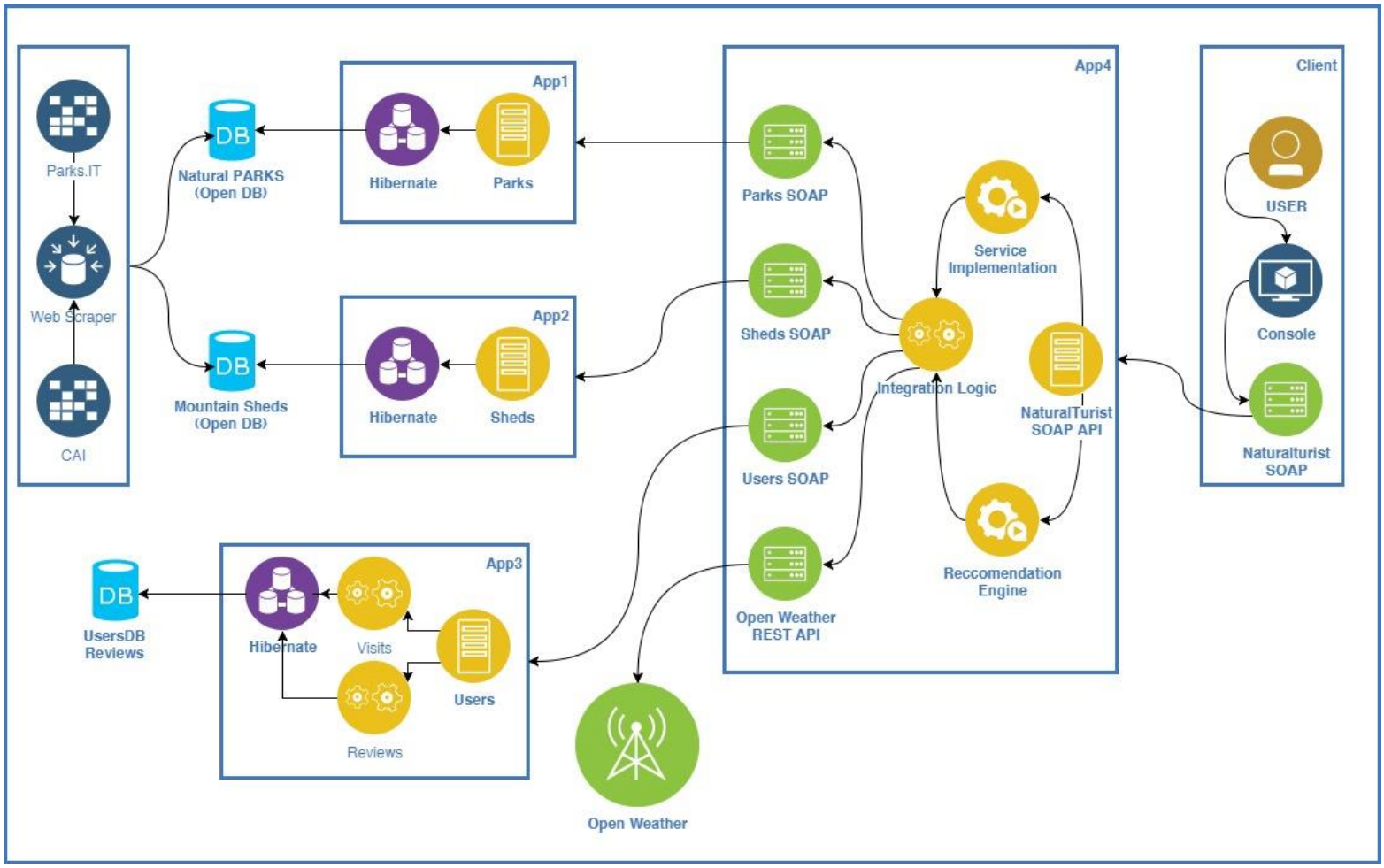
Giulio Dallatorre Final Assignment



This application is aimed at tourists that come in Trentino, the application can search between parks and mountain sheds. The user can leave a review on both and it can also leave a vote (thumbs up/down). The reviews are also tied to the weather.

A user will also get suggestions based on similarities of the places he reviewed and based on the most positively voted places. The suggestions are carried on internally.

App 1, App2 and App3 are contained in the same package in order to follow HEROKU's limit of MAX 5 apps for a free account.

Since I have on my account all the 3 submissions I have only 2 slots left.

The first is taken by App1,2,3 The se apps are independent and don't even share the package. Since they are compiled in a single WAR they do share the endpoint declaration and the libraries.

The second slot is taken up by App4, the orchestrator that includes the integration logic, application logic and recommendation engine.

APP1 – APP2

App 1 and 2 are quite similar… They provide full access to the database through a REST interface. The possible operations are Add, Edit, Remove, Get all the parks, Get a single park, Get a subset of the parks based on search parameters. The specifications of the API can be read in the wiki
https://github.com/dallatorretdu/introsde_2017_assignment_final_model/wiki

I intended to use the Open Data available locally, but the quality of the results is not yet good enough for most situations. E.G. the database named "Points of natural interest in Trento" contained only 1 row with the number of interesting points near Trento "23 points". So in order to preserve my idea I set up 2 independent databases containing the data for the natural barks in the province of Trento and Bolzano (~300 rows) and one for the mountain sheds "Rifugi" and "Bivacchi". Since the data was available but broken in the Open Datasets I decided to scrape the necessary information from the already available websites.
The parks were scraped from Parks.it (you could also buy a cd with all the datasets), and the sheds from CAI's website. The tool used for scraping is "Web Scraper" and the script used for the multicategory, multipage navigation is included in the github project

introsde_2017_assignment_final_model/src/introsde/APP1/parks/persistence/entities/WebScraperConfig.txt

introsde_2017_assignment_final_model/src/introsde/APP2/sheds/persistence/entities/WebScraperConfig.txt

I tried to gather the coordinates of the parks and the mountain sheds, but I could not find a consistent source for these. Only SAT's website had an excel file containing shed name and coordinates, but only for the ones they manage.

App3

App3 manages the users, the registration, user preferences and also manages the reviews and votes (visits) which are related to the Parks or sheds. I did this to preserve the purity of the Parks and Sheds database so in a future it could be used seamlessly with an open Database.

The visits and reviews are directly linked to a user and are not deleted in cascade when a user gets removed from the db, in this way the reviews and votes will be persistent in the system and will not disappear.

App4

This app contains several core modules to integrate all the services. All the SOAP interface have been created using WSimport. The REST open weather interface has been created from scratch, with the help of a tool in order to automatically generate the entities starting from the JSON data. Somehow requesting XML sometimes resulted in an inconsistent and malformed XML document in return.

App4 does not expose to the users the methods to create, remove and edit parks and sheds.

The service implementation includes some good levels of automation like the automatic date on visits, the weather association on reviews, the retrieval of the most rated places and the generation of recommendations. This app works synchronously with the requests from the client.

Recommendation engine

Included in App4 there's my custom-made recommendation engine. I tried to use "Recombee" but I could not figure out how to map my data into a recombee compatible user-transaction-product way, without invent including the complex fields presents in several of the entries of the open databases.

The Recommendation consists of 5 items:

- 1 recommended randomly, but biased towards the preferences of the user.
- 2 recommended based on the most liked places by the other users, still randomly selected but only between the places with the highest score.
- 2 recommended searching similarities between the user's reviewed places. Things kept in consideration are type, altitude, comuni and also similarities in the name to try isolate the differences between park types "parco nazionale dello Stelvio" vs "Biotopo locale di Coredo"

The last type of recommendation is quite complex and works well when the user doesn't have a lot of reviews already, but will potentially include the whole dataset if the reviews are enough.

Persistency

In this project I decided to base my persistency service only on Hibernate, without even calling JPA. The DAO classes are a bit more complicated but I could control which classes to persist and which parameters to adopt for every DAO.

I also use c3p0 as a connection pool manager, this increased by a very good margin the performances of my app.

Client

The client is a relatively simple console (Scanner) application able to "log-in" or register a user, search for users, parks, sheds by ID or keyword. The user can prompt the system to leave a review, a vote or to get the recommendations.

The client only uses the default generated client for App4.

Testing

The application has been developed locally and several Junit 4 test cases have been included in App4. The tests do test App1,2,3 through their WSDL clients. These tests can also be run locally referencing Heroku's server, in order to check the full compatibility of the persistency services. These tests should not be regarded as unitary tests, as they use the network and resemble much more integration tests. The applications have not been developed in a TDD fashion and are not covered of unitary tests.