biking2, Architecture and API

Michael Simons

Table of Contents

Disclaimer

I provide absolutely **no guarantee**, neither for the accuracy of this documentation nor for any property or feature of the software described here.

Do not use this software in critical situations or projects.

1. Introduction and Goals

biking.michael-simons.eu (http://biking.michael-simons.eu) is a project for

- tracking my bike activities
- evaluating technology
- learn stuff
- showcasing my skills

For my information see this blog post (http://info.michael-simons.eu/2014/02/20/developing-a-web-application-with-spring-boot-angularjs-and-java-8/) as a starting point.

I'd like to thank all the people working at the Spring Eco System for doing their amazing work. Also a big thank you to Dr. Gernot Starke and Dr. Peter Hruschka for their inspiring workshop "Mastering Software Architecture" in Munich, December 2015. I took not only a nice *CPSA-F* certificate home, but really valuable and practical patterns for improving "my" software.

The blog post above had song lyrics from Nick Cave & The Bad Seeds as intro (from "Push The Sky Away") and for the architecture i have this:

> *Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.*
>
> — *Edsger W. Dijkstra*
> *On the nature of Computing Science*

1.1. Requirements Overview

What is biking2?

The main purpose of *biking2* is keeping track of bicycles and their milages as well as converting *Garmin Training Center XML (tcx)* (https://en.wikipedia.org/wiki/Training_Center_XML) files to standard *GPS Exchange Format* (GPX) (https://en.wikipedia.org/wiki/GPS_Exchange_Format) files and storing them in an accessible way.

In addition *biking2* is used to study and evaluate technology, patterns and frameworks. The functional requirements are simple enough to leave enough room for concentration on quality goals.

Main features

- Store bikes and their milages
- Convert tcx files to GPX files and provide them in a library of tracks
- Visualize those tracks on a map and provide a way to embed them in other webpages
- Visualize biking activities with images
- Optional, near real time, tracking of a biker

The application must only handle exactly one user with write permissions.

Most bike aficionados have problems understanding the question "why more than one bike?", the system should be able to keep track of everything between 2 and 10 bikes for one user, storing 1 total milage per bike and month. All milages per month, year and other metrics should be derived from this running total, so that the user only need to look at his odometer and enter the value.

The application should store an "unlimited" number of tracks.

The images should be collected from Daily Fratze (https://dailyfratze.de), the source are all images that are tagged with "Radtour". In addition the user should be able to provide an "unlimited" number of "gallery pictures" together with a date and a short description.

1.2. Quality Goals

Table 1. Quality Goals

Nr.	Quality	Motivation
1	Understandability	The functional requirements are simple enough to allow a simple, understandable solution that allows focus on learning about new Java 8 features, Spring Boot and AngularJS.
2	Efficiency	Collecting milage data should be a no brainer: Reading the milage from an odometer and entering it.
3	Interoperability	The application should provide a simple API that allows access to new clients.
4	Attractiveness	Collected milages should be presented in easy to grasp charts.
5	Testability	The architecture should allow easy testing of all main building blocks.

1.3. Stakeholders

The following lists contains the most important personas for this application

Table 2. Stakeholders

Role/Name	Goal/Boundaries
Developers	Developers who want to learn about developing modern applications with Spring Boot and various frontends, preferable using *Java 8* in the backend.
Bikers	Bike aficionados that are looking for a non-excel, self-hosted solution to keep track of their bikes and milages.
Software Architects	Looking for an *arc42* example; want to get ideas for their daily work.
Michael Simons	Improving his skills; wants to blog about Spring Boot; looking for a topic he can eventually hold a talk about; needed a project to try out new *Java 8* features.

2. Architecture Constraints

The few constraints on this project are reflected in the final solution. This section shows them and if applicable, their motivation.

2.1. Technical Constraints

Table 3. List of Technical Constraints

	Constraint	Background and / or motivation
Software and programming constraints		
TC1	Implementation in Java	The application should be part of a Java 8 and Spring Boot show case. The interface (i.e. the api) should be language and framework agnostic, however. It should be possible that clients can be implemented using various frameworks and languages..
TC2	Third party software must be available under an compatible open source license and installable via a package manager	The interested developer or architect should be able to check out the sources, compile and run the application without problems compiling or installing dependencies. All external dependencies should be available via the package manager of the operation system or at least through an installer.
Operating System Constraints		
TC3	OS independent development	The application should be compilable on all 3 mayor operation systems (Mac OS X, Linux and Windows)
TC4	Deployable to a Linux server	The application should be deployable through standard means on a Linux based server
Hardware Constraints		
TC5	Memory friendly	Memory can be limited (due to availability on a shared host or deployment to cloud based host). If deployed to a cloud based solution, every megabyte of memory costs.

2.2. Organizational Constraints

Table 4. List of Organizational Constraints

	Constraint	Background and / or motivation
OC1	Team	Michael Simons
OC2	Time schedule	Start in early 2014 with Spring Boot beta based prototypes running on Java 8 early access builds, first "release" version March 2014 together with the initial release of Java 8. Upgrade to a final Spring Boot release when they are available.

	Constraint	Background and / or motivation
OC3	IDE independent project setup	No need to continue the editor and IDE wars. The project must be compilable on the command line via standard build tools. Due to *OC2* there is only one IDE supporting Java 8 features out of the box: *NetBeans 8* beta and release candidates.
OC4	Configuration and version control / management	Private git repository with a complete commit history and a public master branch pushed to GitHub and linked a project blog.
OC5	Testing	Use JUnit to prove functional correctness and integration tests and JaCoCo to ensure a high test coverage (at least 90%).
OC6	Published under an Open Source license	The source, including documentation, should be published as Open Source under the Apache 2 License.

2.3. Conventions

Table 5. List of Conventions

	Conventions	Background and / or motivation
C1	Architecture documentation	Structure based on the english arc42-Template in version 6.5
C2	Coding conventions	The project uses the Code Conventions for the Java TM Programming Language (http://www.oracle.com/technetwork/java/codeconvtoc-136057.html). The conventions are enforced through Checkstyle.
C3	Language	English. The project and the corresponding blog targets an international audience, so English should be used throughout the whole project.
C4	Naming conventions	There are some naming conventions listed below that are checked and enforced with jQAssistant (https://jqassistant.org)

The following naming conventions are used throughout the project:

All Java types must be located in packages that start with `ac.simons`*.*

CYPHER
```
MATCH
  (project:Maven:Project)-[:CREATES]->(:Artifact)-[:CONTAINS]->(type:Type)
WHERE
  NOT type.fqn starts with 'ac.simons'
RETURN
  project as Project, collect(type) as TypeWithWrongName
```

All Java types must be located in packages that contains the artifactId of the Maven Project.

CYPHER
```
MATCH
  (project:Maven:Project)-[:CREATES]->(:Artifact)-[:CONTAINS]->(type:Type)
WHERE
  NOT type.fqn contains project.artifactId
RETURN
  project as Project, collect(type) as TypeWithWrongName
```

All Java types annotated as `@Entity` *must have a name that ends with the suffix* `Entity`*.*

```cypher
MATCH
  (t:Jpa:Entity)
WHERE
  NOT t.name ends with "Entity"
RETURN
  t as Entity
```

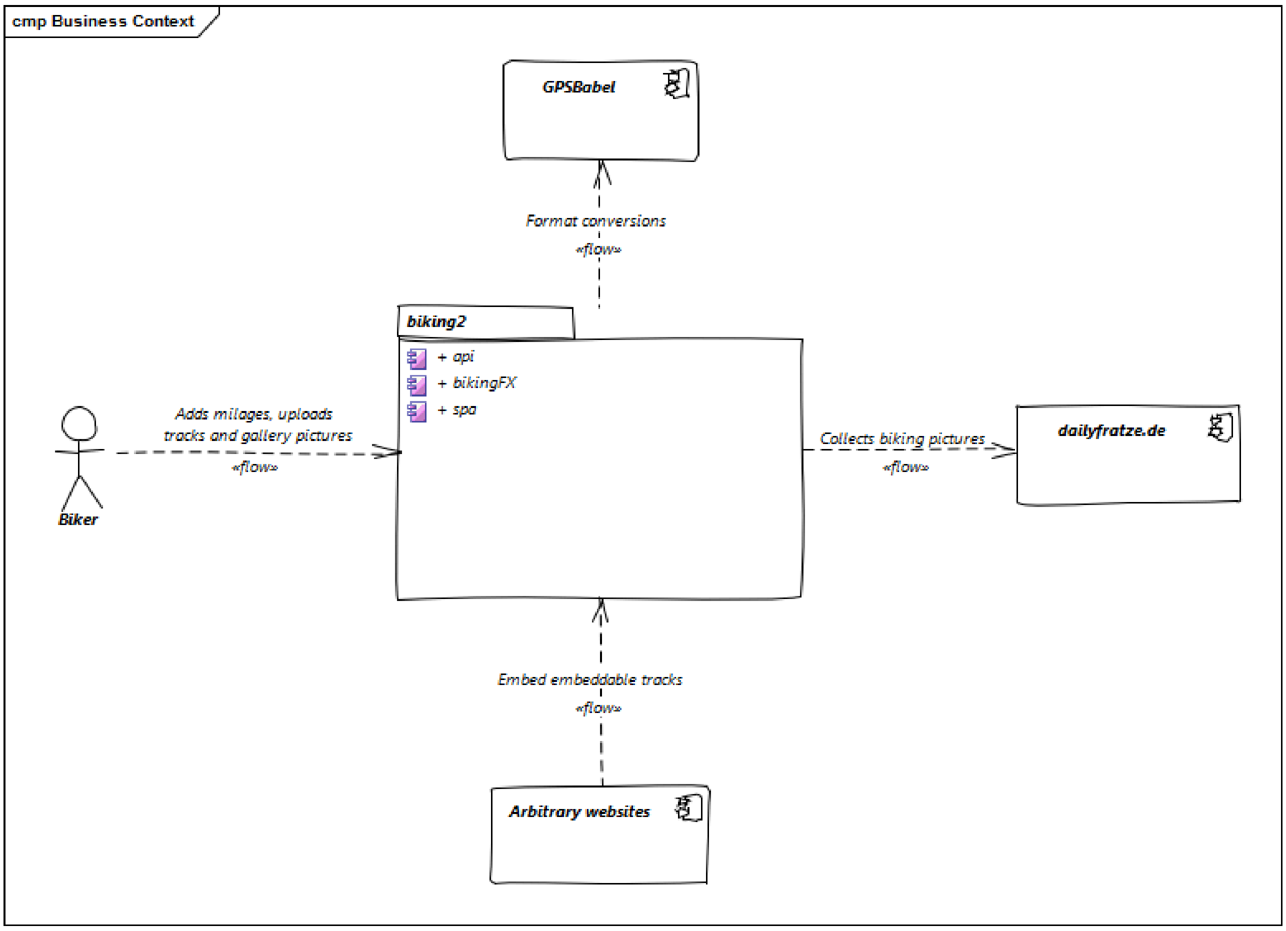
All Java types that represent a repository must have a name that ends with the suffix `Repository`*.*

```cypher
MATCH
  (t:Repository)
WHERE t:Java
  AND NOT t.name ends with "Repository"
RETURN
  t as Repository
```

3. System Scope and Context

This chapter describes the environment and context of *biking2*: Who uses the system and on which other system does *biking2* depend.

3.1. Business Context



Biker

A passionate biker uses *biking2* to manage his bikes, milages, tracks and also visual memories (aka images) taken on tours etc. He also wants to embed his tracks as interactive maps on other websites.

Daily Fratze

Daily Fratze (https://dailyfratze.de) provides a list of images tagged with certain topics. *biking2* should collect all images for a given user tagged with "Theme/Radtour".

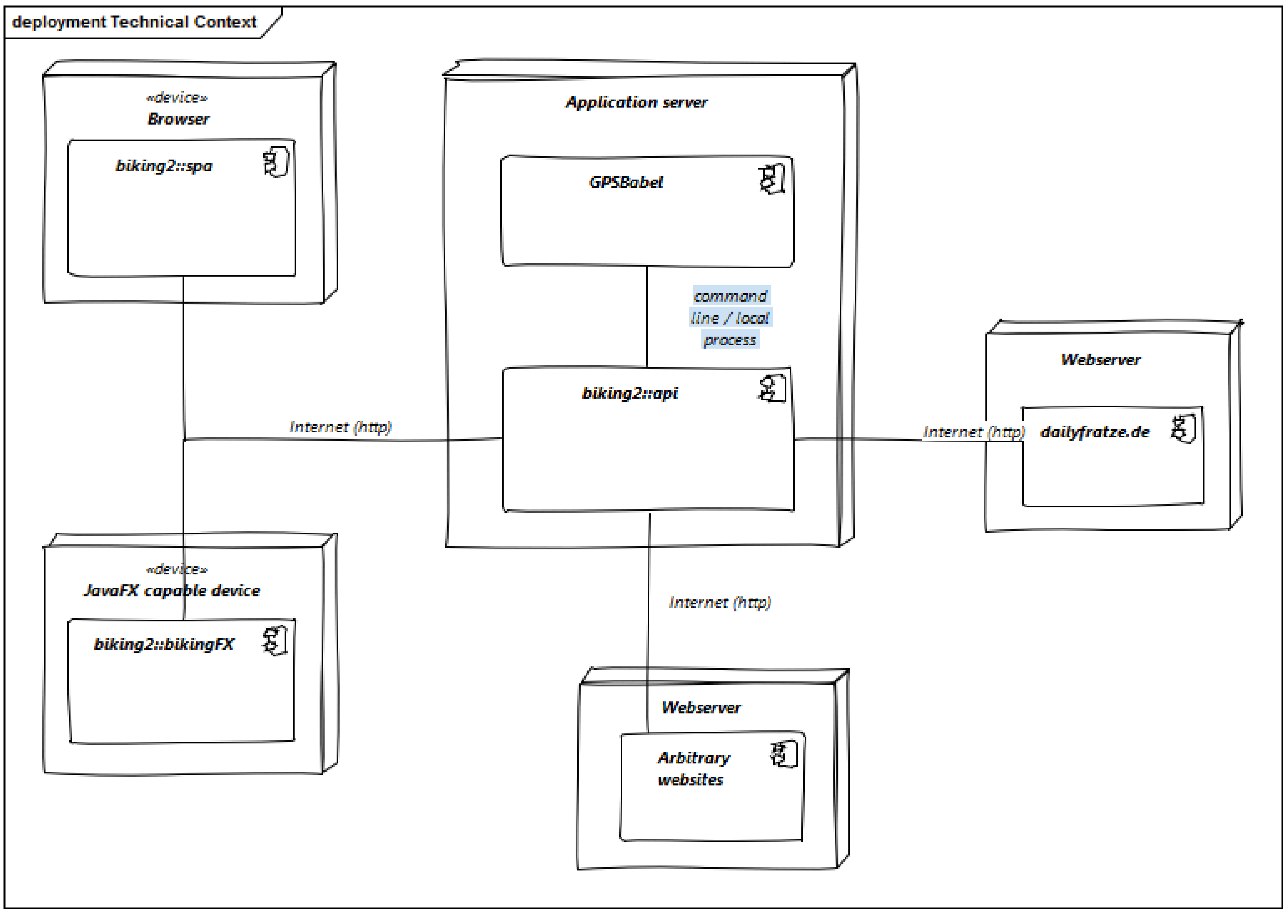
GPSBabel

GPSBabel is a command line utility for manipulating GPS related files in various ways. *biking2* uses it to convert TCX into GPX files. The heaving lifting is done by GPSBabel and the resulting file will be managed by *biking2*.

Arbitrary websites

The user may want to embed (or brag with) tracks on arbitrary websites. He only wants to paste a link to a track on a website that supports embedded content to embed a map with the given track.

3.2. Technical Context



biking2 is broken into 2 main components:

Backend (biking2::api)

The api runs on a supported application server, using either an embedded container or an external container. It communicates via operating system processes with GPSBabel on the same server.

The connection to *Daily Fratze* is http based RSS-feed. The feed is paginated and provides **all** images with a given tag but older images may not be available any more when the owner decided to add a digital expiry.

Furthermore *biking2* provides an oEmbed (http://oembed.com) interface for all tracks stored in the system. Arbitrary websites supporting that protocol can request embeddable content over http knowing only a link to the track without working on the track or map apis themselves.

Frontend (biking2::spa and biking2::bikingFX)

The frontend is implemented with two different components, the biking2::spa (Single Page Application) is part of this package. The spa runs in any modern web browser and communicates via http with the api.

Business interface	**channel**
Format conversions	System processes, command line interface
Collection of biking pictures	RSS feed over Internet (http)
Embeddable content	oEmbed format over Internet (http)
API for business functions	Internet (http)

4. Solution Strategy

At the core of *biking2* is a simple yet powerful domain model based on a few entities of which a "Bike" and it’s "Milage" are the most important.

Although data centric, the application resigns from using too much SQL for creating reports, summaries and such but tries to achieve that with new Java 8 features around streams, lambdas and map/reduce functions.

Building the application with Spring Boot is an obvious choice as one of the main quality goals is learning about it. But furthermore using Spring Boot as a "framework" for Spring Framework allows concentration on the business logic. On the one hand there is no need to understand a complex XML configuration and on the other hand all building blocks are still put together using dependency injection.

Regarding dependency injection and testability: All injection should be done via constructor injection, setter injection is only allowed when there’s no other technical way. This way units under tests can only be correctly instantiated. Otherwise one tends to forget collaborators or even worse: 20 injected attributes may not hurt, but a constructor with 20 parameters will. This hopefully prevents centralized "god classes" that control pretty much every other aspect of the application.

Spring Boot applications can be packaged as single, "fat jar" files. Since Spring Boot 1.3 those files contain a startup script and can be directly linked to */etc/init.d* on a standard Linux systems which serves [TC4].

Interoperability will be achieved by using JSON over simple http protocol for the main API. Security is not the main focus of this application. It should be secure enough to prevent others from tempering with the data, confidentiality is not a main concern (read: passwords can be transmitted in plain over http).

The internal single page application shall be implemented using AngularJS. The deployable artifact will contain this application so there is no need for hosting a second webserver for the static files.

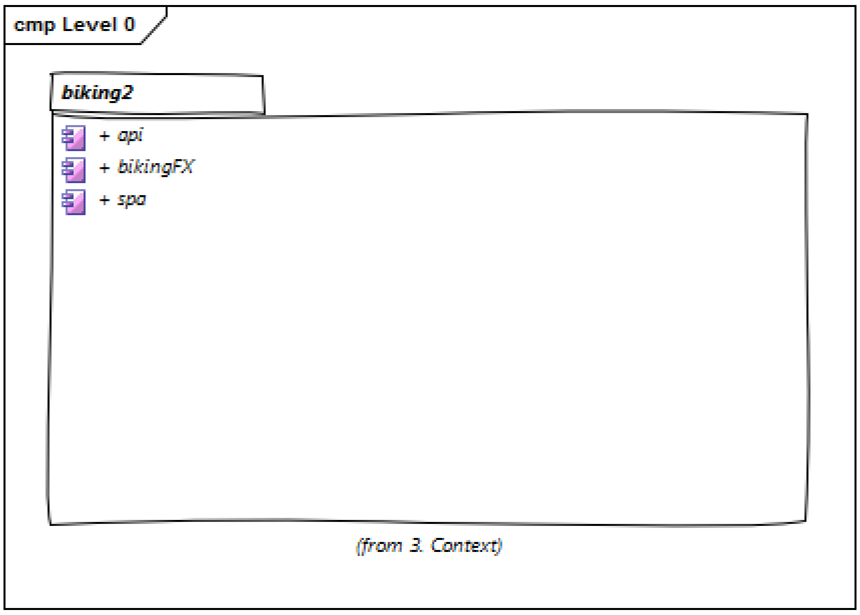
For real time tracking the MQTT protocol will be used which is part of Apache ActiveMQ, supported out of the box by Spring Messaging.

Graphing should not be implemented here but instead the Highcharts (http://www.highcharts.com) library should be used. The configuration for all charts should be computed server side.

> The original installment of this project used Java 8 streams and api heavily to compute statistics (everything under https://biking.michael-simons.eu/milages). Back in 2014 and 2015, when Java 8 was new, it helped a lot to learn that API. Nearly 5 years later, thinking about my history with database and looking at a tons of talk about SQL I gave, I decided it was time to go back to my roots. All the charts are now created with a dedicated statistics service, that is based on jOOQ and typesafe SQL.

5. Building Block View

The application packaged as *biking2.jar* contains two (the api and the spa) of three main parts, as shown in the Business Context:

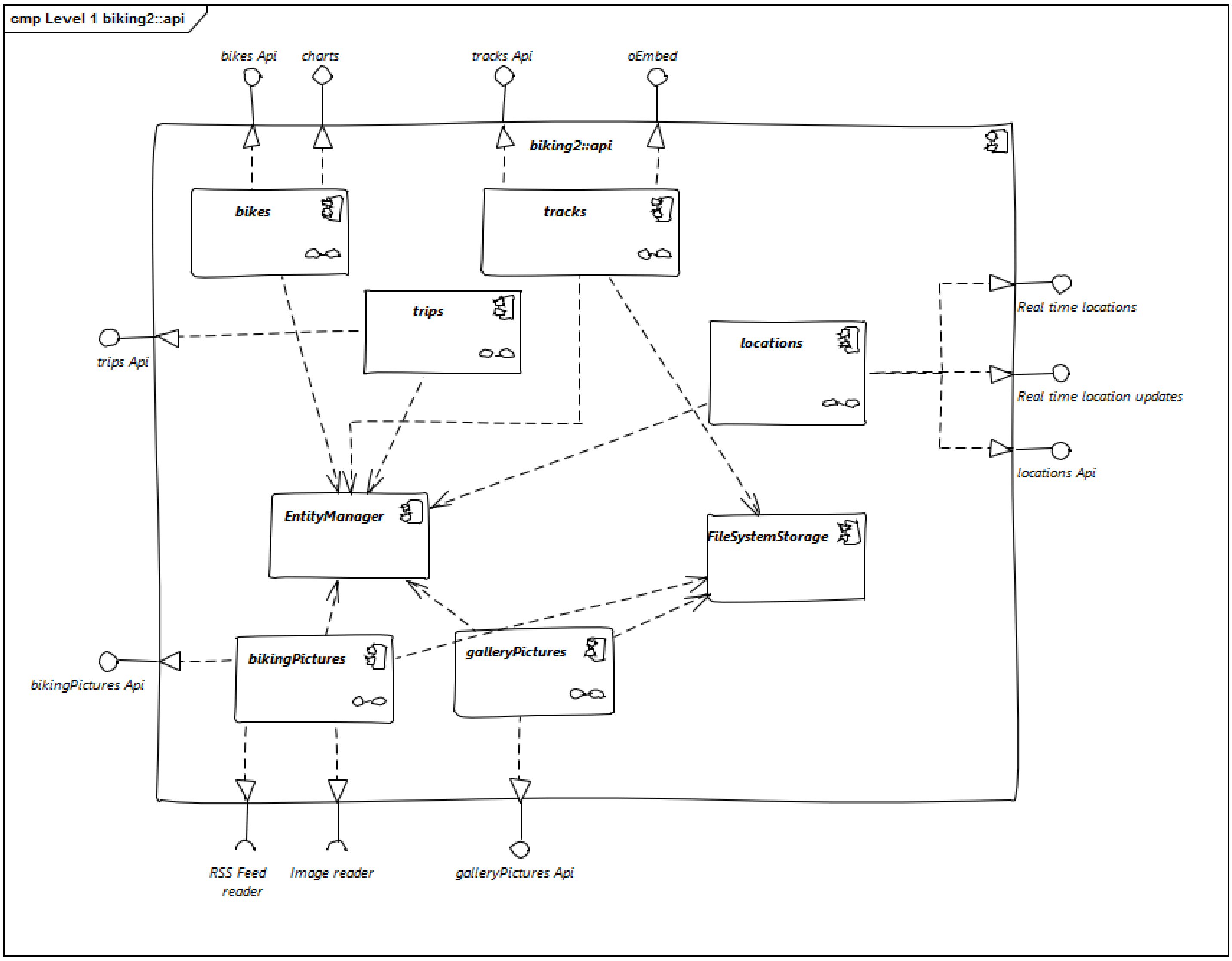


From those two we have a closer look at the api only. For details regarding the structure of an AngularJS 1.2.x application, have a look at their developers guide (https://code.angularjs.org/1.2.28/docs/guide).

To comply with the Java coding style guidelines, the modules "bikingPictures" and "galleryPictures" reside in the Java packages "bikingpictures" and "gallerypictures".

5.1. Whitebox biking2::api

The following diagram shows the main building blocks of the system and their interdependencies:



I used *functional decomposition* to separate responsibilities. The single parts of the api are all encapsulated in their own components, represented as Java packages.

All components depend on a standard JPA EntityManager and some on local file storage. I won't go into detail for those blackboxes.

Contained blackboxes

Table 6. biking2::api building blocks

bikes	Managing bikes, adding monthly milages, computing statistics and generating charts.
tracks	Uploading tracks (TCX files), converting to GPX, providing an oEmbed interface.
trips	Managing assorted trips.
locations	MQTT and STOMP interface for creating new locations and providing them in real time on websockets via stomp.
bikingPictures	Reading biking pictures from an RSS feed provided by *Daily Fratze* and providing an API to them.
galleryPictures	Uploading and managing arbitrary pictures

statistics	Provides an API for statistics

All the blackboxes above should correspond to Java packages. Those packages should have no dependencies to other packages outside themselves but for the support or shared package:

Top level packages should conform to the main building blocks.

```
CYPHER
MATCH (db:Package:Database)
WITH db
MATCH (a:Main:Artifact)
MATCH (a) -[:CONTAINS]-> (p1:Package) -[:DEPENDS_ON]-> (p2:Package) <-[:CONTAINS]- (a)
WHERE not p1:Config
  and not (p1) -[:CONTAINS]-> (p2)
  and not p2:Support
  and p2 <> db
  and not (db) - [:CONTAINS*] -> (p2)
RETURN p1, p2
```

Additional configuration apart from properties lives in the `config` package:

Returns the config package.

```
CYPHER
MATCH (p:Package)
WHERE p.fqn in ['ac.simons.biking2.config']
SET p:Package:Config
return p
```

The `support` package contains technical infrastructure classes while the `shared` package contains application wide messages. Those packages are used as *supporting packages*:

Returns support and shared packages.

```
CYPHER
MATCH (p:Package)
WHERE p.fqn in ['ac.simons.biking2.shared', 'ac.simons.biking2.support']
SET p:Package:Support
return p
```

The `database` package contains generated code for accessing our database via jOOQ:

Returns the package containing generated jOOQ code

```
CYPHER
MATCH (p:Package)
WHERE p.fqn in ['ac.simons.biking2.db']
SET p:Package:Database
return p
```

Interfaces

Table 7. biking2::Interfaces

Interface	Description
bikes Api	REST api containing methods for reading, adding and decommissioning bikes and for adding milages to single bikes.
charts	Methods for retrieving statistics as fully setup chart definitions.
tracks Api	REST api for uploading and reading TCX files.

Interface	Description
trips Api	REST api for adding new trips.
oEmbed	HTTP based oEmbed interface, generating URLs with embeddable content.
Real time locations	WebSocket / STOMP based interface on which new locations are published.
Real time location updates	MQTT interface to which MQTT compatible systems like OwnTracks (http://owntracks.org) can offer location updates.
RSS feed reader	Needs an *Daily Fratze* OAuth token for accessing a RSS feed containing biking pictures which are than grabbed from *Daily Fratze*.
galleryPictures Api	REST api for uploading and reading arbitrary image files (pictures related to biking).

5.1.1. bikes (Blackbox)

Intent/Responsibility

`bikes` provides the external API for reading, creating and manipulating bikes and their milages as well as computing statistics and generating charts.

Interfaces

Interface	Description
REST interface `/api/bikes/*`	Contains all methods for manipulating bikes and their milages.
REST interface `/api/charts/*`	Contains all methods for generating charts.

Files

The `bikes` module and all of its dependencies are contained inside the Java package `ac.simons.biking2.bikes`.

5.1.2. tracks (Blackbox)

Intent/Responsibility

`tracks` manages file uploads (TCX files), converts them to GPX files and computes their surrounding rectangle (envelope) using GPSBabel. It also provides the oEmbed interface that resolves URLS to embeddable tracks.

Interfaces

Interface	Description
REST interface `/api/tracks/*`	Contains all methods for manipulating tracks.
`/api/oembed`	Resolve track URLs to embeddable tracks (content).
`/tracks/*`	Embeddable track content.

Files

The `tracks` module and all of its dependencies are contained inside the Java package `ac.simons.biking2.tracks`.

5.1.3. trips (Blackbox)

Intent/Responsibility

`trips` manages distances that have been covered on single days without relationships to bikes.

Interfaces

Interface	Description
REST interface `/api/trips/*`	Contains all methods for manipulating trips.

Files

The `trips` module and all of its dependencies are contained inside the Java package `ac.simons.biking2.trips`.

5.1.4. locations (Blackbox)

Intent/Responsibility

`locations` stores locations with timestamps in near realtime and provides access to locations for the last 30 minutes.

Interfaces

Interface	Description
REST interface `/api/locations/*`	For retrieving all locations in the last 30 minutes.
WebSocket / STOMP topic `/topic/currentLocation`	Interface for getting notifcation on new locations.
MQTT interface	Listens for new locations coming in via MQTT in OwnTracks format (http://owntracks.org/booklet/tech/json/).

Files

The `locations` module and all of its dependencies are contained inside the Java package `ac.simons.biking2.tracker`. The module is configured through `ac.simons.biking2.config.TrackerConfig`.

5.1.5. bikingPictures (Blackbox)

Intent/Responsibility

`bikingPictures` is used for regularly checking a RSS feed from Daily Fratze collecting new images and storing them locally. It also provides an API for getting all collected images.

Interfaces

Interface	Description
RSS Feed reader	Provides access to the *Daily Fratze* RSS Feed.

Interface	Description
Image reader	Provides access to images hosted on *Daily Fratze*.
REST interface `/api/bikingPictures/*`	Contains all methods for accessing biking pictures.

Files

The `bikingPictures` module and all of its dependencies are contained inside the Java package `ac.simons.biking2.bikingpictures`.

5.1.6. galleryPictures (Blackbox)

Intent/Responsibility

`galleryPictures` manages file uploads (images). It stores them locally and provides an RSS interface for getting metadata and image data.

Interfaces

Interface	Description
REST interface `/api/galleryPictures/*`	Contains all methods for adding and reading arbitrary pictures.

5.1.7. statistics (Blackbox)

Intent/Responsibility

Module for computing statistics, generating charts and providing version and summary information.

Interfaces

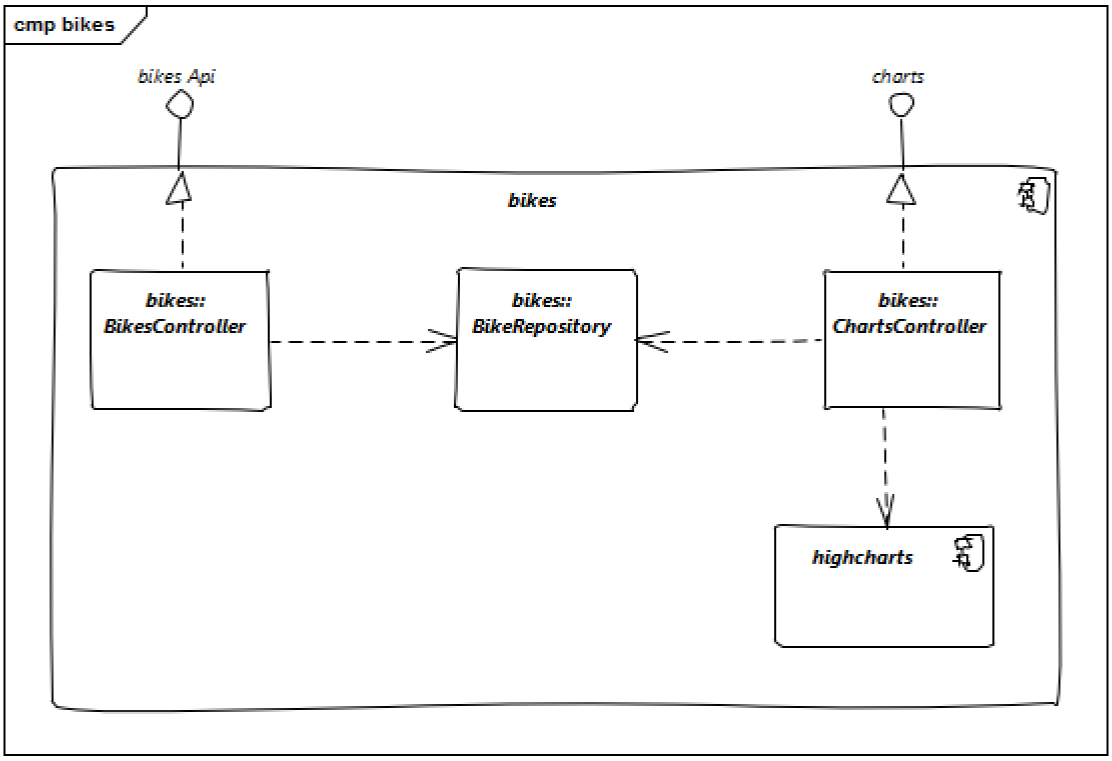
Interface	Description
REST interface `/api/charts/*`	Contains all methods for generating Highcharts Charts (https://www.highcharts.com/).
REST interface `/api/summary/*`	Aggregated values.

Files

The `statistics` module and all of its dependencies are contained inside the Java package `ac.simons.biking2.statistics`.

5.2. Building Blocks - Level 2

5.2.1. bikes (Whitebox)

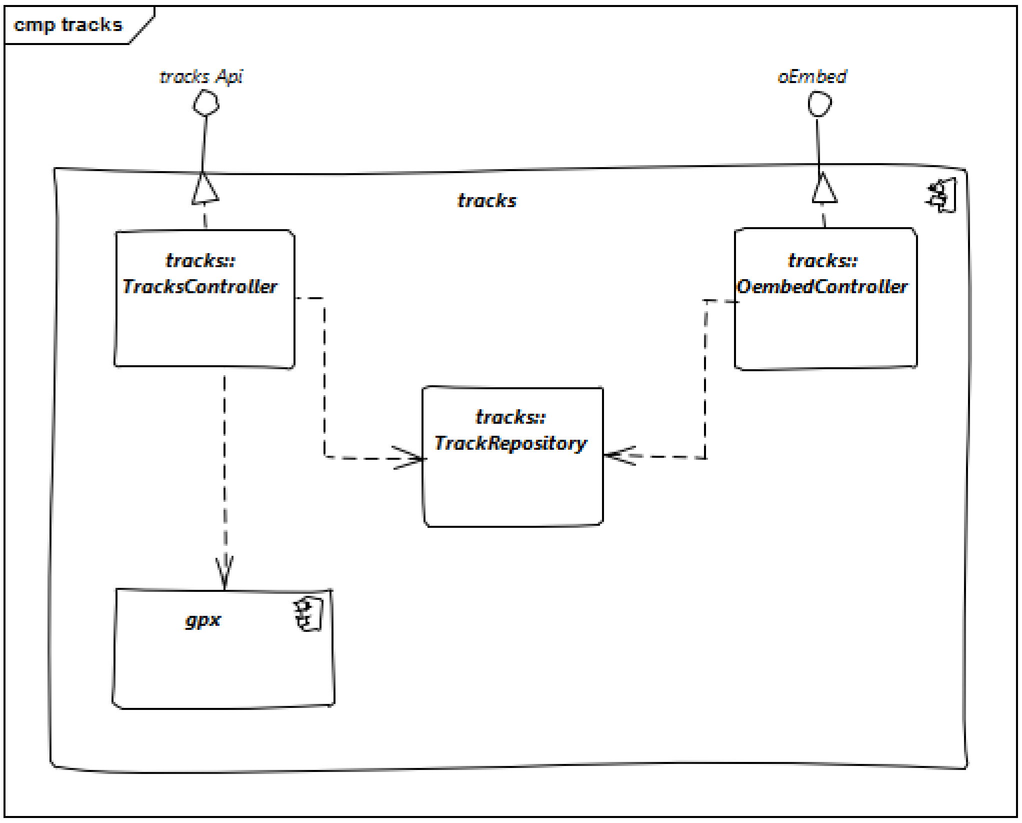


The `BikeRepository` is a Spring Data JPA based repository for `BikeEntities`. The `BikeController` and the `ChartsController` access it to retrieve and store instances of `BikeEntity` and provide external interfaces.

Contained blackboxes

highcharts	Contains logic for generating configurations and definitions for Highcharts (http://www.highcharts.com) on the server side.

5.2.2. tracks (Whitebox)

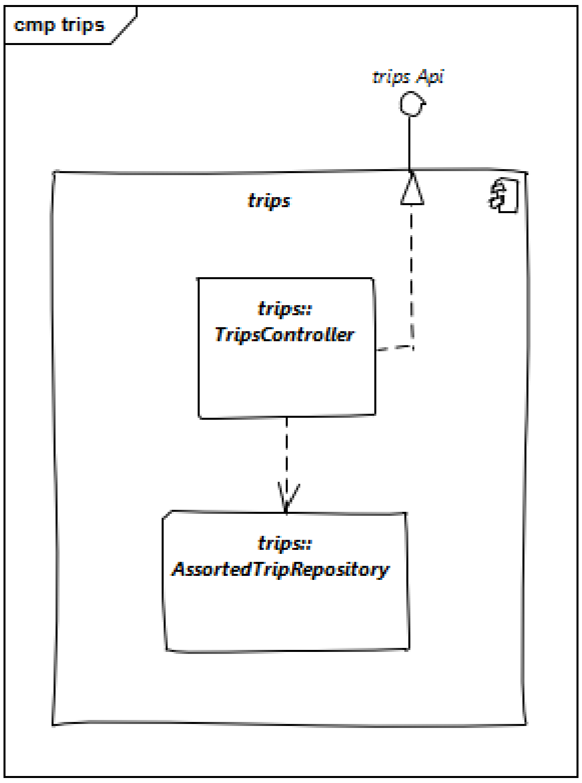


The `TrackRepository` is a Spring Data JPA based repository for `TrackEntities`. The `TracksController` and the `OembedController` access it to retrieve and store instances of `TrackEntity` and provide external interfaces.

Contained blackboxes

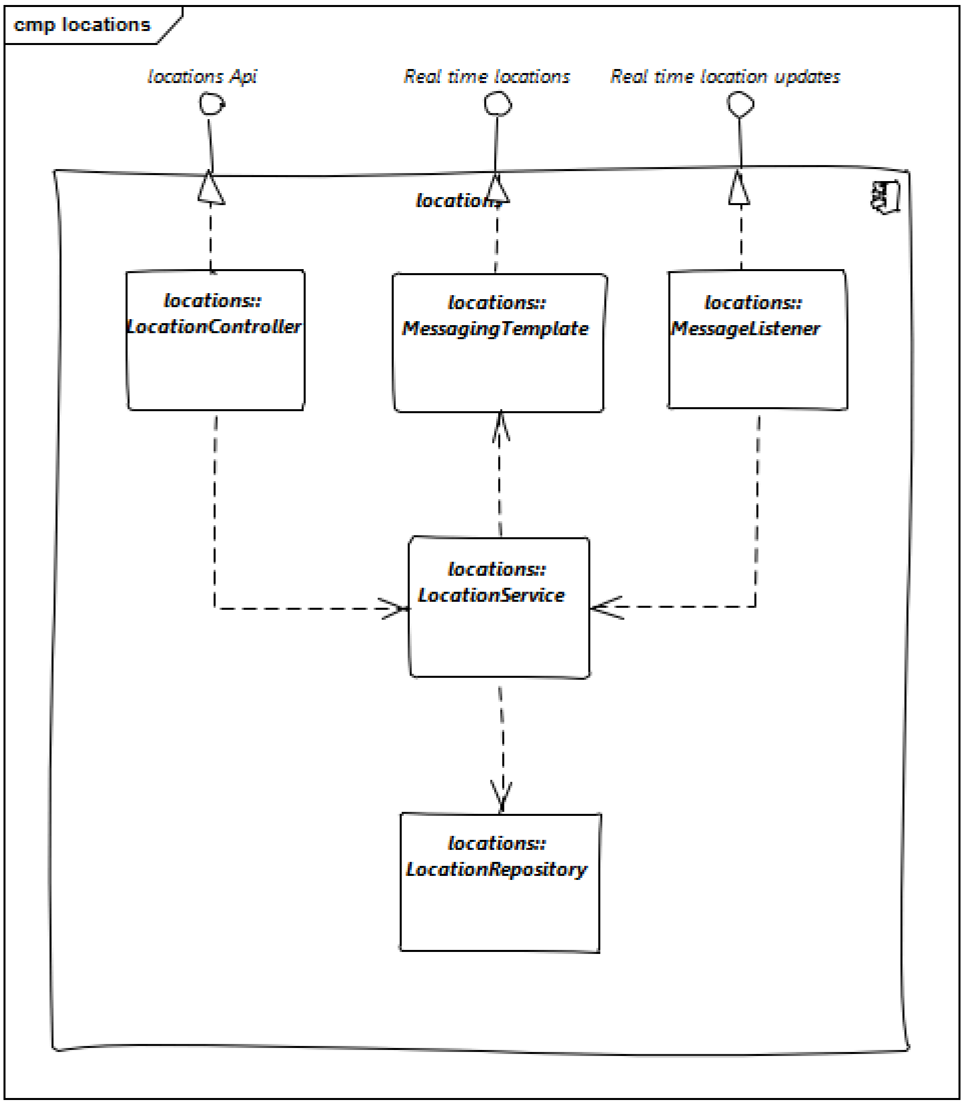
gpx	Generated *JAXB* classes for parsing GPX files. Used by the `TracksController` to retrieve the surrounding rectangle (envelope) for new tracks.

5.2.3. trips (Whitebox)



The `AssortedTripRepository` is a Spring Data JPA based repository for `AssortedTripEntities`. The `TripsController` accesses it to retrieve and store instances of `TrackEntity` and provide external interfaces.

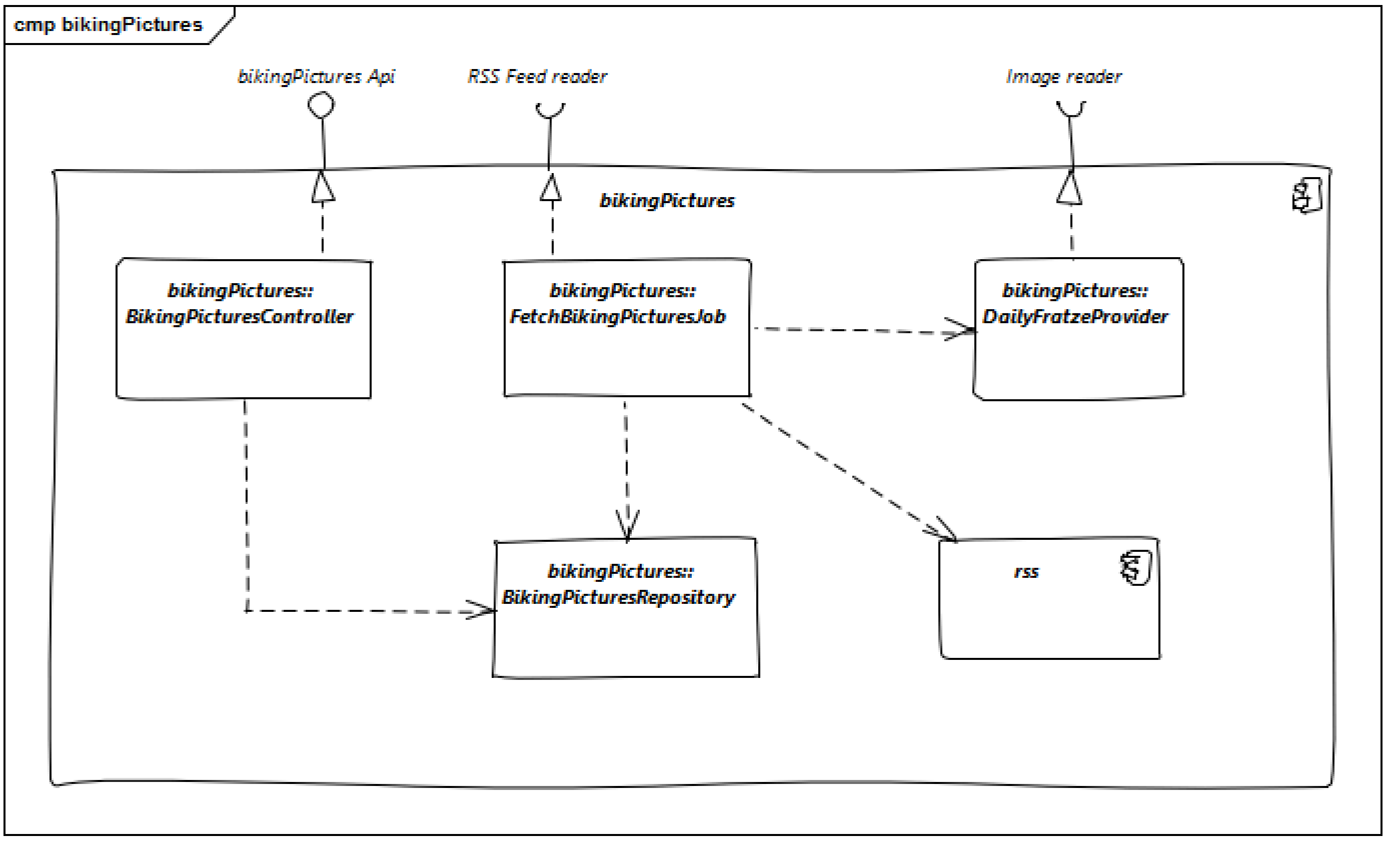
5.2.4. locations (Whitebox)


cmp locations
locations Api
Real time locations
Real time location updates
locations
locations:: LocationController
locations:: MessagingTemplate
locations:: MessageListener
locations:: LocationService
locations:: LocationRepository

Locations are stored and read via a Spring Data JPA based repository named `LocationRepository`. This repository is only accessed through the `LocationService`. The `LocationService` provides real time updates for connected clients through a `SimpMessagingTemplate` and the `LocationController` uses the service to provide access to all locations created within the last 30 minutes.

New locations are created by the service either through a REST interface in form of the `LocationController` or via a `MessageListener` on a MQTT channel.

5.2.5. bikingPictures (Whitebox)

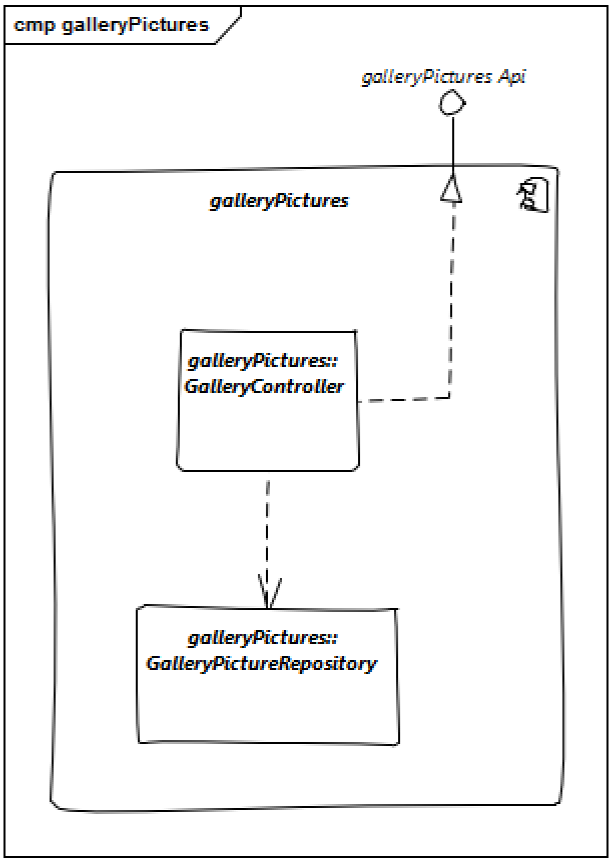


A Spring Data JPA repository named `BikingPicturesRepository` is used for all access to `BikingPictureEntities`, the external REST api for reading pictures is implemented with `BikingPicturesController`. The RSS feed is read from `FetchBikingPicturesJob` by using a JAXBContext "rss". The URLs to the image files which may are protected by various means are provided to the job via a `DailyFratzeProvider`.

Contained blackboxes

rss	Generated *JAXB* classes for parsing RSS feeds. Used by the `FetchBikingPicturesJob` to read the contents of an RSS feed.

5.2.6. galleryPictures (Whitebox)



The `GalleryPictureRepository` is a Spring Data JPA based repository for `GalleryPictureEntities`. The `GalleryController` accesses it to retrieve and store instances of `GalleryPictureEntity` and provide external interfaces.

5.2.7. statistics (Whitebox)

The `StatisticService` is a database centric service that uses an instance of jOOQ's `DSLContext` for creating SQL. That SQL is not generated SQL, but handcrafted. jOOQ is used only to do this in a types safe way.

During build, jOOQ reads a temporary database, created from SQL based migration scripts, and provides several Java classes reassembling a schema. Those classes along with the DSL are used to write SQL.

The statements make heavily use of analytic functions. This is necessary, as the original decision for storing milages has been not storing the amount biked each month, but the accumulated milage on a bike. Therefore, the monthly values need to be computed.

This is done by queries like the following

Query that provides the statistics for the current year

SQL

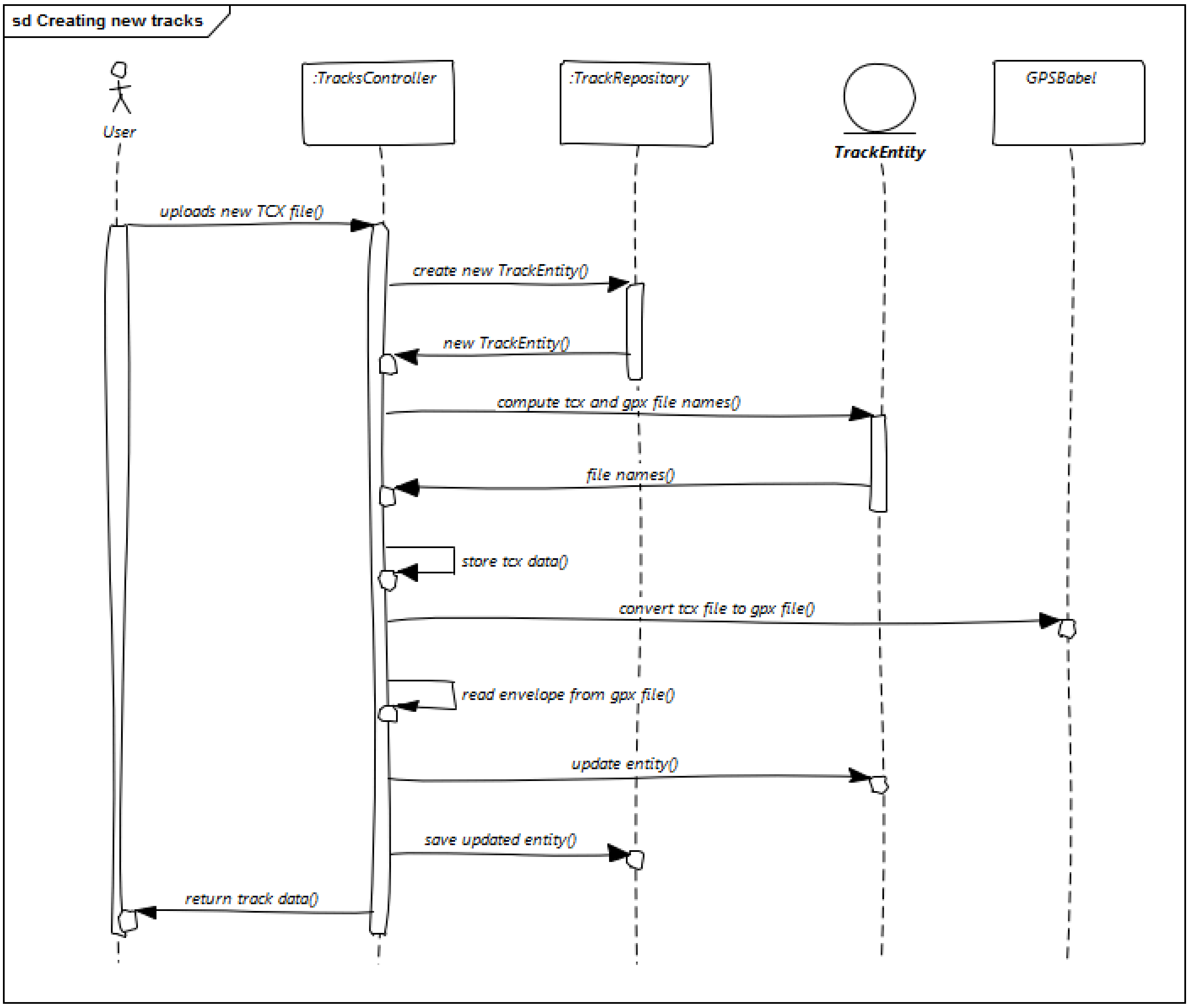
```sql
WITH mm AS (
  SELECT
    bikes.name,
    bikes.color,
    milages.recorded_on,
    (lead(milages.amount) OVER (PARTITION BY bikes.id ORDER BY milages.recorded_on) - milages.amount) value
  FROM bikes
    JOIN milages
      ON milages.bike_id = bikes.id
  ORDER BY milages.recorded_on ASC
)
SELECT
  mm.name,
  mm.color,
  (EXTRACT(MONTH FROM mm.recorded_on) - 1) idx,
  mm.value,
  sum(mm.value) OVER (PARTITION BY mm.recorded_on) total
FROM mm
WHERE (
  mm.value IS NOT NULL
  AND EXTRACT(YEAR FROM mm.recorded_on) >= EXTRACT(YEAR FROM DATE '2019-01-01')
)
ORDER BY mm.name ASC
```

6. Runtime View

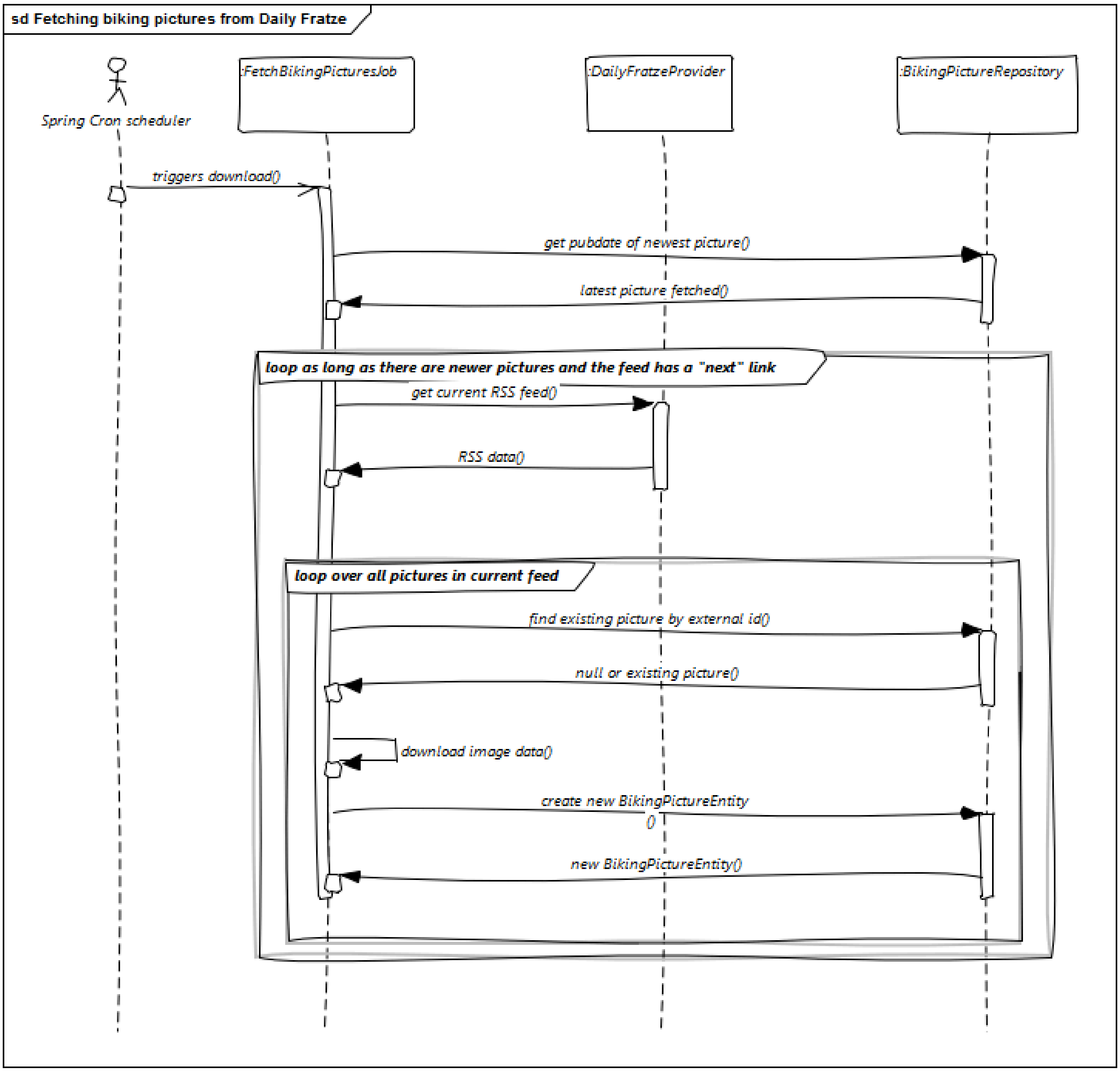
User interaction with *biking2* and error handling is pretty basic and simple.

I picked up two uses cases where an actual runtime view is interesting:

6.1. Creating new tracks



6.2. Fetching biking pictures from *Daily Fratze*


sd Fetching biking pictures from Daily Fratze
Spring Cron scheduler
:FetchBikingPicturesJob
:DailyFratzeProvider
:BikingPictureRepository
triggers download()
get pubdate of newest picture()
latest picture fetched()
loop as long as there are newer pictures and the feed has a "next" link
get current RSS feed()
RSS data()
loop over all pictures in current feed
find existing picture by external id()
null or existing picture()
download image data()
create new BikingPictureEntity ()
new BikingPictureEntity()

7. Deployment View

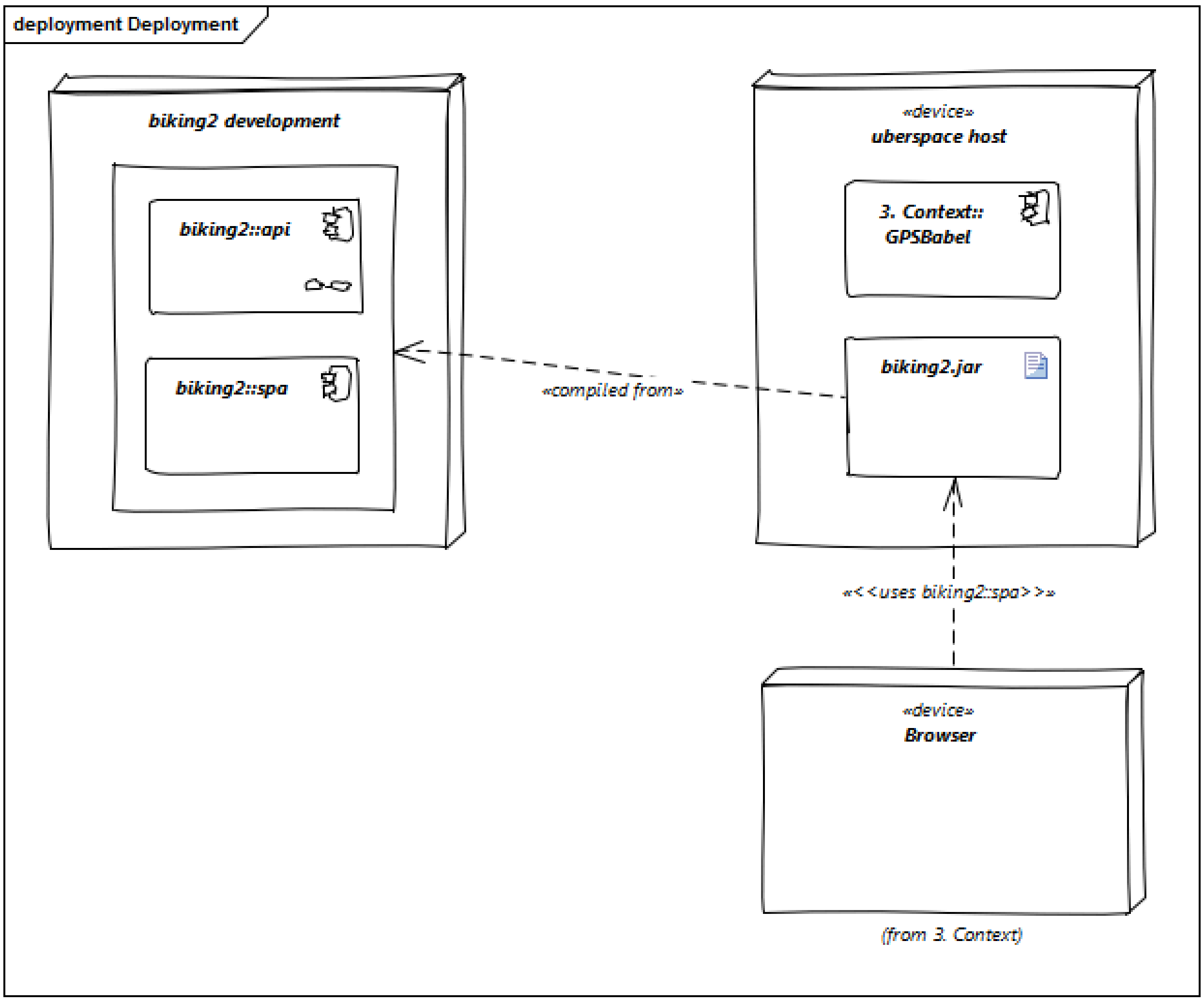


Table 8. Deployment nodes and artifact

Node / artifact	Description
biking2 development	Where *biking2* development takes place, standard computer with JDK 8, Maven and GPSBabel installed.
uberspace host	A host on Uberspace (https://uberspace.de) where biking2.jar runs inside a Server JRE (http://www.oracle.com/technetwork/java/javase/downloads/server-jre8-downloads-2133154.html) with restricted memory usage.
biking2.jar	A "fat jar" containing all Java dependencies and a loader so that the Jar is runnable either as jar file or as a service script (on Linux hosts).
Browser	A recent browser to access the AngularJS biking2 single page application. All major browsers (Chrome, Firefox, Safari, IE / Edge) should work.

8. Concepts

8.1. Domain Models

biking2 is a datacentric application, therefore everything is based around an Entity Relationship Diagram (ER-Diagram):

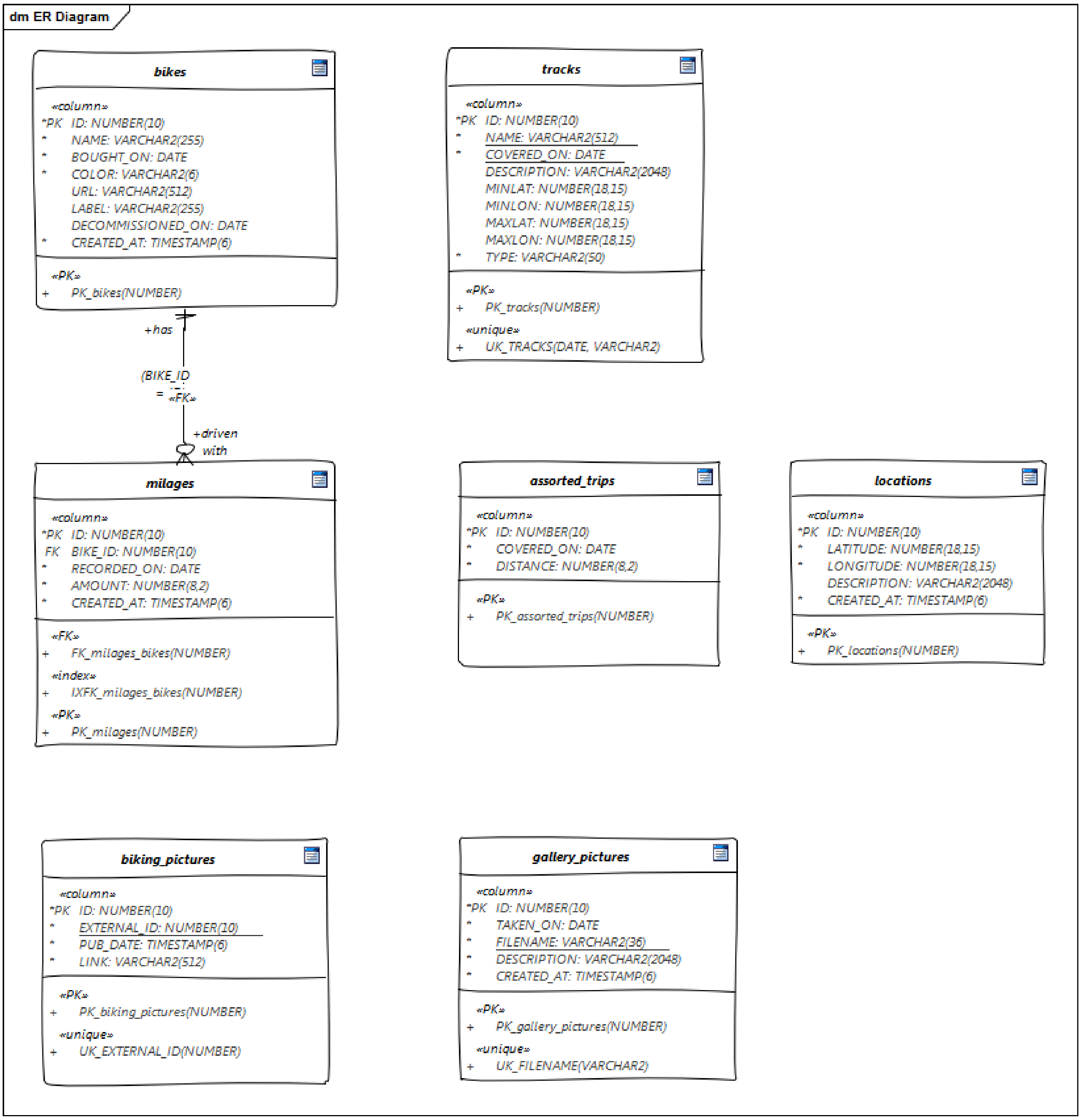


Table 9. Tables

Name	Description

Name	Description
bikes	Stores the bikes. Contains dates when the bike was bought and decomissioned, an optional link, color for the charts and also an auditing column when a row was created.
milages	Stores milages for a bike (when and how much).
tracks	Stores GPS tracks recorded and uploaded with an optional description. For each day the track names must be unique. The columns *minlat*, *minlon*, *maxlat* and *maxlon* store the encapsulating rectangle for the track. The *type* column is constrainted to "biking" and "running".
assorted_trips	Stores a date and a distance on that day. Multiple distances per day are allowed.
locations	Stores arbitrary locations (latitude and longitude based) for given timestamp with an optional description.
biking_pictures	Stores pictures collected from *Daily Fratze* together with their original date of publication, their unique external id and a link to the page the picture originaly appeared.
gallery_pictures	Stores all pictures uploaded by the user with a description and the date the picture was taken. The *filename* column contains a single, computed filename without path information.

Those tables are mapped to the following domain model:

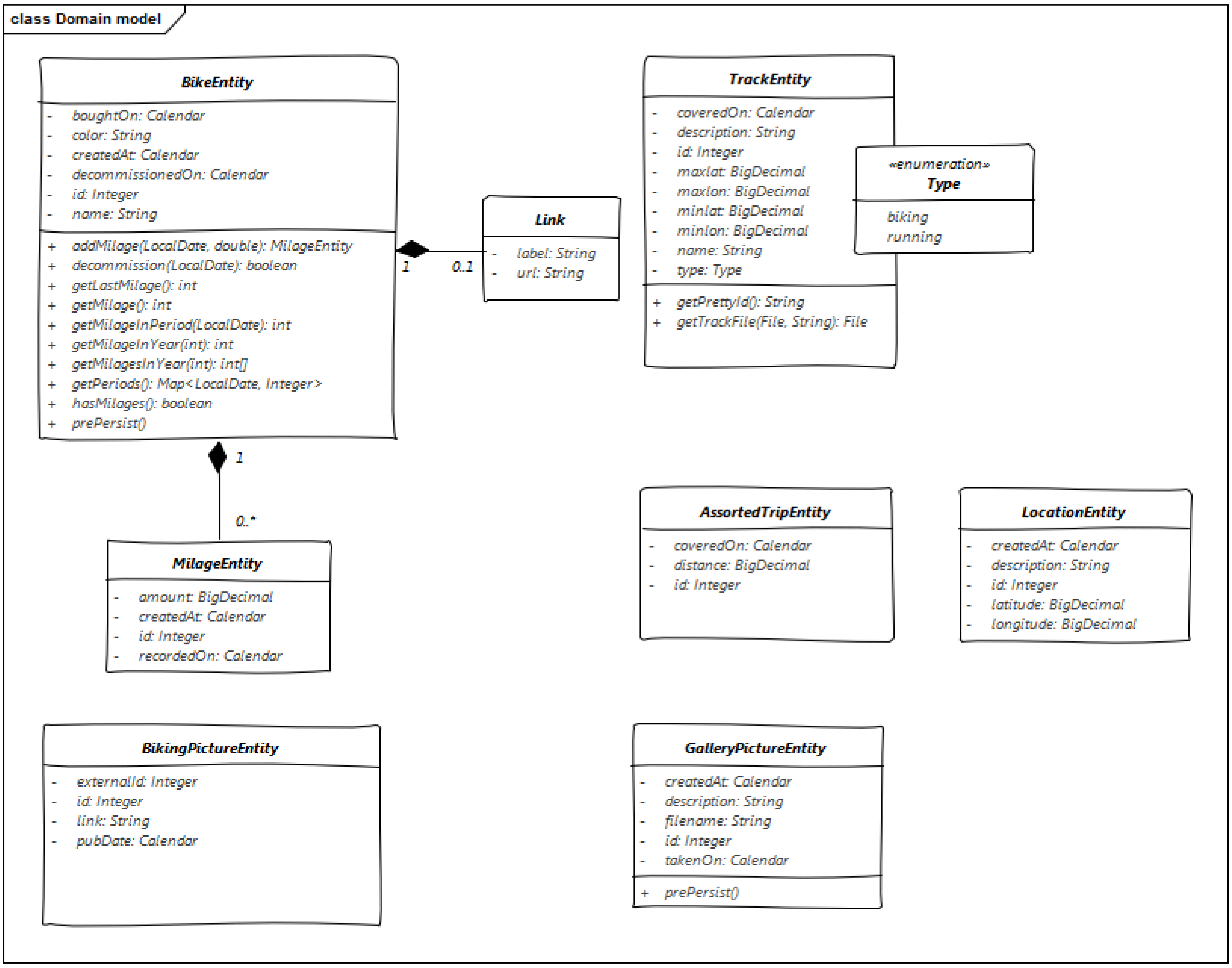


Table 10. Domain model

Name	Description
BikeEntity	A bike was bought on a given date and can be decommisioned. It has a color and an optional link to an arbitrary website. It may or may not have milages recorded. It has some important functions, see Important business methods on BikeEntity
MilageEntity	A milage is part of a bike. For each bike one milage per month can be recored. The milage is the combination of it's recording date, the amount and the bike.
TrackEntity	The representation of *tracks* contents. The type is an enumeration. Notable public operations are `getPrettyId` (computes a "pretty" id based on the instances id) and `getTrackFile` (generates a reference to the GPS track file in the passed data storage directory).
BikingPictureEntity	For handling pictures collected from *Daily Fratze*. The BikingPictureEntity parses the image link on construction and retrieves the unique, external id.
GalleryPictureEntity	A bean for handling the pictures uploaded by the user. `prePersist` fills the `createdAt` attribute prior to inserting into the database.

Name	Description
AssortedTripEntity	This entity captures a distance which was covered on a certain date and can used for keeping track of trips with bikes not stored in this application for example.
LocationEntity	Used in the tracker module for working with real time locations.

Table 11. Important business methods on BikeEntity

Name	Description
`decommission`	Decommissions a bike on a given date.
`addMilage`	Adds a new milage for a given date and returns it. The milage will only be added if the date is after the date the last milage was added and if the amount is greater than the last milage.
`getPeriods`	Gets all monthly periods in which milages have been recorded.
`getMilage`	Gets the total milage of this bike.
`getLastMilage`	Gets the last milage recorded. In most cases the same as `getMilage`.
`getMilageInPeriod`	Gets the milage in a given period.
`getMilagesInYear`	Gets all milages in a year as an array (of months).
`getMilageInYear`	Gets the total milage in a given year.

8.2. Persistency

biking2 uses an H2 (http://www.h2database.com/html/main.html) database for storing relational data and the file system for binary image files and large ascii files (especially all GPS files).

During development and production the H2 database is retained and not in-memory based. The location of this file is configured through the `biking2.database-file` property and the default value during development is `./var/dev/db/biking-dev` relative to the working directory of the VM.

All access to the database goes through JPA using Hibernate as provider. See the Domain Models for all entities used in the application.

The JPA Entity Manager isn't accessed directly but only through the facilities offered by Spring Data JPA, that is through repositories only.

All data stored as files is stored relative to `biking2.datastore-base-directory` which defaults to `./var/dev`. Inside are 3 directories:

- `bikingPictures`: Contains all pictures collected from *Daily Fratze*
- `galleryPictures`: Contains all uploaded pictures
- `tracks`: Contains uploaded GPS data and the result of converting TCX files into GPX files

8.3. User Interface

The default user interface for *biking2* which is packaged within the final artifact is a Single Page Application written in JavaScript using *Angular JS* together with a very default *Bootstrap* template.

For using the realtime location update interface, choose one of the many MQTT clients out there.

There is a second user interface written in Java called bikingFX (http://info.michael-simons.eu/2014/10/22/getting-started-with-javafx-8-developing-a-rest-client-application-from-scratch/).

8.4. JavaScript and CSS optimization

JavaScript and CSS dependencies are managed through Maven dependencies in form of webjars (http://www.webjars.org) wherever possible without the need for brew, npm, bower and the like.

Furthermore *biking2* uses wro4j (http://alexo.github.io/wro4j/) together with a small Spring Boot Starter (https://github.com/michael-simons/wro4j-spring-boot-starter) to optimize JavaScript and CSS web resources.

wro4j provides a model like this:

```xml
<groups xmlns="http://www.isdc.ro/wro">
    <!-- Dependencies for the full site -->
    <group name="biking2">
    <group-ref>osm</group-ref>

    <css minimize="false">/webjars/bootstrap/@bootstrap.version@/css/bootstrap.min.css</css>
    <css>/css/icons.css</css>
    <css>/css/stylesheet.css</css>

    <js minimize="false">/webjars/jquery/@jquery.version@/jquery.min.js</js>
    <js minimize="false">/webjars/bootstrap/@bootstrap.version@/js/bootstrap.min.js</js>
    <js minimize="false">/webjars/momentjs/@momentjs.version@/min/moment-with-locales.min.js</js>
    <js minimize="false">/webjars/angular-file-upload/@angular-file-upload.version@/angular-file-upload-html5-
shim.min.js</js>
    <js minimize="false">/webjars/angularjs/@angularjs.version@/angular.min.js</js>
    <js minimize="false">/webjars/angularjs/@angularjs.version@/angular-route.min.js</js>
        <js minimize="false">/webjars/angularjs/@angularjs.version@/angular-sanitize.min.js</js>
    <js minimize="false">/webjars/angular-file-upload/@angular-file-upload.version@/angular-file-
upload.min.js</js>
    <js minimize="false">/webjars/angular-ui-bootstrap/@angular-ui-bootstrap.version@/ui-bootstrap.min.js</js>
    <js minimize="false">/webjars/angular-ui-bootstrap/@angular-ui-bootstrap.version@/ui-bootstrap-
tpls.min.js</js>
    <js minimize="false">/webjars/highcharts/@highcharts.version@/highcharts.js</js>
    <js minimize="false">/webjars/highcharts/@highcharts.version@/highcharts-more.js</js>
    <js minimize="false">/webjars/sockjs-client/@sockjs-client.version@/sockjs.min.js</js>
    <js minimize="false">/webjars/stomp-websocket/@stomp-websocket.version@/stomp.min.js</js>

        <js>/js/ansi_up.js</js>
    <js>/js/app.js</js>
    <js>/js/controllers.js</js>
    <js>/js/directives.js</js>
    </group>
</groups>
```

This model file is filtered by the Maven build, version placeholders will be replaced and all resources, in webjars as well as inside the filesystem, will be available as `biking.css` and `biking.js`.

How those files are optimized, minimized or otherwise processed is up to wro4js configuration, but minification can be turned off during development.

8.5. Transaction Processing

biking2 relies on Spring Boot to create all necessary beans for handling local transactions within the JPA EntityManager. *biking2* does not support distributed transactions.

8.6. Session Handling

biking2 only provides a stateless public API, there is no session handling.

8.7. Security

biking2 offers security for its API endpoints only via HTTP basic access authentication (https://en.wikipedia.org/wiki/Basic_access_authentication) and in case of the MQTT module with MQTTs default security model. Security can be increased by running the application behind a SSL proxy or configuring SSL support in the embedded Tomcat container.

For the kind of data managed here it's an agreed tradeoff to keep the application simple. See also Safety.

8.8. Safety

No part of the system has life endangering aspect.

8.9. Communications and Integration

biking2 uses an internal Apache ActiveMQ broker on the same VM as the application for providing STOMP channels and a MQTT transport. This broker is volatile, messages are not persisted during application restarts.

8.10. Plausibility and Validity Checks

Datatypes and ranges are checked via JSR-303 (http://beanvalidation.org/1.0/spec/) annotations on classes representing the Domain Models. Those classes are directly bound to external REST interfaces.

There are three important business checks:

1. Bikes which have been decommissioned cannot be modified (i.e. they can have no new milages): Checked in `BikesController`.
2. For each unique month only one milage can be added to a bike. Checked in the `BikeEntity`.
3. A new milage must be greater than the last one. Also checked inside `BikeEntity`.

8.11. Exception/Error Handling

Errors handling to inconsistent data (in regard to the data models constraint) as well as failures to validation are mapped to HTTP errors. Those errors are handled by the frontends controller code. Technical errors (hardware, database etc.) are not handled and may lead to application failure or lost data.

8.12. Logging, Tracing

Spring Boot configures logging per default to standard out. The default configuration isn't change in that regard, so all framework logging (especially Spring and Hiberate) go to standard out in standard format and can be grabbed or ignored via OS specific means.

All business components use the *Simple Logging Facade for Java* (SLF4J). The actual configuration of logging is configured through the means of Spring Boot. No special implementation is included manually, instead *biking2* depends transitively on `spring-boot-starter-logging`.

The names of the logger corresponds with the package names of the classes which instantiate loggers, so the modules are immediately recognizable in the logs.

8.13. Configurability

Spring Boot offers a plethora of configuration options, those are just the main options to configure Spring Boot and available starters: Common application properties (https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html).

The default configuration is available in `src/main/resources/application.properties`. During development those properties are merged with `src/main/resources/application-dev.properties`. Additional properties can be added through system environment or through an `application-*.properties` in the current JVM directory.

During tests an additional `application-test.properties` can be used to add or overwrite additional properties or values.

Those are the *biking2* specific properties:

Table 12. biking2 specific configuration properties

Property	Default	Description
biking2.color-of-cumulative-graph	000000	Color of the cumulative line graph
biking2.dailyfratze-access-token	n/a	An OAuth access token for *Daily Fratze*
biking2.datastore-base-directory	${user.dir}/var/dev	Directory for storing files (tracks and images)
biking2.fetch-biking-picture-cron	0 0 */8 * * *	A cron expression for configuring the `FetchBikingPicturesJob`
biking2.home.longitude	6.179489185520004	Longitude of the home coordinate
biking2.home.latitude	50.75144902272457	Latitude of the home coordinate
biking2.connector.proxyName	n/a	The name of a proxy if *biking2* runs behind one
biking2.connector.proxyPort	80	The port of a proxy if *biking2* runs behind one
biking2.gpsBabel	/opt/local/bin/gpsbabel	Fully qualified path to the *GPSBabel* binary
biking2.scheduled-thread-pool-size	10	Thread pool size for the job pool
biking2.tracker.host	localhost	The host on which the tracker (MQTT channel) should listen
biking2.tracker.stompPort	2307	STOMP port
biking2.tracker.mqttPort	4711	MQTT port
biking2.tracker.username	${security.user.name}	Username for the MQTT channel

Property	Default	Description
biking2.tracker.password	${security.user.password}	Password for the MQTT channel
biking2.tracker.device	iPhone	Name of the OwnTracks device

8.14. Internationalization

Only supported language is English. There is no hook for doing internationalization in the frontend and there are no plans for creating one.

8.15. Migration

biking2 replaced a Ruby application based on the *Sinatra* framework. Data was stored in a SQLite database which has been migrated by hand to the H2 database.

8.16. Testability

The project contains JUnit tests in the standard location of a Maven project. At the time of writing those tests covers >95% of the code written. Tests must be executed during build and should not be skipped.

8.17. Build-Management

The application can be build with Maven without external dependencies outside Maven. *gpsbabel* must be on the path to run all tests, though.

9. Design Decisions

9.1. Using GPSBabel for converting TCX into GPX format

Problem

Popular JavaScript mapping frameworks provide easy ways to include geometries from GPX data on maps. Most Garmin devices however record track data in TCX format, so I needed a way to convert TCX to GPX. Both formats are relatively simple and in case of GPX good documented formats.

Constraints

- Conversion should handle TCX files with single tracks, laps and additional points without problem
- Focus for this project has been on developing a modern application backend for an AngularJS SPA, not parsing GPX data

Assumptions

- Using an external, non Java based tool makes it harder for people who just want to try out this application
- Although good documented, both file types can contain varieties for informations (routes, tracks, waypoints) which makes it hard to parse

Considered Alternatives

- Writing my own converter
- Using existing swiss army knife for GPS data: GPSBabel (http://www.gpsbabel.org)

> *GPSBabel converts waypoints, tracks, and routes between popular GPS receivers such as Garmin or Magellan and mapping programs like Google Earth or Basecamp. Literally hundreds of GPS receivers and programs are supported. It also has powerful manipulation tools for such data. such as filtering duplicates points or simplifying tracks. It has been downloaded and used tens of millions of times since it was first created in 2001, so it's stable and trusted.*

Decision

biking2 uses GPSBabel for the heavy lifting of GPS related data. The project contains a README stating that GPSBabel must be installed. GPSBabel can be installed on Windows with an installer and on most Linux systems through the official packet manager. Under OS X it is available via MacPorts or Homebrew.

9.2. Using local file storage for image and track data

Problem

biking2 needs to store "large" objects: Image data (biking and gallery pictures) as well as track data.

Considered Alternatives

- Using some kind of Cloud storage like S3
- Using local file system

Decision

I opted for local file system because I didn't want to put much effort into evaluating cloud services. If *biking2* should runnable in cloud based setup, one has to create an abstraction over the local filesystem currently used.

9.3. Use a database centric approach

Problem

biking2 was not received as database centric in the beginning. Hibernate entities had been modelled, the database was setup with Hibernates automatic DDL. Analytic functions had not been used, but computation has been done in memory.

Considered Alternatives

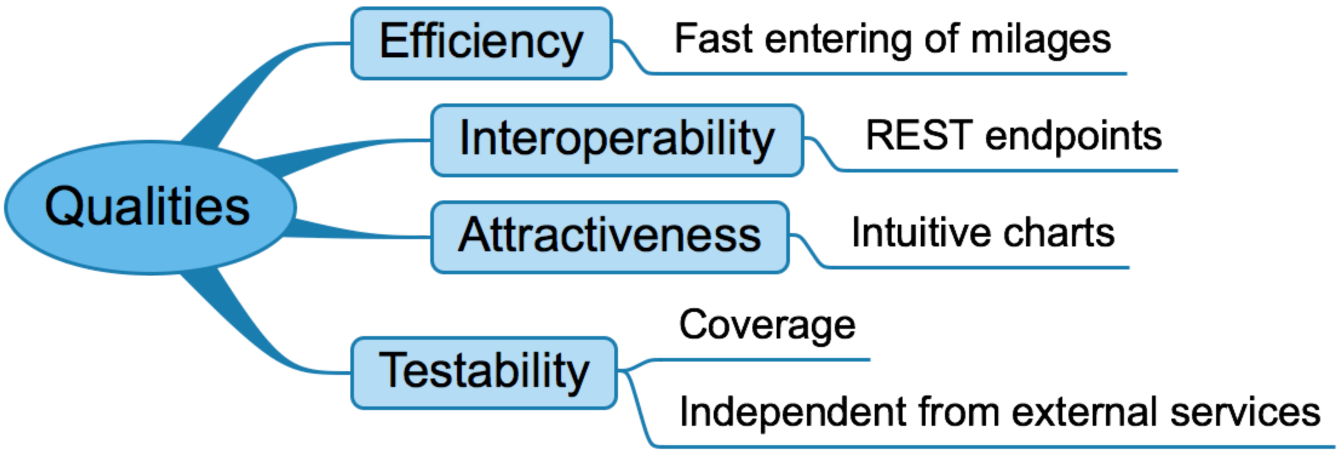
Use a database centric approach as described in "Live with your SQL-fetish and choose the right tool for the job" (https://speakerdeck.com/michaelsimons/live-with-your-sql-fetish-and-choose-the-right-tool-for-the-job)

Decision

In late 2019, that decision has been implemented. Flyway has been introduced to create tables and other migrations via SQL scripts. Based on a database that is actually under our control, we generate a jOOQ schema, on which all SQL generation for computing statistics is done. For more information, have a look at the whitebox view of the statistics modul.

10. Quality Scenarios

10.1. Quality Tree



10.2. Evaluation Scenarios

Testability / Coverage

By using *JaCoCo* during development and the build process (http://info.michael-simons.eu/2014/05/22/jacoco-maven-and-netbeans-8-integration/) ensure a code coverage of at least 95%.

Testability / Independent from external services

The architecture should be designed in such a way that algorithms depending on external services can be tested without having the external service available. That is: All external dependencies should be mockable.

Example: `FetchBikingPicturesJob` needs a resource containing a RSS feed. Retrieving the resource and parsing it are at least two different tasks. Fetching the resource through a separate class `DailyFratzeProvider` makes testing the actual parsing independent from a HTTP connection and thus relatively simple.

11. Technical Risks

biking2 has been up and running for nearly 2 years now, the architecture contains no known risk for my usage scenario.

There is a possibility that the H2 database can be damaged due to an unexpected shutdown of the VM (that is OS or hardware failure). The risk is mitigated through regularly backups of the serialized database file.

12. Glossary

Table 13. Glossar

Term	Synonym(s)	Description
AngularJS		AngularJS (https://en.wikipedia.org/wiki/AngularJS) is an open-source web application framework mainly maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications.
Apache ActiveMQ		Apache ActiveMQ (https://en.wikipedia.org/wiki/Apache_ActiveMQ) is an open source message broker written in Java.
Apache License		The Apache License (http://www.apache.org/licenses/LICENSE-2.0) is a permissive Open Source license, especially designed for free software.
Bootstrap		Bootstrap (http://getbootstrap.com) is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web.
Checkstyle		Checkstyle (http://checkstyle.sourceforge.net) is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard.
Daily Fratze	df, DF, DailyFratze	An online community where users can upload a daily picture of themselves (a selfie, but the site did them before they where called selfies).
Fat Jar		A way of packaging Java applications into one single Jar file containing all dependencies, either repackaged or inside their original jars together with a special class loader.
Flyway		Flyway (https://flywaydb.org/) is a SQL centric tool for applying database migrations in a controlled fashion.
Gallery picture		Pictures from tours provided manually by the hours in addition to the pictures collected automatically from *Daily Fratze*.
Garmin		Garmin (https://en.wikipedia.org/wiki/Garmin) develops consumer, aviation, outdoor, fitness, and marine technologies for the Global Positioning System.
GPSBabel		GPSBabel (http://www.gpsbabel.org) is a command line utility that converts waypoints, tracks, and routes between popular GPS receivers such as Garmin or Magellan and mapping programs like Google Earth or Basecamp.

Term	Synonym(s)	Description
GPX	GPS Exchange Format	GPX, or GPS Exchange Format, is an XML schema designed as a common GPS data format for software applications. It can be used to describe waypoints, tracks, and routes.
JaCoCo		JaCoCo (http://eclemma.org/jacoco/) is a code coverage library which can be used very easily from within NetBeans.
Java 8	JDK 8	The eight installment of the Java programming language (https://en.wikipedia.org/wiki/Java_(programming_language)) and the first one to support functional paradigms in the form Lambda expressions.
jOOQ		Java Object oriented querying: jOOQ (http://www.jooq.org/) is a framework for writing typesafe SQL with a fluent DSL from Java.
JUnit		[JUnit](http://junit.org/junit5/) is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.
MQTT	MQ Telemetry Transport	MQTT (https://en.wikipedia.org/wiki/MQTT) is a publish-subscribe based "light weight" messaging protocol for use on top of the TCP/IP protocol.
NetBeans		NetBeans (https://netbeans.org) is a free and open Source IDE that fits the pieces of modern development together.
OAuth		OAuth (https://en.wikipedia.org/wiki/OAuth) is an open standard for authorization. OAuth provides client applications a 'secure delegated access' to server resources on behalf of a resource owner.
oEmbed		The oEmbed (http://oembed.com) protocol is a simple and lightweight format for allowing an embedded representation of an URL on third party sites.
RSS	Rich Site Summary, Really Simple Syndication	RSS (https://en.wikipedia.org/wiki/RSS) uses a family of standard web feed formats to publish frequently updated information: blog entries, news headlines, audio, video.
SLF4J	Simple Logging Facade for Java	The Simple Logging Facade for Java (SLF4J) (http://www.slf4j.org) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time.
SPA	spa	Single Page Application
Spring Boot		Spring Boot (http://projects.spring.io/spring-boot/) makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

Term	**Synonym(s)**	**Description**
Spring Data JPA		Spring Data JPA (http://projects.spring.io/spring-data-jpa/), part of the larger Spring Data family, makes it easy to easily implement JPA based repositories.
STOMP	The Simple Text Oriented Messaging Protocol	STOMP (https://stomp.github.io) provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers.
TCX		Training Center XML (TCX) is a data exchange format introduced in 2007 as part of Garmin’s Training Center product.

About this template

Appendix A: Api

Bikes

Listing bikes

A `GET` request will list all of the service’s bikes.

Request Parameter

Parameter	Description
`all`	Flag, if all bikes, including decommissioned bikes, should be returned.

Response structure

Path	Type	Description
`[]`	`Array`	An array of bikes
`[].id`	`Number`	The unique Id of the bike
`[].name`	`String`	The name of the bike
`[].color`	`String`	The color of the bike (used in charts etc.)
`[].boughtOn`	`String`	The date the bike was bought
`[].decommissionedOn`	`String`	The date the bike was decommissioned
`[].story`	`Object`	The story of the bike
`[].story.url`	`String`	Link to the story
`[].story.label`	`String`	A title for the story
`[].lastMilage`	`Number`	The last recorded milage of the bike

Example request

```bash
$ curl 'http://biking.michael-simons.eu/api/bikes?all=true' -i -X GET
```

Example response

```http
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Content-Length: 390

[ {
  "id" : 4711,
  "name" : "Bike 1",
  "color" : "FF0000",
  "boughtOn" : "2015-01-01",
  "decommissionedOn" : "2015-12-31",
  "lastMilage" : 200,
  "story" : {
    "url" : "http://test.com/test",
    "label" : "Test Story"
  }
}, {
  "id" : 23,
  "name" : "Bike 2",
  "color" : "CCCCCC",
  "boughtOn" : "2014-01-01",
  "decommissionedOn" : null,
  "lastMilage" : 0,
  "story" : null
} ]
```

Creating a bike

A `POST` request is used to create a new bike

Request structure

Path	Type	Description
`name`	`String`	The name of the new bike
`boughtOn`	`String`	The date the new bike was bought
`color`	`String`	The color of the new bike
`miscellaneous`	`Boolean`	Whether it's a miscellaneous bike or not

Example request

```bash
$ curl 'http://biking.michael-simons.eu/api/bikes' -i -X POST \
    -H 'Content-Type: application/json;charset=UTF-8' \
    -d '{
  "name" : "test",
  "boughtOn" : "2023-08-26T23:06:27.027841+02:00",
  "color" : "cccccc",
  "decommissionedOn" : null,
  "miscellaneous" : false
}'
```

Example response

```http
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Content-Length: 154

{
  "id" : null,
  "name" : "test",
  "color" : "cccccc",
  "boughtOn" : "2023-08-26",
  "decommissionedOn" : null,
  "lastMilage" : 0,
  "story" : null
}
```

Adding milages to bikes

A bike manages its total milage at a given date. To make it easy for the user, no difference needs to be calculated, the user can enter the milage of is bike as stated on the odometer or whatever.

A `POST` request will add a new milage to a given bike.

Path Parameters

Table 14. /api/bikes/{id}/milages

Parameter	Description
`id`	The id of the bike to which a milage should be added

Request structure

Path	Type	Description
`recordedOn`	`String`	The date the new milage was recorded
`amount`	`Number`	The total milage of the bike on the given date

Example request

```bash
$ curl 'http://biking.michael-simons.eu/api/bikes/2/milages' -i -X POST \
    -H 'Content-Type: application/json;charset=UTF-8' \
    -d '{
  "recordedOn" : "2023-08-26",
  "amount" : 23.0
}'
```

Example response

HTTP

```
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Content-Length: 307

{
  "id" : null,
  "recordedOn" : "2023-08-01",
  "amount" : 23.0,
  "bike" : {
    "id" : null,
    "name" : "testBike",
    "color" : "CCCCCC",
    "boughtOn" : "2023-08-26",
    "decommissionedOn" : null,
    "lastMilage" : 23,
    "story" : null
  },
  "createdAt" : "2023-08-26T23:06:26.980972+02:00"
}
```

Adding a story to a bike

Bikes can have an associated story, how they were build or whatever.

A PUT request will update a given bike with a new story.

Path Parameters

Table 15. /api/bikes/{id}/story

Parameter	Description
id	The id of the bike whose story should be updated

Request structure

Path	Type	Description
url	String	Link to the story
label	String	A title for the story

Example request

BASH

```
$ curl 'http://biking.michael-simons.eu/api/bikes/2/story' -i -X PUT \
    -H 'Content-Type: application/json;charset=UTF-8' \
    -d '{
  "url" : "http://planet-punk.de/2015/08/11/nie-wieder-stadtschlampe/",
  "label" : "Nie wieder Stadtschlampe"
}'
```

Example response

The reponse is an updated bike.

HTTP
```
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Content-Length: 270

{
  "id" : null,
  "name" : "test",
  "color" : "000000",
  "boughtOn" : "2023-07-26",
  "decommissionedOn" : null,
  "lastMilage" : 0,
  "story" : {
    "url" : "http://planet-punk.de/2015/08/11/nie-wieder-stadtschlampe/",
    "label" : "Nie wieder Stadtschlampe"
  }
}
```

Deleting a story

An empty `PUT` request deletes a bikes story:

BASH
```
$ curl 'http://biking.michael-simons.eu/api/bikes/2/story' -i -X PUT \
    -H 'Content-Type: application/json;charset=UTF-8'
```

HTTP
```
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Content-Length: 154

{
  "id" : null,
  "name" : "test",
  "color" : "000000",
  "boughtOn" : "2023-07-26",
  "decommissionedOn" : null,
  "lastMilage" : 0,
  "story" : null
}
```

Trips

To keep track of all milage including time not spent on persistent bikes, the user can store assorted trips. A trip is a distance covered at a certain date.

Creating a trip

A `POST` request is used to create a new trip

Request structure

Path	Type	Description
`coveredOn`	`String`	The date of the trip
`distance`	`Number`	Distance covered on the trip

Example request

```bash
$ curl 'http://biking.michael-simons.eu/api/trips' -i -X POST \
    -H 'Content-Type: application/json;charset=UTF-8' \
    -d '{
  "coveredOn" : "2023-08-26",
  "distance" : 23.42
}'
```

Response structure

Path	Type	Description
`id`	`Number`	The unique Id of the trip
`coveredOn`	`String`	The date of the trip
`distance`	`Number`	Distance covered on the trip

Example response

```http
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Content-Length: 67

{
  "id" : 42,
  "coveredOn" : "2023-08-26",
  "distance" : 23.42
}
```

Banner

The application provides a nice banner during startup in the logs. It would be a shame not providing an API for that. To retrieve your ASCII art banner, just juse the following API:

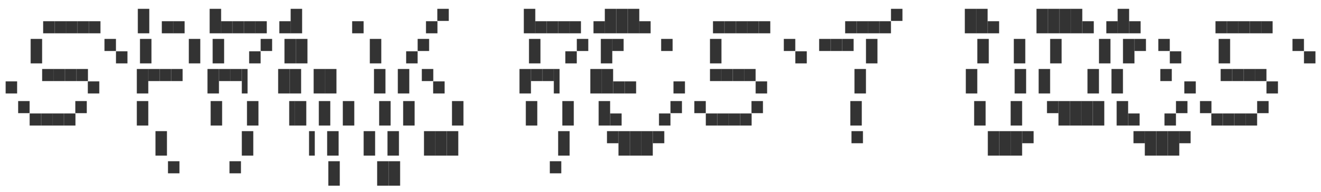
Example request

```bash
$ curl 'http://biking.michael-simons.eu/api/banner' -i -X GET \
    -H 'Accept: text/plain'
```

Example response

HTTP

```
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: text/plain;charset=UTF-8
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Content-Length: 1007
```



Last updated 2015-12-23 12:37:29 +0100