iTixi

Disposoftware für TIXI-Betriebe

Design und Implementation

1 Inhaltsverzeichnis

2 Design

2.1 Domain

Das Resultat einer der fundamentalsten und zugleich wichtigsten Prozesse in der Konzeption einer Unternehmensanwendung ist der Entscheid, wie die Geschäftslogik auf die Software abgebildet werden soll. Dazu können je nach Komplexität der Geschäftslogik unterschiedliche Ansätze verfolgt werden. In diesem Kapitel sollen generelle Architekturentscheide und weitere Top-Level-Strukturen beschrieben werden. Informationen zur konkreten Implementation finden sich im Kapitel 10 Implementierung.

2.1.1 Architekturentscheid Domain Model / Domain-Driven-Design

Nach Fowler [1] können grundsätzlich die folgenden drei Pattern unterschieden werden, nach denen Geschäftslogik innerhalb einer Software organisiert werden kann: Transaction Script, Domain Model und Table Module (es wird im Rahmen dieser Dokumentation davon ausgegangen, dass die genannten Pattern bekannt sind). Da die Wahl eines dieser Pattern massgeblichen Einfluss auf die Gesamtarchitektur des Systems hat, haben wir dieser Entscheidung einen hohen Stellenwert beigemessen.

Fowler hat den Versuch unternommen, die Pattern als Funktion der Beziehung zwischen der Komplexität der Geschäftslogik und dem Aufwand, der geleistet werden muss, wenn sich die Geschäftslogik ändert oder erweitert, aufzuzeichnen. Die dahinterstehenden Überlegungen hatten einen grossen Einfluss auf unsere Entscheidung.

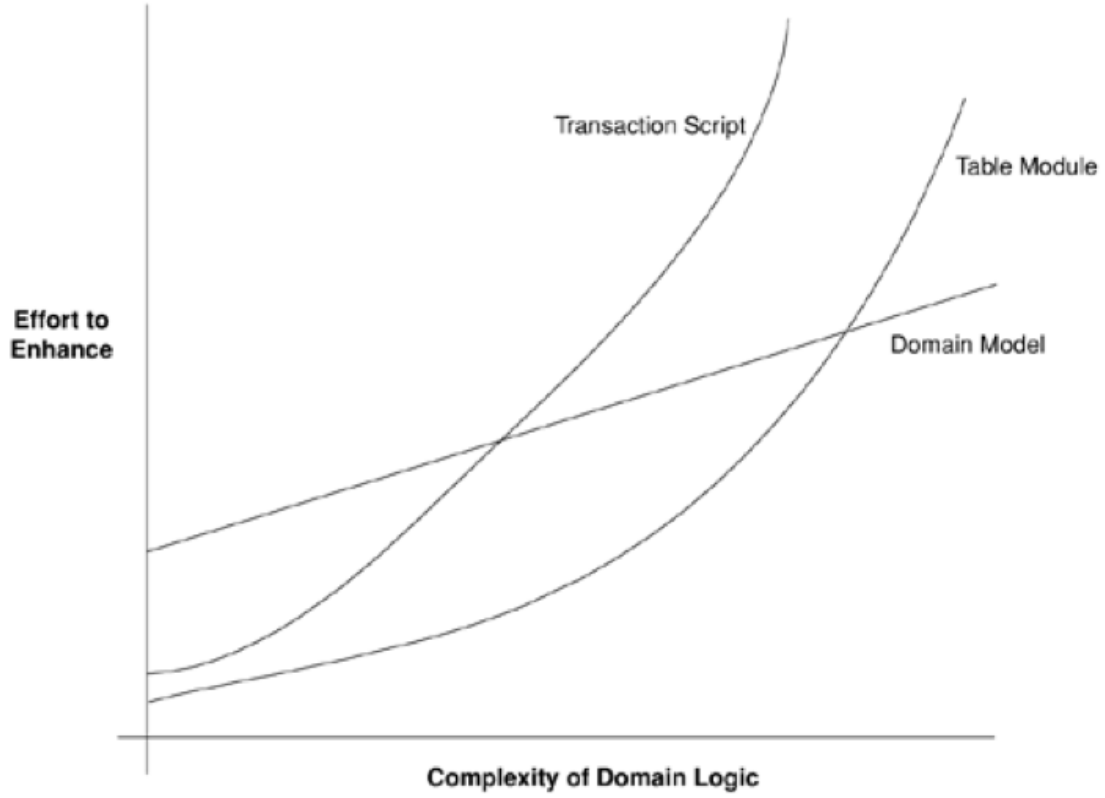


Abbildung 1: Beziehung Komplexität und Entwicklungsaufwand (Fowler [1], S.42)

Aufgrund der in Kapitel 6 beschriebenen funktionalen Anforderungen an das System, sind wir von einer relativ hohen Komplexität ausgegangen, die wir möglichst modular und gleichmässig an die Komponenten verteilen wollten. Zudem sollte das System die Möglichkeit bereithalten, zukünftige Schnittstellen zu ergänzenden

Komponenten mit möglichst geringem Entwicklungsaufwand bereitstellen zu können. Aus diesen Überlegungen und dem Umstand, dass wir als Projektteam sehr stark mit den objektorientierten Ansätzen verwurzelt sind, haben wir uns entscheiden, die Geschäftslogik nach einem Domain Model und die Applikation nach dem Konzept des Domain-Driven-Design's zu organisieren. Nachfolgend soll der Entscheid als Y-Statement zusammengefasst werden:

> *Im Kontext der Architekturentscheidung zur Organisation der Geschäftslogik innerhalb der iTIXI-Gesamtapplikation, konfrontiert mit vielen Anforderungen aus unterschiedlichen Funktionsbereichen, der Forderung nach einem möglichst generischen Ansatz um zukünftigen Anpassung und Erweiterung begegnen zu können und unseren Vorkenntnissen in objektorientierten Programmierparadigmen, haben wir uns für das Domain Model Pattern und gegen das Transaction Script oder Table Module Pattern entschieden, um zu erreichen, dass die Applikation trotz ihrer hohen Komplexität wartbar, änderbar und erweiterbar ist; wir nehmen dafür in Kauf, dass die Entwicklung nicht nach der Rapid-Application-Development (RAD) Methode erfolgen kann und bedingt durch den hohen Abstraktionsgrad einen höheren Einarbeitungsaufwand für zukünftige Entwickler mit sich bringt. Auch nehmen wir in Kauf, dass dadurch zu Beginn einen erheblichen Mehraufwand an Entwicklungsarbeit geleistet werden muss, da bereits bestehende Teile der Applikation neu programmiert werden müssen und dadurch die ursprüngliche Aufgabenstellung des Projektes angepasst werden muss.*

Y-Statement Architekturentscheid Domain Model

2.1.2 Entscheid OR-Mapper

Eine Grundproblematik bleibt die Abbildung des Domain Models auf die Persistenzschicht in einem relationalen Umfeld. Fowler [1] nennt dabei zwei grundlegende Strategien: Active Record und Data Mapper. Da wir es mit einem reichen Domain Model zu tun haben und gängige Techniken wie Vererbung etc. einsetzen, kam für uns der selbstverwaltende Ansatz des Active Record Pattern nicht in Frage und wir entscheiden uns, das Data Mapper Pattern einzusetzen. Da die Implementation dieses Patterns in Eigenregie aufgrund der fehlenden Zeit nicht möglich war und innerhalb der gewählten Technologie und des Frameworks eine gute Lösung existierte, entschieden wir uns, Doctrine als Data Mapper einzusetzen.

> *Im Kontext des Strategientscheids, wie das Domain Model auf die Datenbankschicht in einem relationalen Umfeld abgebildet werden soll, konfrontiert mit einem reichen und komplexen Domain Model, haben wir uns für das Data Mapper Pattern und im speziellen für Doctrine als Implementationslösung eines Drittanbieters und gegen das Active Record Pattern entschieden, um als machbare und performante Lösung das Domain Model Pattern innerhalb eines objektorientierten Umfeldes umsetzen zu können; wir nehmen dafür in Kauf, dass Doctrine eine Abhängigkeit gegenüber eines Drittanbieters darstellt.*

Y-Statement Entscheid OR-Mapper

2.1.3 Domain Model

Nachfolgend lässt sich das durch iterative Prozesse fortan weiterentwickelte Domain Model in einer geeigneten Notation abbilden. Durch die Wahl eines OR-Mappers beruht die Notation auf einer Mischform von UML 2.0 und Elementen aus der Datenbanksicht. Es werden Teilansichten aus dem ganzen Model entnommen, die Gesamtsicht ist für die Dokumentation zu gross, ein hochauflösendes Dokument ist der CD beigelegt (02-Abbildungen/Domain Model.pdf) und taucht als A4 Ausdruck im Appendix **Fehler! Verweisquelle konnte nicht gefunden werden.** auf.

Bei der Modellierung wurden aus Platzgründen nicht alle Attribute und Funktionen aufgelistet, die wichtigsten Assoziationen und Attribute sind aufgeführt.

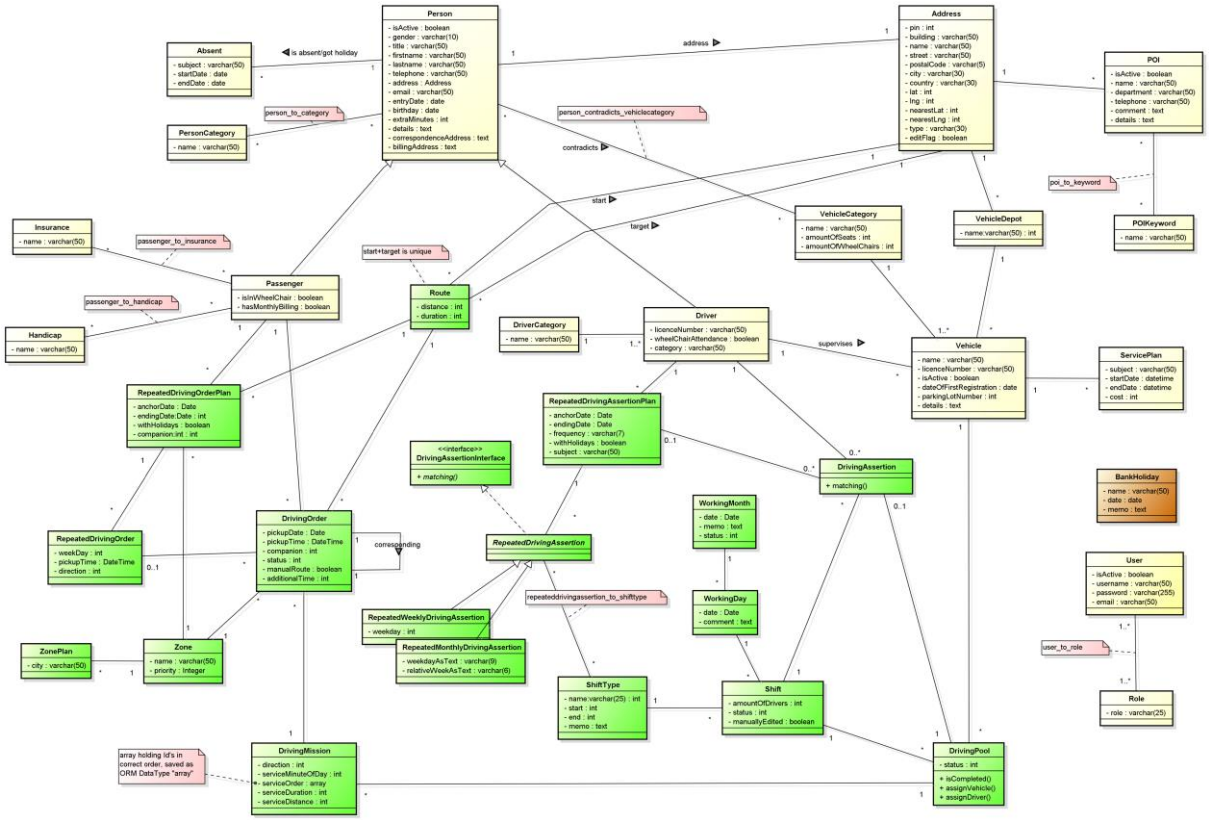


Abbildung 2: Domain Model Gesamtübersicht

Die wichtigsten Aspekte werden nachfolgend in den Teilansichten nur in einer hohen Abstraktion und nicht im Detail erläutert.

2.1.3.1 Elemente zur Stammdatenverwaltung

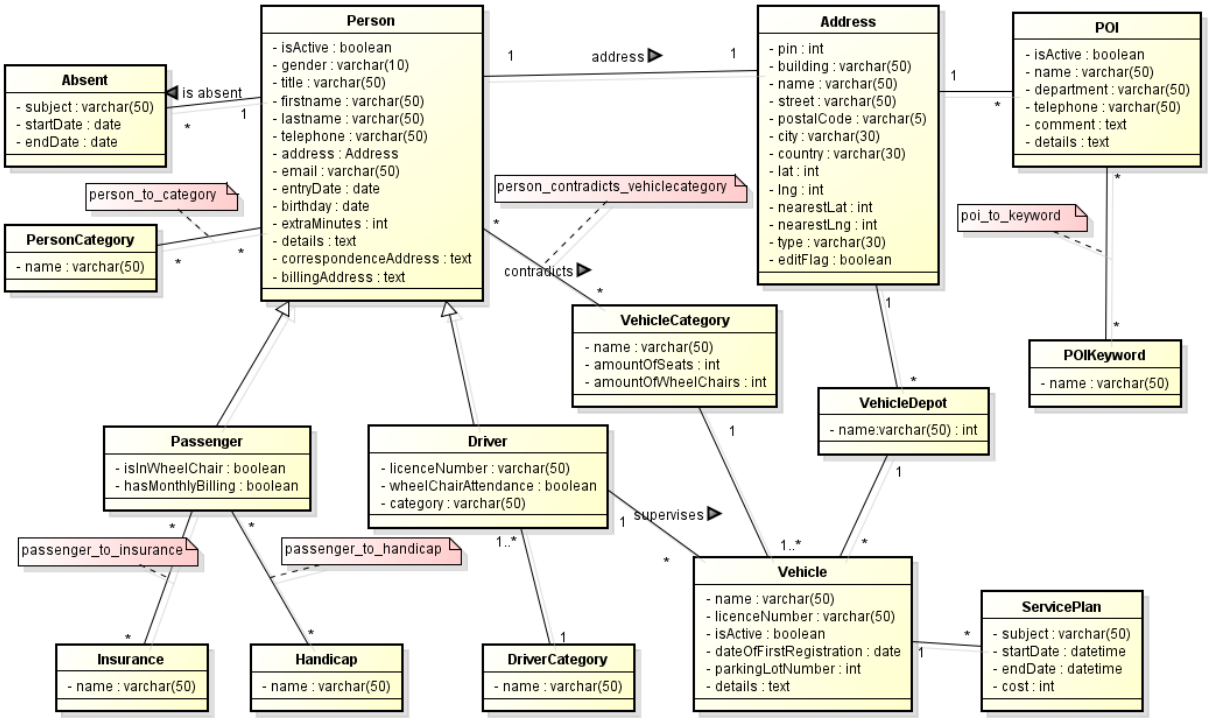


Abbildung 3: Domain Model Stammdatenverwaltung

Um Datenredundanz zu vermeiden, erben Driver und Passenger gemeinsame Attribute der Klasse Person. Drei verschiedene Strategien ermöglichen dabei das Abbilden einer Vererbung per OR-Mapper auf eine relationale Datenbank (Inheritance Mapping):

- Mapped Superclass
- Single Table Inheritance[1]
- Class Table Inheritance[2]

Für die Abbildung von Passenger und Driver wurde ein Single Table Inheritance Mapping gewählt, dies ermöglicht performante Abfragen durch eine Minimierung von JOIN‘s, da alle Daten in einer Tabelle gehalten werden und über ein Discriminator Attribut unterschieden werden.

Auch bei der Address werden alle Attribute aus Performance-Gründen in derselben Klasse gehalten, da wiederum auf Datenbank-Ebene schnelle Abfragen möglich sind und zudem die Volltextsuche einen Index über die Kolumnen der Tabelle erstellen kann. Hier wird bewusst Redundanz eingegangen.

[1] Infos gemäss Fowler auf http://martinfowler.com/eaaCatalog/singleTableInheritance.html
[2] Infos gemäss Fowler auf http://martinfowler.com/eaaCatalog/classTableInheritance.html

2.1.3.2 Elemente zum Fahrauftrag

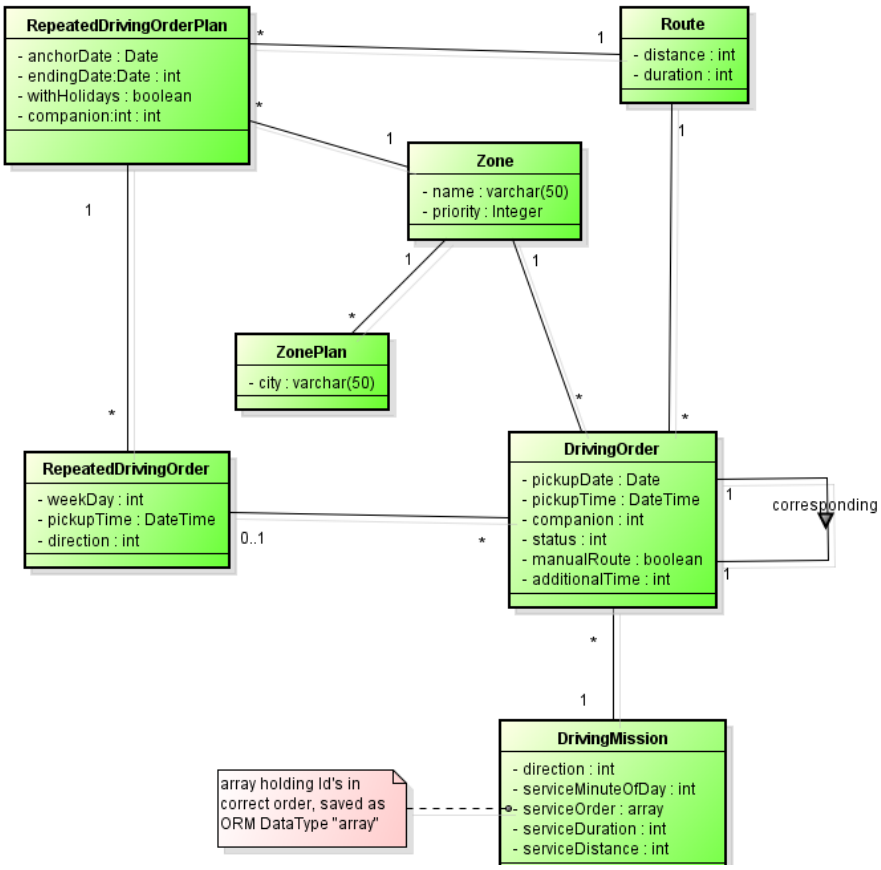


Abbildung 4:Domain Model Fahrauftrag

Die Fahrauftragserfassung beinhaltet Einzel- (DrivingOrder) und wiederkehrende Aufträge/Daueraufträge (RepeatedDrivingOrder). Dieser werden dabei zu einem Fahrgast/Passenger assoziiert (nicht ersichtlich in Abbildung 4). Beim Erstellungsprozess werden aus RepeatedDrivingOrders einzelne DrivingOrders erstellt, die Information aus welchem Dauerauftrag diese stammen wird durch die Assoziation erhalten. Damit eine gemeinsame Bearbeitung von Hin- und Rückfahrten für einen Auftrag möglich ist, wird eine selbstreferenzierte Assoziation auf DrivingOrder erstellt. Kommt ein Fahrauftrag zustande, oder werden mehrere Fahraufträge zu einem Sammelauftrag zugeordnet, erwächst daraus eine DrivingMission.

2.1.3.3 Elemente der Ressourcenplanung

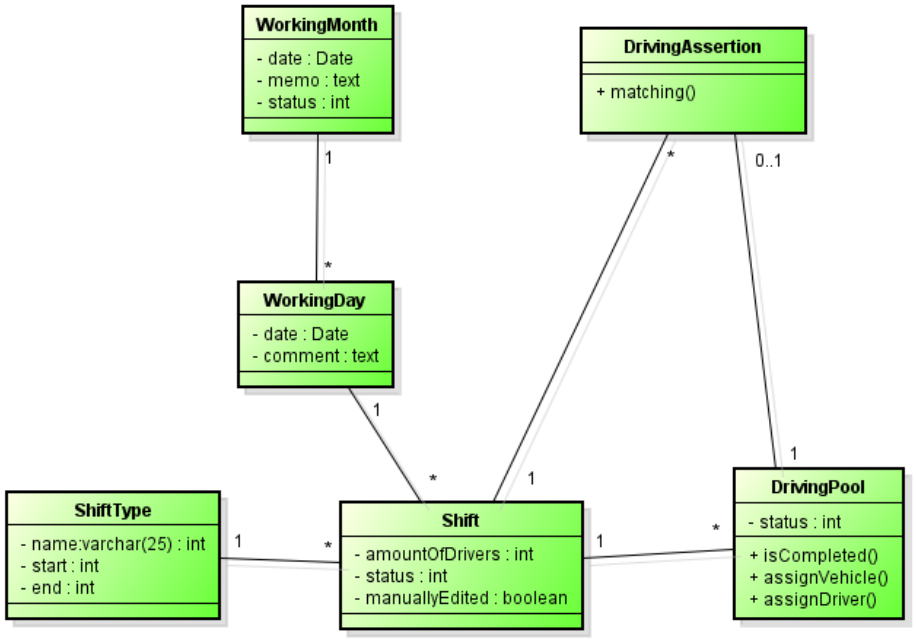


Abbildung 5: Domain Model Ressourcenplanung

Die Abbildung zu einem Produktions- und Monatsplan ist über WorkingMonth, WorkingDay und einzelnen Shifts gegeben, die über einen ShiftType korrespondierende Start- und End-Zeiten zuordnet. Aus benötigten Anzahl Fahrer pro Schicht/Tag/Monat erwächst ein DrivingPool, der für die Zuordnung von bereitgehaltenen Fahrern und benötigten Fahrern dient und beim Zustandekommen einer Fahrt die Assoziation zu einer DrivingMission herstellt.

2.1.3.4 Elemente der Fahrerzuteilung

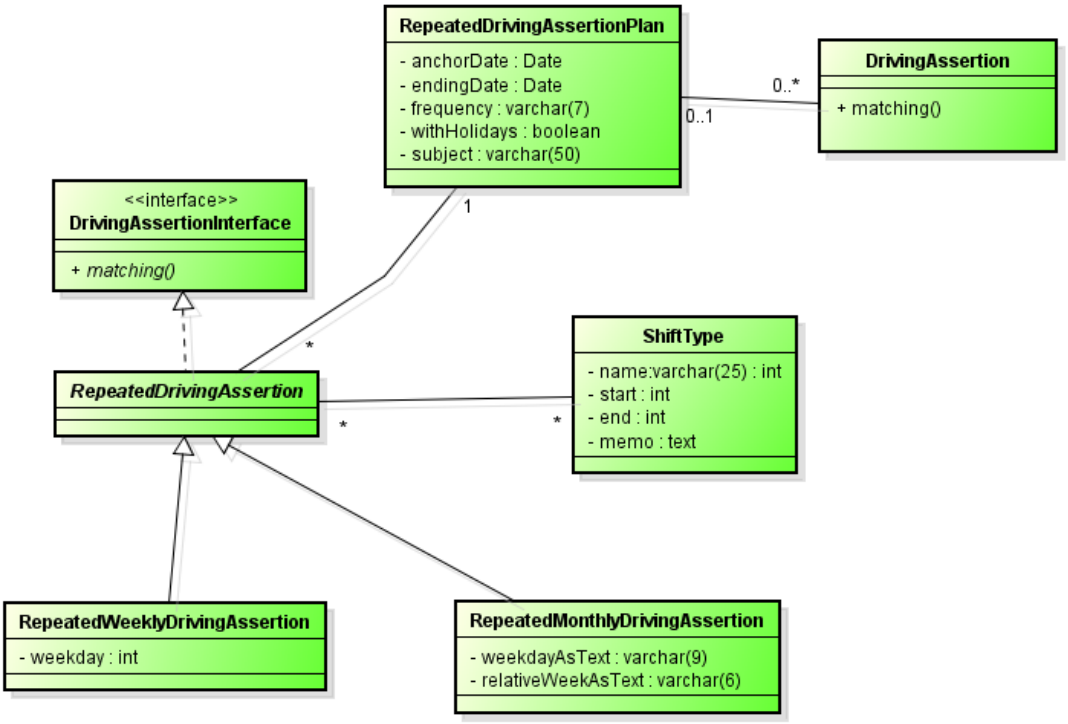


Abbildung 6: Domain Model Fahrerzuteilung

Für die Bereitstellung eines Fahrers wird ein Dauereinsatz (RepeatedDrivingAssertion) erstellt um die möglichen Schichten (ShiftType), Tage (weekday), Wochen (RepeatedWeeklyDrivingAssertion) und Monate (RepeatedMonthrlyDrivingAssertion) abzubilden an denen er Einsätze leisten kann.
Beim Erstellungsprozess werden pro Tag und Schicht DrivingAssertion Objekte erstellt, die für die Zuweisung von Einsätzen im Einsatzplan dient.

3 Implementierung

3.1 Systemübersicht

Die iTIXI-Applikation wurde auf Basis des PHP Frameworks Symfony 2.3 erstellt und nach den Prinzipien des Domain-Driven-Design organisiert. Ziel dieses Abschnittes ist es, einen Überblick über die Komponenten, deren Funktion und Zusammenspiel zu geben und ihre Rolle innerhalb der Domain Driven Design-Konzeptes zu klären. Es wird davon ausgegangen, dass der Leser mit den Symfony-Grundkonzepten vertraut ist.

3.1.1 Gesamtsicht

Um möglichst die Symfony-Standardkonzepte verwenden zu können, haben wir uns entschieden, mit einem sehr dünnen Client zu arbeiten, der vorwiegend Information anzeigt und nur sehr wenig Applikationslogik hält. Die Logik befindet sich demnach überwiegend auf der Serverseite. Das nachfolgende Komponentendiagramm soll einen Überblick über die Struktur der Applikation geben, wobei sich die Bezeichnung der Komponenten (in der Grafik gelb dargestellt) mit der effektiven Dateistruktur des Symfony Source-Ordners deckt. Nicht alle Komponenten sind auch Symfony-Komponenten. Symfony-Komponenten sind daran zu erkennen, dass sie den Bezeichner *Bundle* im Namen tragen. Die blauen Bereiche stellen die logischen Schichten gemäss Domain Driven Design-Terminologie dar. Alles, was nicht als Client bezeichnet ist, ist der Serverseite zuzuordnen.

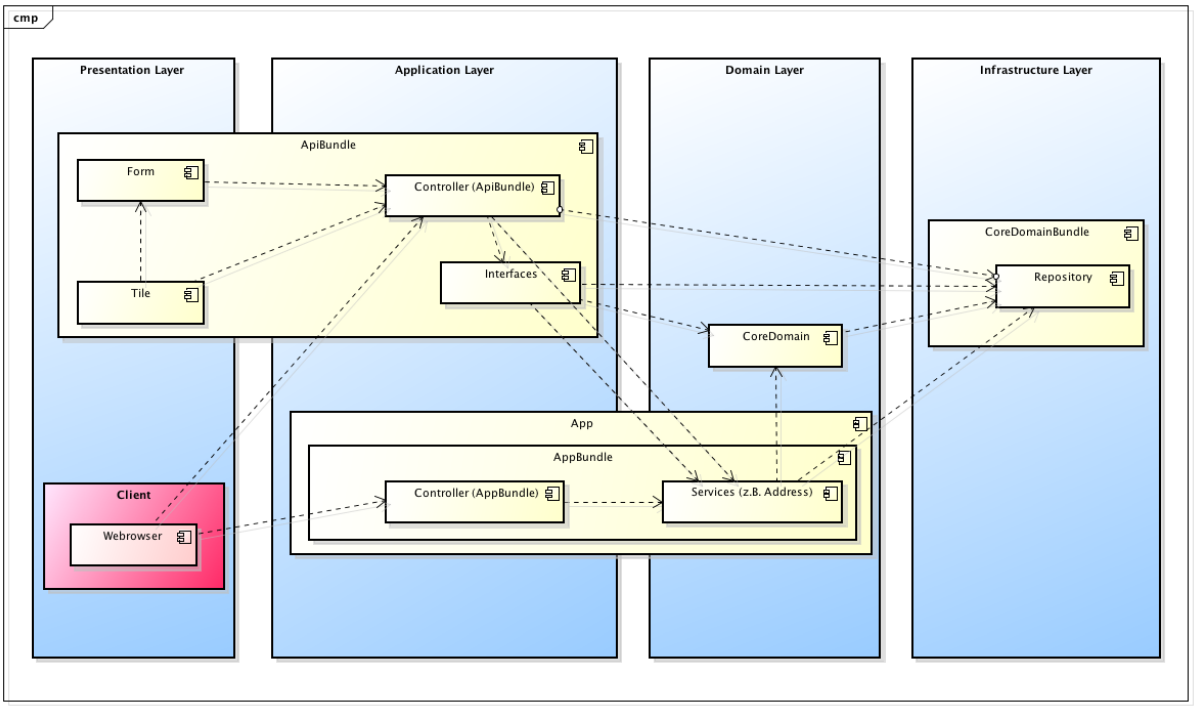


Abbildung 7: Systemgesamtsicht

3.1.2 ApiBundle

Das ApiBundle stellt den Haupteintrittspunkt des Clients in die Applikation dar. Es übernimmt einerseits klassische Frontend-Aufgaben, indem es über die Tile- und Formkomponenten Ein- und Ausgabemasken zur Verfügung stellt und andererseits über die Funktionen der Controller-Klassen, die über HTTP erreichbar sind, das API gegenüber dem Backend bzw. der Domain-Schicht definiert.

3.1.2.1 Controller (ApiBundle)

Paketstruktur:

Controller-Klassen:

`src/Tixi/ApiBundle/Controller`

App-Globale Routing-Konfiguration:

`app/config/routing.yml`

Funktions- und Implementationseckpunkte im Symfony-Konzept:

- Die Controller-Klassen der Controller-Komponente sind klassische Symfony-Controller[3] und erben von `Symfony\Bundle\FrameworkBundle\Controller\Controller`
- Das URL-Routing wird über @Route-Annotations innerhalb der Controller-Klassen gesteuert, was dem System über den folgenden Eintrag in der globalen Routing-Konfiguration bekanntgegeben wird:

```
tixi_api:
      resource: "@TixiApiBundle/Controller/"
      type: annotation
      prefix:   /
```

Beziehung zu anderen Komponenten:

- Client: Die Funktionen der Controller-Klassen sind über HTTP aufrufbar und stellen damit den wichtigsten Eintrittspunkt des Clients (Browser) in die Applikation dar
- ApiBundle\Interfaces: Benutzt die Assembler-Klassen zur Serialisierung und Deserialisierung von Entitäten in Data Transfer Objects und umgekehrt
- ApiBundle\Form: Steuert den Eingabeprozesse über die Form-Komponente
- ApiBundle\Tile: Steuert den Ausgabeprozess über die Tile-Komponente
- CoreDomainBundle\Repository: Lädt und speichert Entitäten aus der Persistenzschicht über die entsprechenden Funktionen der Repository-Klassen

Aufgabe im System:

Die Controller-Komponente ist konzeptionell im Application Layer angesiedelt, der als Mediator zwischen dem Presentation Layer, also der User-Eingabeseite und dem Domain Layer fungiert.

Der Controller hat im Wesentlichen die Aufgabe, entweder Objekte aus dem Domain Layer zu serialisieren und zur Anzeige an eine View, in unserem Konzept einem Tile, zu delegieren, oder Usereingaben, die typischerweise über ein Form übermittelt werden, zu deserialisieren und korrespondierende Entitäten aus dem Domain Layer zu erzeugen oder zu modifizieren.

Das nachfolgende Sequenzdiagramm (Abbildung 8) zeigt das vereinfachte Zusammenspiel der Komponenten, wie sie den für einen Controller typischen Prozess des Anzeigens eines Formulars bedienen:

[3] Symfony Controller: http://symfony.com/doc/current/book/controller.html

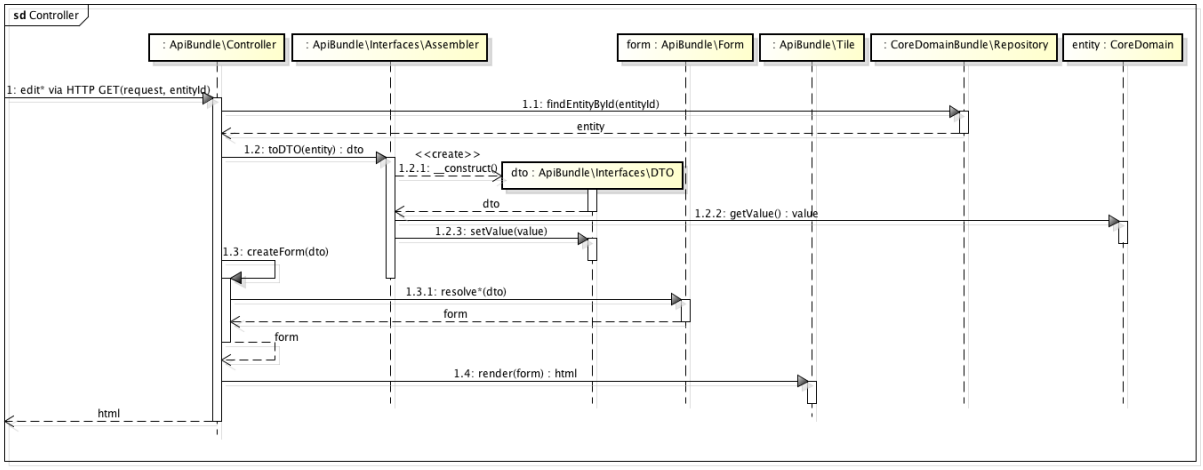


Abbildung 8: Sequenzdiagramm Controllerprozess

Eigenheiten der Implementation:

Der Response, der an den Client übermittelt wird, ist durch die Tile-Komponente generiertes HTML.

3.1.2.2 Interfaces (Assembler und DTO's)

Paketstruktur:

Assembler- und DTO-Klassen:

`src/Tixi/ApiBundle/Interfaces`

Assembler Service-Definitionen:

`src/Tixi/ApiBundle/Resources/config/services.xml`

Funktions- und Implementationseckpunkte im Symfony-Konzept:

- Keine direkte Beziehung zu Elementen im Symfony-Konzept
- Die Assembler-Instanzen werden als Services über den Service-Container[4] zugänglich gemacht

Beziehung zu anderen Komponenten:

- App\AppBundle\div. Services: Assemblers nutzen teilweise Services aus der App-Komponente zur Diensterbringung.
- CoreDomain: Erzeugt oder ändert Entitäten des CoreDomains über entsprechende Factory-Methoden oder Update-Methoden.
- CoreDomainBundle\Repository: Liest oder schreibt Entitäten aus der Persistenzschicht über entsprechende Repository-Funktionen.

Aufgaben im System:

Die serverseitige Kommunikation zwischen dem Presentation Layer und dem Application Layer erfolgt konsequent über sogenannte Data Transfer Objects (DTO's). Die Idee dahinter ist, dass keine direkte Beziehung zwischen den Feldern von Eingabe- bzw. Ausgabemasken der Views und den Eigenschaften der Entitäten des Domain Layers bestehen muss. Vielmehr können z.B. über eine Eingabemaske und die damit verbundene Eingabetransaktion Eigenschaften von verschiedenen Entitäten geändert werden. Jeder Eingabe- und jeder Ausgabetransaktion ist demnach ein entsprechendes DTO zugeordnet.

[4] Symfony Service Container: http://symfony.com/doc/current/book/service_container.html

Das Überführen von Entitäten in DTO's und umgekehrt wird durch sogenannte Assembler übernommen, die als Services über Symfony's Service-Container bereitgestellt werden.
Die Assembler-Klassen und DTO-(pseudo)Klassen werden von der Interface-Komponente bereitgestellt.

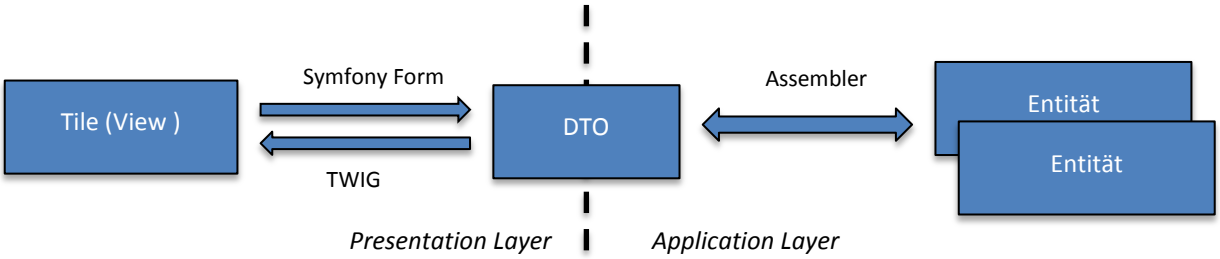


Abbildung 9 Illustration des Data Transfer Object (DTO)-Konzepts

3.1.2.3 Form und Validation

Paketstruktur:

Form-Type-Klassen:

`src/Tixi/ApiBundle/Form/*`

Twitter-Bootstrap Form Template:

`src/Tixi/ApiBundle/Resources/views/Form/fieldsBootstrap.html.twig`

Custom Form Template Config:

`app/config/config.yml`

Funktions- und Implementationseckpunkte im Symfony-Konzept:

- Die FormType-Klassen folgen dem Symfony-Form-Konzept[5] und erben, teilweise indirekt über die CommonAbstractType-Klasse, von `Symfony\Component\Form\AbstractType`
- Die Form-Validators werden in den entsprechenden FormTypes über das constraints-Attribut bekanntgegeben. Zusätzlich wird auch clientseitig validiert (siehe Eigenheiten der Implementation)
- Als Data-Klassen werden DTO's und nicht direkt Entitäten eingesetzt (siehe Eigenheiten der Implementation)
- Das Twitter-Bootstrap Template wird als Default-Template im App-Config eingebunden

Beziehung zu anderen Komponenten:

- ApiBundle\Controller: Die Controller-Komponente steuert den Form-Prozess anhand der Form-Status
- ApiBundle\Tile: Die Form-Komponente wird von der Tile-Komponente zur Darstellung eines Forms innerhalb einer View benutzt.
- ApiBundle\Interfaces: Die Datenklassen der Forms sind DTO's.

Aufgabe im System:

Die Form-Komponente kapselt den Prozess, wie Benutzereingaben in das System gelangen. Diese erfolgen typischerweise über ein HTML-Webformular.

Dazu gehört das Rendern der Formularelemente auf der Clientseiten, das serialisierte Übertragen der Formularwerte zwischen dem Server (Symfony/PHP) und dem Client (Browser) und die Validierung der clientseitigen Eingaben.

[5] Symfony Forms: http://symfony.com/doc/current/book/forms.html

Eigenheiten der Implementation:

DTO's als Data-Klassen:

Um die in den Aufgaben der Interface-Komponente beschriebene Entkoppelung zwischen Entitäten und Eingabemasken umsetzen zu können, werden DTO's als Form-Data-Klassen eingesetzt. Diese werden in den entsprechenden Types durch das Überschreiben der setDefaultOptions()-Funktion bekanntgemacht

(Codebeispiel aus dem DriverType):

```
public function setDefaultOptions(OptionsResolverInterface $resolver) {
      $resolver->setDefaults(array(
          'data_class' => 'Tixi\ApiBundle\Interfaces\DriverRegisterDTO'
      ));
  }
```

Server- und Clientseitige Validierung:

Für die serverseitige Validierung setzen wir die von Symfony angebotenen Form-Validator ein, die wir über das constraints-Attribut direkt über das FormBuilderInterface zum entsprechenden FormType hinzufügen.

Um ein besseres Usability-Verhalten zu erreichen, haben wir uns entschieden, neben der serverseitigen Validierung zusätzlich auch clientseitig zu validieren. Dies erreichen wir, indem wir das HTML5 Validation-Konzept einsetzen.

Symfony setzt bei FormTypes, die als required definiert sind, automatisch das required-Attribut auf dem korrespondierenden HTML-Input-Type auf den Wert "required" (required="required"). Über die HTML5-JavascriptAPI kann dann das Formular über die Methode checkValidity() auf den Status der als required gesetzten Felder abgefragt werden. Standardmässig wird dabei auf "NotBlank" geprüft, also auf nicht leer. Zusätzlich zum Required-Attribut kann noch ein Pattern-Attribut gesetzt werden, das den geforderten Input als RegEx-Pattern definiert.

Die Symfony-Validators setzen meistens auch gleich das Pattern-Attribut auf dem Input Element korrekt. Einzig beim RegEx-Validator musste teilweise das HTML-Pattern manuell überschrieben werden, da sich die RegEx-Formate zwischen den Systemen teilweise unterscheiden.

Die clientseitige Validierung wird per Javascript über das eigens dafür geschriebene FormValidationController-Objekt gemacht. Dieses ist im File *web/bundles/tixiapi/js/FormValidationController.js* spezifiziert und wird standardmässig über das Form-Tile eingebunden.

3.1.2.4 Tile-Konzept

Paketstruktur:

Tile-Klassen:

`src/Tixi/ApiBundle/Tile`

Tile View-Ressourcen:

`src/Tixi/ApiBundle/Resources/views/Tile`

Funktions- und Implementationseckpunkte im Symfony-Konzept:

- Das Tile-Konzept entspricht konzeptionell dem Symfony-Konzept der View
- Der TileRenderer, der für das Rendern der Tile-Instanzen in HTML zuständig ist, ist über den Service *tixi_api.tilerenderer* eingebunden.

Beziehung zu anderen Komponenten:

ApiBundle\Controller: Die Tile-Klassen werden typischerweise in den jeweiligen Controllern instanziiert, verschachtelt und gerendert.

ApiBundle\Form: Die clientseitige HTML-Repräsentation einer Form-Instanz wird über eine entsprechende Tile-Klasse gesteuert.

Aufgabe im System:

Das Tile-System ist konzeptionell dem Presentation Layer zugeordnet und definiert, wie die System-Elemente - meistens Forms - auf der Client-Seite gerendert werden. Die Kernelemente des Tile-Systems sind die Tile-Klassen, die, analog zum Symfony-Form-System, wiederverwendbare und, durch das Prinzip der Selbstreferentialität, verschachtelbare View-Elemente beschreiben.

Eigenheiten der Implementation:

Das Tile-System besteht aus den Tile-Klassen, die alle die abstrakte Klasse AbstractTile erweitern. Zu jeder Instanz einer Tile-Klasse, einem Tile, können über deren add-Methode weitere Tiles hinzugefügt werden. Die zu einem Tile hinzugefügten Tiles werden als Kinder (children) bezeichnet. Aus der Sicht der Kinder wird das Tile, zu dem sie hinzugefügt worden sind, als Eltern-Tile (parent) bezeichnet. Dadurch entsteht eine Baumstruktur von Tiles, wobei ein Tile die Wurzel des Baumes bildet und als root-Element bezeichnet wird.

Jeder Tile-Klasse ist ein Template zugeordnet, dessen Pfad durch das Überschreiben der abstrakten getTemplateName()-Methode des `AbstractTile` bekanntgegeben wird. Das Template kann in einer der von Symfony unterstützen Template-Sprachen verfasst werden. In unserem Fall ist dies TWIG.

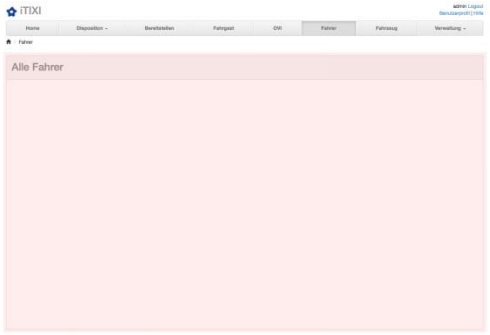
Das root-Element kann dem TileRenderer zum rendern übergeben werden, der den Tile-Baum rekursiv durchläuft und dabei, angefangen mit dem tiefsten, die Elemente gemäss ihrem definierten Template rendert. Es wird mit der Symfony-Template-Engine gerendert, die in der Systemkonfiguration definiert ist (bei iTIXI ist dies wiederum TWIG).

Die View-Parameter, die im jeweiligen Template zur Verfügung stehen sollen, werden durch Überschreiben der getViewParameters()-Methode des `AbstractTile` bekanntgegeben. Die Parameter des Eltern-Tile werden an die Kind-Tiles weitergereicht und stehen dort ebenfalls zur Verfügung. Standardmässig stehen die gerenderten Kind-Elemente dem Eltern-Tile zur Verfügung und können dort entweder über den Tile-Namen, der durch überschrieben der getName-Methode des AbstractTile definiert wird, oder über weitere optionale View-Identifiers, die durch überschreiben der getViewIdentifiers-Methode des AsbtractTile angegeben werden, angesprochen werden.

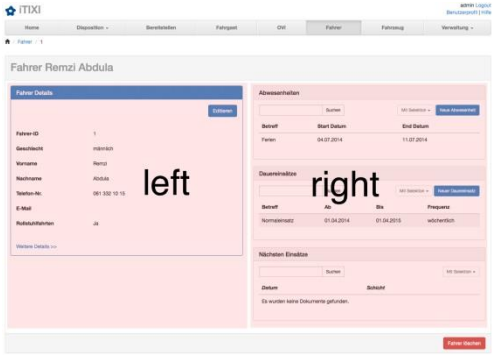
Die Tile-Klassen werden typischerweise in einem Controllern instanziert, verschachtelt und gerendert. Tileweise werden fertig konfigurierte Tiles auch von Services geliefert.

Die wichtigsten Tile's, die im Rahmen dieser Arbeit erstellt wurden sind die folgenden:

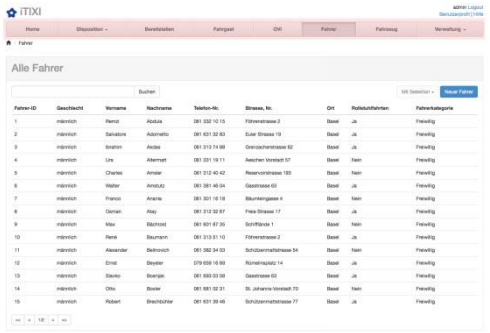
RootPanel



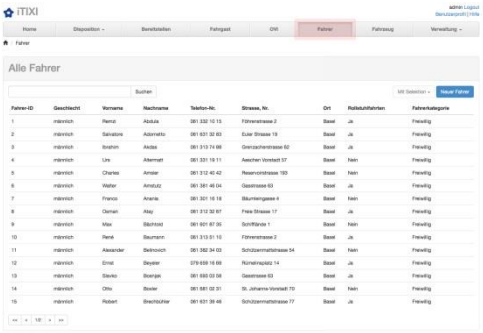
PanelSplitterTile



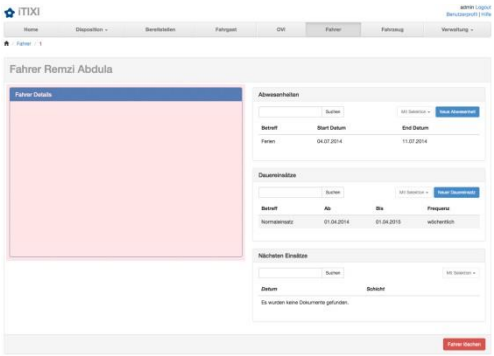
MenuTile



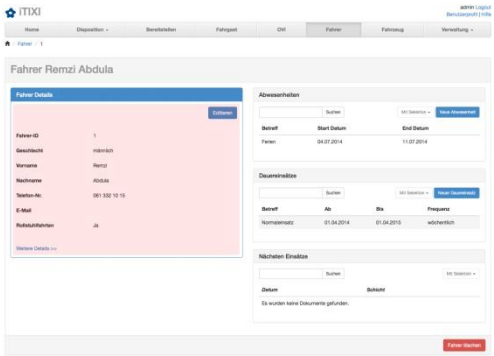
MenuItemTile



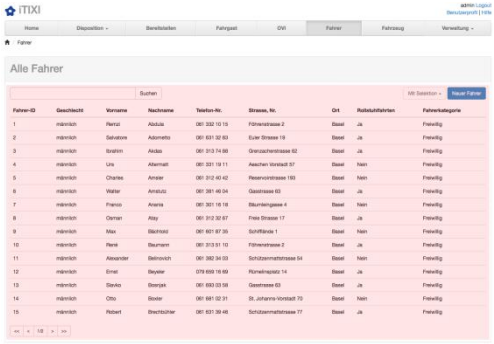
PanelTile



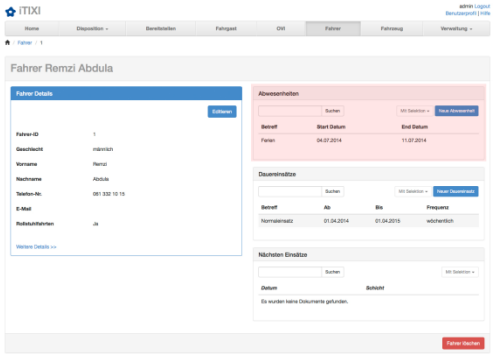
AbstractFormViewTile



DataGridTile



DataGridEmbeddedTile



Für die weiteren Tile-Klassen und die genauen Parametern, die für die Erzeugung der Tile-Instanzen erforderlich sind, sei an dieser Stelle auf den Code bzw. auf die Code-Dokumentation verwiesen.

Das nachfolgende, vereinfachte Code-Beispiel, soll einen Eindruck vermitteln, wie ein typsicherer Tile-Baum innerhalb des Controllers aufgebaut wird:

```
$rootPanel = new RootPanel(...);
$panelSplitter = $rootPanel->add(new PanelSplitterTile(...));
$formPanel = $panelSplitter->addLeft(new PanelTile(...));
$formPanel->add(new DriverRegisterFormViewTile(...));
$absentGridPanel = $panelSplitter->addRight(new PanelTile(...));
$absentGridTile = $dataGridHandler->createEmbeddedDataGridTile(...);
$absentGridPanel->add($absentGridTile);
$rootPanel->add(new PanelDeleteFooterTile(...));
return new Response($tileRenderer->render($rootPanel));
```

3.1.3 App und AppBundle

Grundsätzlich wurde bei der Implementierung darauf geachtet, Businessfunktionalität wo immer möglich direkt auf den zugehörigen Entitäten zu behandeln. Es gibt jedoch auch Operationen, die konzeptionell nicht direkt einer Instanz eines Domain Object's zugeordnet werden können oder für die Diensterbringung noch weitere Entitäten oder Services involviert sind. Diese Operationen wurden zu Services zusammengefasst und stehen über die App-Komponente zur Verfügung. Grundsätzlich werden alle Services durch Interfaces beschrieben, die direkt im Paket *src/Tixi/App* abgelegt sind. Die Implementation der Services unter Symfony findet sich in der AppBundle-Komponente und werden über den Symfony Service-Container zur Verfügung gestellt.

Im Regelfall werden die Services von der ApiBundle-Komponente genutzt, wo sie direkt aus den Controllern oder indirekt über die Assembler-Klassen der Interface-Komponente aufgerufen werden.

Einzelne Service-Funktionen werden jedoch auch direkt vom Client über XMLHttpRequests (AJAX) angesprochen. Die Kommunikation erfolgt dabei über die Controller-Klassen der AppBundle-Komponente.

In diesem Abschnitt sollen die Controller-Komponenten beschrieben werden, sowie die Services aufgelistet werden. Die wichtigsten Services werden in späteren Abschnitten beschrieben.

3.1.3.1 Controller (AppBundle)

Paketstruktur:

Controller-Klassen:

`src/Tixi/App/AppBundle/Controller`

App-Globale Routing-Konfiguration:

`app/config/routing.yml`

Funktions- und Implementationseckpunkte im Symfony-Konzept:

- Die Controller-Klassen sind klassische Symfony-Controller und erben von `Symfony\Bundle\FrameworkBundle\Controller\Controller`
- Das URL-Routing wird über @Route-Annotations innerhalb der Controller-Klassen gesteuert, was dem System über den folgenden Eintrag in der globalen Routing-Konfiguration bekanntgegeben wird:

```
tixi_app:
      resource: "@TixiAppBundle/Controller/"
      type: annotation
      prefix:   /
```

Beziehung zu anderen Komponenten:

- Client: Die Funktionen der Controller-Klassen sind über XMLHttpRequets erreichbar und werden von verschiedenen Javascript-Client-Libraries aufgerufen
- App\AppBundle*diverse Services*: Die Controller-Komponente nutzt zur Verarbeitung der Client-Requests das Funktionsangebot verschiedener Services

Aufgabe im System:

Wie auch die Controller-Komponente des ApiBundles bieten die Controller-Klassen des AppBundles über HTTP ansprechbare Eintrittspunkte in das System an. Sie haben im Wesentlichen die Funktion, Aufrufe an die entsprechenden Services zu dispatchen, das Resultat zu serialisieren und als JSON auszuliefern.

Eigenheiten der Implementation:

Anders als im ApiBundle wird jedoch nicht HTML an den Client zurückgegeben sondern JSON.

3.1.3.2 Übersicht Services

Name	Service-Name	Klasse unter `Tixi\App\AppBundle\*`
Address-Service	tixi_app.addressmanagement	Address\AddressManagementImpl
Dispositions-Service	tixi_app.dispomanagement	Disposition\DispositionManagementImpl
Dokument-Service	tixi_app.documentmanagement	Document\DocumentManagementKnp
Fahrereinsatz-Service	tixi_app.drivingassertionmanagement	Driving\DrivingAssertionManagementImpl
Fahrauftrags-Service	tixi_app.drivingordermanagement	Driving\DrivingOrderManagementImpl
E-Mail-Service	tixi_app.mailservice	Mail\MailServiceSwiftMailer
Fahrtoptimierungs-Service	tixi_app.ridemanagement	Ride\RideManagementImpl
Routenverwaltung-Service	tixi_app.routemanagement	Routing\RouteManagementImpl
Routenabfrage-Service	tixi_app.routingmachine	Routing\RoutingMachineOSRM
Zoneplan-Service	tixi_app.zoneplanmanagement	ZonePlan\ZonePlanManagementImpl

3.1.4 CoreDomain

Die CoreDomain-Komponente stellt das zentrale Element des Domain Layers dar. Darin enthalten sind die Objekte, die im Konzept eines Rich Domain Model die Daten für den Betrieb sowie die eigentliche Businessfunktionalität halten. Die meisten Objekte sind klassische Entities, die neben den Daten auch möglichst viel Logik bereitstellen. Ebenfalls im CoreDomain enthalten sind die als Interface definierten Repositories, die als Implementation des Repository Patterns den Zugriff auf den Entity-Pool kapseln.

Paketstruktur:

Entities und zugehörige Repositoriy-Interfaces:

`src/Tixi/CoreDomain`

Funktions- und Implementationseckpunkte im Symfony-Konzept:

- Die CoreDomain-Komponente hat keine direkte Entsprechung im Symfony-Konzept
- Die Entities sind als Doctrine-Entities konzipiert

Beziehung zu anderen Komponenten:

ApiBundle\Interfaces: Entities werden grundsätzlich im Rahmen von Benutzer-Eingabeprozessen von den Assembler-Klassen der Interface-Komponente anhand der Daten aus den entsprechenden DTO's erzeugt oder geändert oder, ebenfalls von den Assembler Klassen, für Ausgabeprozesse serialisiert.

AppBunle\div. Services: Diverse Services nutzen für die Erbringung ihrer Leistung die Daten und Funktionen der Entities oder Erzeugen oder Ändern Entities anhand von definierten Geschäftsprozessen.

CoreDomainBundle\Repository: Die Repository-Klassen implementieren die im CoreDomain spezifizierten Interfaces in der gewählten Persistenztechnologie, im Rahmen dieser Implementation in Doctrine.

Aufgabe im System:

Wie bereits im Eingangstext besprochen, halten die Objekte der CoreDomain-Komponente alle für den Betrieb relevanten Daten und implementieren soviel der Geschäftslogik, wie möglich. Sie repräsentieren in ihrer Gesamtheit das, was als Rich Domain Model bezeichnet wird.

Die Repositories stellen die nötigen Funktionen zur Verfügung, um Entitäten anhand von bestimmten Eigenschaften aus dem Pool aller Entitäten zu laden und in diesem abzulegen. Dieses Zugriffspattern wird als Repository Pattern bezeichnet und stellt ein sogenanntes "collection-like-interface" für die Entitäten bereit.

Eigenheiten der Implementation:

Es wurde darauf geachtet, die Komponente des CoreDomains soweit als möglich und sinnvoll von Symfony zu entkoppeln und möglichst als Plain Old PHP Objects zu konzipieren, um grundsätzlich das Prinzip der technologieunabhängigen Wiederverwendbarkeit zu gewährleisten. Um das ohnehin schon komplexe Organisationskonzept der Applikation nicht noch weiter zu verkomplizieren, haben wir uns als Kompromiss zwischen formaler DDD-Architektur und Praktikabilität der Lösung darauf geeinigt, die Entitäten über entsprechende Annotations als Doctrine-Entitäten zu konzipieren. Dabei verstehen wir die Information über Annotations als lose Koppelung zu der gewählten Doctrine-Technologie, die im Falle eines Technologiewechsels schnell geändert werden kann. Als einziger Konzeptbruch kann der Einsatz der für Doctrine nötigen Definition von Arrays als Doctrine\Common\Collections\ArrayCollection-Klassen gesehen werden. Wir sind jedoch der Ansicht, dass auch dieser Umstand im Falle eines Technologiewechsels im Persistenz Layer ohne grosse Komplikationen gelöst werden kann, da sich die ArrayCollections genau gleich wie reguläre PHP-Arrays verhalten.

3.1.5 CoreDomainBundle

Das CoreDomainBundle hat einzig die Funktion, die im CoreDomain beschriebenen Repositories in der gewählten Persistenz-Technologie zu implementieren.

Paketstruktur:

Repository-Implementationen:

`src/Tixi/CoreDomainBundle/Repository`

Service-Konfiguration:

`src/Tixi/CoreDomainBundle/Resources/config/services.xml`

Funktions- und Implementationseckpunkte im Symfony-Konzept:

- Die Repositories stehen als Symfony-Services über den Service-Container zur Verfügung

Beziehung zu anderen Komponenten:

CoreDomain: Die Repository-Klassen des CoreDomainBundle implementieren die im CoreDomain spezifizierten Repositories.

ApiBundle\Interfaces, ApiBundle\Controller, AppBundle\div. Services: Die Repositories werden dazu verwendet, um Entitäten anhand bestimmter Eigenschaften aus der Persistenz Schicht zu laden und zu speichern.

Aufgabe im System:

Die konkreten Implementationen der Repositories stellt als Infrastrukturkomponente Zugriff auf die Entitäten sicher, indem sie aus Sicht der Applikation transparent zwischen RAM und MySQL-Datenbank vermittelt.

Eigenheiten der Implementation:

Die konkreten Instanzen der Repository-Implementationen werden im Symfony-Container über die Abstraktion von Aliasen gebunden. Dies ermöglicht theoretisch den Austausch von Doctrine als OR-Mapper rein durch eine Änderung der Systemkonfiguration.

Beispiel dieses Implementationsmerkmal anhand des Vehicle-Repository's:

```
<services>
       <service id="vehicle_repository"
       alias="vehicle_repository.doctrine"></service>


      <service id="vehicle_repository.doctrine"
       class="%vehicle_repository.doctrine.class%" ...>
              ...
      </service>
</services>
```

3.2 iTIXI als Informationssystem

Ein wesentlicher Bestandteil von iTIXI, der im Rahmen dieser Arbeit entwickelt wurde, ist die Implementation der Prozesse und Elemente eines klassischen Informationssystems, die für das Erstellen, Anzeigen, Modifizieren und Löschen (Create, Read, Update, Delete - CRUD) von Daten notwendig sind. Dabei sind in unserem Implementationskonzept grundsätzlich die Komponenten in der Funktion beteiligt, wie sie in der Gesamtsicht beschrieben wurden. Ziel dieses Abschnittes ist es, Besonderheiten der Implementation unter dem Gesichtspunkt von iTIXI als Informationssystem darzulegen.

3.2.1 Menükonzept / Sitemap

Bis auf die Bereiche Disposition, Verwaltung und Bereitstellen, können alle Aktionen klar mit einer Entität verknüpft werden. Die Masterview der jeweiligen Entität bildet den Eintrittspunkt zu allen möglichen Folgeaktionen, die im Zusammenhang mit dieser Entität stehen.

Abbildung 10: Menükonzept

Primärer Funktionspunkt	Sekundärer Funktionspunkt	@Route
Home		/
Disposition		/disposition
	Produktionsplan	/disposition/productionplan
	Monatsplan	disposition/monthlyplan
Bereitstellen		(nicht Teil dieser Arbeit)
Fahrgast		/passengers
OVI		/pois
Fahrer		/drivers
Fahrzeug		/vehicles
Verwaltung		/management
	Benutzer	/management/users
	Zone	/management/zones
	Zonenplan	/management/zoneplans
	Fahrzeugkategorie	/management/vehiclecategories
	Fahrzeugdepot	/management/vehicledepots
	Fahrerkategorie	/management/drivercategories
	Personenkategorie	/management/personcategories
	OVI-Keyword	/management/poikeywords
	Behinderung	/management/handicaps
	Versicherung	/management/insurances
	Schichten	/management/shifttypes
	Feiertage	/management/bankholidays

3.2.1.1 Master-Detail-Konzept

Jeder im Menükonzept aufgeführte Funktionspunkt führt primär auf die Master-View dieser Funktionsgruppe, die bei der Stammdatenverwaltung der jeweiligen Entität entspricht. Ausgehend von dieser Master-View können dann Folgeaktionen auf der Detailstufe der jeweiligen Instanz ausgeführt werden. Dies entweder direkt über das Menü des Master-DataGrids (Abbildung 11) oder indirekt über die Detail-View (Abbildung 12) der Instanz, die durch einen Doppelklick erreicht wird.

Nachfolgend sollen anhand des Beispiels "Fahrer" die wichtigsten Elemente der Views dargestellt werden:

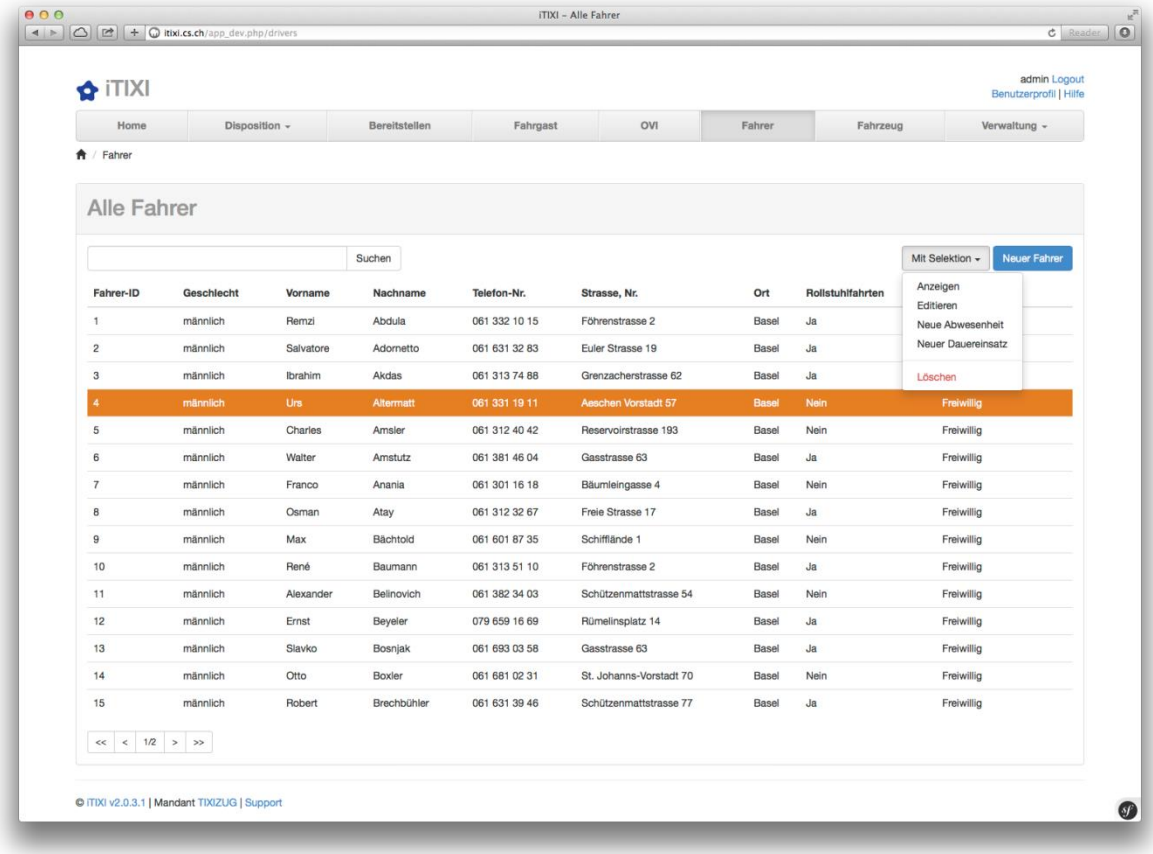


Abbildung 11: View Fahrer – Master-DataGrid

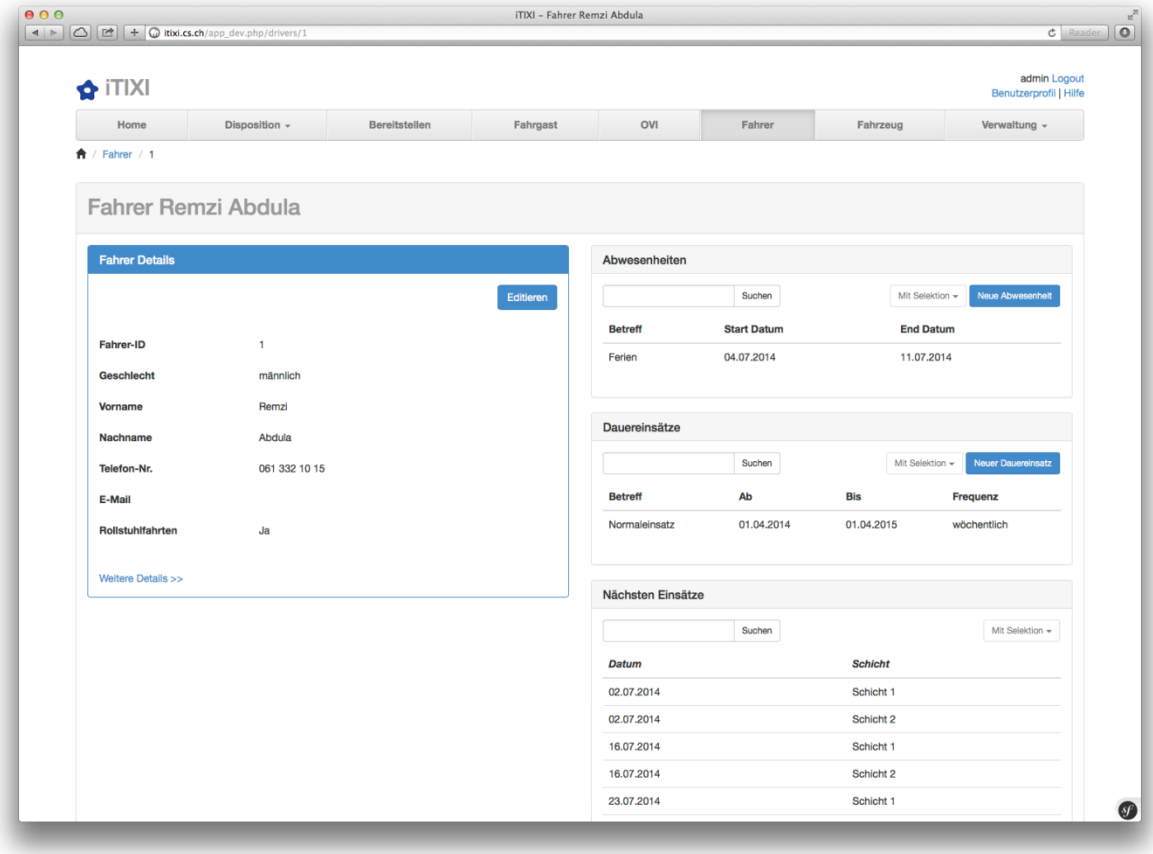


Abbildung 12: View Fahrer – Detail-View

3.2.2 ID-Konzept

Um das Frontend-Testing ermöglichen zu können, wurde ein ID-Konzept für alle informationsrelevanten View-Elemente erarbeitet. Dabei werden grundsätzlich Elemente mit lesendem, schreibenden und ausführenden Charakter unterschieden und mit den Funktionsprefixes *fpr_* (lesend), *fpw_* (schreibend), *fpx_* (ausführend) klassifiziert. Das Konzept im weiteren Sinne lässt sich am besten anhand des Beispiels der Entität Fahrzeug darstellen:

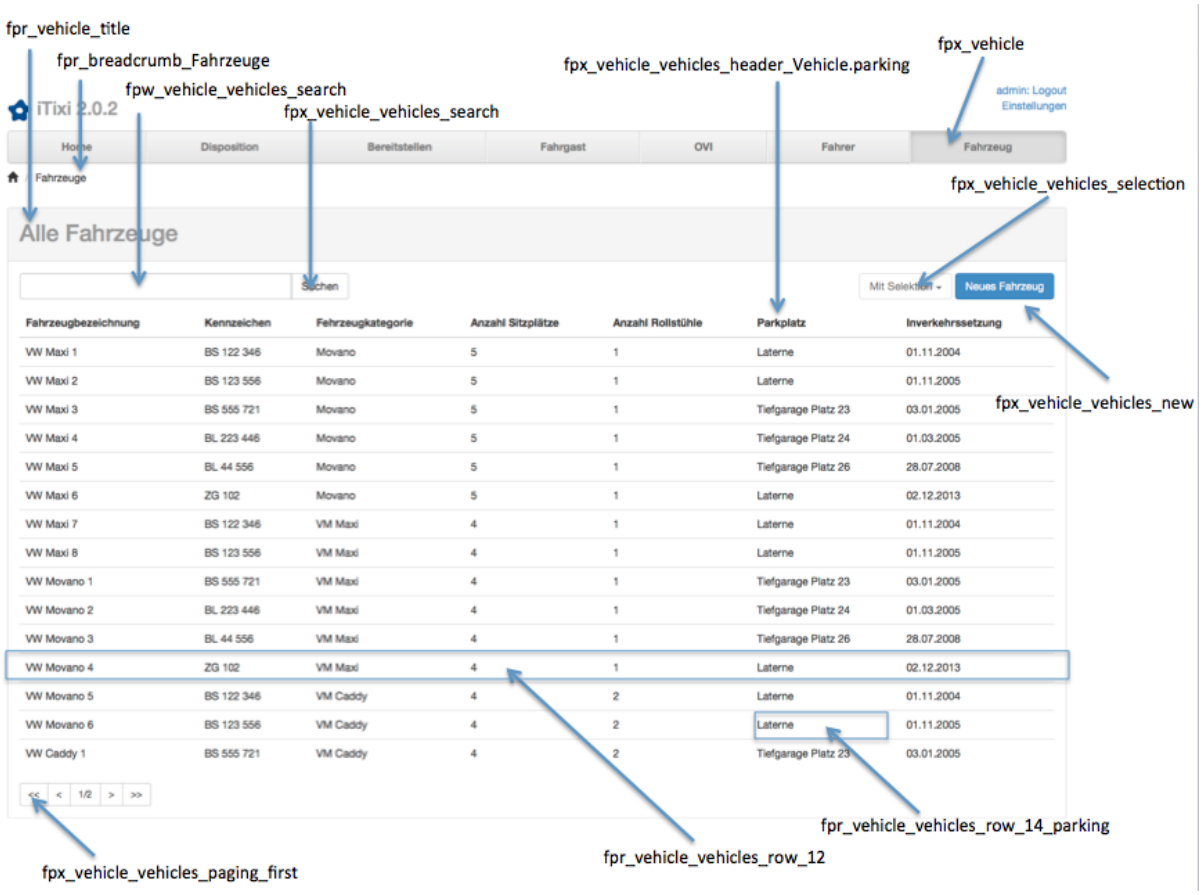


Abbildung 13: ID-Konzept Master-View

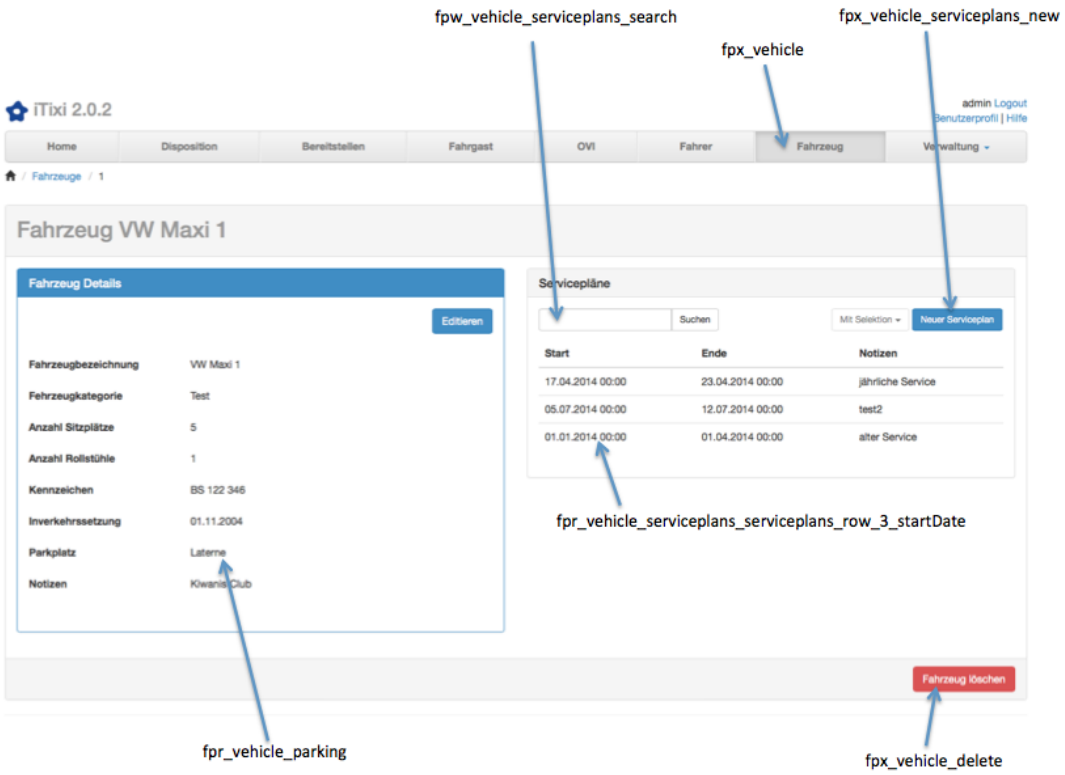


Abbildung 14: ID-Konzept Detail-View get

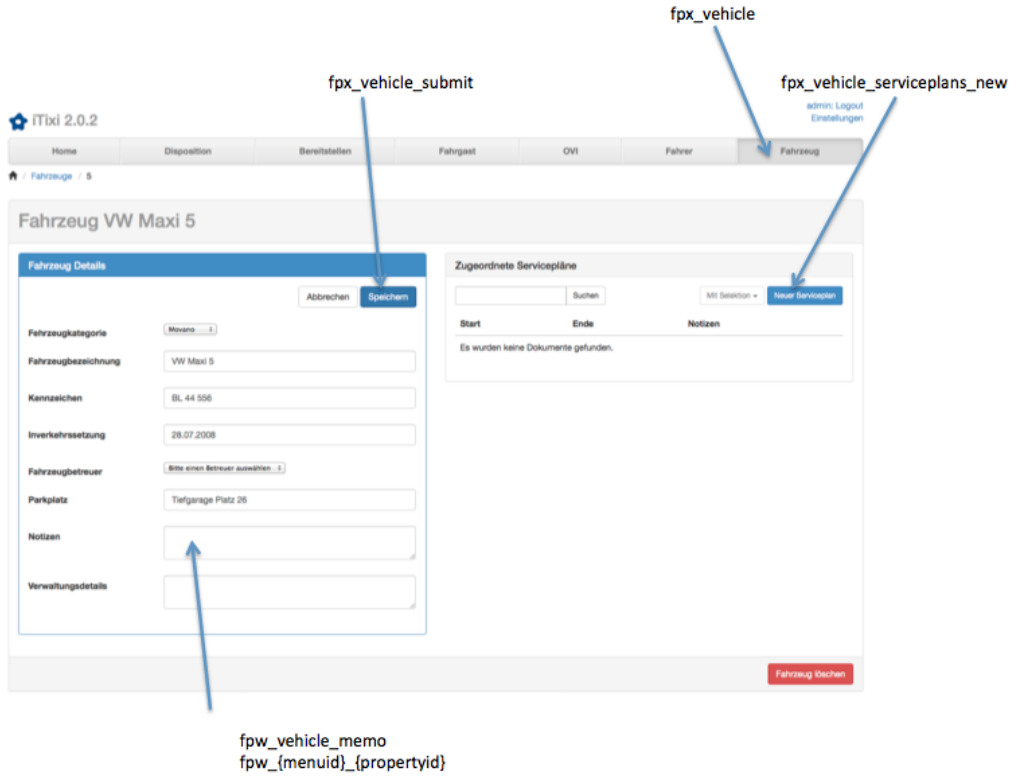


Abbildung 15: ID-Konzept Detail-View edit

3.2.3 Controller-API-Funktionen

Als Eintrittspunkte zum Frontend dienen durch Symfony's Route-Mapping @Route die Controller-Actions zum Aufbau der Response, welche sich für die Entitäten immer gleich aufbaut. Nachfolgend am Beispiel des Fahrers aufgelistet:

getDriversAction() zur Aufbereitung aller Fahrer-Entities für die Master-DataGrid View.

getDriverAction() zur Aufbereitung einzelner Fahrer-Entitiies für die Detail-View.

newDriverAction() zur Erstellung eines neuen Fahrers in der Detail-View Edit.

editDriverAction() zur Bearbeitung eines bestehenden Fahrers in der Detail-View Edit.

```
/**
* Class DriverController
* @Route("/drivers")
*/
class DriverController extends Controller {

        /**
        * @Route("", name="tixiapi_drivers_get")
        */
        public function getDriversAction(...) { ... }

        /**
        * @Route("/{driverId}")
        */
        public function getDriverAction( ... ) { ... }

        /**
        * @Route("/new", name="tixiapi_driver_new")
        */
        public function newDriverAction(Request $request) { ... }

        /**
        * @Route("/{driverId}/delete",name="tixiapi_driver_delete")
        */
        public function deleteDriverAction( ... ) { ... }

        /**
        * @Route("/{driverId}/edit", name="tixiapi_driver_edit")
        */
        public function editDriverAction( ... ) { ... }

}
```

3.2.4 Data Grid

Ein zentrales Element des Informationssystems sind Listen. Hierzu wurde eine eigene Implementierung realisiert für ein geeignetes Pagination grosser Datenmengen.

Diese findet sich im Package: `src\Tixi\ApiBundle\Shared\DataGrid`

> *Aus gesundheitlichen Gründen konnte dieses Kapitel nicht mehr durch CS vervollständigt werden*

3.2.5 Adressverwaltung

Ein weiteres zentrales Konzept für die iTXI-Applikation stellt die Adressverwaltung dar. Im Kontext von iTIXI hat eine Adresse grundsätzlich zwei Funktionen: Einerseits repräsentiert sie eine Postadresse und andererseits beschreibt sie einen Punkt auf der Erde, der eindeutig über geographische Koordinaten definiert ist.

Ziel der Adressverwaltung ist es, eine möglichst redundanzfreie und exakte Adressdatenbank mit Objekten aufzubauen, deren Datenstruktur sowohl aus der strukturierten Postadresse, als auch den zugehörigen Koordinaten besteht.

Um das definierte Ziel zu erreichen ist es notwendig, den Eingabeprozess durch den Benutzer so gut wie möglich zu unterstützen. Dafür wurde ein komplexes Adress-Konzept erarbeitet und implementiert, dessen Komponenten in diesem Abschnitt konzeptionell erklärt werden sollen.

Um das Konzept zu erklären, haben wir uns entschieden, zuerst den Prozess aus Benutzersicht aufzuzeigen und dann die nötigen Systemkomponenten, die dieses Verhalten ermöglichen, aufzuzeigen.

3.2.5.1 Adresseingabemaske / Benutzerinteraktion

Grundsätzlich können die folgenden Benutzerinteraktionen im Zusammenhang mit der Adresseingabemaske identifiziert werden:

Leeres Adressfeld: Der Benutzer trifft auf ein Adress-Formularfeld

Abbildung 16: Adresseingabemaske - Adressfeld

Adressvorschläge: Der Benutzer beginnt, die Postadresse einzugeben. Nach jedem eingegebenen Zeichen erhält der Benutzer immer stärker eingeschränkte Adressvorschläge, sofern das System eine Zuordnung machen kann. Falls keine Zuordnung gefunden werden kann, ist die Vorschlagsliste leer und der Benutzer ist aufgefordert, die Adresse manuell über den Link "Adresse manuell hinzufügen" einzugeben. Grundsätzlich kann die Adresse immer auch manuell eingegeben werden.

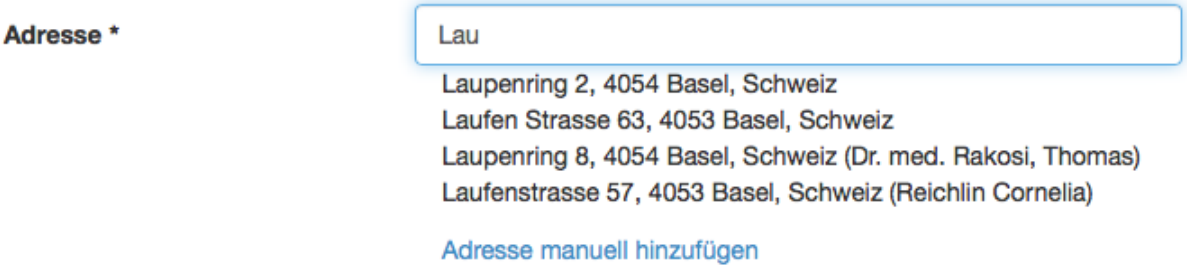


Abbildung 17: Adresseingabemaske - Adressvorschläge

Adressanzeige: Durch das Darüberfahren mit der Maus über die Adressvorschläge, wird die der Adressen direkt in einer Google Maps Karte angezeigt.

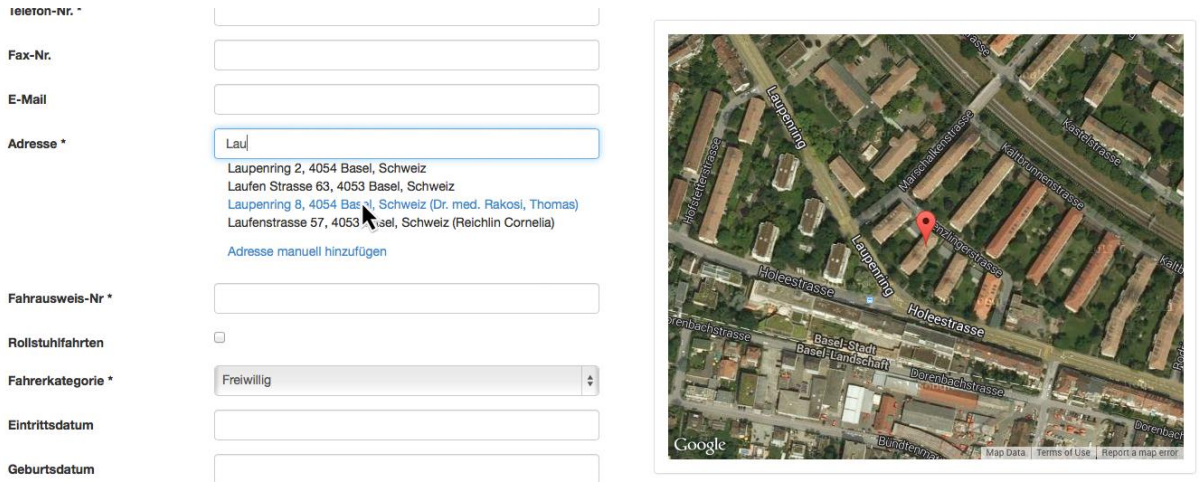


Abbildung 18: Adresseingabemaske - Adressanzeige

Adressauswahl: Durch Klicken kann eine Adresse ausgewählt werden

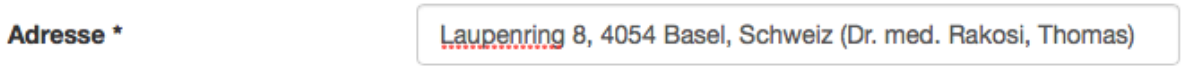


Abbildung 19: Adresseingabemaske - Adressauswahl

Adresse editieren: Durch (erneutes) Anwählen eines ausgefüllten Adressfeldes, wird die Adresse in einer Google Maps Karte angezeigt und der Benutzer hat die Möglichkeit, über den Link "Adresse manuell editieren" die Felder der Postadresse sowie auch die zugeordneten Koordinaten manuell zu korrigieren, sollten die Angaben nicht korrekt sein (was jedoch der Ausnahme entspricht). Die Korrektur der Koordinaten kann auch direkt über das verschieben des Markers auf der Google Maps Karte erfolgen. Eine Korrektur an einer Adresse führt dazu, dass das zugehörige Adressobjekt angepasst wird, wodurch eine Korrektur immer systemweit erfolgt.

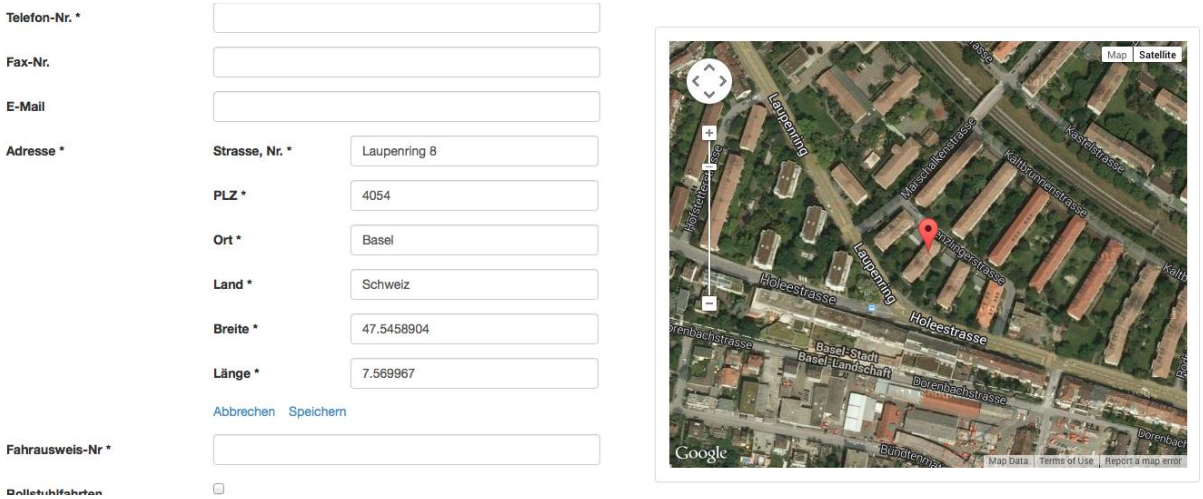


Abbildung 20: Adresseingabemaske - Adresse manuell editieren

Manuelle Adresseingabe: Wie bereits angesprochen, kann eine Adresse immer auch direkt manuell eingegeben werden. Dies sollte jedoch nur im Ausnahmefall gemacht werden, falls wirklich kein Adressvorschlag zutrifft. Für das Auffinden der Koordinaten kann erneut der Marker auf der Google Maps Karte verschoben werden. Dazu wird zuerst ein Ort über das innerhalb der Karte zur Verfügung stehende Eingabefeld gesucht und von dort ausgehend, die korrekte Position durch ziehen des Markers ermittelt. Die Koordinaten können natürlich auch manuell eingeben werden.

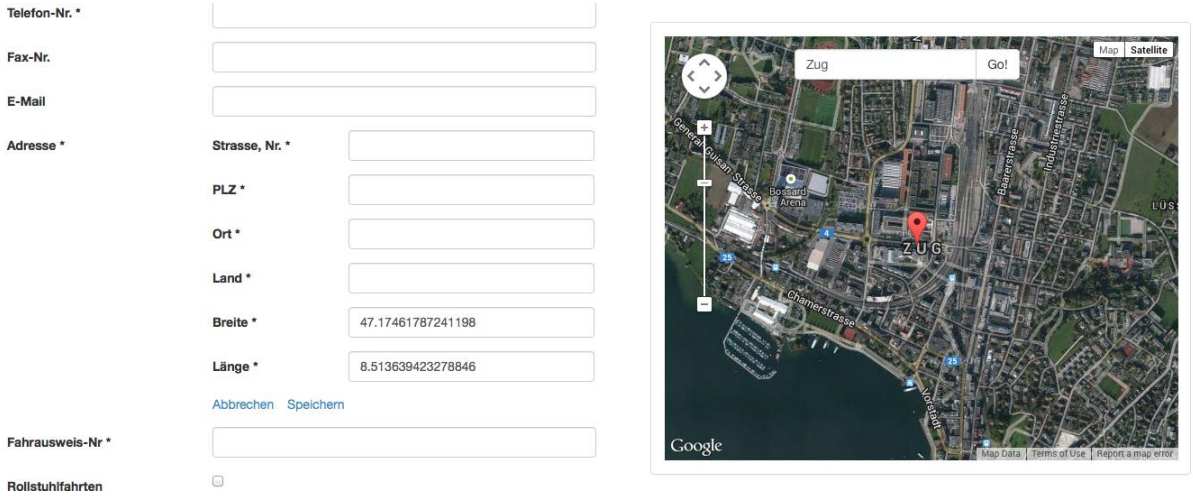


Abbildung 21: Adresseingabemaske - Manuelle Adresseingabe

3.2.5.2 Übersicht Systemkomponenten

Die Implementation des Adressverwaltungskonzeptes kann grundsätzlich in zwei Hauptkomponenten unterteilt werden: Einerseits in die Komponente, die die Adresseingabemaske rendert, die Interaktion mit der Eingabemaske steuert und die Kommunikation mit dem Adress-Service organisiert.

Diese Komponente soll als Adress-Umsystem (Adress-SurroundingSystem) bezeichnet werden. Auf der anderen Seite steht die Komponente, die als Adress-Service bezeichnet wird und die für das generieren und zuordnen von Adress-Entitäten anhand eines unstrukturierten Suchstrings verantwortlich ist.

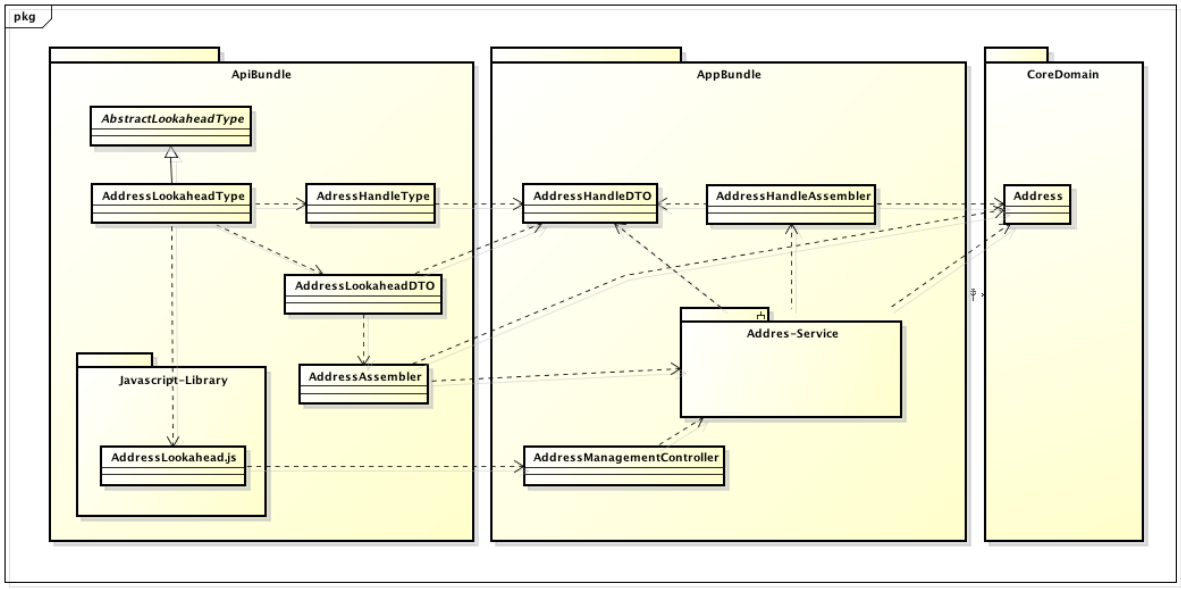


Abbildung 22: Das Adress-Umsystem

Das Adress-Umsystem kommuniziert mit dem Adress-Service grundsätzlich auf zwei Arten:

Einerseits, um anhand eines unstrukturierten Suchstrings als Inputparameter, Objekte zu erhalten, die alle nötigen Informationen enthalten, um daraus eine konkrete Instanz einer Adress-Entität zu erzeugen bzw. einer bestehenden Adress-Entität zuzuordnen. Der Suchstring wird per XMLHttpRequest von der Eingabemaske über den `AddressManagementController` an den Address-Service übertragen. Die Antwort-Objekte werden als AddressHandleDTO's bezeichnet und werden in serialisierter Form als JSON zurück an den Client übertragen, wo sie als Auswahlmöglichkeiten präsentiert werden (vergl. Abbildung 17).

Andererseits bietet der Adress-Service eine Funktion an, um AdressHandleDTO's in konkrete Adress-Objekte zu transformieren und, wenn nötig, zu persistieren. Diese Funktion wird benutzt, um aus den Form-Data-Klassen, die ein AddressLookaheadType und somit ein Adresseingabefeld enthalten, konkrete Adress-Entitäten zu extrahieren.

3.2.5.3 Adress-Service

Die wichtigste Komponente bei der Adressverwaltung ist der Address-Service, der im AppBundle implementiert ist. Kernaufgabe dieses Service ist es einerseits, einem nicht exakten Adresssuchstring, der über das Adress-Umsystem kommt, sogenannte AdressHandleDTO-Objekte zuzuordnen und diese andererseits, in einem separaten Prozess, auf konkrete Adress-Entitäten abzubilden. Als Besonderheit des Systems ist dabei die Einbindung der Google Geocoding API zu erwähnen, über die durch Einsatz eines Wrappers AdressHandles direkt aus den Adressdaten von Google erzeugt werden, falls die Suche in der lokalen Datenbank kein Treffer ergibt.

Das nachfolgende Klassendiagramm soll die Struktur des Adress-Service in der Übersicht zeigen:

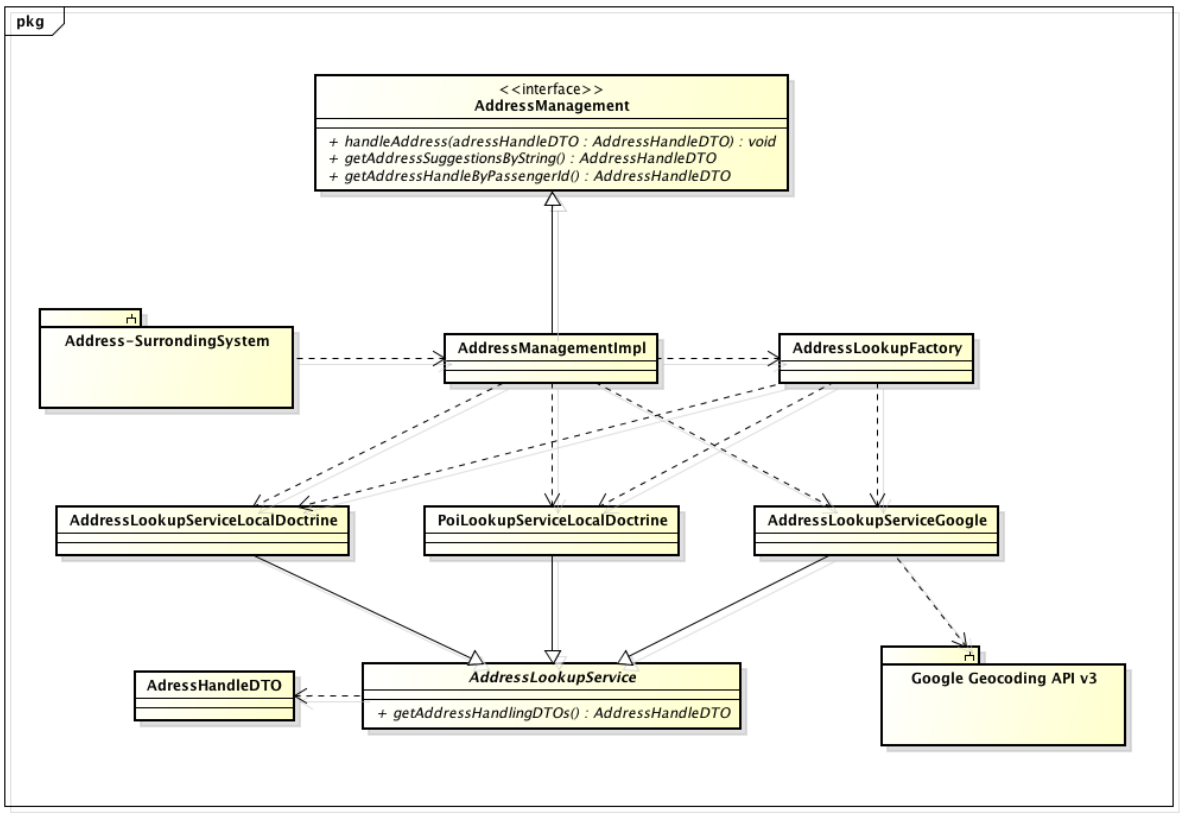
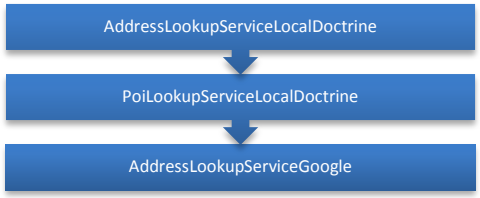


Abbildung 23: Die Komponenten des Adress-Service

Service-Trail Google Geocoding API

Der Versuch, passende Adress-Handle-Objekte zu einem unstrukturierten Suchstring zu finden, erfolgt über sogenannte AddressLookupService-Klassen bzw. über Klassen, die von dieser abstrakten Klasse erben. Jede dieser Klassen bedient sich einer anderen Quelle. Der Address-Service definiert einen sogenannten Service-Trail, der festlegt, in welcher Reihenfolge welche LookupService-Klassen nach möglichen Adress-Handle-Objekten durchsucht werden. Kann eine LookupService-Klasse kein Resultat für einen Suchstring liefern, wird zur nächsten Klasse in der Kette gesprungen, bis entweder ein oder mehrere Resultate gefunden wurden oder kein weiterer Service mehr zur Verfügung steht.

Der Service-Trail, wie er im Rahmen dieser Arbeit entwickelt wurde, ist wie folgt definiert:



AddressLookupServiceLocalDoctrine:

Zuerst wird die lokale Adress-Datenbank über den `AddressLookupServiceLocalDoctrine` durchsucht. Da dabei über alle Properties von potentiell zehntausenden von Address-Entitäten gesucht werden muss, musste der Service aus Gründen der Performanz sehr direkt mit der gewählten Datenbanktechnologie, sprich MySQL, verzahnt werden. Dazu wurde ein Symfony-Command erstellt, der MySQL Full-Text-Indexe über die Address-Table und die POI-Table anlegt. Diese werden durch absetzen der folgenden SQL-Kommandos erzeugt:

```
ALTER TABLE `address` ADD FULLTEXT `address_fts_idx` (`name`, `street`, `postalCode`, `city`,
`country`, `source`);


ALTER TABLE `poi` ADD FULLTEXT `poi_fts_idx` (`name`);
```

PoiLookupServiceLocalDoctrine:

Es ist auch möglich, Adressen der OVI's über den OVI-Name zu suchen. Dazu wird in zweiter Instanz über den `PoiLookupServiceLocalDoctrine` die POI-Table durchsucht.

AddressLookupServiceGoogle:

In letzter Instanz des Service-Trails steht die AddressLookupServiceGoogle-Klasse, die in der Funktion eines Wrappers, Suchanfragen an die Google Geocoding API weiterleitet.

3.3 Disposition

In diesem Abschnitt sollen die Implementationskonzepte für die wichtigsten Prozesse der Disposition erklärt werden.

3.3.1 Produktionsplan verwalten

Der Produktionsplan ist keiner konkreten Entity im Business Model zugeordnet sondern wird über Ein- und Ausgabeprozesse definiert, die mehrere Entitäten umfassen. Das Ziel des Produktionsplan ist es, auf Monatsbasis die Anzahl Fahrzeuge zu definieren, die für eine bestimmte Schicht zur Verfüg stehen.

Basis für den Produktionsplan bildet die Gesamtheit der im Domain Model unter 2.1.3.3 beschriebenen Funktionsgruppe von Business-Entitäten.

Gesteuert wird der Prozess im Wesentlichen durch den Dispositions-Service, der über den Symfony-Container zur Verfügung steht und vom `ProductionPlanAssembler` für die Erzeugung und Modifikation bedient wird. Die System-Prozesse, die im Zusammenhang mit der Eröffnung und Verwaltung eines Produktionsplanes stehen, lassen sich am besten über die Benutzereingabeprozesse erklären.

3.3.1.1 Übersicht Systemkomponenten

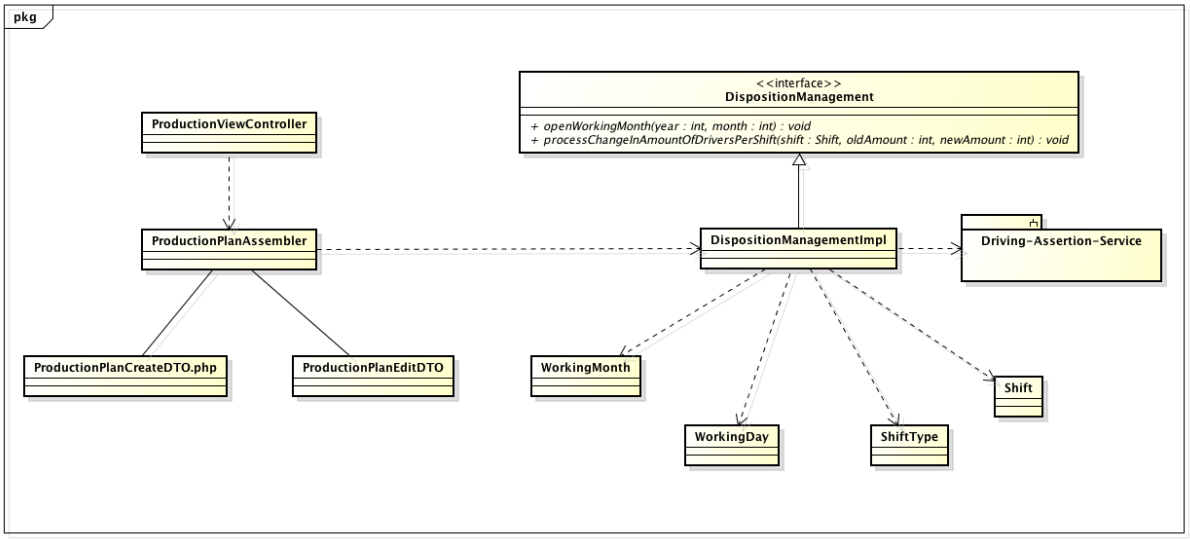


Abbildung 24: ProductionPlan Service Komponenten

Die API-Funktionalität für die Benutzerinteraktion in Zusammenhang mit dem Produktionsplan wird vom `ProductionViewController` bereitgestellt.

3.3.1.2 Produktionsplan eröffnen

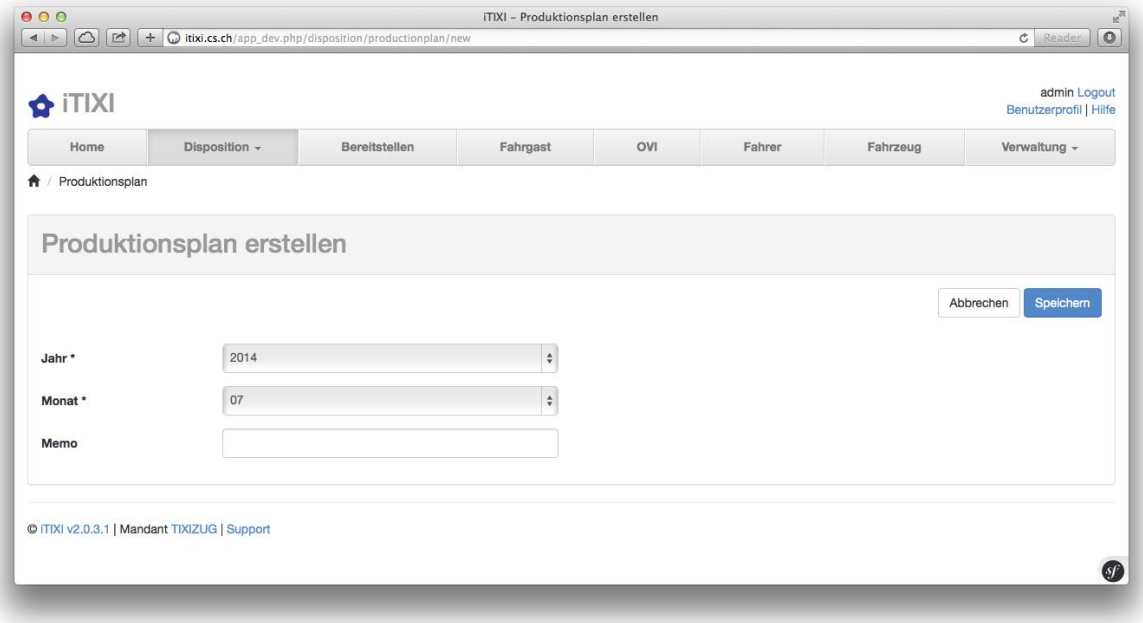


Abbildung 25: View - Produktionsplan eröffnen

Wird ein neuer Produktionsplan erstellt, so wird im Dispositions-Service (`DispositionManagementImpl`) die Methode `openWorkingMonth()` aufgerufen. Diese legt die Grundlage für einen neuen `WorkingMonth` an, sprich erstellt für alle Tage dieses Monats alle Schichten anhand der in der Verwaltung festgelegten `ShiftTypes`.

3.3.1.3 Produktionsplan editieren

Über die in den Shift-Entities für einen bestimmten WorkingMonth enthaltenen Informationen, kann über den ProductionPlanAssembler ein DTO geniert werden, das als Grundlage für eine Form steht, über das die Anzahl Fahrzeuge für eine bestimmte Schicht an das System übermittelt werden kann.

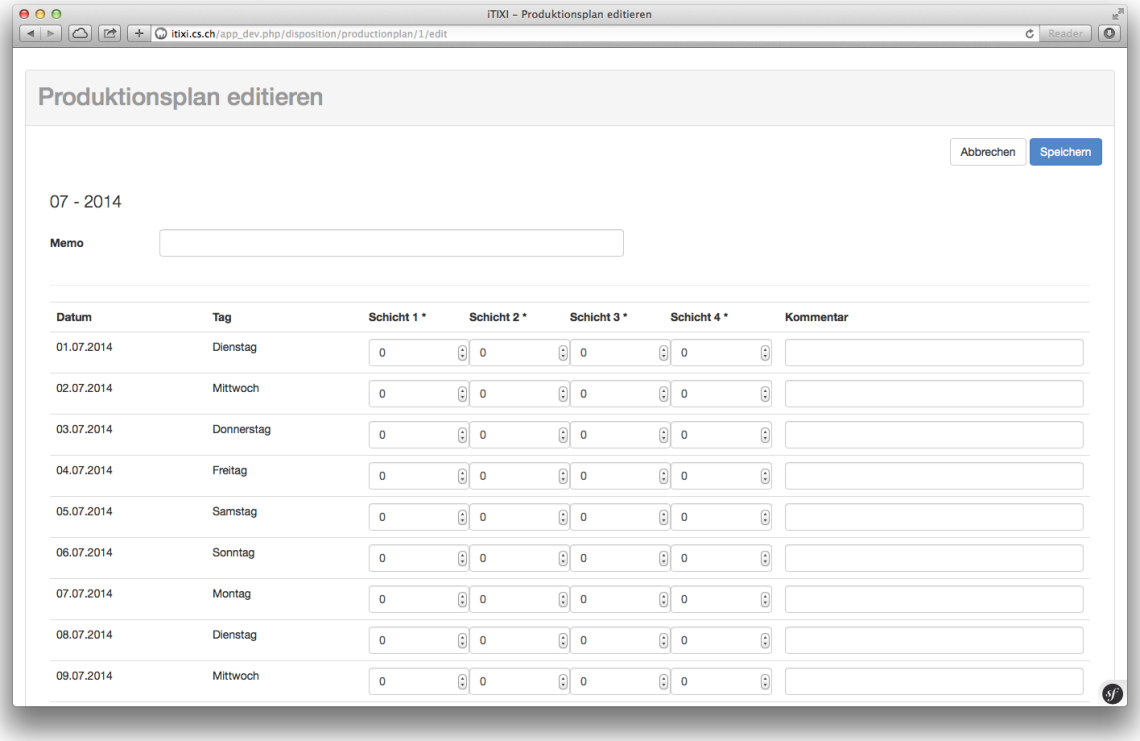


Abbildung 26: Produktionsplan editieren

Wird die Anzahl der Fahrzeuge im Produktionsplan verändert, so hat dies bindenden Einfluss auf alle angegliederten Prozesse der Disposition. Über den ProductionPlanAssembler wird beim Eintreten einer Änderung an einem `ProductionPlanEditDTO` ein Prozess angeworfen, der für das Change-Management zuständig sind. Dieser ist im `DispositionManagementImpl` angesiedelt als `processChangeInAmountOfDriversPerShift()` und erstellt oder löscht anhand der geänderten Anzahl die dazugehörigen DrivingPools.

3.4 Routen Abfrage

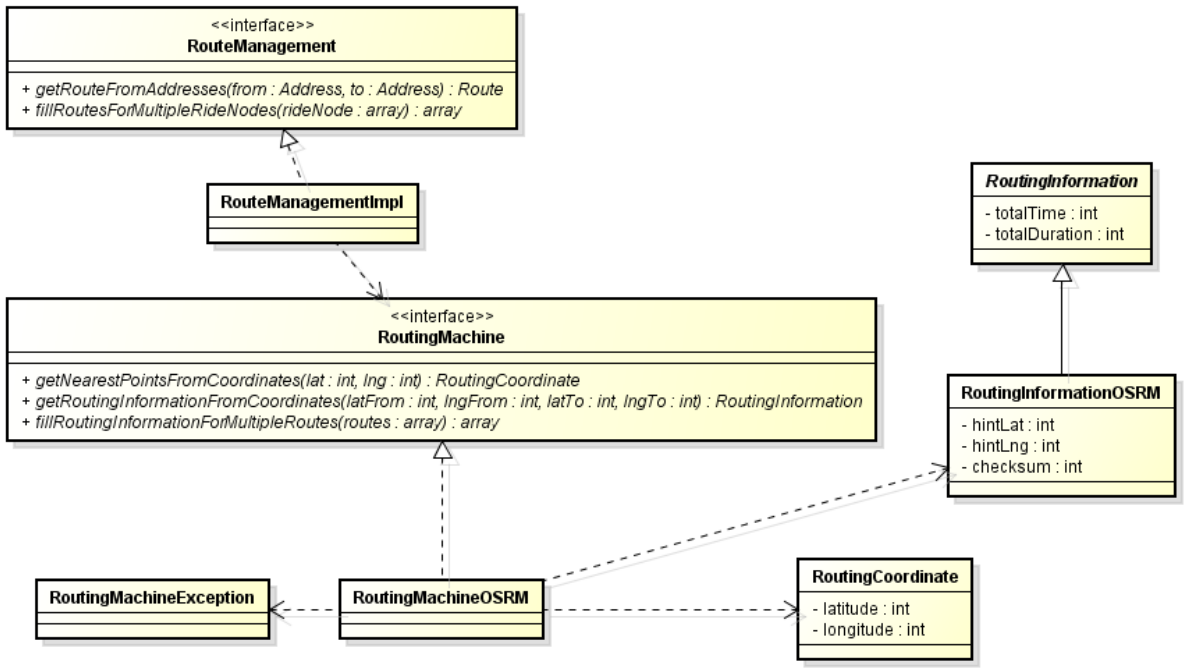


Abbildung 27: RouteManagement und RoutingMachine

Für die Abfrage von Routen wird der Service von OSRM verwendet. Im Kontext der iTIXI Applikation sind zwei Interfaces im Package App definiert:

- `RouteManagement`
- `RoutingMachine`

Mit `RouteManagementImpl` als Realisierung von `RouteManagement` und `RoutingMachineOSRM` als mögliche Realisierung einer `RoutingMachine` über OSRM.

3.4.1 RouteManagement

Für iTIXI werden die Routenabfragen für zwei verschiedene Adressen benötigt bei der Fahrauftragserfassung; über `getRouteFromAddresses()` wird ein `Route` Objekt erstellt mit den Routeninformationen zwischen Start- und Ziel-Adresse. Zwei Werte sind dabei immer erforderlich: Dauer und Distanz. Die abstrakte Klasse `RoutingInformation` sollte diese beiden Informationen immer beinhalten für die Berechnung mit der RoutingMachine.

Der weitere Anwendungsfall ist die mengenmässig Abfrage an Routen für die Fahrtoptimierung, hier werden viele Routeninformationen für die möglichen Leerfahrten berechnet. Bei der Fahrtoptimierung wird mit RideNode Objekten gearbeitet, das RouteManagement stellt deshalb eine geeignete Funktion über `fillRoutesForMultipleRideNodes()` zur Verfügung.

Für einen Fahrauftrag wird immer ein Route Objekt persistiert, da hier auch die Start und Ziel-Adresse festgehalten werden und somit die Informationen zum Fahrauftrag über Route und Address immer abrufbar sind. Man bekommt die Informationen über die Route eines Fahrauftrags somit auch wenn keine RoutingMachine (Lokaler Service OSRM oder andere Online-Dienste) erreichbar ist.

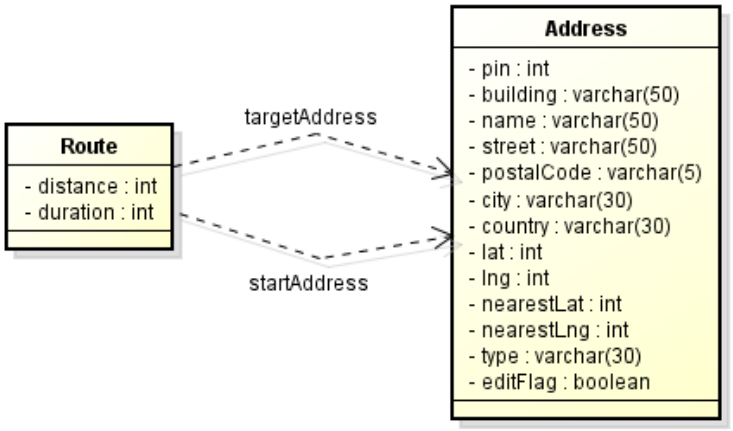


Abbildung 28: Detail Route und Address Beziehung

Eine Route definiert sich immer über die Start und Ziel-Adresse, die Identifier von targetAddress und startAddress bestimmen deshalb den Primary Key von Route:

```
UNIQUE KEY `search_idx` (`address_start_id`,`address_target_id`),
KEY `IDX_2C420799CB2071A` (`address_start_id`),
KEY `IDX_2C420798FCF372E` (`address_target_id`),
CONSTRAINT `FK_2C420798FCF372E` FOREIGN KEY (`address_target_id`) REFERENCES `address` (`id`),
CONSTRAINT `FK_2C420799CB2071A` FOREIGN KEY (`address_start_id`) REFERENCES `address` (`id`)
```

Bei der Abfrage einer Route für zwei Adressen wird zuerst in der Datenbank nach einer vorhandenen Route gesucht, dies kommt vor wenn zwei gleiche Fahraufträge erfasst werden. Wenn keine Route vorhanden ist werden über die RoutingMachine die passenden Informationen abgefragt.

Das Repository für die Route stellt die passenden Abfrage-Funktionen für vorhandene Routen zur Verfügung:

```
/**
 * @param Address $from
 * @param Address $to
 * @return Route
 */
public function findRouteWithAddresses(Address $from, Address $to) {
    $qb = $this->createQueryBuilder('e')
        ->where('e.startAddress = :startAddressId')
        ->andWhere('e.targetAddress = :targetAddressId')
        ->setParameter('startAddressId', $from->getId())
        ->setParameter('targetAddressId', $to->getId());
    return $qb->getQuery()->getOneOrNullResult();
}
```

Erst beim speichern eines Fahrauftrags werden auch dazugehörige Route-Objekte angelegt, falls nicht schon vorhanden.

Während der Erfassung eines Fahrauftrags haben wir den Fall, dass Dauer und Distanz in der entsprechenden View angezeigt werden, zu diesem Zeitpunkt aber noch keine Address-Objekte vorhanden sind.

Durch den Address-Lookahead sind beim Client die Koordinaten einer Adresse bekannt, ob diese Informationen aus der Datenbank oder über einen Online-Dienst kommen ist aber nicht gegeben.

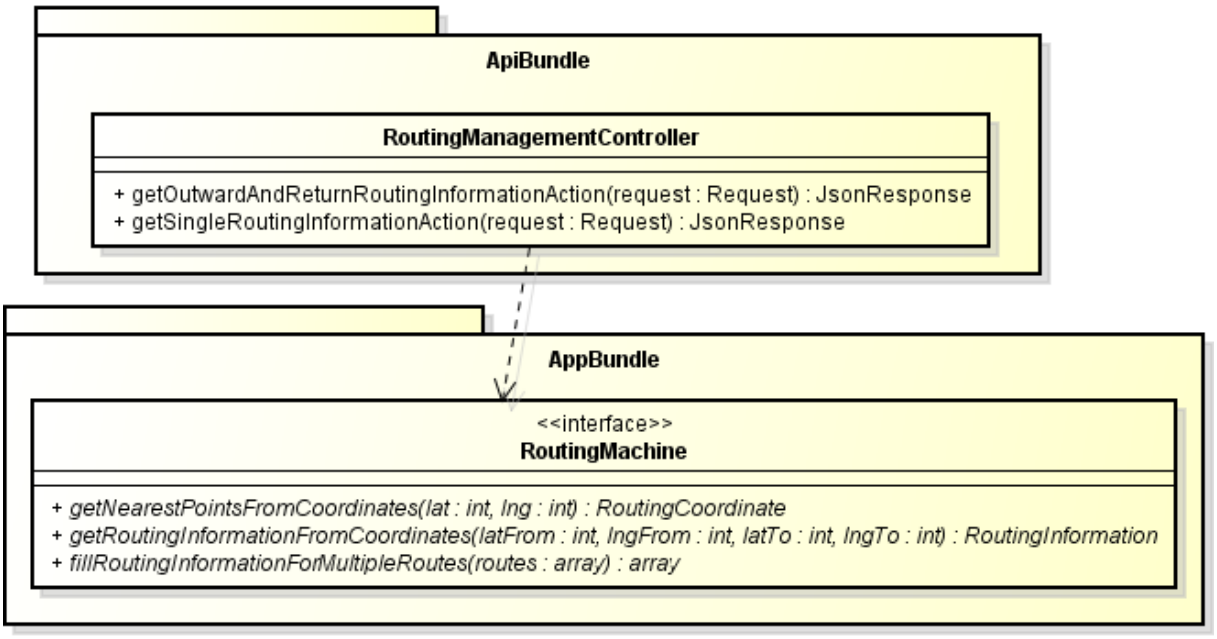


Abbildung 29: RoutingManagementController Zugriff

Der Controller `RoutingManagementController` erwirbt deshalb direkt über eine RoutingMachine die Informationen zu Dauer und Distanz und stellt diese dem Client als JSON zur Verfügung:

Methode	`getOutwardAndReturnRoutingInformationAction()`
API-Call	`/service/routing?latFrom=0.0&lngFrom=0.0&latTo=0.0&lngTo=0.0`
Response	`{"status":"0","routeOutwardDuration":1,"routeOutwardDistance":0,"routeReturnDuration":1,"routeReturnDistance":0}`

Für die Abfrage werden gleich Hin- und Rückfahrt berechnet, dabei werden einfach die Koordinaten vertauscht und dementsprechend zwei Abfragen über die RoutingMachine ausgeführt. Der übliche Fall beim Erfassen eines Auftrags ist mit einer Rückfahrt verbunden, es muss somit nur eine Abfrage vom Client aus getätigt werden.

3.4.2 RoutingMachine

3.4.2.1 OSRM Services

Für die Implementation zur Routen-Berechnung wird OSRM genutzt. Der OSRM Server bietet ein webbasiertes API an und ermöglich einfache Abfragen über HTTP. Die Response kann wahlweise über JSON oder JSONP erfolgen. Die OSRM Server API bietet drei Services an[6]:

Service	Nearest node im Strassennetzwerk zu einer Koordinate abfragen
API-Call	http://seiout.ch:8080/locate?loc=47.096531,8.463097
Response	{"version":0.3,"status":0,"mapped_coordinate":[47.096569,8.463191],"transactionId": "OSRM Routing Engine JSON Locate (v0.3)"}

Nearest Node sucht den nächstgelegenen Punkt auf einem Strassennetzwerk.

Service	Nearest point eines Strassensegments zu einer Koordinate abfragen
API-Call	http://seiout.ch:8080/nearest?loc=47.096531,8.463097
Response	{"version":0.3,"status":0,"mapped_coordinate":[47.096569,8.463097],"name":"Dorfplatz","transactionId":"OSRM Routing Engine JSON Nearest (v0.3)"}

Der Nearest Point sucht die nächstgelegene Koordinate, die auf irgendeinem Strassensegment vorhanden ist.

Diese Koordinate ist, wie im Kapitel Grundlagen erwähnt, notwendig für die Routen-Berechnung.

Service	Routen zu einer Anzahl Koordinaten abfragen
API-Call	http://seiout.ch:8080/viaroute?loc=47.498796,7.760499&loc=47.049796,8.548057
Response	{"version": 0.3,"status":0,"status_message": "Found route between points","route_geometry": "..","route_instructions": [],"route_summary":{"total_distance":108756,"total_time":4000,"start_point":"","end_point":"Parkstrasse"},"alternative_geometries": [".."],"alternative_instructions":[[]],"alternative_summaries":[{"total_distance":117105,"total_time":4321,"start_point":"","end_point":"Parkstrasse"}],"route_name":["",""],"alternative_names":[["",""]],"via_points":[[47.498796,7.760499],[47.049796,8.548057]],"hint_data": {"checksum":1617606390, "locations": ["KodMAAAAAAAbAAAAAAAAAAAAAAAAAAPA_LMbUAnNqdgA", "VYMVAA8FAAASAAAAIQAAAIxX3Wb-Xdc_ROzNAtluggA"]},"transactionId": "OSRM Routing Engine JSON Descriptor (v0.3)"}

Für die Abfrage von Routen-Informationen können mehrere Koordinaten (bis max. 25) im Request mitgegeben werden, wobei für unsere Routen-Berechnung meist nur Start und Ziel Koordinaten notwendig und keine weiteren „Via“ Punkte zu berücksichtigen sind. Mehrere Koordinaten würde man einfach nacheinander aufführen: `viaroute?loc=lat1,lon1&loc=lat2,lon2&loc=lat3,lon3&loc=lat4,lon4&...&latX,lonX`

[6] Siehe auch OSRM-Wiki: https://github.com/DennisOSRM/Project-OSRM/wiki/Server-api

3.4.2.2 OSRM Requests

Um einen Request ab zu setzten wird die cURL[7] Extension von PHP verwendet. Über einige cURL Parameter kann festgelegt werden, ob z.B. ein HTTP Header mitgegeben wird (für OSRM Web API nicht benötigt).

Da OSRM bei der TIXI Applikation auf der gleichen Server-Instanz betrieben wird und nur ein HTTP API anbietet, werden aus Performancegründen alle unnötigen Verifizierungen bei cURL deaktiviert.

```php
/**
 * @param $requestUrl
 * @return resource
 */
private function createCurlRequest($requestUrl) {
    $curl = curl_init();
    $options = array(
        CURLOPT_URL => $requestUrl,
        CURLOPT_POST => 0,
        CURLOPT_TIMEOUT => 30,
        CURLOPT_HEADER => 0,
        CURLOPT_RETURNTRANSFER => 1,
        CURLOPT_DNS_USE_GLOBAL_CACHE => 1,
        CURLOPT_DNS_CACHE_TIMEOUT => 3600,
        CURLOPT_SSL_VERIFYPEER => 0,
        CURLOPT_SSL_VERIFYHOST => 0,
        CURLOPT_FOLLOWLOCATION => 0,
        CURLOPT_IPRESOLVE => CURL_IPRESOLVE_V4,
    );
    curl_setopt_array($curl, $options);
    return $curl;
}
```

Abbildung 30: cURL Request mit Parametern für OSRM Abfrage

Weiter wird durch curl_multi eine parallele Abfrage ermöglicht indem mehrere cURL Handles werden dabei gleichzeitig abgesetzt und parallel abgearbeitet. Dies ergibt einen enormen Geschwindigkeitsvorteil gegenüber mehreren einzelnen Handles.

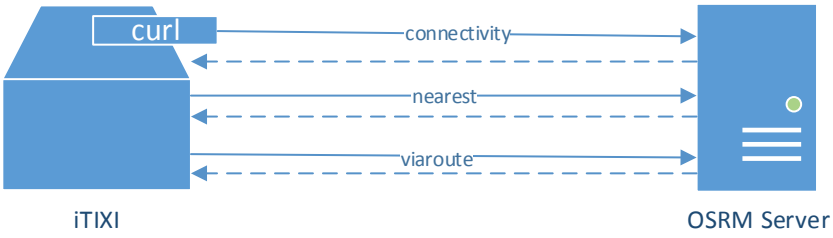


Abbildung 31: cURL single Request OSRM

Eine einzelne Routing-Information wird abgefragt, indem zuerst eine Verbindung über cURL aufgebaut wird und die Konnektivität zum OSRM Server testet. Danach wird ein cURL Handle erstellt um einen Request für die Abfrage der Nearest Point Koordinaten auszuführen. Wurde für die Abfrage eine bestehende Adresse aus der Datenbank gewählt, sind die Nearest Point Koordinaten meistens schon bekannt und der Request ist nicht nötig. Anschliessend wird ein cURL Handle erstellt um einen Request die Abfrage der Route zwischen diesen beiden Nearst Point Koordinaten auszuführen.

Um mehrere Abfragen gleichzeitig zu tätigen, was gerade für die Fahrtoptimierung benötigt wird um eine grosse Menge an Routen für die möglichen Leerfahrten zu berechnen. Über curl_multi werden mehrere Handles gleichzeitig erstellt, abgesetzt und verarbeitet.

[7] cURL: Client for URLs, verfügbar als Extension zu PHP

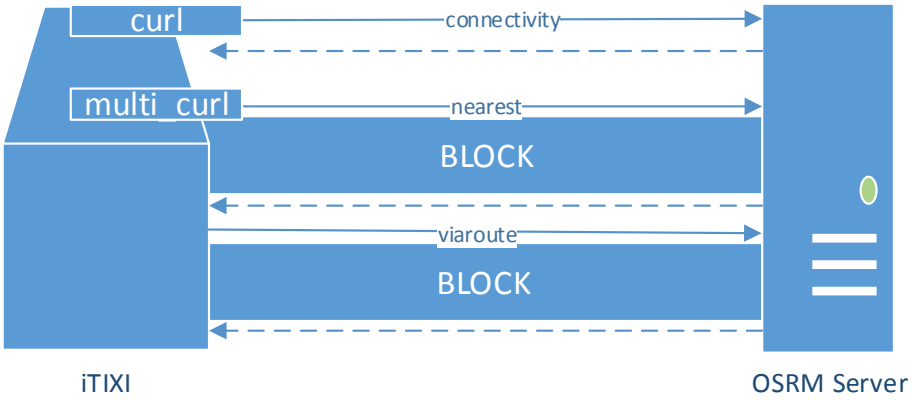


Abbildung 32: cURL multi Request OSRM

Bei unserer Implementierung wird über eine BLOCKSIZE die Anzahl der maximal simultan laufenden cURL Handles für curl_multi angegeben. Pro Block wird immer diese Anzahl cURL Handles ausgeführt, auf die Responses gewartet und die Resultate weiter verarbeitet.

Dabei spielt die Bandbreite und Verbindungsqualität eine Rolle bei der Anzahl an Handles, die gleichzeitig möglich sind. Die Performance von curl_multi variiert je nach Netzwerk-Stack des Betriebssystems und der kompilierten cURL Library. Eine Blocksize von 40 hat sich durch Tests als guter Mittelwert bestätigt zwischen Stabilität und Geschwindigkeit. Über das Abfragen einiger hundert Zufallsrouten wurden sporadisch verschiedene Blocksizes an einem Entwickler-Laptop ausprobiert. Ist eine zu schlechte Verbindung vorhanden, kann cURL nicht alle Handles absetzen und die parallelen Abfragen zu OSRM schlagen fehl.

Device: Lenovo mit IntelPRO, Windows 7 x64, easyPHP + cURL Extension
Verbindung: 100Mbit Ethernet an der HSR, Latenz zu OSRM Server seiout.ch: 32ms

Blocksize	Benötigte Zeit für 912 Abfragen
100	2.74s
80	2.80s
60	3.09s
40	3.52s
20	5.10s
10	8.13s

Schlägt eine Verbindung fehl oder konnten keine Daten vom OSRM Server abgerufen werden, wird eine `RoutingMachineException` geworfen mit Begründung über den Fehler.

Mit dem parallelen Absetzen von Requests errechnen sich mehr als 100 Routen pro Sekunde. Da wir bei den meisten Adressen die Nearest Point Koordinaten zusätzlich abspeichern, entfällt diese Abfrage und es wird zusätzliche Zeit eingespart. Bei der Fahrtoptimierung ist die Berechnung von Leerfahrten nötig zwischen Ziel-Adresse des einen Fahrauftrags und Start-Adresse des anderen Fahrauftrags – hier werden die Nearest Point Koordinaten immer vorhanden sein und minimiert die benötigte Zeit für das Berechnen der optimalen Fahrten um ein Weiteres.

3.5 Fahrwegsoptimierung

Wie im Kapitel Grundlagen erwähnt, wurde versucht eine mögliche Implementierung der Fahrtoptimierung zu realisieren. Im Package `App` wird dieser Service durch das Interface `RideManagement` definiert.

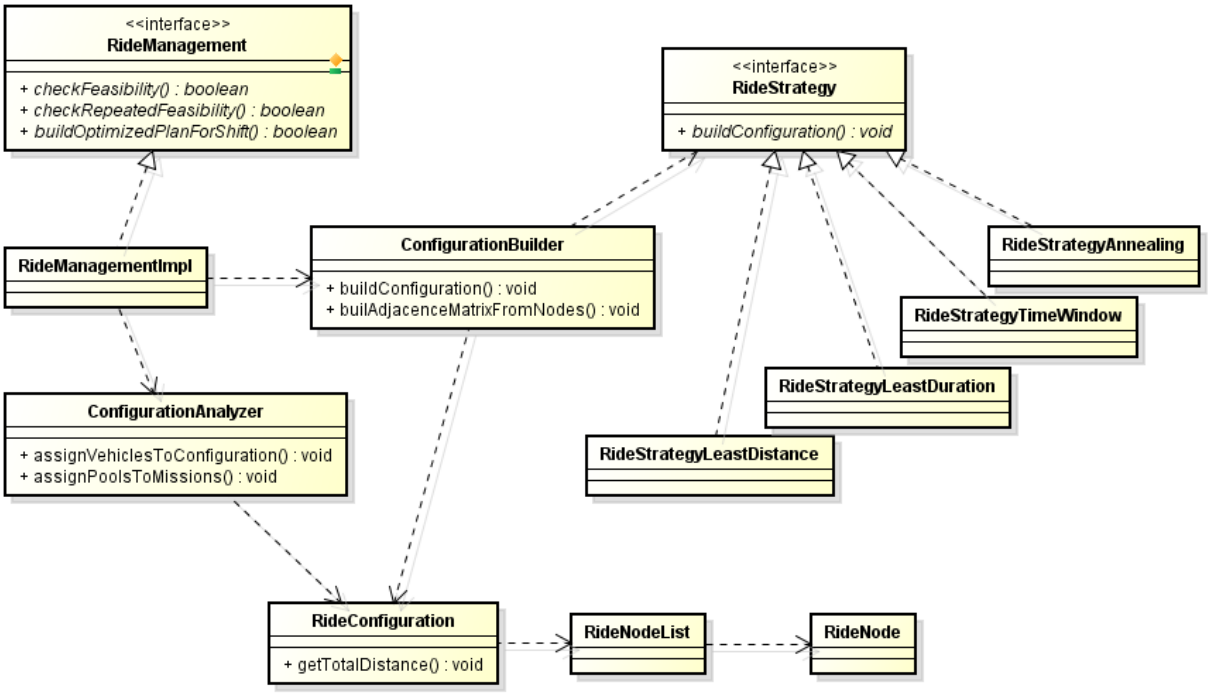


Abbildung 33: RideManagement Klassen Übersicht

Die Konfigurationsbildung lehnt den im Kapitel Grundlagen – DARP definierten Begriffen an.

Es wurde darauf geachtet, dass der Prozess der Konfigurationserzeugung selbst austauschbar ist, d.h. es sollen verschiedene Algorithmen einsetzbar sein. Deshalb wurden Konfigurations-Strategien definiert und gemäss dem Muster des Strategy Patterns[8] implementiert.

3.5.1 Konfigurationserzeugung

Die Klasse `ConfigurationBuilder` hält eine Strategie vom Typ `RideStrategy` als Member-Variable und führt die Konfigurationsbildung `buildConfiguration()` auf einer konkreten Strategie aus, z.B. RideStrategyAnnealing.

[8] Strategy Pattern: Entwurfsmuster aus GoF

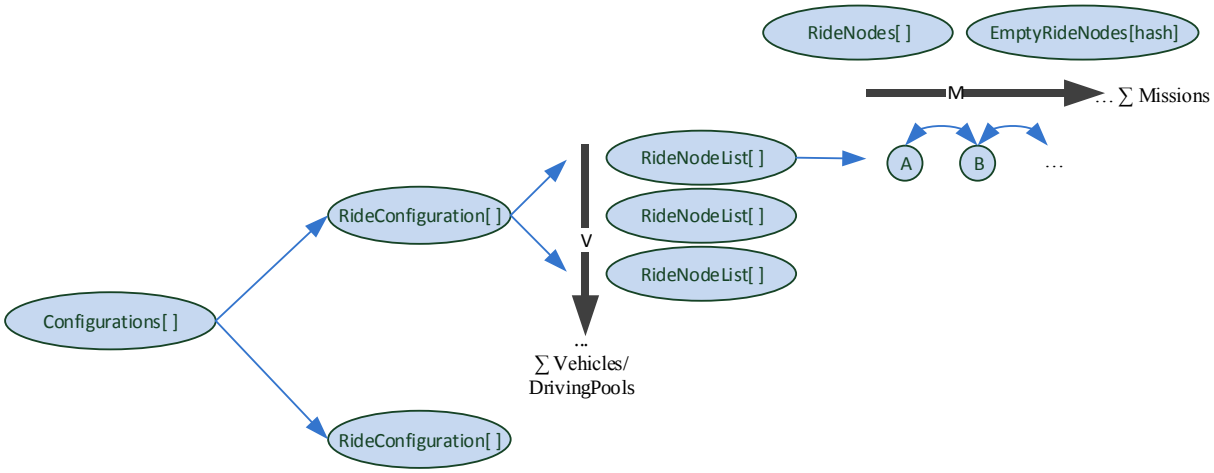


Abbildung 34: Aufbau der Rideconfigurations

Im `ConfigurationBuilder` wird ein Array mit den erzeugten RideConfigurations gehalten, woraus dann die beste Konfiguration genommen werden kann für weitere Zuteilungen.

Die RideConfigurations werden durch eine Strategie erzeugt, in dem pro benötigte Fahrzeuge eine RideNodeList angelegt wird, die eine LinkedList von RideNodes pflegt (siehe Abbildung 34).

Die Menge an RideNodes ergibt sich aus allen DrivingMissions die optimiert werden.

Unsere Implementierung sieht vor, dass pro Schicht optimiert werden kann. Alle `DrivingMissions` die Ihre Startzeit innerhalb einer Schicht haben werden dazu übernommen (siehe Abbildung 35).

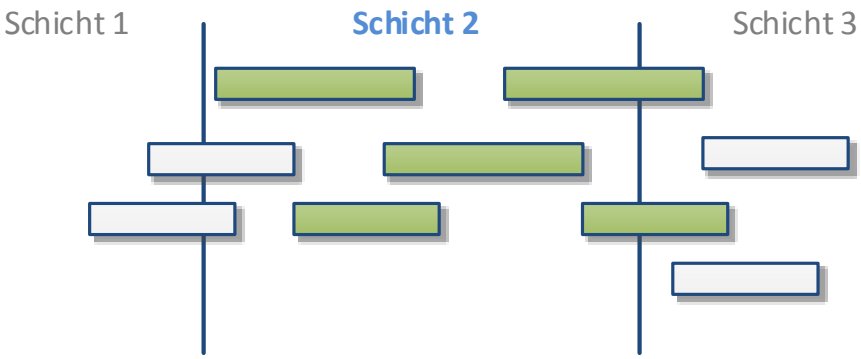


Abbildung 35: Beispiel DrivingMissions in Schicht2

Aus den DrivingMissions werden dann RideNodes erzeugt, wobei nur noch Startzeit in Minuten + Fahrtdistanz + Fahrtdauer + zusätzliche Minuten (Boarding Times etc.) interessieren für die Berechnung. Ein RideNode dient in diesem Sinne als DTO für die Konfigurationsbildung innerhalb einer Strategie.

Die Leerfahrten können dann aus allen vorhandenen RideNodes zusammengestellt und über die RoutingMachine abgerufen werden. Diese werden in einer geeigneten HashTable EmptyRideNodes hinterlegt, welche aus RideNodes besteht die keiner DrivingMission entstehen und als Key einen geeigneten Hash aus den Start und Ziel Koordinaten haben. Der HashKey wird per MD4-Hashfunktion berechnet aus einem zusammengesetzten String von Latitude + Longitude der Startadresse und Zieladresse.

Dieser Hash entspricht damit genau einer Route. Sollte eine gleiche Leerfahrt entstehen zwischen zwei RideNodes, so wird diese unter dem gleichen Value in der Hashtable gespeichert und muss somit nur einmal

abgefragt werden. MD4 wurde gewählt, da diese Hashfunktion in PHP gemäss Benchmarks[9] sehr schnell berechnet ist und mit einem 128-Bit Hash genügend grosse Sicherheit gegenüber möglichen Kollisionen bietet.

Über die Abfrage eines Hashs zwischen zwei RideNodes (z.B. Zieladresse von RideNode 1 und Startadresse von RideNode 2) kann somit innert $O(1)$ die entsprechende Leerfahrt herausgelesen werden.

Aus allen Fahrten und Leerfahrten wird dann eine Adjacence-Matrix erstellt, die Auskunft über machbare aufeinanderfolgende RideNodes gibt, und wenn diese machbar sind, uns gleich die entsprechende Leerfahrt zurückgibt. Mit dieser Adjacence-Matrix als Basis, werden dann die Konfigurationen gemäss einer Strategie erstellt:

RideStrategyTimeWindows

- erstellt aufgrund bereits zugeteilten DrivingMissions und DrivingPools eine Konfiguration von RideNodes und fügt nicht zugeteilte RideNodes, die zur bestehenden Konfiguration in einem Zeitbereich machbar sind, hinzu. Diese Konfiguration dient nur dazu, über Zeitslots herauszufinden, ober ein Fahrauftrag überhaupt machbar wäre und nicht schon mehr Fahraufträge zeitlich im gleichen Fenster liegen als die Menge an vorhandener Fahrzeuge.

RideStrategyLeastDistance

- gemäss einfachem Algorithmus. Hier werden verschiedene RideNodes als Start für eine Fahrt genommen und darauf immer die nächste RideNode, welche machbar ist, mit kleinster Distanz zugeteilt. Aus den verschiedenen machbaren Konfigurationen mit unterschiedlichen Start Nodes wird die günstigste gewählt.

RideStrategyLeastDuration

- gemäss einfachem Algorithmus, einfach darauf ausgelegt alle möglichen RideNodes nach Fahrtzeit in einer Konfiguration zu haben. Bei RideStrategyLeastDistance ist es möglich, nicht machtbare Konfigurationen zu erzeugen wenn nicht alle Aufträge einer RideNodeLists, bzw. einem Fahrzeug, zugeordnet werden können.

RideStrategyAnnealing

- erstellt zuerst eine machbare Konfiguration mit einem einfachen Algorithmus wie *RideStrategyLeastDuration*. Diese Initialkonfiguration wird gemäss dem Optimierungsverfahren Simulated Annealing auf ein globales Minimum gebracht.

Die RideNodeList hält beim Einfügen und Entfernen von RideNodes immer folgende Daten fest:

- gesamte Distanz aller Fahrten und Leerfahrten
- totale Dauer aller Fahrten und Leerfahrten
- maximale Anzahl Passagiere
- maximale Anzahl Rollstühle
- nicht gewünschte Fahrzeugkategorien durch Passagiere

Durch die Summe aller RideNodeList Daten hält eine RideConfiguration dann die gesamte Distanz und Dauer für eine Konfiguration fest, aufgrund dieser man die verschiedenen Konfigurationen messen kann.

[9] http://benchmarks.ro/2009/04/hashing-algorithms-benchmarked/#benchmark_code

3.5.2 Konfigurationsanalyse

Sind erst einmal günstige Konfigurationen zusammengestellt, werden zu der bestmöglichen noch Fahrer und Fahrzeuge zugeordnet. Dies wird über die Klasse `ConfigurationAnalyzer` bestimmt, ein *Analyzer* untersucht eine Konfiguration und ordnet weitere Objekte zu.

Verschiedene *Analyzer* als Strategien zu implementieren sahen wir momentan nicht als Notwendigkeit, die Zuordnung ist meist fix geregelt durch Constraints und für unser Beispiel recht klar welche Fahrer und Fahrzeuge für einen Ride geeignet sind. Trotzdem wäre es vorstellbar, unterschiedliche Constraints zu verfolgen und diese als weitere Strategien zu implementieren.

Vorhandene Vehicles werden über `assignVehiclesToConfiguration()` geprüft auf die Verträglichkeit mit einem Ride. Die Liste der Vehicles wird nach Fahrzeuggrösse sortiert um kleinere und meist energieeffizientere Fahrzeuge bei der Auswahl zu bevorzugen. Der Sortierprozess beruht sich hierbei nur auf die Summer der Anzahl Sitzplätze und Rollstuhlplätze. Es werden alle RideNodeLists mit den Fahrzeugen verglichen und die Verträglichkeit anhand der festgehaltenen Informationen in einer RideNodeList bestimmt.

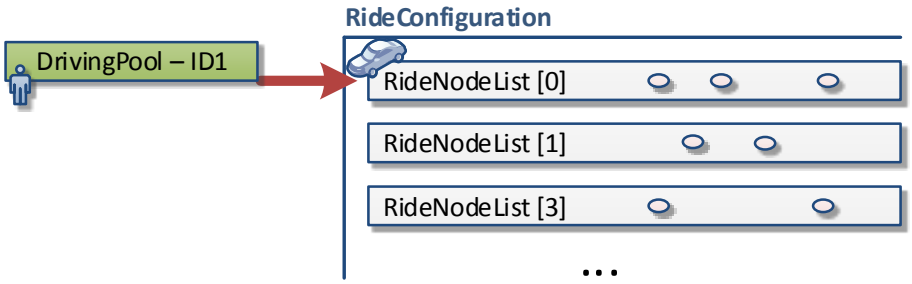


Abbildung 36: Zuteilung Fahrer/Fahrzeuge zu Fahrten

Die vorhandenen DrivingPools, welche bei der Monatsplanung einen Fahrer zugeordnet bekamen, werden nun über `assignPoolsToMissions()` den mit Fahrzeug verknüpften RideNodeLists zugeteilt, wobei auf die Verträglichkeit mit den Fahrern geachtet wird – Fahrer zu Fahrzeugkategorien und wenn der Fahrer Extra-Minuten hat, ob die nacheinander folgenden Aufträge mit dieser zusätzlichen Zeit machbar sind.

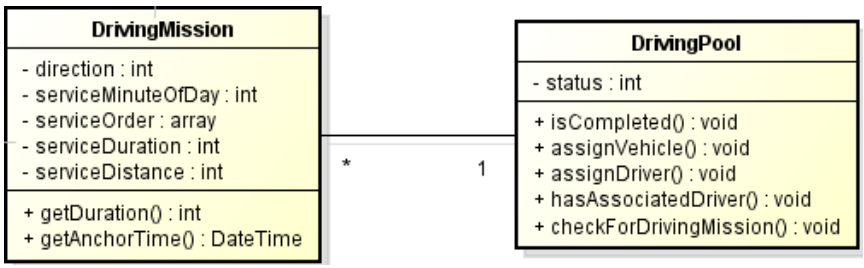


Abbildung 37: DrivingMission zu DrivingPool

Ist die Zuteilung erfolgreich, wird jeweils die aus den RideNodes enthaltene DrivingMission mit dem verträglichen DrivingPool verknüpft. Die Optimierung einer Schicht wird dann beendet.

3.5.3 Annealing

Um kurz auf den Annealing Prozess einzugehen, soll nachfolgende Vereinfachung der Implementation zur Erläuterung dienen.

```
$temperature = 10000.0;
$coolingRate = 0.999;
$absoluteTemperature = 0.0001;
//loop until system has cooled, this is our break criterion
while ($temperature > $absoluteTemperature) {
        //swap two nodes
        $nextConfiguration = $this->getNextRandomConfiguration($currentConfiguration);
        if ($nextConfiguration !== null) {
                $nextDistance = $nextConfiguration->getTotalDistance();
                $deltaDistance = $nextDistance - $distance;
                //random next double 0.0 > ran < 1.0
                $ran = mt_rand(1, 100000000) / 100000000;
                //we accept the configuration if distance is lower or satisfies Boltzmann condition
                if (($deltaDistance < 0) || ($distance > 0 && exp(-$deltaDistance / $temperature) > $ran)) {
                        $distance = $nextDistance;
                        $currentConfiguration = $nextConfiguration;
                        //keep track of the best configuration
                        if ($distance < $bestDistance) {
                                $bestDistance = $nextDistance;
                                $configurations[] = clone $nextConfiguration;
                        }
                }
        }
        $temperature *= $coolingRate;
}
```

Abbruchkriterium ist die Temperatur, wenn diese unter die *absoulteTemperature* fällt endet die Prozedur. Weiter denkbar wären noch Zeit Kriterien; falls der Prozess eine gewisse Zeit überschreitet, soll beendet werden. Bei den Tests wurde aber nie eine Ausführungszeit von 1min überschritten.

über `getNextRandomConfiguration()` werden aus der momentan behandelten Konfiguration zufällig zwei RideNodes miteinander vertauscht und die gehaltenen Informationen aktualisiert, aber nur wenn diese machbar sind. Bleibt die Konfiguration machbar, wird überprüft, ob damit eine bessere Distanz erreicht wird. Wenn eine günstigere Distanz erreicht wurde oder die Wahrscheinlichkeit durch die Boltzmann Verteilung gegeben ist, wird diese Konfiguration aus Ausgangslage für die nächsten Berechnungen gesetzt. Ist die entstandene Distanz günstiger als die globale Distanz *bestDistance*, wird die Konfiguration zu einem Set hinzugefügt, aus dem nach dem Annealing die Beste genommen wird.

Nach jedem Schritt kühlt die Temperatur durch die gegebene *coolingRate* ab.

Um die entstandene Konstellation zu veranschaulichen ohne der implementierten View zur Tagesplanung, wurden die Resultate per *print()* ausgegeben.

Im Beispiel von Abbildung 38 wurden 1608 Routen (die möglichen Leerfahrten) innert 8 Sekunden abgefragt.

59 Aufträge wurden für eine Schicht bearbeitet. Die Initialkonfiguration über RideStrategyLeastDuration ergab eine Gesamtdistanz von 282.969km, über den Annealing Prozess wurde eine Gesamtdistanz von 200.761km innert 4 Sekunden erreicht. Die Tabelle zeigt eine Zuweisung von DrivingPools (ID links), die maximale Anzahl Rollstühle (W) und Passagiere (P) in einer RideNodeList, das zugewiesene Fahrzeug und nachfolgend die Fahraufträge/DirivingMissions (ID in Klammern) hintereinander.

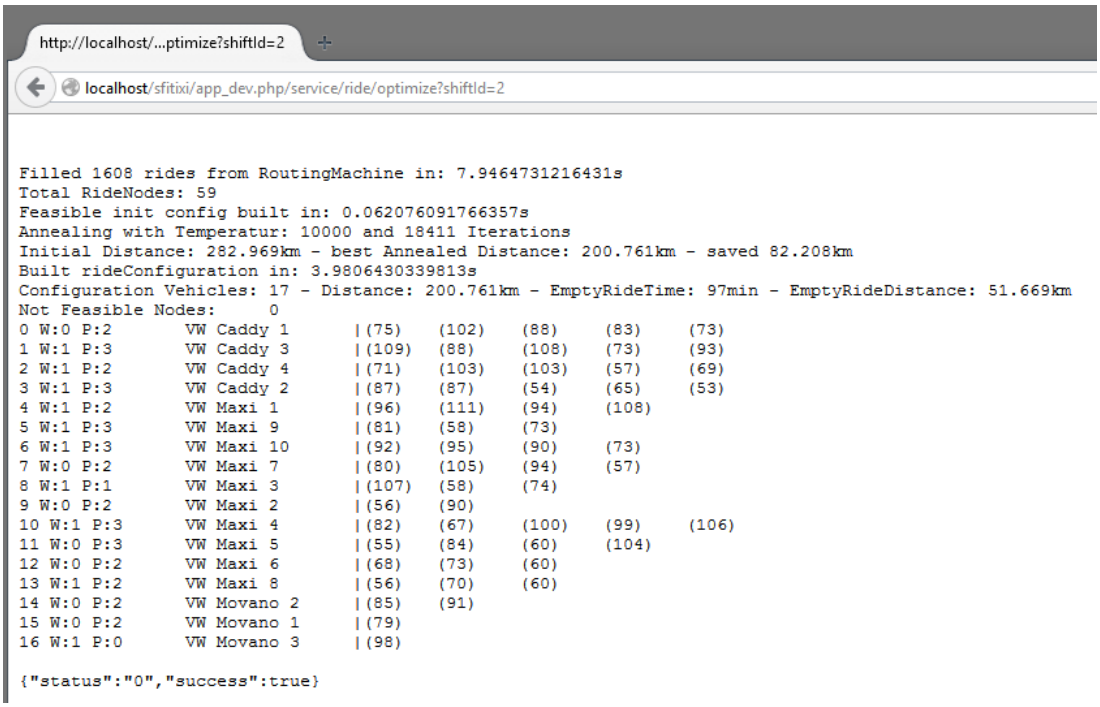


Abbildung 38: Print einer optimierten Schicht

Weitere Testresultate sind im Appendix aufgeführt. Die Tests wurden hier auf einem leistungsstarken Arbeitsplatz (Core i7 3.4GHz) weshalb die Ausführungszeiten recht kurz sind.

3.6 Security

Das gesamte Security-Konzept wurde von Symfony übernommen. Eine Formular-basierte Anmeldung mit Schutz vor Cross-site request forgery (CRSF) zur Authentisierung von Benutzern, Access-Controls von URL's und Access-Control-Lists (ACL) zur Authorisierung und Freigabe von Ressourcen, diese Konzepte werden durch eine PHP Library von Symfony zur Verfügung gestellt. Die Verbindung zwischen Client und Server soll, wie im Kapitel Deployment aufgeführt, durch HTTPS gesichert sein.

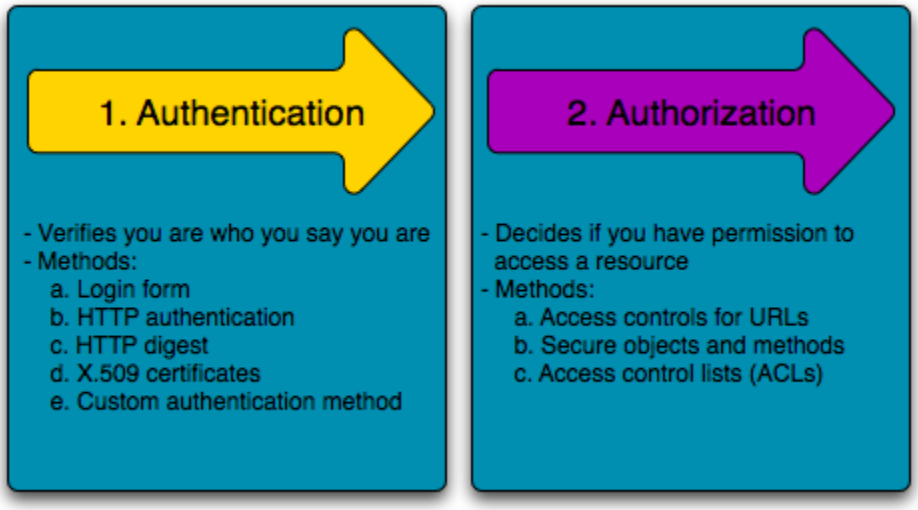


Abbildung 39: Authentisierung und Authorisierung gemäss Symfony (Quelle: symfony.com)

Die Security Klassen wurden gesondert von der Domain Implementierung als zusätzliches Bundle realisiert, da diese sehr nahe an Symfony anzugliedern sind und nicht zur eigentlichen Implementierung gehören.

Über einen UserProvider kann die Ablage und Verwaltung von Benutzern und Rollen für Symfony individuell gestaltet werden. Rollen (bzw. ROLES) entsprechen soweit eine Benutzergruppe, derer man verschiedene Berechtigungen erstatten kann. Ein Benutzer kann in mehreren Benutzergruppen zugeordnet sein.

Bei iTIXI werden Benutzerobjekte und Rollen in der MySQL Datenbank persistiert. Benutzerkennwörter werden per BCrypt[10] zu einem geeigneten Hash für die Ablage umgewandelt.

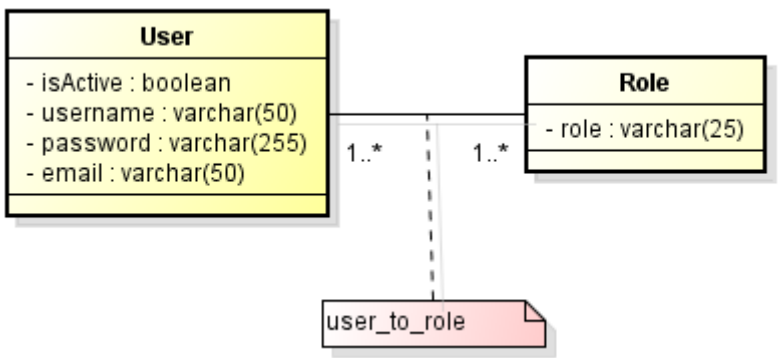


Abbildung 40: User und Role Klassen

[10] BCrypt: Kryptologische Hashfunktion, speziell für das Speichern von Passwörtern

3.6.1 Roles

Durch die Nutzung von ROLES im Symfony, lassen sich einfach gewisse Ressourcen an bestimmte Benutzer freigeben, z.B. der Zugriff über einen Controller:

```
if (false === $this->get('security.context')->isGranted('ROLE_MANAGER')) {
    throw new AccessDeniedException();
}
```

Abbildung 41: Zugriff gestatten an ROLE_MANAGER in einem Controller

Symfony erlaubt den Aufbau einer gewissen Rollen-Hierarchie, für iTIXI wurde diese adaptiert für:

Rolle	Beschreibung
ROLE_DISPO	Disponent, Benutzer der Disposition
ROLE_MANAGER	Manager, Benutzer in der Geschäftsleitung
ROLE_ADMIN	Administrator, Supporter der iTIXI Applikation

Zugriffe auf Freigaben für ROLE_DISPO sind auch ROLE_MANAGER, bzw. für die höheren Rollen möglich.

Verlangt eine Freigabe mindestens die Rolle ROLE_MANAGER, so haben tiefere Rollen wie ROLE_DISPO keinen Zugriff, höhere Rollen wie ROLE_ADMIN erhalten aber Zugriff.

```
role_hierarchy:
    ROLE_DISPO:
    ROLE_MANAGER:       ROLE_DISPO
    ROLE_ADMIN:         ROLE_MANAGER
```

Abbildung 42: ROLE Hierarchie in security.yml von Symfony

3.6.2 OAuth 2.0

Für die weitere Entwicklung der iTIXI Applikation wurde ein Symfony Bundle eingerichtet um einen OAuth 2.0 Server zu betreiben. Dies kann zukünftig gebraucht werden für API-Zugriffe von einer Buchhaltungssoftware oder eines Telefon-Systems, die keine Formular-basierte Anmeldung durchführen können. Weiter möchten wir hierbei nicht auf diese Bundle eingehen, Informationen erhält man über FOSOAuthServerBundle[11].

[11] FOSOAuthServerBundle: https://github.com/FriendsOfSymfony/FOSOAuthServerBundle

4 Deployment

Das Deployment ist nicht Teil der Aufgabenstellung, aber dennoch notwendig um einen Testbetrieb zu gewährleisten. Wie bei der kontinuierlichen Integration erwünscht, wird stets eine aktuelle Version auf einem Server ausgerollt. Auf unserem eigenen Buildserver *tixi.seiout.ch* wurde die entsprechende Konfiguration (Abbildung 43) an Technologien ausgeführt für den Betrieb von iTIXI, wie es in einer ausgerollten und produktiven Umgebung der Fall wäre.

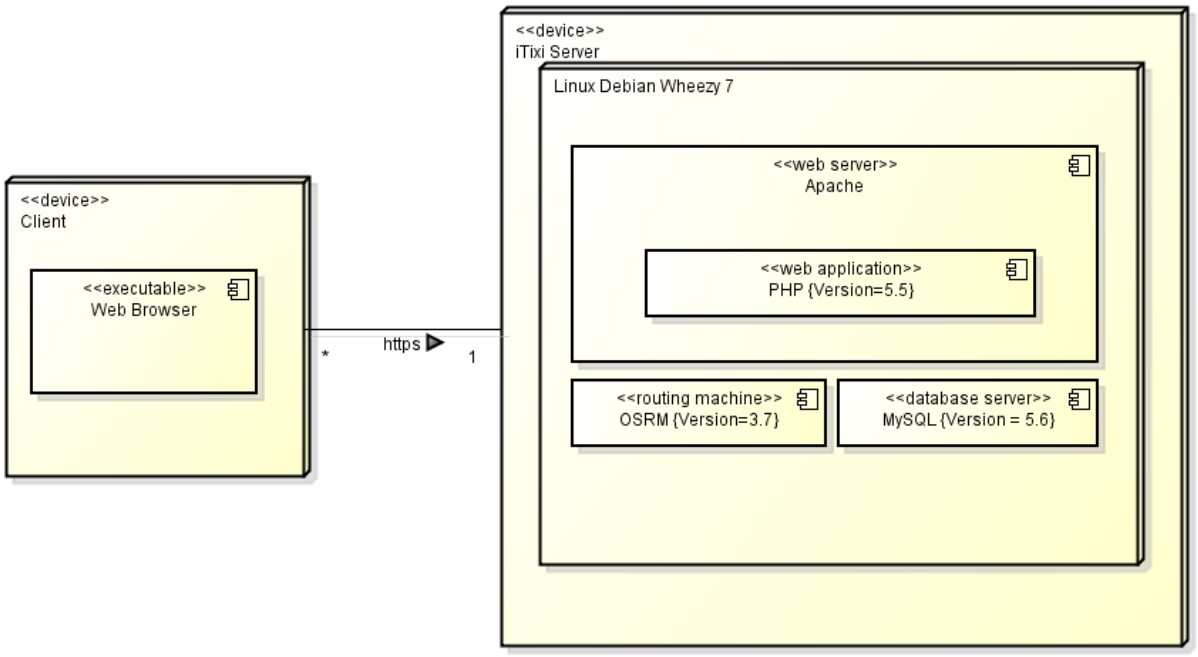


Abbildung 43: Deployment-Diagramm

Vorausgesetzt wird eine lauffähige LAMP Umgebung mit den Mindestversionen:

- PHP 5.5
- MySQL 5.6

Apache sollte über ein SSL-Zertifikat verfügen um eine HTTPS Verbindung zu ermöglichen, wie z.B. https://tixi.seiout.ch/tixi/

Speziell sind die Anforderungen an OSRM v3.7. Zusammen mit den anderen Technologien wurde dafür ein Dokument *Technologie-Deployment* erstellt, welches im Appendix (**Fehler! Verweisquelle konnte nicht gefunden werden.**) verfügbar ist. Dieses Dokument wurde auch dem Projektpartner zur Verfügung gestellt für die Auswahl eines geeigneten Hosting-Providers.

4.1 Build

4.1.1 Umgebungsparameter

Auf jeder Umgebung müssen Parameter für Symfony, Bundles und weitere Libraries definiert werden.

Diese werden in der Datei *app/config/parameter.yml* definiert:

```
parameters:
    database_driver: pdo_mysql
```

```
database_host: localhost
database_port: 3306
database_name: tixi
database_user: mysql_tixi
database_password: password
mailer_transport: smtp
mailer_host: 127.0.0.1
mailer_user: null
mailer_password: null
locale: de
secret: ThisTokenIsNotSoSecretChangeIt
time_zone: Europe/Zurich
wkhtmltopdf_path: /usr/bin/wkhtmltopdf
```

Dazu gehören Zugriffsinformationen für Database und SMTP Maildienst, die Zeitzone sowie speziell für die PDF-Erzeugung eine ausführbare Binary zu wkhtmltopdf.

Durch SwiftMail kann möglichst simpel der Maildienst von Google genutzt werden. Dazu müssen nur folgende Parameter gesetzt werden:

```
mailer_transport: gmail
mailer_host: null
mailer_user: myaccount@gmail.com
mailer_password: mypassword
```

4.1.2 Applikations Parameter

Einstellungen zur iTIXI Applikation werden über die Datei *app/config/tixi.yml* definiert:

```
parameters:
     tixi_parameter_app: iTIXI
     tixi_parameter_version: v2.0.3.2
     tixi_parameter_client: TIXIZUG
     tixi_parameter_client_page: http://www.tixizug.ch

     tixi_parameter_files_directory: %kernel.root_dir%/cache/files
     tixi_parameter_admin_mail: itixizug@gmail.com

     tixi_parameter_google_apikey: AIzy…w00uAcAv-…
     tixi_parameter_osrm_server: seiout.ch:8080
```

Dazu gehören Applikationsname, Version und Mandant die im Frontend angezeigt werden.

Mit tixi_parameter_files_directory wird ein Directory bestimmt für die Ablage erzeugter Dateien (z.B. PDF Rapporte). Eine Mailadresse zu einem Administrator sollte über tixi_parameter_admin_mail bestimmt werden für die Zustellung von Informationen/Fehlern zum System. Hinzu kommen der Google API Key für den Mandanten, sowie Adresse und Port eines OSRM-Server.

4.1.3 Symfony + Composer

Alle benötigten Daten für den Buildprozess werden vom Repository aus zur Verfügung gestellt.

Für Symfony werden einige Dependencies zu PHP benötigt, zudem werden einige Bundles genutzt. Diese können komfortabel über den Composer bezogen werden. Fehlen gewisse Angaben in der parameter.yml, wird zur manuellen Eingabe dieser Parameter bei der Ausführung vom Composer aufgeboten.

4.1.4 Doctrine + Database

Um eine frische Installation anzulegen, werden Tabellen für die Datenbank per Doctrine erzeugt. Alle nachfolgenden Befehle werden dabei im Symfony Root-Verzeichnis ausgeführt. Symfony bietet über *console.php* eine Verwendung von Commands an mit der zugehörigen Umgebung.

Befehl für Database Erzeugung:

```
php app/console doctrine:database:create
```

Bei einer lauffähigen Umgebung, die Updates erhält, muss nur das Datenbankschema aktualisiert werden:

```
php app/console doctrine:schema:update –force
```

Da Doctrine über keine Angaben zur Volltext-Suche von MySQL mit InnoDB verfügt, wird bei einer Neuanlegung der Datenbank nachträglich ein Volltext-Index erzeugt über einen von uns erstellten Symfony Command:

```
php app/console project:build-fulltext
```

Dieser fügt per ALTER TABLE einen Volltext-Index zu den benötigten Kolumnen in der Adress-Tabelle hinzu.

Für den weiteren Build wurden auf unserem Buildserver jeweils noch Testdaten aus einer .SQL Datei hinzugefügt.

4.2 Technologie-Stack

Folgende Technologien werden für den Betrieb von iTIXI auf einem Linux-Server aufgeführt:

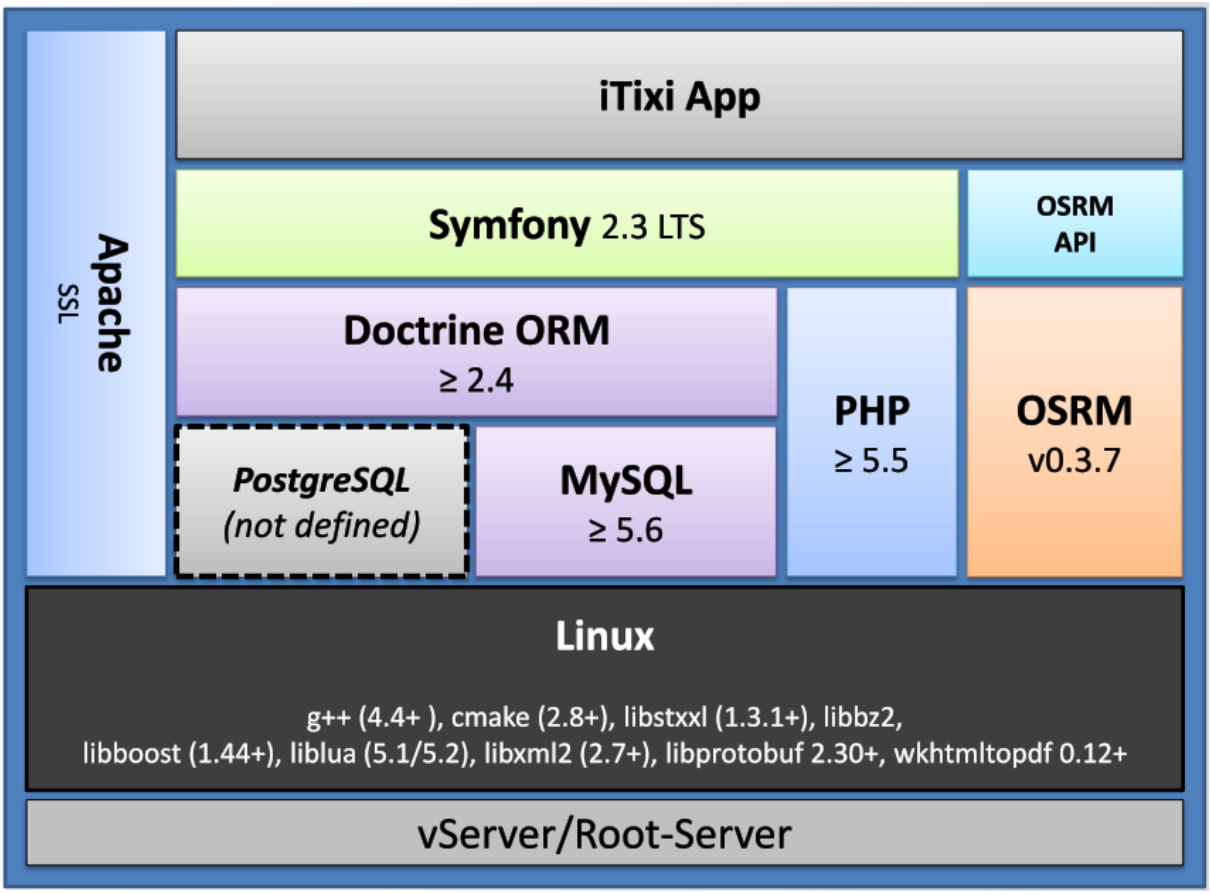


Abbildung 44: Technologie-Stack iTIXI

Dabei wäre auch PostgreSQL als Alternative denkbar und dank OR-Mapper Doctrine2 ohne Probleme austauschbar anstelle von MySQL. Nebst Parametern müsste nur die Erstellung und Verwendung der Volltextsuche unter Umständen angepasst werden.

4.2.1 OSRM Update

Da sich OSRM auf die Daten von OpenStreetMap stützt, sollten die OSM Dateien von Zeit zu Zeit aktualisiert und für OSRM neu vorberechnet werden. Auf einem Linux Server wäre dazu ein BASH-Skript geeignet, das z.B. jedes Wochenende ausgeführt wird:

```
./stop.sh      (aktuellen OSRM Prozess stoppen)

(vorhandene Datei löschen oder verschieben)
cd build && yes | rm switzerland-latest.osm.pbf

(aktuelle OSM herunterladen)
wget http://download.geofabrik.de/europe/switzerland-latest.osm.pbf

(vorhandene konvertierte OSM Dateien löschen und neu konvertieren)
yes | rm swiss.o5m && ./osmconvert switzerland-latest.osm.pbf -o=swiss.o5m

(konvertierte OSM Datei umwandeln)
yes | rm swiss.osm.pbf && ./osmconvert swiss.o5m -o=swiss.osm.pbf

(OSM Dateien für OSRM extrahieren und berechnen lassen)
./osrm-extract build/swiss.osm.pbf
./osrm-prepare build/swiss.osrm

./start.sh     (OSRM Prozess neu starten)
```

4.3 Commands

Um den Betrieb von iTIXI zu gewährleisten, werden einige Symfony Commands angeboten, die über einen Scheduler vom Betriebssystem aus angesteuert werden können. Diese sind für folgende Aufgaben gedacht:

4.3.1 Adresskoordinaten abrufen

Alle in der Datenbank vorhandenen Adressen werden auf die Vollständigkeit von Koordinaten und Nearest Points überprüft und allenfalls ergänzt über Google Maps API und OSRM.

Die könnte täglich am Abend terminiert werden. Soweit werden aber durch die Adressverwaltung alle neu erfassten Adressen immer mit den vollständigen Koordinaten erstellt. Genutzt wurde der Command vorallem bei der Daten-Migration, bzw. Integration von Testdaten, damit alle Adressobjekte über eine Koordinate verfügten.

Command: `project:query-address-coordinates %limit%`

Argument: %limit% bestimmt die maximale Anzahl abzurufender Addressen an. Default: 1000

Häufigkeit: täglich

Sendet ein Informationsmail mit der Menge an aktualisierten Adressen an den Administrator.

4.3.2 Routen aktualisieren

Da zu einem Fahrauftrag eine Route abgespeichert wird, kann es sein, dass diese mit der Zeit nicht mehr so aktuell sind. Über den Command werden Routen aus der Datenbank geladen, die älter als einen Monat sind. Diese werden anschliessend über OSRM erneut abgefragt und wenn nötig aktualisiert.

Command: `project:route-update`

Argument: keine

Häufigkeit: monatlich

Sendet ein Informationsmail mit der Menge aktualisierter Routen an den Administrator.

4.3.3 Fahrtoptimierungen vorausberechnen (angedacht)

Denkbar ist, alle Schichten und Tage über die Fahrtoptimierung bereits zu berechnen, damit in der Tagesplanung Fahraufträge bereits in einer möglichen Form aufgelistet sind und es für die Disponenten vereinfacht, einen zukünftigen Tag zu untersuchen bei der Erfassung weiterer Fahraufträge.

Command: `project:ride-optimization %day%`

Argument: %day% gibt den Tag in Form von dd.mm.YYYY an, der optimiert werden soll
Default: 01.07.2024 (so ergeben aus den Testdaten)

Häufigkeit: täglich