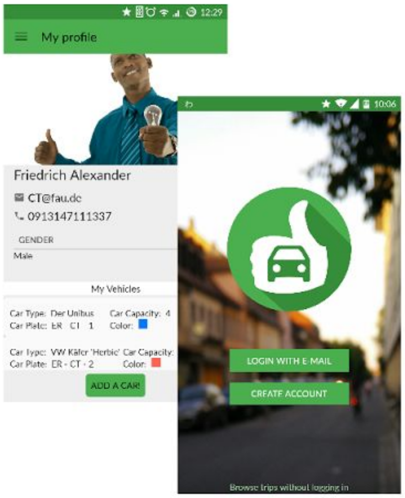
CroudTrip!

sponsored by Elektrobit

The CroudTrip! application wants to revolutionize the car-ride-sharing market with its easy, user-friendly and highly automated way of organizing shared Trips!

The Product



Friedrich Alexander
✉ CT@fau.de
☎ 0913147111337

GENDER
Male

My Vehicles

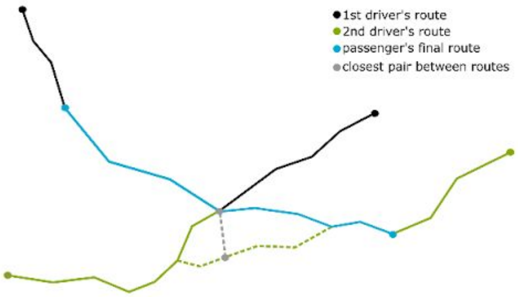
Car Type: Der Unibus Car Capacity: 4
Car Plate: ER CT 1 Color: 🟦

Car Type: VW Käfer 'Herbie' Car Capacity:
Car Plate: ER - CT - 2 Color: 🟥

- Offer and join shared Trips at short-notice!
- For drivers: Easily find passengers on the way you are going anyway ... and earn money with it!
- For passengers: Reach your destination comfortably!
- We will automatically match you to the best offer in real-time!
- Simply check-in and check-out of your Trips using NFC on your device!
- No direct Trips? No problem - Join a SuperTrip! with multiple drivers!

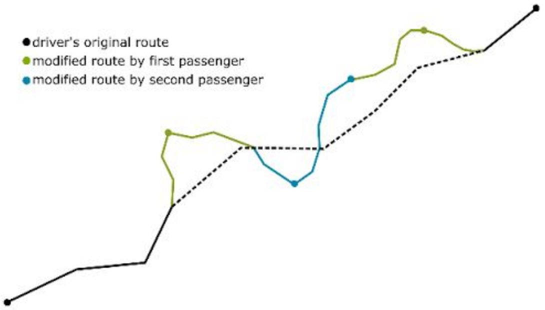
The Concept

SuperTrip!



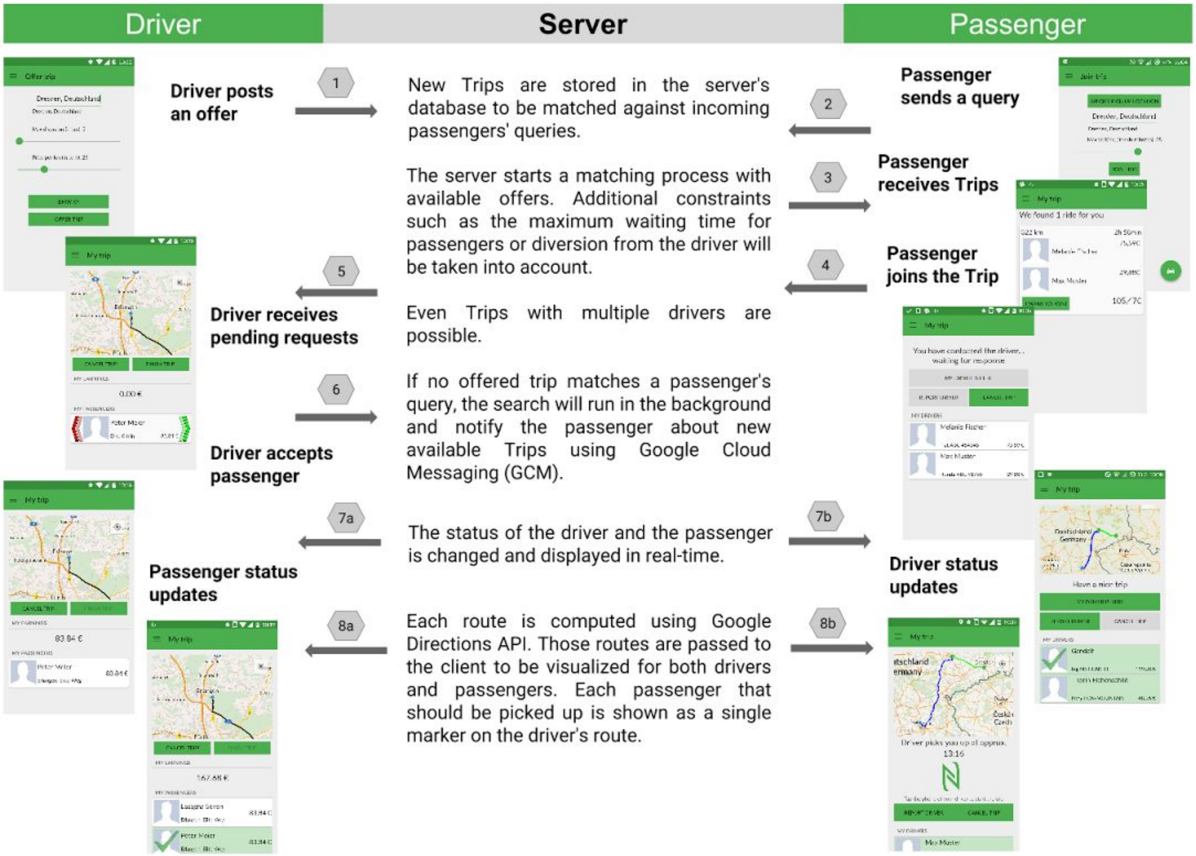
- Combine multiple offered routes to serve passengers even if there is no direct connection available
- Find routes which can pick up a passenger from his start position or drive to his final destination
- Subdivide those routes, compute the closest pair of those waypoints and use it as a “connection point”
- If the distance of the closest pair is too large, start a recursive matching process with these two waypoints

Multiple Passengers


● driver's original route
● modified route by first passenger
● modified route by second passenger

- Match multiple passengers with one driver who will pick them up and bring them to their destinations in an optimal order
- Optimal order is constrained by given internal order of each waypoint pair, because each passenger has to be picked up before the driver reaches his destination location
- Compute optimal order by solving the Travelling Salesman Problem via Brute Force (max. 4 passengers)

The Interactions



Driver posts an offer (1)

New Trips are stored in the server's database to be matched against incoming passengers' queries.

Passenger sends a query (2)

The server starts a matching process with available offers. Additional constraints such as the maximum waiting time for passengers or diversion from the driver will be taken into account.

Passenger receives Trips (3)

Passenger joins the Trip (4)

Driver receives pending requests (5)

Even Trips with multiple drivers are possible.

Driver accepts passenger (6)

If no offered trip matches a passenger's query, the search will run in the background and notify the passenger about new available Trips using Google Cloud Messaging (GCM).

Passenger status updates (7a)

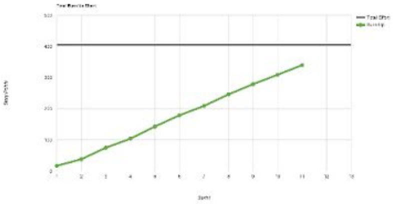
Driver status updates (7b)

The status of the driver and the passenger is changed and displayed in real-time.

(8a) (8b)

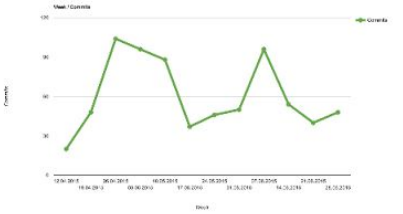
Each route is computed using Google Directions API. Those routes are passed to the client to be visualized for both drivers and passengers. Each passenger that should be picked up is shown as a single marker on the driver's route.

The Process



Total Messages

27% #channels, 1% groups, 72% DMs



- Total # of story points: 370
- Development Speed: 30.9
- 13.23% of total effort used for bugfixing

- Slack as main communication tool
- Integrations for Travis CI, Github and Crashlytics

- Total # of Commits: 727
- Lines of Java Code: 15362
- Lines of Comments: 3938

The Team

Alexander Popp (SD), Philipp Eichhorn (SD), Frederik Simon (SD), Vanessa Lange (SD), Nazeeh Ammari (SD), Ricarda Hohn (PO), Michael Weber (PO), Maximilian Capraro (University Coach)

The Sponsor

Bernd Hardung, Elektrobit

Claus Stellwag, Elektrobit

Elektrobit

The University

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

CroudTrip Research Documents

Multiple Passengers for drivers

Goal

Drivers usually have multiple seats in their cars and therefore could transport more than only one passenger. But adding new passengers to an existing trip is not as simple as it seems.

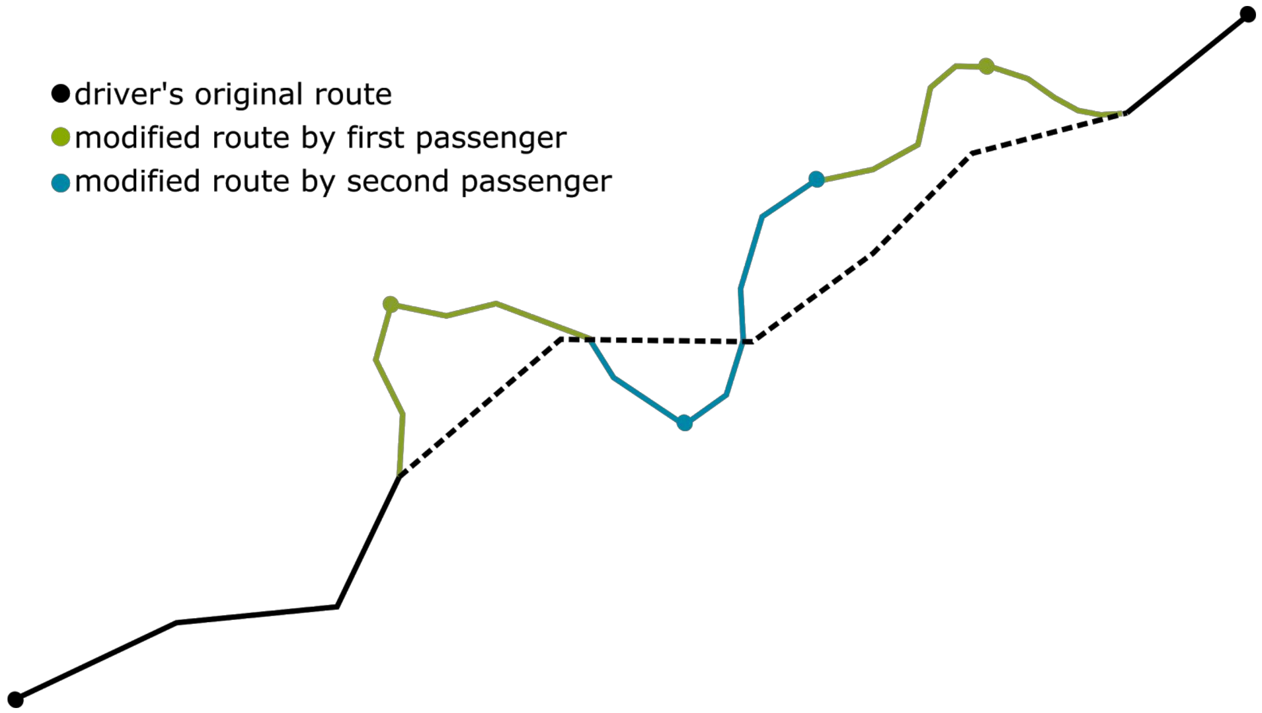
Problems and Restrictions

Calculating the optimal route for multiple passengers joining one trip is a traveling salesman problem with additional constraints (e.g. start point cannot come after end point). As such there are two general ways of solving it:

- Brute force
- Heuristics

Additional considerations regarding the Google directions API:

- Direction requests can contain a number of waypoints which should be included in the route
- Optionally those waypoints can be optimized by Google to yield the shortest possible route
- Max. 8 way points (= max. 4 passengers)
- 2500 requests per day
- Only 2 requests per second


● driver's original route
● modified route by first passenger
● modified route by second passenger

Brute force

Traditional traveling salesman problem have a complexity of O(n!), where n is the number of waypoints. In our complexity is slightly lower as some routes are not possible due passengers going only from A to B but not from B to A. Unfortunately the number of solutions is still quite high:

Passenger Count	Possible routes
1	1
2	6
3	90
4	2520
5	113400

Since we are restricted to only 4 passengers per driver (see the max 8 waypoints for a route from google), brute force solving the TCP with 4 passengers seems feasonable. However, due to the tight quota restrictions we cannot perform 2520 request per passenger query, hence we need an alternative solution for deciding which route is the shortest one without using the Google Directions API.
One solution is to approximate the distance between two waypoints by the computing airline distance between them. The result will therefore not be the optimum solution, but should be good enough for our needs.

TSP solving takes place after the passenger sends a query to the server. After TSP solving we will check if the max diversion and waiting time parameters are fulfilled and will either show this result to the passenger as a potential match or throw it away.

Multiple passenger considerations

- Include max driving time for passenger (e.g. passenger wants to be at destination in max. 10 minutes)
- Before adding another passenger to route, check if previous passenger constraints are still met (max waiting time and max driving time)
- Inform passenger about change time when driver arrives (e.g. he is picking up another passenger before)
- After passenger cancels trip inform driver about update
- Store route legs within the routes to save additional API calls (to get distances from one waypoint to another)
- Show driver a view as soon as he gets near to a passenger where he could choose “picked up passenger” or “passenger was not here”. Server will handle route calculation after this answer.

Integrating SuperTrips!

Goal

Connect multiple Drivers in one so called super trips. Each driver of a SuperTrip! will take the passenger a piece of the total route and so the passenger is even able to reach his destination, even if no direct connection with only one driver is possible.

Main Restrictions

- Directions API calls: we should do as few direction calls as possible
- Time: The user wants to get results as quickly as possible, solving TSP in background and searching for super trips could take a while

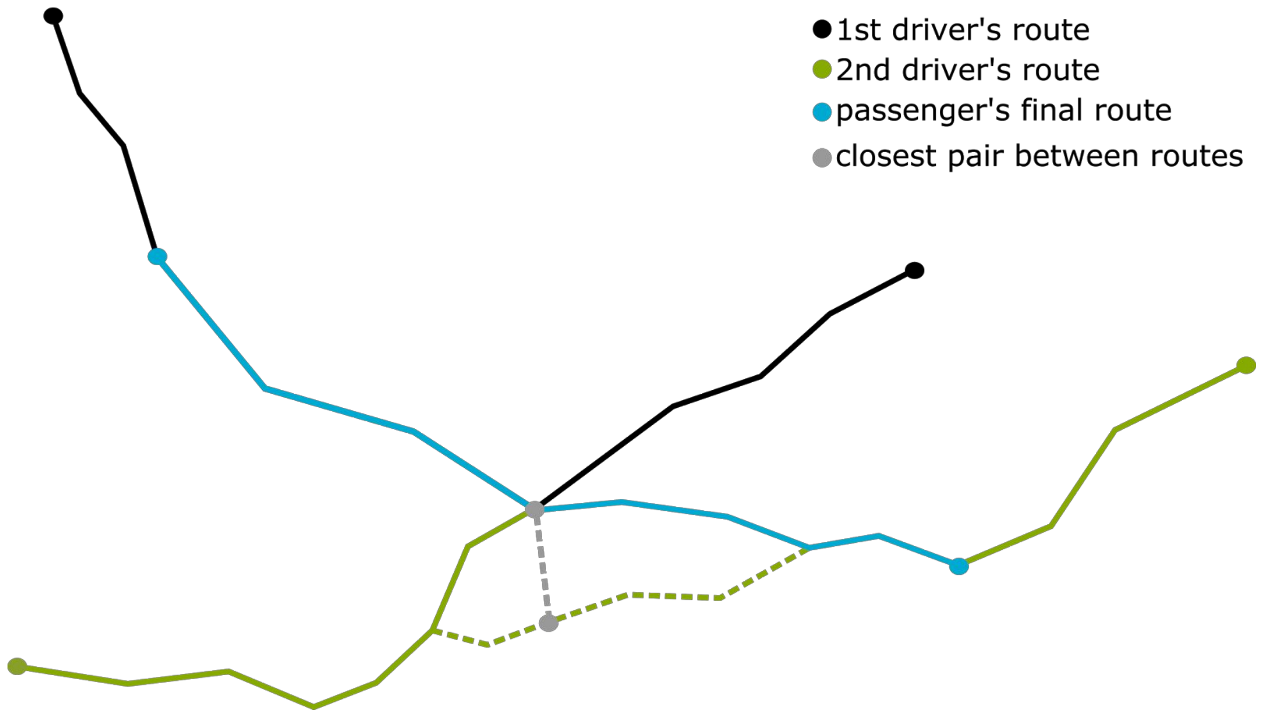
Basic Solution ideas

- Return as soon as possible: maybe it will not be the best result, but the user will get a result in time, with as few direction calls as possible

First algorithm thoughts to find super trips

Passenger Start (B), Passenger Destination (C)

1. If we don't find an offer that is taking the passenger from B to C, we will do an airline search to find possible routes that are picking the passenger up from B (pick-up-offers) and another airline search to find possible routes that are taking the passenger to C (drop-offers).
   So now we have two lists with all the routes that could pick up the passenger and that could drop off the passenger somewhere. We "just" have to connect these offers.
2. Compute routes containing B/C and subdivide those routes into multiple waypoints (we have the polyline information which is a decoded location information, so this should be possible). We just need all the points after pick up at A and for the other offers all the points before drop-off at C.
3. Now we have pairs of subdivided routes and we can run the closest pair algorithm on it to find the closest corresponding point pair that both drivers have on their trip. Name this Pair (E,F).
4. We check, if the first driver is able to drive to F (the new drop off for the passenger) or if the second driver is able to pickup the passenger at E.
5. Check if the arrival times at those points will match.
6. If one is possible we found a super trip consisting out of two drivers.
7. If it is not possible for either of them we have to solve a new trip request from E to F. So the problem gets recursively called to find a driver that can bring our passenger from E to F.
8. We should limit the depth of the recursion, since it is no fun to switch the car all the time.



Some Problems that could occur during super trips:

- Passenger is in the car of driver A and driver B cancels his trip.
- What happens if a driver accepts additional passengers after a passenger within a super trip joined? We cannot guarantee that the arrival time will still be the same.
- Shall we really compute all the cheapest super trips for the second cheapest price? Or shall we just say: As long as we find one super trip he could take this one.
- A passenger that should be taken up by the driver while super trip computation was running, does cancel his trip. We will have a time divergence at the meeting point.
- How long shall the maximum waiting time at the meeting point be?

Google Directions API calls:

1. Let’s say we find **n** offers over B, and **m** offers over C.
2. (n+m) directions calls (maybe even some more because we have to calculate diversions by using old direction routes (that ideally should be cached)
3. no direction calls
4. (n+m) direction calls (maybe even some more because we have to calculate diversions by using old direction routes (that ideally should be cached)

=> 2(n+m) direction calls. So in worst case we have 4n additional calls.
Currently we have n directions calls per query.
Hence we get at a maximum of 5n*depth direction calls per query.

Ideas to save some more direction calls

- Do a first arrival time check before point (4). If the arrival times differs too much (maybe taking distance between E & F into account) it is not probable that both drivers will be at the connection point at the same time.
- Check for distance between E & F. If the distance is too big, store this result for later and try to find another one. If E&F are far away from each other it is rather unlikely that one of the driver will make the diversion, but it's rather more likely that we need another driver in this trip (recursive call is expensive).
- Store every route that is computed while computation runs and reuse as many of the routes as possible (e.g. the current route of that driver needs only to be computed one time within the whole query process).
- We could use stored routes even for a rough check after (1). We could insert an additional Closest Pair algorithm to compute pairs of offers that are rather close together (and it is more likely that this will be a good super trip).

Finding good connection points for SuperTrips!

Goal

Connecting Super Trips means that two drivers will drop respectively pick up the passenger at a connection point that is not defined by the passenger, but is a result of our closest pair algorithm on the matching routes. Since this closest pair computation only relies on the distances between two coordinates, it can easily happen that the computed connection point is not a good location to stop the car (for example on the highway). So our goal is to find a better connection point for both routes where it is easily possible to stop the car and drop the passenger.

Considerations

Once we have computed the closest pair and therefore have two possible connection points, we can easily start a Google Places API call on both of these points and check the nearby surrounding for so called point of interests (POI). These POI are usually locations where it should be easy to stop the car. Instead of now testing our original connection points we could check whether both drivers could be connected at one of these POI.

We don't really have any additional effort except for the Places API call, because it does not matter whether we check both routes against the closest pair results or some POI.

Problems

There are still some problems that matter for our project:

Direction Calls

If we have many POIs we maybe have to check multiple points and therefore to compute multiple routes to theses POIs. So we would need some more direction calls that we cannot save. What we could do is to limit the count of POIs that we are checking and matching the first fitting result. But we will probably check more than only 2 POIs.

Not available POIs

We also have to be aware that it can easily happen, that there is no valid car route for a specific POI (for example if the POI is within a pedestrian area). In that case we have spent a directions call for nothing and have to test additional ones. It is even possible that we cannot reach any of the given POIs or there is no POI in the surrounding. For our prototype we could probably ignore that case and just use the original closest pair as connection resul

Implementation

Implementation should be not that difficult, since we only have to implement the Places API calls in our server. But there is currently no library that handles these API calls (there are libs for the other APIs we are using), so we would be forced to implement the connection to Google on own, which should cost some time, but not be impossible.

Future Work

An additional thing that could be implemented (but not in our prototype) is to use Google’s Road API (or any similar navigation API) which would give you the possibility to get the speed limit for a specific location. So one could easily check the speed limit at the connection points and if it is rather slow (in the city for example) that one could assume that the driver will be able to find a position to stop the car and do the POI-search only for higher speed limits (highways) where it is not that easy to stop. Unfortunately this speed limit feature is only available for Google Customers and so we cannot use it within our project.

What would be possible even without paying Google is to snap the connection points to roads and compare the names of the roads and only to the POI check for highways and similar roads. But that’s probably not necessary within our time limits for this project.

Integrating ComboTrips

Goal

To find the shortest / cheapest / fastest route between two points regardless of the transportation type (foot, bike, car, bus, train, plane, boat, …).

Approach

In order to create routes which are made up of arbitrary transportation mediums, a generic framework is required where new transportation vendors can be added. Possible details which are required for new transportation types could include:

- Start / End location of trip (trivial for cars / planes, bus / train routes could be “split” at each station)
- Expected departure and arrival times
- Whether or not those times are flexible (car might be willing to wait, train not)
- Route (including possible way points. E.g. car driver prefers this street vs that street)
- Whether or not detours are allowed on the route (car can pick up passenger at arbitrary locations, plane cannot)
- Price (including where + how to conduct payment)

Integration

Other than regular cars, CroudTrip will have to integrate with external services for fetching data about public transport. Realistically this integration should be done in such a way that there is no work required by public transport providers and the CroudTrip development team can handle all of the actual work. Some considerations include:

- Up to date data: public transport is often combined with last minute changes. Update frequency needs to adjust for that
- Use existing GTFS (General Transit Feed Specification) platforms for subscribing to public transit information where possible

Routing

Even without public transport the problem of calculating optimal routes for passengers is a core problem for CroudTrip. Essentially the problem is a traveling salesman problem and as such it is impossible to determine the optimal route once sufficient possibilities exist (e.g. enough public transport providers have been added).

A couple of techniques which can be used to reduce the amount of data which has to be analyzed:

- Using quadtrees for filtering out trips which are no where near the passenger early on (https://en.wikipedia.org/wiki/Quadtree)
- Starting with “long distance” public transportation types first: when trying to go from Paris to New York its very likely a passenger will be traveling most of the distance via plane. Possible ordering: Plane, Train, Car, Bike, Foot

Using external services like Rome2Rio

Services like Rome2Rio (http://www.rome2rio.com/) have already done a huge amount of work in regards of combining various public transport mechanisms. Coverage of Rome2Rio can be found at http://www.rome2rio.com/coverage.

While integrating with such a services reduces the development effort, it can also result in a reduced user experience as the resulting routes no longer have to be optimal, since CroudTrip can only combine its car trips with those of Rome2Rio by “smart guessing”. To illustrate the problem here’s a simple example:

Assume a route A → B → C → D where a passenger wants to go from A to B. In addition there is a driver going from A to C and a train (data from Rome2Rio) B to D. Since CroudTrip does not know that this train exists at first (it would have to “guess” B and D to do so) CroudTrip could try to bring passenger from A to C using the car and then use an alternative service to go from C to D. However, if all information would have been available to one application, the route A → B by car and B → D by train would have probably been optimal.

Moovel is a service similar to Rome2Rio (https://www.moovel.com/en/DE/) but does not offer a public API for using their data.

Client considerations

The current client implementation of CroudTrip was designed to deal with trips via cars which can be very dynamic (routes can change, pickup / dropoff locations vary) and where there is always an explicit step of passengers accepting drivers and via versa. Public transport does not have any of this. As such the client UI varies depending on the scenario but public transports do not seem to impose any major difficulties here.

Glossary

ComboTrip	A Combo Trip! is a Trip by the passenger that connects individual routes by different drivers and public transport to a specific destination.
Connection Point	A SuperTrip! connection point is the exact location where 2 different routes cross and a passenger on a SuperTrip! is dropped of and picked up again. This is a point which both individual drivers pass on their route.
SuperTrip	A SuperTrip is a Trip by a passenger, that connects various individual routes by different drivers to reach the destination.