Lösungsskizze

Erstellung neuer geographischer Routen anhand von existierenden Routen

1 Projektbeschreibung

Ziel dieses Projektes ist es, eine Webanwendung zu entwickeln, die eine soziale Plattform bietet, um Touren mit anderen Leuten zu planen, neue Routen zu erstellen und diese zu bewerten. Kern dieser Anwendung soll ein Algorithmus sein, der auf der Basis schon bestehender Routen automatisch oder manuell eine neue Route berechnen und hinzufügen soll.

Die Anwendung teilt sich in folgende Komponenten auf:

- Import-/Export-Funktionalität
- soziale Komponente
  - Benutzerverwaltung
  - Forum
  - Bewertungsfunktion
  - ...
- Persistenz der Routen
- Sammeln von Routen
- Berechnung neuer Routen
- Visualisierung der Routen

Da die Entwicklung aller Komponenten wahrscheinlich den Zeitrahmen dieser Veranstaltung überschreiten würde und dieses Projekt als *Proof-of-Concept* dienen soll, wird die „soziale Komponente“ im Rahmen der Ausarbeitung außen vor gelassen.

2 Installation

Damit die Anwendung kompilations- und lauffähig ist, müssen folgende Programme vorinstalliert sein:

- Java 8 JDK[1]
- PostgresSQL 9.6.1[2]
- PostGIS 2.3.1 [3]
- Eclipse Neon mit Maven-Plugin [4]
- WildFly 10.1 Application Server[5]

[1] `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`
[2] `https://www.postgresql.org`
[3] `http://postgis.net/`
[4] `https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/neon3`
[5] `http://wildfly.org/`

Die Quelldateien können unter `https://github.com/simhue/wisscontmgmtprak` heruntergeladen werden. Zunächst muss in der Datenbank das Schema angelegt werden. Das dazugehörige SQL-Skript liegt unter `sql\createdb.sql`. Im Skript müssen ggf. noch Benutzer und Datenbankname angepasst werden.

Sobald das Schema angelegt ist, kann der *TourenCrawler* kompiliert und gestartet werden. Dazu müssen der *TourenCrawler* und *TourenComputer* in Eclipse als Maven-Projekte hinzugefügt werden.

1. Eclipse starten und neuen Workspace anlegen
2. File → Import → Maven → Existing Maven Projects Ordner auswählen, in dem die Projekte liegen
3. Projekte auswählen und hinzufügen
4. Im TourenCrawler-Projekt die Datei `config.properties` öffnen und die Datenbankdaten eintragen.
5. Rechtsklick auf TourenCrawler-Projekt → Run As... → Java Application
6. `de.unileipzig.contentmanagement.praktikum.Crawler` aus der Liste Auswählen und Ok drücken

Jetzt wird der *TourenCrawler* gestartet und er importiert 100 Strecken in die Datenbank.

Damit der *TourenComputer* eine Verbindung mit der Datenquelle aufbauen kann, müssen die Verbindungsdaten im WildFly hinterlegt werden.[6] Darauf achten, dass der JNDI-Name der Verbindung als `java:/PostgresDS` gespeichert wird.

Bevor der *TourenComputer* gestartet werden kann, muss der Schlüssel für die *GoogleMaps*-API[7] noch in die Konstante `G_MAPS_KEY` der Klasse `de.contmgmt.praktikum.MapBean` eingetragen werden. Wenn das erledigt ist, kann der *TourenComputer* als Server-Anwendung gestartet werden (Rechtsklick auf *TourenComputer*-Projekt → Run As → Run on server). Ggf. muss hier noch der WildFly-Server als Server hinzugefügt werden.[8] Sobald die Webanwendung gestartet wurde, kann der *TourenComputer* unter `http://localhost:8080/MotorcycleTouringComputer/` aufgerufen werden.

3 TourenCrawler

Um eine Grundlage zu bieten, damit eine neue Strecke berechnet werden kann, müssen vorerst existente Routen von anderen Plattformen gesammelt und in der Datenbank abgelegt werden. Als Basis dient eine Auswahl von Strecken im `KML`[9]-Format der Zeitschrift *MOTORRAD*[10]. Die Strecken https://www.postgresql.org/download/werden vom *TourenCrawler* iterativ heruntergeladen, vom `KML`-Format in das Datenbankschema überführt und anschließend abgespeichert. In diesem Vorgang werden `KML`-Dokumente, die keine Koordinate enthalten, verworfen.

Da die Routen manuell von Nutzern angelegt wurden, sind einige Strecke mangelhaft. Das kann bedeuten, dass GPS-Koordinaten abseits von befahrbaren Straßen hinzeigen oder dass eine Folge von GPS-Koordinaten nur approximativ einen Straßenverlauf widerspiegelt. Ein ungefährer Straßenverlauf beeinträchtigt die Qualität der Route, wenn zwei aufeinanderfolgende Koordinaten eine bestimmte

[6] `https://yellowshoes.de/2015/01/05/wildfly-jdbc-treiber-und-datasource-einrichten/`
[7] `https://developers.google.com/maps/documentation/javascript/get-api-key?hl=de`
[8] `http://www.mastertheboss.com/jboss-server/wildfly-8/configuring-eclipse-to-use-wildfly-8`
[9] `https://de.wikipedia.org/wiki/Keyhole_Markup_Language`
[10] `http://maps.motorradonline.de/`

Entfernung zueinander überschreiten. Durch diese „Lücke“ ist unklar, wo die eigentliche - vom Autor gewollte - Strecke entlangführen soll. Das macht eine mögliche automatische Angleichung schwierig, da interessante Teilabschnitte der Strecke übersprungen werden könnten.

Auch ist es herausfordernd, eine legitime Maximalentfernung zwischen zwei Koordinaten zu finden. Eine Idee war, dass sich die Qualität einer Strecke anhand der Anzahl der Koordinaten relativ zur Gesamtlänge der Route berechnen lässt. Dafür wurde mit Hilfe der *PostGIS*-Stored-Function `ST_Length(linestring)`[11] die Länge von 898 Strecken ermittelt und daraus die ungefähre durchschnittliche Entfernung $296m$ berechnet. Ein Überschreiten dieses Durchschnittswertes wurde als Ausschlusskriterium genommen. Bei Betrachtung des Ergebnisses des Testimports fiel auf, dass ein Großteil der Strecken verworfen und unsaubere Strecken nicht gefiltert wurden. Da die „Lücken“ vereinzelt in sonst fehlerlosen Strecken auftreten können, muss jeder Koordinatenpunkt p_i mit dem folgenden Punkt p_{i+1} bzw. die Entfernung der beiden Punkte zueinander verglichen werden[12]. Nach Auflistung der größten Entfernungen ließ sich $1500m$ als akzeptabler Grenzwert festlegen. Die größere Entfernung verhindert, dass Strecken verworfen werden, die willentlich längere Geraden enthalten (z.B. eine gerade Landstraße oder Autobahn). Dieser Ansatz hat sich als der erfolgreichere Filter herausgestellt und wird weiterhin als solcher genutzt.

4 TouringComputer

4.1 Visualisierung der Routen

Zur Visualisierung wird das *PrimeFaces*[13]-Framework in der Community-Edition benutzt. Das Framework ermöglicht eine einfache Realisierung von einem responsiven Design und bietet eine Schnittstelle zu der *GoogleMaps*-API[14]. Damit die Anwendung einen Ausgangspunkt hat, muss dieser als erstes gesetzt werden. Entweder kann der Benutzer einen Punkt auf der Karte anklicken oder er gibt eine Adresse in das Feld „Von“ ein und klickt auf den Setzen-Button. Bei der zweiten Option wird die Adresse mittels *GeoCoding*[15] der *GoogleMaps*-API in eine geographische Koordinate überführt und auf der Karte dargestellt.

Nachdem ein Startpunkt selektiert wurde, muss noch ein Umkreis ausgewählt werden, in dem Routen dargestellt werden sollen. Dafür gibt der Nutzer eine Kilometer- oder Meilenanzahl an - die Einheit kann ausgewählt werden - und klickt auf den Suchen-Button. Im Folgenden wird eine SQL-Anfrage an die Datenbank gesendet. Die Ergebnismenge der Anfrage enthält alle geographischen Objekte, die sich innerhalb eines Umkreises um den gesetzten Punkt befinden.

```
SELECT route.id, route.description, points.id, points.point FROM route
   JOIN routepoints ON route.id = routepoints.routeid
   JOIN points ON routepoints.pointid = points.id
   WHERE ST_DistanceSphere(point, ST_MakePoint(longitude, latitude)) <= radius *
       distanceUnit
   ORDER BY route.id, points.id
```

Diese SQL-Anfrage ist auch der Grund warum entschieden wurde, dass jede Koordinate einer Route einzeln als Punkt in die Datenbank abgelegt wird und nicht gesammelt als *LineString*-Objekt[16]. Da

[11] `https://postgis.net/docs/ST_Length.html`
[12] `ST_Distance(p1, p2) https://postgis.net/docs/ST_Distance.html`
[13] `https://www.primefaces.org/`
[14] `https://developers.google.com/maps/?hl=de`
[15] `https://developers.google.com/maps/documentation/javascript/geocoding?hl=de`
[16] gezogene Linie entlang der Reihenfolge der inkludierten Punkte

die Umkreisabfrage alle geometrischen Objekte zurückgibt, würde das Resultat *LineStrings* als Ganzes enthalten. Das bedeutet, dass auch Punkte außerhalb des Radius zurückgegeben werden. Die Stored-Function `ST_DumpPoints` gibt eine Liste aller Punkte innerhalb eines geometrischen Objektes zurück, jedoch ist dann nicht gewährleistet, dass die Punkte in der Reihenfolge aufgelistet werden, wie sie im *LineString* definiert sind. Die einzelnen Punkte in der Datenbank können anhand der automatisch inkrementierten ID auf- oder absteigend geordnet werden.

Die Resultate werden in entsprechende POJOs geladen. Jede Route wird als *LineString* in das *GoogleMaps*-Modell hinzugefügt, damit dieses die Routen anzeigen kann. Jede Route wird dabei anders farblich kenntlich gemacht, damit der Benutzer die einzelnen Strecken visuell unterscheiden kann. Zur besseren Erkennung eines Streckenverlaufs kann jede Strecke angeklickt werden. Darauf wird die Transparenz der Strecke aufgehoben und die Strecke hebt sich deutlich von den anderen ab.

Abbildung 1: 20km Umkreis

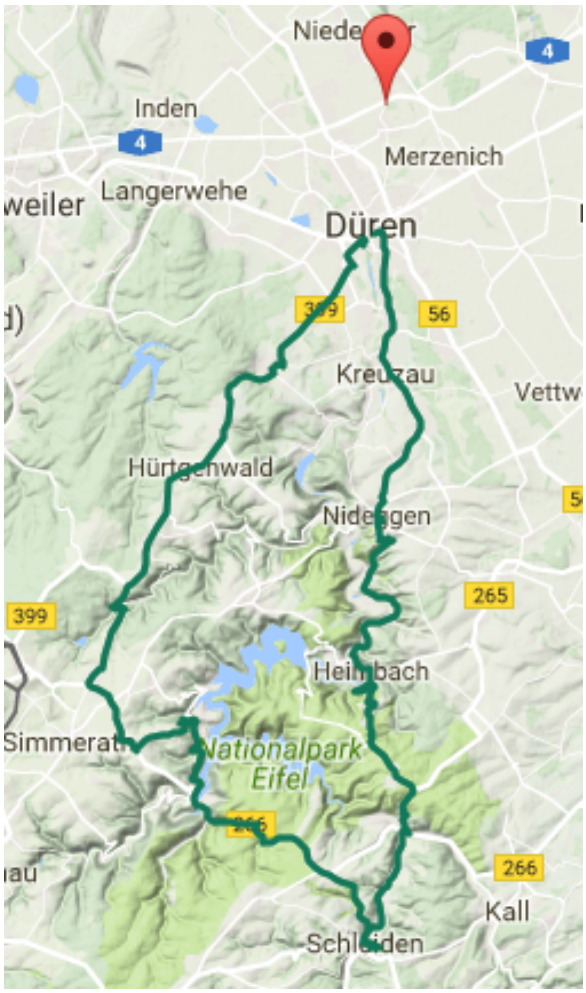


Abbildung 2: 200km Umkreis

Dadurch, dass Punkte außerhalb des Betrachtungsradius ignoriert werden, treten „Lücken“ auf, wie sie im *TourenCrawler*-Kapitel beschrieben wurden (s. Abb. 1). Hier ist es schwieriger eine Maximalentfernung zu definieren wie beim Importvorgang, da dort die Routen manuell angelegt wurden und es wahrscheinlicher ist, dass längere Geraden gewollt sein können. Das Angleichen der Gerade an Straßenverläufe durch die *SnapToRoads*-Funktion der *Google-Roads*-API ist problematisch, da diese API am effektivsten funktioniert, wenn zwei Punkte nahe zueinander sind.[17] Jedoch wird in der Dokumentation der API nicht genau erläutert, welche Entfernung optimal ist. Da die ID's der Punkte innerhalb einer Route beim Speichern inkrementiert werden, ist die Differenz der ID's von zwei aufeinanderfol-

[17] `https://developers.google.com/maps/documentation/roads/snap?hl=de#demo`

genden Punkten stets 1. Eine größere Differenz deutet darauf hin, dass Punkte der eigentlichen Route, wie sie in der Datenbank abgespeichert wurde, sich nicht innerhalb des Suchgebiets befinden. Wenn das der Fall ist, handelt es sich hier um eine Lücke. Dieser Ansatz ist unabhängig von der Länge der Lücke. Die Gesamtroute kann man nun zwischen den Endpunkten der Lücke aufteilen und diese Route als zwei separate Routen behandeln. Es muss noch untersucht werden, ob der Ansatz in der Praxis wie erwartet funktioniert.

4.2 Routenberechnung

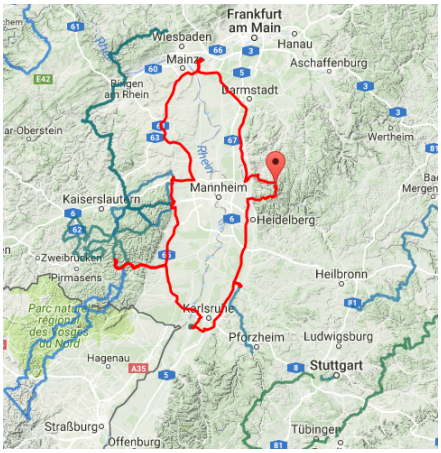


Abbildung 3: In Rot: neue Streckensegmente

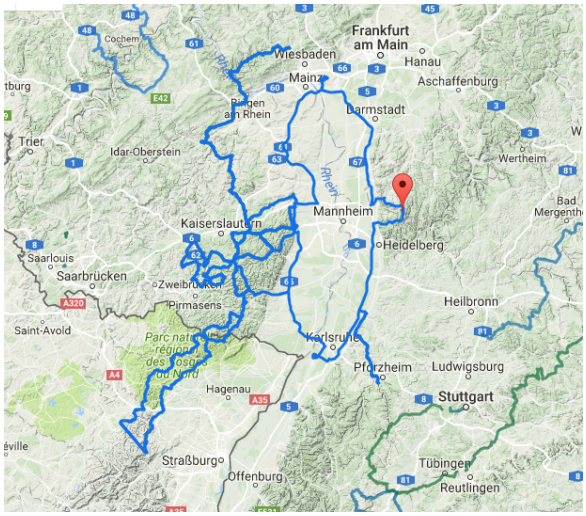


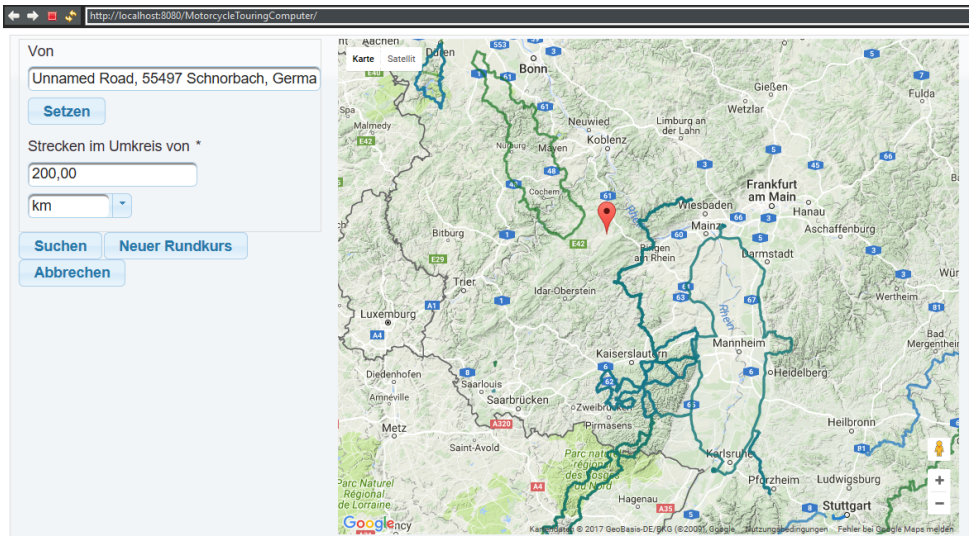
Abbildung 4: Hervorgehoben: neue gespeicherte Route

Der Algorithmus der Routenberechnung nimmt vorher gesetzten Startpunkt als Ausgangspunkt. Die Endpunkte jeder Route im Suchkreis werden gesammelt und mit Hilfe von `ST_ClosestPoint` wird der geographisch naheliegendste Punkt aus der Menge zurückgegeben. Durch die *GoogleMaps*-API wird eine neue Teilstrecke zwischen diesem und dem Startpunkt berechnet. Die neue Teilstrecke wird der neuen Gesamtstrecke als erstes Segment hinzugefügt und der Startpunkt, der auch gleichzeitig das Ende der gesamten Route sein soll, wird der Menge hinzugefügt. Darauf wird die Strecke des Endpunktes ebenfalls hinzugefügt. Der zweite Endpunkt der Strecke wird als neuer Ausgangspunkt gesetzt. Die bisher benutzten Punkte (Startpunkt und die Endpunkte der naheliegendsten Strecke) werden aus der gesammelten Menge entfernt. Mit dem neuen Ausgangspunkt wird der Vorgang wiederholt, bis der Zielpunkt (gleich dem Startpunkt) als naheliegendster Punkt zurückgegeben wird.

Dem Benutzer werden die neuen Teilsegmente als rote Linien präsentiert (Abb. 3). Ist der Benutzer mit dem Resultat zufrieden, kann er die neue Strecke speichern, anschließend auswählen und als `KML`-Dokument exportieren.

5 Fazit und Ausblick

Die Webanwendung befindet sich noch im frühen Status der Entwicklung, jedoch bietet die Anwendung eine gute Basis für eine soziale Plattform, in der sich Menschen über gemeinsame Touren austauschen können. Die Entwicklung hat gezeigt, dass es möglich ist aufgrund bestehender Routendaten eine neue Strecke berechnen zu lassen. Wegen der prototypischen Implementation des *TourenComputers* wurden nur die Endpunkte einer jeden gefundenen Route im Suchkreis als Zielpunkte für die *GoogleMaps*-Routenberechnung benutzt. Ein Ziel ist es, dass jede Koordinate einer Route als Anker benutzt werden



kann. Daraus ergeben sich dann für die weitere Berechnung mehrere Optionen, da der weitere Verlauf der Ankerroute nun in zwei Richtungen weitergehen kann. Entweder könnte man den Verlauf in beide Richtungen folgen und dem Benutzer mehrere Ergebnisse anbieten oder man nimmt jeweils nur eine Richtung (z.B. die Richtung des kürzen oder längeren Teilstücks).

Auch sollen noch weitere Filteroptionen für die Routenberechnung hinzugefügt werden. So soll der Benutzer zum Beispiel auswählen können, dass nur Strecken im Suchkreis gesucht werden, die eine Mindestbewertung haben oder einem bestimmten Höhenprofil entsprechen. Folglich soll die Anwendung noch um ein Bewertungssystem erweitert werden und das Höhenprofil einer Strecke soll ersichtlich werden.

Die *GoogleMaps*-API bietet schnelle Beantwortung aller Anfragen, jedoch ist die Anzahl der täglichen Anfragen limitiert.[18] Um der Einschränkung zu entgehen, könnte man das System auf eine ähnliche API (z.B. OpenStreetMaps[19]) umstellen.

[18] `https://developers.google.com/maps/documentation/geocoding/usage-limits?hl=de`
[19] `https://www.openstreetmap.de`