This thesis is part of the over-arching project of crafting an end-user-friendly client-application for the Web of Needs (related publications), short WoN. The main focus was to research ways of structuring the JavaScript-based client-application; thus it consisted of researching and experimenting with state-of-the-art web-application architectures and tooling, adapting and innovating on them for the particular problem space, as well as identifying a migration path for updating the existing code-base. To define the requirements, we first need to take a high-level look over what the Web of Needs is and how people can interact with it.

Web of Needs

It is a set of protocols (and reference implementations) that allow posting documents, for instance describing supply and demand. Starkly simplified examples would be “I have a couch to give away” or “I’d like to travel to Paris in a week and need transportation”. These documents, called “needs” can be posted on arbitrary data servers (called “WoN-Nodes”). There they’re discovered by matching-service, that continuously crawls the nodes it finds. Additionally, to get faster results, nodes can notify matchers of new needs. These then get compared with the ones the matcher already knows about. If it finds a good pair – e.g. “I have a couch to give away” and “Looking for furniture for my living room” – the matcher notifies the owners of these needs. They can then decide whether they want to contact each other. If they send and accept each other’s contact request, they can start chatting with each other. The protocol in theory can also be used as a base-level for other interactions, like entering into contracts or transferring money.

Data on WoN-Nodes

Needs, connections between them and any events on those connections are published on the WoN-Nodes in the form of RDF, which stands for Resource Description Framework. In it, using a variety of different syntax-alternatives, data is structured as a graph that can be distributed over multiple (physical) resources. Edges in the graph in their basic, most primitive form are described by triples of subject (the start-node), predicates (the edge-type) and object (the target-node). Note that subject and object need to be Unique Resource Identifiers (URIs). Additionally, when using URIs, that also are Uniform Resource Locators (URLs) – together with the convention to publish data for an RDF-node at that URL – data-graphs on multiple servers can easily be linked with each other, thus making them Linked Data. This is a necessary requirement for the Web of Needs, as data is naturally spread out across several servers, i.e. WoN-Nodes.

Some example triples taken from a need description could look something like:

```
<https://node.matchat.org/won/resource/need/7666110576054190000>
<http://purl.org/webofneeds/model#hasBasicNeedType>
<http://purl.org/webofneeds/model#Demand> .
```

```
<https://node.matchat.org/won/resource/need/7666110576054190000>
<http://purl.org/webofneeds/model#hasContent>
_:c14n0 .

<https://node.matchat.org/won/resource/need/7666110576054190000>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://purl.org/webofneeds/model#Need> .

_:c14n0
<http://purl.org/dc/elements/1.1/title>
"Transportation Paris-Charles de Gaulle to City Center" .

_:c14n0
<http://purl.org/webofneeds/model#hasTextDescription>
"I'd like to travel to Paris in a week and need ⏎
transportation (e.g. ride-sharing) from the airport ⏎
to the city-center. :)" .
```

As you can see, this way of specifying triples, called N-Triples, isn't exactly developer-friendly; the subject is repeated and large parts of the URIs are duplicate. The short-URIs starting with an underscore (e.g. `_c14n0`) are called blank-nodes and don't have a meaning outside of a document.

There's several other markup-languages/serialization-formats for better writing and serving these triples, e.g. Turtle, N3, RDF/XML and JSON-LD. The same example above in JavaScript Object Notation for Linked Data (JSON-LD) would read as follows:

```
{
  "@id":"need:7666110576054190000",
  "@type":"won:Need",
  "won:hasBasicNeedType":"won:Demand",
  "won:hasContent": {
    "dc:title":
      "Transportation Paris-Charles ⏎
      de Gaulle to City Center",
    "won:hasTextDescription":
      "I'd like to travel to Paris in ⏎
      a week and need transportation ⏎
      (e.g. ride-sharing) from the ⏎
      airport to the city-center . :)"
  },

  "@context":{
     "need": "https://node.matchat.org/won/resource/need/",
     "rdfs":"http://www.w3.org/2000/01/rdf-schema#",
     "dc":"http://purl.org/dc/elements/1.1/",
     "won":"http://purl.org/webofneeds/model#",
     "won:hasBasicNeedType":{
        "@id":"won:hasBasicNeedType",
        "@type":"@id"
```

```
      }
    }
  }
```

As you can see, JSON-LD allows to nest nodes and to define prefixes (in the `@context`). Together this allow to avoid redundancies. The other serialization-formats are similar in this regard (and are used between other services in the Web of Needs); however, as JSON-LD also is valid JSON/JS-code, it was the natural choice for using it for the JS-based client-application.

WoN-Owner-Application

Interaction Design

Among the three services that play roles in the web of needs – matchers, nodes and owner-applications – the work I did has its focus on the latter of these. It provides people a way to interact with the other services in a similar way that an email-client allows interacting with email-servers. Through it, people can:

- Create and post new needs. Currently these consist of a simple data-structure with a subject, textual description and optional tags or location information.
- View needs and all data in them in a human-friendly fashion
- Share links to posts with other people
- Immediately get notified of and see matches, incoming requests and chat messages
- Send and accept contact/connection requests
- Write and send chat messages

For exploring these interaction, several prototypes – both paper-based and (partly) interactive – had already been designed, the latest of which was a (graphical) overhaul by Ulf Harr.

screens of latest prototype

Technical Requirements

On the development-side of things, the requirements were:

- As subject of a research-project, the protocols can change at any time. Doing so should only cause minimal refactoring in the owner-application.
- In the future different means of interactions between needs – i.e. types need-to-need connections – will be added. Doing so should only cause minimal changes in the application.
- Ultimately the interface for authoring needs should support a wide range of ontologies respectively any ontology people might want to use for descriptions. Adapting the authoring guys or even just adding a few form input widgets should be seamless and only require a few local changes.

- We didn't want to deal with the additional hurdles/constraints of designing the prototype for mobile-screens at first, but a later adaption/port was to be expected. Changing the client application for that should require minimal effort.

The previous iteration of the prototype had already been implemented in angular-js 1.X. However, the code-base was proving hard to maintain, as we continuously had to deal with bugs that were hard to track down, partly because JavaScript's dynamic nature obscured where they lived in the code and mostly because causality in the angular-app became increasingly convoluted and hard to understand. The application's architecture needed an overhaul to deal with these issues, hence this work you're reading. Thus, additional requirements were:

- Causality in the application is clear and concise to make understanding the code and tracking down bugs easier.
- Local changes can't break code elsewhere, i.e. side-effects are minimized.
- Responsibilities of functions and classes are clear and separated, so that multiple developers can easily collaborate.
- The current system state is transparent and easily understandable to make understanding causality easier.
- Lessens the problems that JavaScript's weakly-typed nature causes, e.g. bugs causing exceptions/errors way later in the program-flow instead of at the line where the problem lies.