examples include the `persistence` component of the app and the ratings on the routing server. To provide this storage, a docker instance running MongoDB is provided.
There are 3 user accounts for the database: one admin account, an account for the client and one for the routing server. Their credentials and privilege levels are detailed in table 6. The different databases, collections and their purposed is described in table 7.

user	password	privileges
lopeningent	drigmongo007	manage all user accounts
client	dynamicrunninginghentSmart	read/write on DRIG_unittest and DRIG_userdata
server	dynamicrunninginghentGraph	read/write on edges

Table 6: MongoDB credentials and privileges

database name	purpose
DRIG_userdata	data saved by `persistence` component of the client
DRIG_unittest	copy of DRIG_userdata for testing
edges	saves the ratings on the graph. The `rating` collection is for develop, `rating_production` for production.

Table 7: MongoDB databases and uses

5 Guidelines

This section provides guidelines on how to install the different applications in a test environment, how to modify them, and how to check whether the tests are passing.

5.1 Develop Environment

This section describes the steps to take to set up a develop environment for our application.

5.1.1 Client-side

Install Android Studio (`https://developer.android.com/studio/index.html`).
Clone the git repository into a local directory.
Open the `DynamicrunninginGhent` folder in Android Studio.
(Recommended) Keep Android Studio updated.

5.1.2 Server-side

Running the server is currently only possible on Linux and Mac.

On Linux/Mac: Install python2.7 and python2.7-dev through the main package manager, or from source. `libgeos` and `libffi` are also necessary.
(Optional) Install a virtual environment.
Install Rust 1.16, stable version (other stable versions should work as well, but are currently untested).
Clone the repository into a local directory. When in the `drigServer/` directory, run:

```
cd djangoserver/
pip install -r requirements.txt
```

Every time you want to locally start the server, run (in the same directory):

```
cargo run --release
python manage.py runserver <port number>
```

Testing can be done externally as well. To do this, navigate to the `drigServer/tests/` directory, and run:

```
cargo run <hostname>:<port> <lat> <lon>
```

E.g.

```
cargo run localhost:8000 51.0 3.7
```

On Windows: Install a Linux VM and follow the Linux instructions.

5.2 Commits

GitHub is used in this project to distribute and manage the code between developers. Two repositories are made, one for the server application and one for the client application. Each repository has two main branches:

- The master branch: contains the source code that is production ready
- The develop branch: contains the latest developed changes.

These two branches have the same lifetime as our project and will never be removed. Next there are three supporting branches. These are necessary when more than one developer is developing code. These branches have a limited lifetime and are deleted whenever the task is finished. The supporting branches are:

- Feature branch : whenever a new feature needs to be added, a new branch is made out of the develop branch. Every developer is working on the feature branch. The code can be pushed from the feature branch to the develop branch whenever the feature is implemented.
- Release branch : This branch will be used before every sprint to fix bugs on the code or to complete features when these are insufficient. The code is merged with the master branch and develop branch when the release is ready.
- Hotfix branch: Whenever a bug gets discovered in the master branch, then this bug needs to be solved as soon as possible. Therefore this branch is branched of the master branch. The hotfix branch will be merged back with the master and develop branch whenever the bug is fixed.

On our wiki is a document with commands to create and end these supporting branches in GitHub.

5.3 Testing and Code Quality

Concerning testing it is expected that each developer writes tests for his own code. Not all code has to be unit-tested (this is impossible for e.g. Android Activity classes) but it is important to strive for all code being covered by some kind of test (for activities this would be UI tests/Monkey testing).
For code quality, it is important to regularly check SonarQube and fix bugs, vulnerabilities and try to minimize code smells. Each developer takes care of his/her own code as to avoid unwanted changes. 20% of lines should be comments.
For further details concerning testing and code quality, refer to the Testing and Quality Assurance document.

5.3.1 Server recommendations

Most of the server has been written in a functional, hierarchical style. This has two major implications:

1. Many components inside the server lack any classes, only containing functions that transform data.

2. Not a single element, let it be a function, class, file, or component, contains a cycle.

Please adhere to the aforementioned rules.

6 Releasing new version

When a new release is comming out we want to make sure the release goes as smooth as possible. However stress can influence the smoothness of the release. Important steps can be forgotten or can be executed wrong. Therefore following steps are listed to ease the task of the person responsible for the release.

1. Update the release version in the Mobile.gradle file.

2. Input the following git commands in command prompt to start release:

   - git checkout -b release#number of release# develop //creating the release branch
   - ./bump-version.sh #number of release# //setting version certain release number
   - git commit -a -m Bumped version to // commit release number

3. Input the following git commands in command prompt to end release:

   - git checkout master //change back to master branch
   - git merge -no-ff release#number of release# //merge code from release branch with master branch
   - git push origin master //pushing master branch to the remote master branch on the ugent Git server
   - git checkout develop //change back to develop branch
   - git merge -no-ff release#number of release# //merge code from release branch with develop branch
   - git branch -d release#number of release# // deleting release branch

- git push origin develop //pushing develop branch to the remote develop branch on the ugent Git server

4. Generate Signed apk: in Android the apk can be signed. The wearable app can only be installed if the APK is signed. The signing procedure is done by going to build, generate signed APK in Android Studio. There you give in the path from the keystore. The password from the keystore and key is "DRIGclient", the alias is DRIG. Afterwards you click on next. In build type release needs to be selected. The signature version needs to be "V1(Jar signature)". The APK can be generated if these two options are filled in.
5. Wait for SonarQube and Jenkins: The build first needs to pass in Jenkins and the code needs to pass also the quality gate in SonarQube. Therefore wait 15 minutes after commit.
6. Add release in Github: The release can be added in Github by clicking on release in GitHub. On this webpage the tag, title and description of the release can be added. Do not forget to add a link to the URL of the release notes with the following markdown input [title of URL](www.url.be) in the description field.
7. Release notes: The release notes will need to be added to the wiki whenever the release is approaching. Every feature needs to be added to the wiki whenever it is finished. Stress will increase if the release notes are postponed to the release date. The release notes are started as soon as possible to avoid stressful circumstances at the end of the release.

The team has made an agreement about the last development day of features. No new features are allowed to be pushed to the git server two days before the release date. This will give us plenty of time to test every new feature, to fix bugs and to solve the SonarQube issues.

7 Credentials

Jenkins tutors:tutors
Admin access can be requested by mail via olivier.cammaert@ugent.be.

SonarQube tutors:tutors
Admin access can be requested by mail via olivier.cammaert@ugent.be.

Fabric Access to Fabric can be requested by sending a mail to lorenz.vanherwaarden@ugent.be.

Firebase Admin access to Firebase can be requested by sending a mail to maxim.deweirdt@ugent.be.