A Client-side JavaScript Architecture

For the Web-of-Needs Owner-Application Prototype

Renée Singer

25.01.2018

Contents

List of Figures

List of Listings

1 Authorship Declaration

Hereby I declare, that I authored this work on my own and that all sources and aids have been listed in their entirety and that all parts of the work – including tables, maps and figures – stemming from other works in exact wording or spirit, have been marked as quote and contain a reference to the source.

2 Abstract

This thesis is part of the over-arching Web of Needs (see ref. “Webofneeds” n.d.; for related publications see ref. “Web of Needs Publications List” 2013) project – short WoN – and, somewhat more particular, of developing an end-user-friendly client-application (see ref. “Mat(ch)²at” n.d.) prototype/demonstrator for it, that allows testing the protocol and helps with communicating the WoN’s potential to people. The main focus of the work done for this thesis 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 laid out in chapter 3, as well as identifying a migration path for updating the existing code-base.

As laid out in chapter 5, we used a variant of the (ng-)redux architecture (see secs. 4.1.6, 4.1.7 for the original architecture), but added a “messaging-agent” more akin to the runtime in the Elm-architecture (see sec. 4.1.8). The action-creators, who handle non-socket network communication, use an RDF-store for caching the RDF-data used throughout the Web of Needs. To allow using newer language features and bundle the application Webpack (sec. 5.3.6) with Babel (sec. 5.3.1.5) is used. Styling is done in SCSS (sec. 5.3.2) using BEM (sec. 5.3.3) as naming convention.

3 Problem Description

As mentioned already in the abstract, the challenge to be tackled by this work was to find (and adapt) a state-of-the-art architecture and tooling for the Web of Needs Owner client application and a migration path of the JavaScript code-base to these, while addressing the requirements laid out in this chapter. To define said requirements, we first need to take a high-level look at what the Web of Needs is and how people can interact with it.

3.1 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-services, that continuously crawl the nodes they can find. Additionally, to get results more quickly, 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.

3.2 RDF-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 (see ref. "Resource Description Framework" n.d.). In RDF, using a variety of different syntax-alternatives, data is structured as a graph that can be distributed over multiple (physical) resources. Every graph-edge is defined by a subject (the start-node), a predicate (the "edge-type") and object (the target-node).

Note that the subject always needs to be an Unique Resource Identifiers (URIs). For the case, when those URIs are also Uniform Resource Locators (URLs) there is the convention to host that data under

that URL. This allows easily linking data graphs on multiple servers, thus making them Linked Data (see ref. “Linked Data” n.d.). 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/post on the WoN-Node running at http://node.matchat.org/resource/ (accessed 18.06.2018) could look something like the following ones:

Listing 3.1: Excerpt of a need description (N-Triples)

```
<https://node.matchat.org/won/resource/need/ow14asq0gqsb>
<http://purl.org/webofneeds/model#is>
_:b0 .

<https://node.matchat.org/won/resource/need/ow14asq0gqsb>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://purl.org/webofneeds/model#Need> .

_:b0
<http://purl.org/dc/elements/1.1/title>
"Simple easel to give away" .

_:b0
<http://purl.org/dc/elements/1.1/description>
"I've got an old easel lying around at my place that is \
mostly just catching dust. If there is any aspiring landscape \
painters that would like to have it: poke me :)" .
```

As can be seen, this way of specifying triples, called N-Triples, isn’t well-suited for direct reading or authoring; the subjects (`.../need/ow14asq0gqsb` and `_:b0`) are repeated and large parts of the URIs are duplicate. The short URIs starting with an underscore (e.g. `_:b0` are called blank-nodes and don’t have meaning outside of a document and can reoccur in other documents as opposed to Unique Resource Identifiers. there is also a convention that when using URLs used as subject-URIs (e.g. https://node.matchat.org/won/resource/need/ow14asq0gqsb) it should be possible to access these to get a document with the triples for that subject.

There are several other markup-languages respectively serialization-formats for easier writing and clearer serializations for these triples, e.g. Turtle/Trig, JSON-LD and the somewhat verbose RDF/XML. The same example, but in JavaScript Object Notation for Linked Data (JSON-LD) would read as follows:

Listing 3.2: Excerpt of a need description (JSON-LD)

```
{
  "@id": "need:ow14asq0gqsb",
  "@type": "won:Need",
```

```
  "won:is": {
    "@id": "_:b3", // <-- optional
    "dc:title": "Simple easel to give away"
    "dc:description": "I've got an old easel lying \
    around at my place that is mostly just catching \
    dust. If there is any aspiring landscape painters \
    that would like to have it: poke me :)",
  },

  "@context": {
    "dc": "http://purl.org/dc/elements/1.1/",
    "need": "https://node.matchat.org/won/resource/need/",
    "won": "http://purl.org/webofneeds/model#"
  }
}
```

As can be seen above, JSON-LD allows to visually represent the nesting (`need:ow14asq0gqsb won:is _:b3`) and to define prefixes (in the `@context`). Together this allows to avoid redundancies. The other serialization-formats are similar in this regard (and are used between other services in the Web of Needs) – see below for a turtle-serialization of the same triples:

Listing 3.3: Excerpt of a need description (TTL)

```
@prefix dc:    <http://purl.org/dc/elements/1.1/> .
@prefix need:  <https://node.matchat.org/won/resource/need/> .
@prefix won:   <http://purl.org/webofneeds/model#> .

need:ow14asq0gqsb
  a             won:Need ;
  won:is        [
    dc:title         "Simple easel to give away" ;
    dc:description   "I've got an old easel lying
      around at my place that is mostly just catching
      dust. If there is any aspiring landscape painters
      that would like to have it: poke me :)"
  ]
```

However, as JSON-LD also constitutes valid JSON/JS-object-literal-syntax, it is the natural choice for using it in the JS-based client-application and was already being used in the existing code-base.

3.3 WoN-Owner-Application

3.3.1 Interaction Design

Among the three services that play roles in the web of needs – matchers, nodes and owner-applications – the work at hand has its focus on the latter of these. It provides people with a way to interact with the other services in a similar way to how 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 line, a long textual description and optional tags or location information.
- **View needs/posts** and all data in them in a human-friendly fashion
- **Share links** to needs/posts with other people
- **Notifications:** 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 interactions, several prototypes had already been designed and implemented. The first were paper-based or simple clickable dummies, that weren't fully interactive. The last prototype before the one described in this work had been implemented using Angular 1.X and its MVC-architecture (sec. 4.1.3). For this iteration new graphic designs were made, that necessitated to leave the Bootstrap-theme we had previously been using behind and develop and maintain our own (S)CSS (see section 5.3.2). See figs. 5.2, 5.3, 5.4, 5.5, 5.6 for screenshots of the GUI.

3.3.2 Technical Requirements

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

- **Networking:** The application needs to be able to keep data in sync between the JS-client and the Java-based servers. This happens through a REST-API and websockets. Most messages arrive at the WoN-Owner-Server from the WoN-Node and just get forwarded to the client via the websocket. The only data directly stored on and fetched from the Owner-Server are the URIs and private keys[^cryptography happens on the WoN-Owner-Server] of needs/posts owned by an account, as well as information which messages have been seen. All other data lives on the WoN-Node-Servers, that have no concept of user-accounts.
- **Adaptability and Extendability:** As subject of a research-project, the protocols can change at any time. Doing so should only cause minimal refactoring in the owner-application. Planned features/changes include integrating payment-services, "personas" (i.e. signature-identities) or

"agreements" (i.e. a mechanism to make formalized contracts via messages exchanged over the connections by formally agreeing with the contents of other messages).

- **Many Ontologies:** Ultimately the interface for authoring needs should support a wide range of ontologies[1] respectively any ontology people might want to use for describing things and concepts. Adapting the authoring GUIs or even just adding a few form input widgets should be seamless and only require a few local changes.
- **Mobile:** We[2] 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 needed to require minimal effort.
- **Responsiveness:** It should be possible to build an application that feels responsive when using it. This means low times till first meaningful render and complete page-load. This in term implies a reduction of round-trips and HTTP-requests and use of caching mechanisms for data and application code. But "feeling responsive" also means that operations that take a while despite all other efforts need to show feedback to the user (e.g. spinning wheels, progress bars, etc) to communicate that the application hasn't frozen.
- **Thin Application-Server:** The WoN-Owner-Server should be as thin as possible, for this reason and to allow for an app that can do without hard page-loads it is developed as a client-side "Single Page Application" in JavaScript. Native applications were considered, but they don't possess the OS-independence and simple delivery, that the web-plattform has to offer.
- Runs on **ever-green browsers**: As it's a research-prototype there is less need to support old browsers, like the pre-edge internet-explorer.
- **DX:** Good developer experience, i.e. new language features to allow more expressive, robust and concise code, warnings about possible bugs where possible, auto-completion, jump-to-definition, documentation on mouse-hover, etc.
- **Learnable in project:** Any new technologies needed to be feasible to learn within the project's scope.
- **Retain code-base:** The more of the old, AngularJS1.x-based code-base that could be kept, the better in regard to the project-scope/budget.

The previous iteration of the prototype had already been implemented in Angular-JS 1.X. However, the code-base was proving hard to maintain. We continuously had to deal with bugs that were hard to track down, partly because JavaScript's dynamic nature obscured where they originated 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, which gave rise to the work at hand. Thus, additional requirements were:

- **Causality** in the application is **clear** and concise to make understanding the code and tracking

[1] Ontologies can be described as data-structure-descriptions, i.e. schemata, for RDF-data. E.g. the current demo-ontology defines that needs can have a title, a description, a location, tags, etc.

[2] My colleagues at the researchstudio Smart Agent Technologies and I

down bugs easier.
- Local changes can't break code elsewhere, i.e. **side-effects of changes are minimized**.
- **Responsibilities** of functions and classes are **clear** and separated, so that multiple developers can easily collaborate.
- **Clear System State:** The current system state is transparent and easily understandable to make understanding causality easier.
- **Localizable Errors:** The solution should lessen the problems that JavaScript's weakly-typed nature causes, e.g. bugs causing errors way later in the program-flow instead of at the line where the problem lies.
- **Reduces code-redundancies**
- Makes **code conciser** and clearer to the reader

4 State of the Art

4.1 Frameworks and Architecture

The following patterns all help with developing graphical user interfaces, by providing separation of concerns and decoupling. This makes it easier for multiple developers to collaborate and allows better reasoning about the app's behavior. It also makes adapting the application easier when the understanding of the design problem changes.

The presented architectural patterns for client-side JavaScript-applications, at the time of writing encompass all widely used and sufficiently distinct I could discern during my technology- and literature research. This selection provides the reference for choosing an architecture for the Web of Needs owner application, which architecture is based off of redux (sec. 4.1.6) in general and ng-redux (sec. 4.1.7) in particular. The reasons for this design-decision will be discussed in section 5 "Suggested Solution".

4.1.1 Model-View-Controller

On of the, if not the most classical architectural pattern typically used in front-end-programming is Model-View-Controller. It was first introduced in the 70s at the Palo Alto Research Center (Reenskaug 1979) and first formally published by Krasner and Pope (1988b). As it's still widely used and Angular's MVC (sec. 4.1.3) is a variant thereof it shall be shortly described here for the sake of completeness. The pattern mainly consists of three types of building blocks (as can also be seen in figure fig. 4.1):

- **controllers** contain the lion's share of the business logic. User input gets handled by them and they get to query the model. Depending on these two information sources they decide what messages to send to the the model, i.e. the controller telling the model to change. Usually there is one controller per view and vice-versa.
- **models** hold the application's state and make sure it's consistent. If something in the data changes, it notifies views and controllers depending on it. These notifications can be parametrized, telling the dependents what changed.
- **views** are what the outside world/user's get to see. When the model changes, the view gets notified and – depending on the data passed along and what it reads from the model – updates

accordingly. Especially in HTML-applications, views (and thereby controllers) tend to be nested (e.g. the entire screen – a column – a widget in it – a button in the widget)

Note, that there is a wide range of different interpretations of this architectural pattern, that organize models, views and controllers differently. Section 4.1.3 describes one of these (angular 1.X' MVC) in more detail.

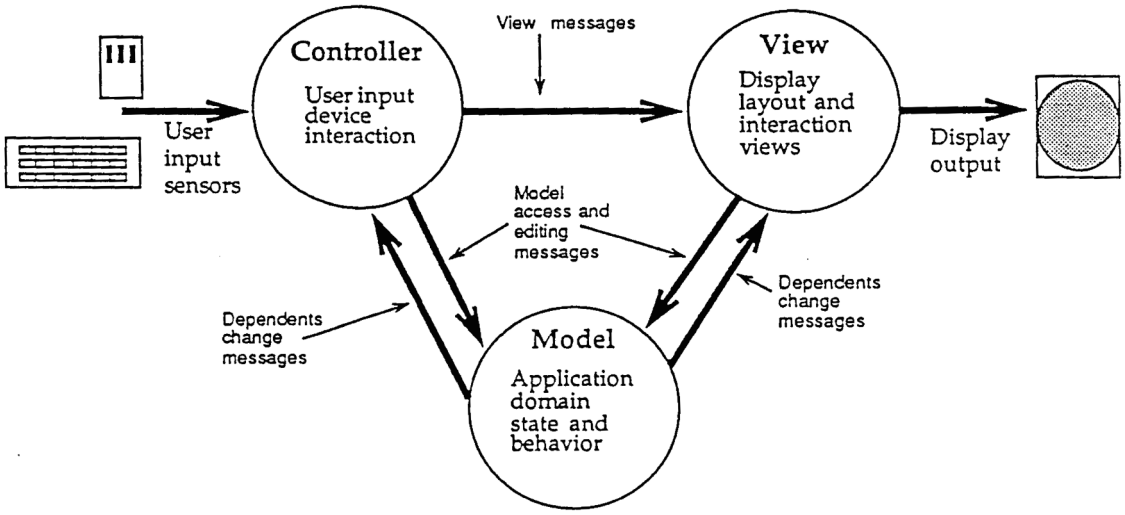

Figure 4.1: MVC-architecture (Krasner and Pope 1988a)

4.1.2 Model-View-ViewModel

This architectural pattern, also known as "Model-View-Binder" is similar to MVC but puts more emphasis on the separation between back-end and front-end. Its parts are the following (see also fig. 4.2):

- **The model** is the back-end business-logic and state. It can be on a different machine entirely, e.g. a web server.
- **The view-model** contains the front-end logic and state. It is a thin binding layer, that processes inputs and that manages and provides the data required by the view.
- **The view** is a stateless rendering of the data retrieved from the view-model; in the case of some frameworks, this happens via declarative statements in the view's templates, that automatically get updated when the data in the view-model changes. User-input events raised in the view get forwarded to the view-model.

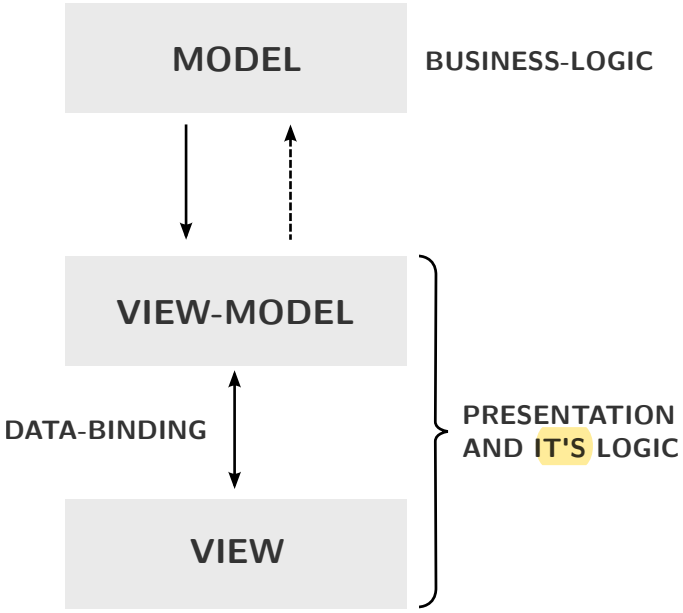

Figure 4.2: MVVM-architecture (source: https://en.wikipedia.org/wiki/File:MVVMPattern.png, accessed 18.06.2018)

4.1.3 Angular 1.x MVC

Angular 1.x is a JavaScript-framework that roughly follows the MVC/MVVM architectures, but has a few conceptual variations and extensions.

For views it uses a template-syntax. See the following snippet for an example from the webofneeds-codebase, that renders to an `<h2>`-header and a paragraph with a description, if the description is present (the `ng-show` is a conditional):

Listing 4.1: Example angular template code from WoN-codebase

```
...
<h2 class="post-info__heading"
  ng-show="self.post.getIn([
    'won:hasContent','won:hasTextDescription'
  ])">
    Description
</h2>
<p class="post-info__details"
  ng-show="self.post.getIn([
    'won:hasContent','won:hasTextDescription'
  ])">
    {{ self.post.getIn(['won:hasContent','won:hasTextDescription']) }}
```

```
13 </p>
14 ...
```

These are either specified in an HTML-file and then later linked with a controller or are a string in the declaration of something called a “directive” (which are custom HTML tags or properties). Every template has a scope object bound to it and can contain expressions – e.g. those in curly braces – that have access to that scope object. For the code-example ~~in~~ above this means, that – in the HTML that the user gets to see – the curly braces will have been replaced by the result of `self.post.getIn(['won:hasContent','won:hasTextDescription'])` (the `getIn` is there because `post` is an immutable-js (see ref. “Immutable.js” n.d.) object). Practically every time the result of that expression changes, angular will update the displayed value. Basically every expression causes a “watch” to be created (this can also be done manually via `$scope.watch`). During what's called a “digest-cycle” it checks all of these watch-expressions for changes and then executes their callbacks, which in the case of the curly-braces causes the DOM-update. These digest-cycles are triggered by a lot of provided directives or if you call the function `$apply`. Some thought needs to be given to managing these to avoid bad performance and also because you can't call `$apply` while already in a digest-cycle.

Beyond the curly braces, angular also provides a handful of other template-utilities in the form of directives. For instance the property-directive `ng-repeat` allows iterating over a collection as follows:

Listing 4.2: Example ng-repeat usage

```
1 <div ng-repeat="el in collection">{{el.someVar}}</div>
```

Or, similarly, `ng-show="someBoolVar"` conditionally displays content.

Note that these template-bindings are bi-directional, i.e. the code in the template can change the the values in the model (the template's “scope”). Additionally, templates/directives can be nested within each other. By default, their scopes then use JavaScript's prototypical inheritance (see ref. “Inheritance and the Prototype Chain” n.d.) mechanism, i.e. if a value can't be found on the template's/directive's scope, angular will then go on to try to get it from the on the one wrapping it (and so on) This allows writing small apps or components where all data-flows are represented and all code contained in the template. For medium-sized or large apps however, the combination of bi-directional binding and scope inheritance, ~~-~~can lead to hard-to-follow causality, thus hard-to-track-down bugs and thus poor maintainability.

Also, using scope inheritance reduces reusability, as the respective components won't work in other contexts ~~anymore~~.

For all but the very smallest views/components the UI-update logic will be contained in Angular's controllers, however. They are connected with their corresponding templates via the routing-configuration if they're (top-level) views (more on that later in section 5.1.5) or by being part of the

same directive/component (see sec. 5.1.4). Controllers have access to their template's scope and vice versa.

Theoretically, it's possible to reuse controllers with different templates, but this can lead to hard-to-track-down and I'd advise against doing that. When nesting templates and thus their associated controllers, the latter form ~~something like a prototypical~~ inheritance chain: If a variable isn't found on the controller, respectively its scope, the default is to check on its parent and its parent's parent, etc, up to the root-scope. Note, that scopes can be defined as isolated (see sec. 5.1.4.4) – for views in their routing configuration and for directives in their declaration. This allows to avoid this behavior, which ~~I'd recommend~~ for predictability- and thus maintainability-reasons.

These scopes (models), templates (views) and controllers constitute a classical MVC-architecture (see section 4.1.1). However, angular also has the concept of services: ~~Essentially,~~ they are objects that controllers can access and that can provide utility functions, manage global application state or make HTTP requests to a web server. Controllers can't gain access to each other – except for nesting / prototypical inheritance – but they can always request access to any service (via dependency injection, i.e. listing the required service's name when initializing your controller or service). Examples of services are, for instance, `$scope` that, among others, allows registering custom watch-expressions with angular outside of templates, ~~like so:~~

Listing 4.3: Example of listening for changes of a variable in angular

```
1 var myApp = angular.module('myApp', []);
2 myApp.controller('PostController', function ($scope) {
3   $scope.post = { text: 'heio! :)' };
4   $scope.$watch('post.text', function(currentText, prevText) {
5     console.log('Text has been edited: ', currentText);
6   });
7 });
```

Another example for a service would be `linkeddata-service.js` that had already been written for the first won-owner-application prototype and that is still in use. It can be used to load and cache RDF-data[1].

Considering services, it's the angular framework can also be viewed through the lense of MVVM (see section 4.1.2), with templates as views, scopes and controllers as view-models and services as models or as proxies for models on a web server (as we did with `linkeddata-service.js`).

Note~~,~~ that Angular 1.x uses its own module system to manage directives, controllers and services. If you include all modules directly via `<script>`-tags in your `index.html`, this mechanism makes sure they're executed in the correct order. However, this also means, that if you want to combine all

[1] see section 3.2 for more on RDF

your scripts into one `bundle.js`[2] you'll have to specify the same dependencies twice – once for your bundling module system and once for Angular's, as can be seen in the code-sample below:

Listing 4.4: Duplicate dependency declaration: ES6-modules and Angular's dependency injection

```
/* es6 imports for bundling */

import angular from 'angular'
import createNeedTitleBarModule from '../create-need-title-bar';
import posttypeSelectModule from '../posttype-select';

//...

class CreateNeedController { /* ... */ }

//...

/* angular module declaration */

export default angular.module(
  /* module's name: */
  'won.owner.components.createNeed',
  [ /* module's dependencies: */
    createNeedTitleBarModule,
    posttypeSelectModule,
    // ...
  ])
  .controller(
    'CreateNeedController',
    [
      '$q', '$ngRedux', '$scope', // services for ctrl
      CreateNeedController // controller factory/class
    ])
.name;
```

As you can see writing applications in angular requires quite a few concepts to get started (this section only contains the essentials, you can find a full list in the angular documentation (see ref. "AngularJS: Developer Guide: Conceptual Overview" n.d.). Accordingly, the learning curve is rather steep, especially if you want to use the framework well and avoid a lot of the pitfalls for beginners, that otherwise result in hard to debug and unmaintainable code.

[2] Bundling for instance helps to reduce the number of HTTP-requests on page-load and thus its performance. It can be done by using a build-tool like Browserify, Webpack or JSPM plus a module system like AMD, CommonJS or the standardized ES6-modules (see ref. "ECMAScript 2015 Language Specification – ECMA-262 6th Edition" 2015, sec. 15.2.2. Imports).

4.1.4 React

React is a library that only provides the view and view-model of application architectures. It provides a mechanism to define custom components/HTML-tags (comparable to directives in Angular 1.X and webcomponents in general) as a means to achieve separation of concerns and code reusability. These components are stateful (thus the view-model) and contain their own template code, usually specified in the form of inline-HTML that is processed to calls to the React-libary (more on that below). For all but the smallest applications – where the state can be fully contained in the components – you'll need some extra architecture in addition to React, e.g. to handle the application-state or manage HTTP-requests and websockets. Usually the code to do these things are structured using the Flux (see section 4.1.5) or more recently the Redux-architectures (see section 4.1.6).

~~In any way~~, to get to the bottom of what distinguishes React, one should ~~first~~ start by talking about the big problem of the Document Object Model: When there is a large number of nodes on the screen, manipulating several quickly one after each other can take quite a while, causing the whole interface to noticeably lag as every changed node causes a reflow of the layout and rerendering of the interface. React is the first of a row of libraries to use a light-weight copy of the DOM (called "Virtual DOM". The idea is to only directly manipulate the VDOM and then apply the differential / cumulative change-set to the actual DOM in one go. This means a performance gain where multiple operations are applied to the same node or multiple nodes at the same time as React makes sure that the slow reflow and rerendering only happens once. From a development perspective, this diff'ing-process means, that there is no need to manage DOM state changes and intermediate states; the template code in the components can be written as if they were rendered completely new every cycle, i.e. only a direct mapping from data to desired HTML needs to be provided and React handles the changes to get there.

As a notable difference to Angular, React's data-flow is unidirectional, meaning a component can read the data it gets via its HTML-tag-properties, but it can't modify them. This is a useful guarantee, to avoid bugs like when you use a component, don't know it modifies its parameter variables (intentionally or as a bug) and thus influences your unsuspecting parent component as a side-effect. Intended child-to-parent communication can be done explicitly via events published by the child (or via callback functions).

4.1.5 Flux

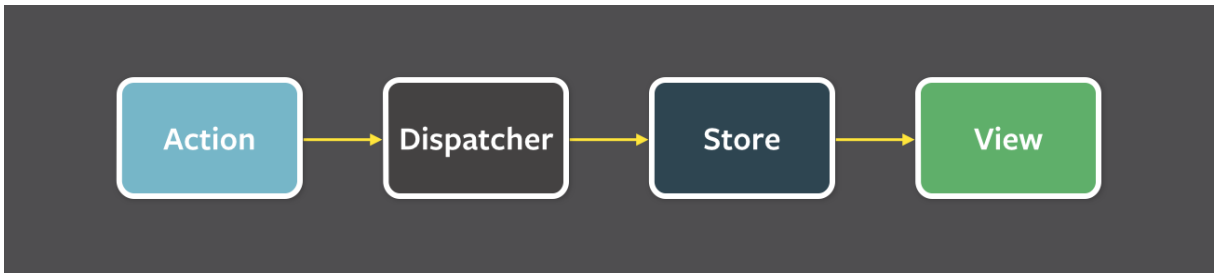

Figure 4.3: Core pipeline of the Flux-architecture (source: https://facebook.github.io/flux/img/flux-simple-f8-diagram-1300w.png, accessed 18.06.2018)

When you start reading about React you'll probably stumble across Flux (see fig. 4.3) rather earlier than later. It is the architecture popularized alongside of React and akin to MVC in that it separates handling input, updating the state and rendering the GUI.

However, instead of having bi-directional data-flow between the architectural components, Flux' is uni-directional and puts most of its business logic into the stores that manage the state. To give an example of a flow through this loop: For example, a user clicks on a map widget with the intent of picking a location. The widget's on-click method would then create an object, called an action, that usually contains type-field like `"PICK_LOCATION"` and any other data describing the user-interaction, such as geo-coordinates. That on-click method then goes on to pass the action object to the globally available dispatcher, which broadcasts it to all stores. Every store then decides for itself in what way it wants to update the data it holds. For instance, a `locationStore` could update the geo-coordinates it holds. The stores would then notify all components that are listening to them in particular, that their state has changed (but not in what way). The affected components, e.g. the map and a text-label below it, poll the store for the data and render themselves anew (as if it was the first time they were doing this) – e.g. the map would place a singular marker on the coordinates it gets from the store and the label would write out the coordinates as numbers.

Because of the last point – the components rendering themselves "from scratch" every time, i.e. them being an (ideally) state-less mapping from app-state to HTML – this architecture pairs well with React's VDOM.

When there is preprocessing that needs to be done on the data required for the action-object – e.g. we want to resolve the geo-coordinates to a human-friendly address-string – action-creators are the usual method to do so (see fig. 4.4). These are functions that do preprocessing – including HTTP-requests for instance – and then produce the action-objects and dispatch them.

Though being an architecture, i.e. a software-pattern, per se, usually one will use one of many ready

made dispatchers and also a store-prototype to inherit from, that will reduce the amount of boiler-plate code necessary to bootstrap a Flux-based application.

Stores can have dependencies among each other. These are specified with a function along the lines of `B.waitFor(A)`, meaning that the store B only starts processing the action once A has finished doing so. Managing these dependencies in a medium-sized to large application can be quite complex, which is where Redux (see below) tries to improve over Flux.

In general, using Flux profits from using immutable data-structures for the state (e.g. those of immutable-js (see ref. “Immutable.js” n.d.)). Without these, components could accidentally modify the app-state by changing fields on objects they get from the stores, thus having the potential for hard-to-track-down bugs.

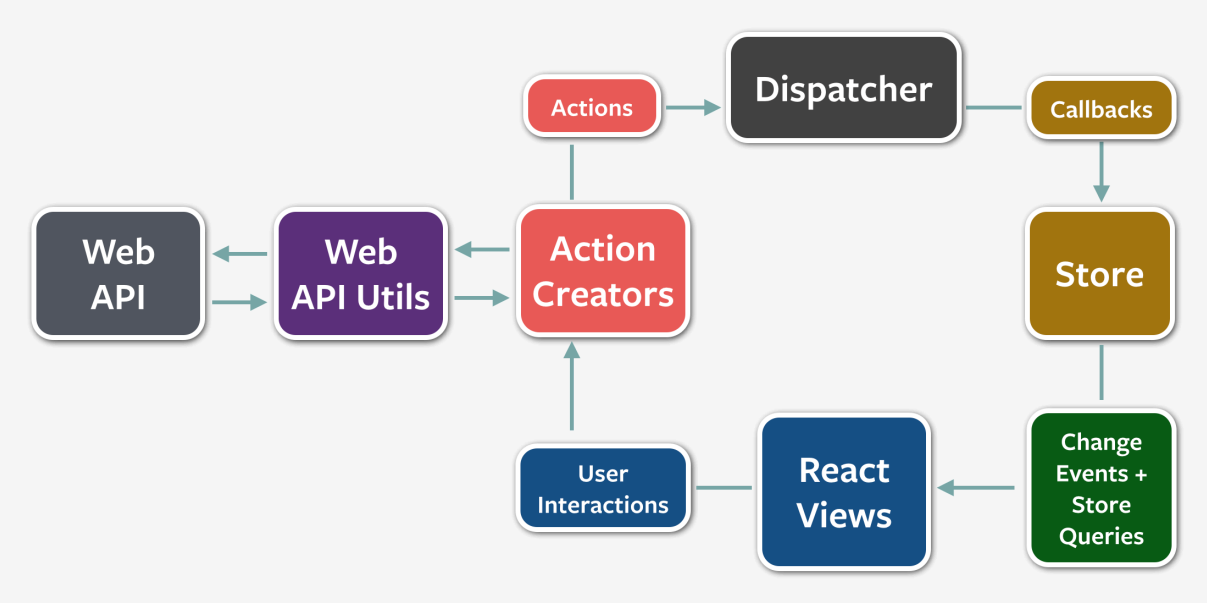

Figure 4.4: Full Flux-architecture incl. networking (source: https://facebook.github.io/react/img/blog/flux-diagram.png (accessed 2017-02-21))

4.1.6 Redux

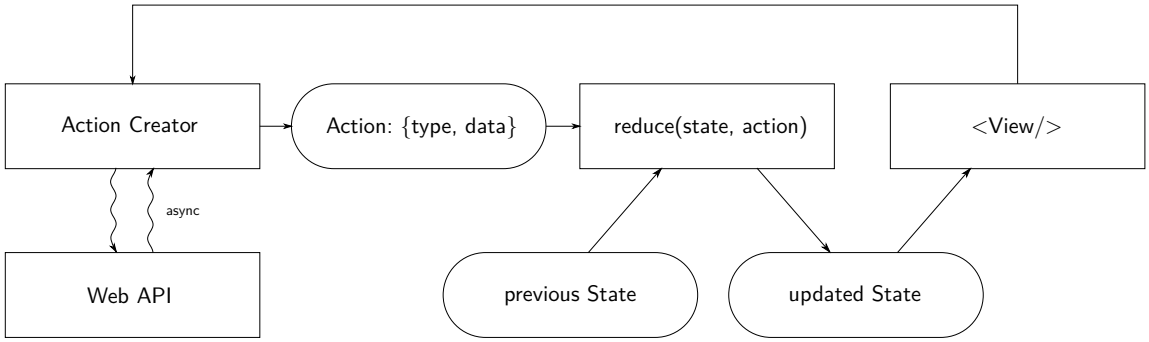

Figure 4.5: The redux-architecture

The developers/designers of Redux list the object-oriented Flux- (see above) and functional Elm-architecture (see below) as prior art (see ref. “Prior Art - Redux” n.d.). Redux mainly differs from Flux in eschewing the set of stateful stores, for the Elm-like solution of having a single object as app-state, that a single reducer-function `(state, action)=> state'` gets applied to for every new action, thus updating the state (see fig. 4.5). As such there formally is also no need for a dispatcher, as there is only a single function updating the state. However, in practice usually a very thin utility library is used, that manages state and reducers and provides a `dispatch`-function with which the reduction can be triggered. Separation of concerns – achieved in Flux via its larger number of stores – can be achieved in Redux by having the reducer function call other functions, e.g. one per subobject/-tree of the state.

As the simplest implementation of this architecture consists of only a single function and a component that feeds actions into it, the learning curve is relatively flat compared to Flux and almost flat compared to Angular’s MVC.

Redux profits from immutable data-structures for the app-state even more than Flux. The reducer function is supposed to be stateless and side-effect-free (i.e. pure). In this particular case this means that parts of the system, that still hold references to the previous state, shouldn’t be influenced by the state-update. If they want the new state, they’ll get notified through their subscription. Using immutable data guarantees this side-effect-freeness to some extent; nothing can prevent any point in the code from accessing the global `window`-scope in JavaScript though – however it’s very bad practice to do so and thus should universally be avoided. This property also means that you should try to move as much business logic as possible to the reducer, as it’s comparatively easy to reason about and thus debug. For all things that require side-effects (e.g. anything asynchronous like networking) action-creators are the go-to solution – same as in Flux.

4.1.7 Ng-Redux

Ng-Redux (see ref. Buchwalter and Januska 2018) is a framework that is based on the Redux-architecture and is designed to be used with Angular applications. The latter handles the Components/Directives and their updates of the DOM, whereas Ng-Redux manages the application state. In this combination, the frameworks binds functions to the angular controllers to trigger any of the available actions. Even more importantly, it allows registering a `selectFromState`-function that gets run after the app-state has been updated and the result of which is then bound to the controller. Ng-Redux also provides a middleware-system for plugins that can modify actions and state before and after a reduction step and can trigger side-effects. For example "thunk" provides a convenient way to handle asynchronicity in reducers, by passing a dispatch-function to them that they can call at any later point in time (e.g. when an HTTP request returns). Another example is the `ngUiRouterMiddleware` that allows interfacing with browsers' history-API (and thus URL in the URL-bar). That middleware also conveniently adds this information (e.g. current route and route-parameters) to the application state, where components can retrieve them like any other part of the state.

4.1.8 Elm-Architecture

Elm (see ref. "Elm Language" n.d.) is a functional language whose designers set out to create something as accessible to newcomers as Python or JavaScript. It can be used to build front-end web applications. The original Elm-architecture was based on functional reactive programming – i.e. using streams/observables like CycleJS' MVI (see below) that it inspired as well – but they have since been removed to make it more accessible to newcomers. The current architecture (see ref. "The Elm Architecture · an Introduction to Elm" n.d.), in its basic form, requires one to define the following three functions and then, in the main function, pass these three to one of several startup functions (e.g. `Html.beginnerProgram`):

- `model : Model`, that initializes the app-state.
- `update : Msg -> Model -> Model` is a function that takes a `Msg` and a `Model` and returns an (updated) `Model`. This function performs the same role as `reduce` in Redux, with `Msg`s in Elm being the equivalent to actions in Redux.
- And lastly, `view : Model -> Html Msg` to produce the HTML from the model. In Redux a library like react or, as is the case of the work at hand, angular would be used to do this rendering of `Model` to HTML.

As Elm is a pure (i.e. side-effect-free) language, these can't handle asynchronity yet (e.g. HTTP-requests, websockets) or even just produce random numbers. The full architecture, that handles these side-effects, looks as follows (and is run via `Html.program`):

- `init : (Model, Cmd Msg)` fulfills the same role as `model`, but also defines the first `Cmd`. These commands allow *requesting* for side-effectful computations like asynchronous operations (e.g. HTTP-requests) or random number generation. The result of the `Cmd` is fed back as `Msg` to the next `update`.
- the function `update : Msg -> Model -> (Model, Cmd Msg)`, in this variant of the architecture, also returns a `Cmd` to allow triggering messages ("actions" in redux-terms) depending on user input or the results of previous `Cmd`s. This allows keeping all of the business-logic in the `update`-function (as compared to Flux'/Redux' action-creators) but trades off the quality, that every user-input or websocket message can only trigger exactly one action and thus exactly one update (thus making endless-loops possible again)
- `subscriptions : Model -> Sub Msg` allows to set up additional sources for `Msg`s beside user-input, things that *push*, e.g. listening on a websocket.
- `view : Model -> Html Msg` works the same as in the simple variant.

4.1.9 CycleJS MVI

CycleJS is a framework based on "functional reactive programming" (short FRP). The framework's is following a Model-View-Intent architecture that is similar to the Redux- and (original) Elm-architectures.

As an FRP-based framework, it uses observables/streams of messages for its internal data-flows. These can be thought of as as Promises that can trigger multiple times, or even more abstract, as pipes that manipulate data flowing through. These observables/streams can be composed to form a larger system. The integral part developer's using the framework need to specify is a function `main(sources)=> ({ DOM: htmlStream})` (see fig. 4.5) that takes a driver "`sources`" like the DOM-driver that allows creating stream-sources (e.g. click events on a button). One would then apply any data-manipulations in the function and return a stream of virtual DOM. In the very simple code-example given below, for every input-event a piece of data/a message would travel down the chained functions and end up as a virtual DOM object. This `main`-function is passed to the `run`-function to start the app. A simple "hello world"-application for CycleJS could look like the following:

Listing 4.5: Example CycleJS app

```
1 import {run} from '@cycle/xstream-run';
2 import {div, label, input, hr, h1, makeDOMDriver} from '@cycle/dom';
3
4 function main(sources) {
5   const sinks = {
6     DOM: sources.DOM.select('.field').events('input')
```

```
7         .map(ev => ev.target.value) // get text from field
8         .startWith('') // initial value / first stream-message
9         .map(name =>
10          div([
11            label('Name:'),
12            input('.field', {attrs: {type: 'text'}}),
13            hr(),
14            h1('Hello ' + name),
15          ])
16        )
17    };
18    return sinks;
19  }
20
21  run(main, { DOM: makeDOMDriver('#app-container') });
```

For more complex applications, an architecture similar to Redux/Elm, called “Model-View-Intent” is recommended. For this, the stream in `main` is split into three consecutive sections:

- **Intent**-functions that set up the input streams from event-sources (e.g. DOM and websockets) and return “intents” that are equivalent to Flux’/Redux’ actions and Elm’s messages.
- The **model**-stage is usually implemented as a function that is `reduce`’d over the model (equivalent to how Redux deals with state-updates)
- And lastly the **view**-stage takes the entire model and produces VDOM-messages.

Separation of concerns happens by using sub-functions or splitting the stream at each stage (or starting with several sources at the intent-stage) and combining them at the end of that respective stage. Thus at the boundary between each of the three stages all streams are unified into one stream that is connected to the next stage.

5 Suggested Solution

As already mentioned in the problem description (chapter 3), the rework and restructuring started with a codebase using Angular (see section 4.1.3), all modules included one-by-one in an `index.jsp`, and some bootstrap-theme for styling. Bugs were hard to solve due to the "grown" code-base and the somewhat ambiguous architecture stemming both the wide range of concepts in angular that required understanding and best-practices as well as our grasp of them. Additionally, the visual style was neither polished nor projecting a unique identity.

As part of a research-project together with our partner Meinkauf, the Researchstudio Smart Agent Technologies was tasked with developing a platform-independent mobile application and used Ionic (see ref. "Ionic Framework" n.d.), i.e. a tooling default, that at the time consisted of Phonegap (see ref. "PhoneGap" n.d.), Angular 1.x, SCSS (see section 5.3.2), ionic-specific CSS and its own command-line-tool. This project presented a good opportunity to try out a different architecture, to deal with the ambiguities and maintenance problems we were experiencing with the Web of Needs owner-application.

5.1 Architecture

We're using a variation of the redux-architecture (see sections 4.1.6 and 4.1.7 respectively) for the won-owner-webapp JavaScript-client.

This section will document in what ways our architecture diverges from or builds on top of basic redux, as well as list experiences and style-recommendations derived from using it.

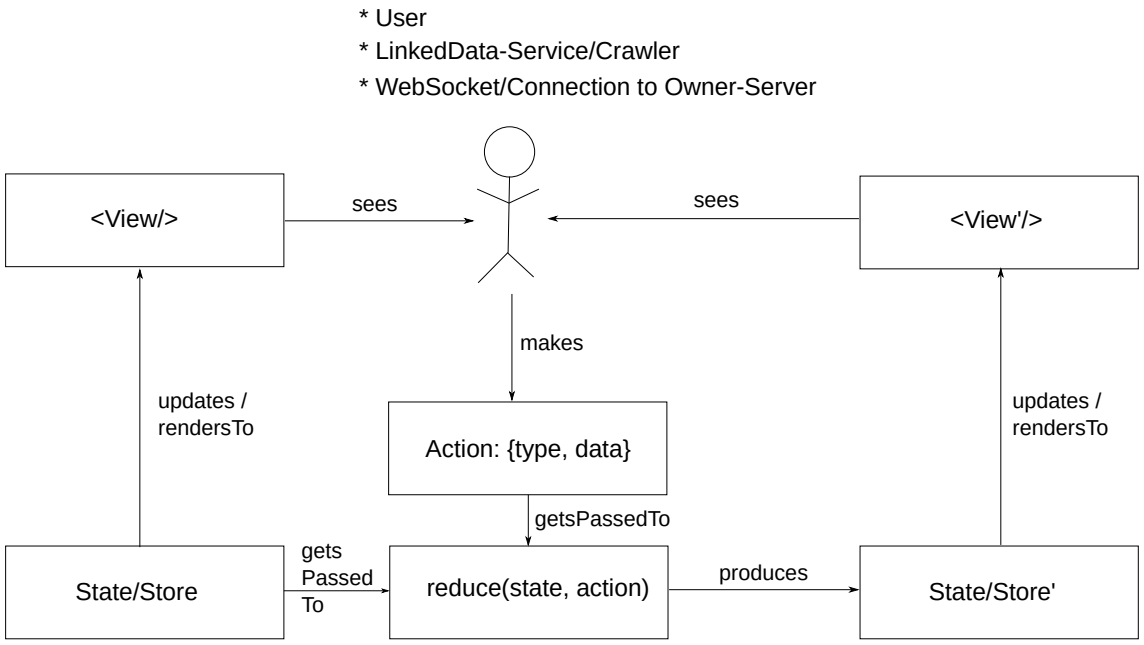

Figure 5.1: Redux architecture in client-side owner-app

5.1.1 Action Creators

Can be found in `app/actions/actions.js`

Anything that can cause **side-effects** or is **asynchronous** should happen in these (tough they can also be synchronous – see `INJ_DEFAULT`) They should only be triggered by either the user or a push from the server via the `messagingAgent.js`. In both cases they cause a **single**(!) action to be dispatched and thus passed as input to the reducer-function.

All actions are declared in the `actionHierarchy`-object in `action.js`. From that two objects are generated:

- `actionTypes`, which contains string-constants (e.g. `actionTypes.drafts.change.title === 'drafts.change.title'`)
- `actionCreators`, which houses the action creators. For the sake of injecting them with ng-redux, they are organized with `__` as separator (e.g.
- `actionCreators.drafts__change__title('some title')`

The easiest way to create actions without side-effects is to just place an `myAction: INJ_DEFAULT`. This results in an action-creator that just dispatches all function-arguments as payload, i.e.

```
actionCreators.myAction = argument => ({type: 'myAction', payload: argument
})
```

Actions and their creators should always describe **high-level user stories/interactions** like `matches.receivedNew` or `publishPost` (as opposed to something like `matches.add` or `data.set`) Action-creators encapsule all sideeffectful computation, as opposed to the reducers which (within the limits of JavaScript) are guaranteed to be side-effect-free. Thus, we should do **as much as possible within the reducers**. This decreases the surprise-factor, coupling and bug-proneness of our code and increases its maintainability.

5.1.2 Actions

They are objects that serve as input for the reducer. Usually they consist of a type and a payload, e.g.:

Listing 5.1: Example action object

```
{
  type: ”needs.close”
  payload: {
    ownNeedUri: ”https://node.matchat.org/won/resource/need/1234”
  }
}
```

These should describe high-level interactions from the user (or server if initiated there).
A full list of action-types, used in the owner-application
can be found in `app/actions/actions.js`.

5.1.3 Reducers

Can be found in `app/reducers/reducers.js`

These are **side-effect-free**. Thus, as much of the implementation as possible should be here instead of in the action-creators to profit from this guarantee and steer clear of possible sources for bugs that are hard to track down.

Usually they will consist of simple switch-case statements. A simple reducer that would keep track of own needs could (in-part) look as follows:

Listing 5.2: Simple example reducer.

```
import { actionTypes } from '../actions/actions';
import Immutable from 'immutable';
```

```
import won from '../won-es6';

const initialState = Immutable.fromJS({
    //...
    ownNeeds: {},
});

export default function(allNeeds = initialState, action = {}) {
  switch(action.type) {
    case actionTypes.logout:
      return initialState;

    case actionTypes.needs.close:
      return allNeeds.setIn([
        "ownNeeds", action.payload.ownNeedUri, 'won:isInState'
      ], won.WON.InactiveCompacted);

    //...

    default:
      return allNeeds;
}
```

5.1.4 Components

They live in `app/components/`.

Top-level components (views in the angular-sense) have their own folders (e.g. `app/components/create-need/` and are split in two files). You'll need to add them to the routing (see below) to be able to switch the routing-state to these.

Non-top-level components are implemented as directives. A very simple demo-component, that would render the title and description of a need to the DOM-tree and allow closing it (i.e. making it unreachable to contact requests) via a click on "[CLOSE]", would look as follows:

Listing 5.3: Example component.

```
import angular from 'angular';
import 'ng-redux';
import { actionCreators }  from '../actions/actions';
import { attach } from '../utils.js'
import { seeksOrIs, connect2Redux } from '../won-utils'
```

```
6
7  // angular utilities required to integrate the component
8  // into the redux architecture via the `connect2Redux`-
9  // function.
10 const serviceDependencies = ['$ngRedux', '$scope'];
11
12 // factory function for the directive:
13 function genComponentConf() {
14
15   const template = `
16     <h1>{{ self.needContent.get('dc:title') }} [DEMO]</h1>
17     <p>{{ self.needContent.get('won:hasTextDescription') }}</p>
18     <a ng-click="self.needs__close(self.need.get('@id'))">
19         [CLOSE]
20     </a>
21   `;
22
23   class Controller {
24     constructor() {
25
26       // does `controller.<serviceName> = <serviceName>`
27       // for all services injected via `arguments`
28       attach(this, serviceDependencies, arguments);
29
30       const selectFromState = (state) => {
31         const need =
32           state.getIn(['needs', 'ownNeeds', this.needUri]);
33
34         // need and needContent will be bound to the
35         // controller (=scope) and thus be available
36         // in the template.
37         return {
38           need,
39           needContent: need && seeksOrIs(need),
40         }
41       };
42       connect2Redux(
43         selectFromState, // result will be bound to `this`
44         actionCreators, // will be bound to `this`
45         ['self.needUri'], // component property used in select
46         this // the controller
47       );
48     }
```

```
49    }
50    Controller.$inject = serviceDependencies;
51
52    // directive configuration:
53    return {
54      scope: { needUri: '=' }, // available property on HTML-tag
55
56      controller: Controller,
57      template: template
58
59      restrict: 'E', // directive only usable as HTML-tag
60      controllerAs: 'self', //ctrl available as `self` in template
61      bindToController: true, //ctrl is scope for template
62    }
63  }
64
65  export default angular.module(
66    'won.owner.components.demoComponent',
67    [ /* here: any dependencies using angular */ ]
68  )
69  .directive('wonDemoComponent', genComponentConf)
70  .name
71  // ^ exported name used by importing component in dependency-array
```

The component can then be used by a **parent component** via:

Listing 5.4: Usage in parent component.

```
1  // ...
2
3  import demoComponentName from './demo-component.js'
4
5  // ...
6
7  function genComponentConf() {
8    const template = `
9      <h1>All Owned Needs</h1>
10     <won-demo-component
11       ng-repeat="uri in self.needUris"
12       needUri="uri">
13     </won-demo-component>`
14
15   //...
```

```
16 }
17
18 export default angular.module(
19   'won.owner.components.demoParent',
20
21   // so angular knows to run the child first:
22   [ demoComponentName ]
23 )
24 .directive('wonDemoParent', genComponentConf)
25 .name
```

As you can see, there is quite a bit boiler-plate required by angular. All that is required by (ng-)redux is the listener to the state set up by `connect2Redux`.

Among the boiler-plate there is a few details I'd like to point out, that make working with Angular 1.X a lot less painful. I'll go through it top-to-bottom in the following sub-sections.

5.1.4.1 Service Dependencies

The `serviceDependencies` lists the angular services, that will be passed to the constructor of the directive. Assigning that array with the dependency-names to the Controller class via `$inject` makes sure Angular does just that, even if the code is minified. Per default angular reads the names of the arguments of the constructor, but during minification that information is lost. By setting `strictDi: true` when starting up angular in `app/app_jspm.js` we make sure angular complains if the injection array isn't there. The `attach`-function then takes the constructor's arguments (i.e. the injected service dependencies) and assigns them as properties to the controller-object.

5.1.4.2 Template Strings

The template strings (`const template = '...'`) describe the HTML that the user gets to see. When a component is first rendered, angular parses the string and generates the required HTML, starts up any required child-components and directives, and evaluates any expressions in double curly braces and then replaces them with the result of that respective expression. E.g. `<h1>{{ self.needContent.get('dc:title')}} [DEMO]</h1>` might become `<h1>Couch to give away [DEMO]<h1>`. It also makes sure that whenever these expressions change, the DOM is updated. To do this it sets up a so called "watch" per expression. Every time a `$digest`-cycle is triggered, all watch-expressions are evaluated and necessary changes to the DOM made one at a time (this is also what makes Angular 1.x terribly imperformant compared to virtual-DOM frameworks like React and the Elm-runtime, that batch updates). `$ngRedux` makes sure a `$digest`-cycle is triggered

every time the redux state has been updated. Managing these `$digest`-cycle can be a bit of a hassle at times and the occasional source of a hard-to-track-down bug.

Also, in the template, the `ng-click="self.needs__close(self.need.get('@id'))"` sets up a listener for a click event on the element, that executes the code in the double quotes, in this case it calls the action-creator `needs__close` with a specific need-uri, that makes an HTTP-request to the server and on success creates an action-object and dispatches it, thus triggering a state-update.

5.1.4.3 Listening for State-Changes

Ng-redux provides us with the utility function `$ngRedux.connect(selectFromState, actionCreators)(controller)` that `connect2Redux` uses internally. What it does is to set up a listener on the state managed by ng-redux. Every time the state is updated, `selectFromState` is run on it. Its return object is then assigned property-by-property to the `controller`. As a convenience-feature, the functions in `actionCreators` are wrapped with a call to `$ngRedux.dispatch` and also get assigned to the `controller` when the component is initialized. Otherwise it would be necessary to write `self.$ngRedux.dispatch(self.someAction(...))` everywhere in the component that the action is triggered.

Note, that `selectFromState` can be used to transform the data, that should be stored in a normalized, redundancy-free fashion in the state, into something that is easier to consume in the state. Frequently used selection-functions can be found in `app/selectors.js`. Many of these use reselect[1] that allows caching the results of computations until their dependencies change. This way, if e.g. the list of connections with their related needs and events is needed by multiple components on the screen, the filter and group operations are only run once (instead of once per component selecting that data).

As a secondary function, `connect2Redux` also unregisters any listeners and watches when the component is removed.

5.1.4.4 Essential Component Boilerplate

Some hard lessons went into using the following in the directive configuration:

Listing 5.5: Essential directive configuration boilerplate.

```
1 {
2   scope: { },
3   //...
```

[1] https://github.com/reactjs/reselect

```
4    restrict: 'E',
5    bindToController: true,
6    controllerAs: 'self',
7  }
```

Of these, the first is the most important. It allows specifying custom properties for the component. However, even when there are no properties, one should always specify an (empty) scope object. This "isolates" the scope in angular-terms. Without it, when a property is requested (e.g. in the template) and it's not found on the directive itself, angular will continue to look for the property in the scope of the enclosing directive or view. Not only that it will read data from there, when variables are assigned, it will also write there(!). Thus, when you assign to a variable that reads the same, as a parent component's, you'll change the value there as well, causing ~~(~~almost certainly unintended~~)~~ consequences there.

The `restrict` ensures that the directive is only used as HTML-tag. Usually it would also be usable as HTML-tag-property or even class. Unless you're doing something along the lines of `ng-click` (that sets up click-handlers on an arbitrary HTML-tag) I wouldn't recommend using the property and definitely would always advise against using directives via class names. Neither of these is suited well for having inner HTML.

Of the other options `bindToController` ensures that the controller is used as scope, thus avoiding juggling two JavaScript-objects and wondering on which the data is. `controllerAs` binds exposes the controller to the template as `'self'` (in this case).

5.1.5 Routing

We use ui-router (see ref. *Ui-Router* n.d.) and in particular the redux-wrapper (see ref. Fenton [2015] 2018) ~~for it.~~

Routing(-states, aka URLs) are configured in `configRouting.js`. State changes are triggered via the asynchronous action creator `actionCreators.router__stateGo(stateName)`. The current routing-state and -parameters can be found in our app-state:

Listing 5.6: Routing parameters in redux state.

```
1  $ngRedux.getState().get('router')
2  /* =>
3  {
4    currentParams: {...},
5    currentState: {...},
6    prevParams: {...},
7    prevState: {...}
```

```
8 }
9 */
```

5.1.6 Server-Interaction

As specified in the problem description (in particular sec. 3.3.2), the application needs to fetch linked-data via a REST-API and receive updates via a web-socket. This section covers how this is done in the implemented architecture.

For any **REST**-style requests, `fetch(...).then(data => {...dispatch(...); })` is used in action-creators. If they're **linked-data-related**, the utilities in `linkeddata-service-won.js` are used. They do standard HTTP(S) but make sure to cache as much as possible via the local triplestore[2]. However, in the future this custom caching layer can be replaced by using HTTP2 to load the large number of RDF-documents[3] in one round-trip and let the browser-chache handle repeated requests. One advantage of the triple-store is that it stores the RDF in its natural state and additional data can just be "poured"/merged it. Anything, e.g. data related to a need, can be retrieved from the store using a SPARQL-query[4] and molded into a desired JSON-data-structure via JSON-LD-framing[5]. One price here however is one of performance – some SPARQL-queries performed very badly and needed to be replaced by work-arounds – and another price is complexity, as the custom caching logic written to avoid unnecessary HTTP-requests yet keep the store in synch with the node-server is a frequent source of hard to track down bugs.

JSON-LD send is **send** to the server **via a websocket-connection**. For this case-statements for the respective action are added in `message-reducers.js` that adds them to the message-queue in `state.getIn(['messages', 'enqueued'])`. The messaging agent picks theses up and pushes them to the websocket it manages.

New messages are **received via the web-socket**. This allows the server to push-notify the client. The messaging agent contains a series of handlers for different message-types that then dispatch corresponding actions. It conceptually acts similar to a user faced with output from the system and capable of returning input, but towards the network (see fig. -@#fig:adapted-redux). It can trigger actions, like a user could via e.g. button presses, and does so when it receives messages via the web-socket. On the other side, it sees the result of the state, like a user would – except it doesn't get to see rendered DOM but rather it's message queue, that it forwards to the owner-server.

[2] RDFstore.js: https://github.com/antoniogarrote/rdfstore-js
[3] ad large number of documents: when your entire state contains of a single contact request, you still need to load 6 documents, in 3-5 round-trips: your need, its connection container, the connection to the other person's need, its event container, the event, and lastly the other person's need.
[4] SPARQL Protocol and RDF Query Language: W3C recommendation at https://www.w3.org/TR/sparql11-query/
[5] JSON-LD-framing: W3C editor's draft at https://w3c.github.io/json-ld-framing/

5.2 Views and Interactions

For the sake of completeness and to illustrate the usefulness, this section will give a very brief overview over the GUI built with this works' architecture and tooling and how it ties into the architecture at large. For a detailed code example of a simple component see fig. 5.3 in sec. 5.1.4.

The figures in this section illustrate the process of authoring a new need fig. 5.2, getting matches to it fig. 5.3, making a contact request fig. 5.4, that is then accepted by the other person fig. 5.5, and chatting with them fig. 5.6. This is the core workflow in the WoN-ownerapplication-prototype and cover the interactions layed out in the problem-description-subsection 3.3.1. Note, that the in the given example the person using application was already logged in (using an anonymous account). If they weren't they could either start by signing up via a standard signup-form or just going through the same process of authoring a need, that will create an anonymous account for them as a side-effect and adding a private ID to the URL, so they can bookmark it via their browser.

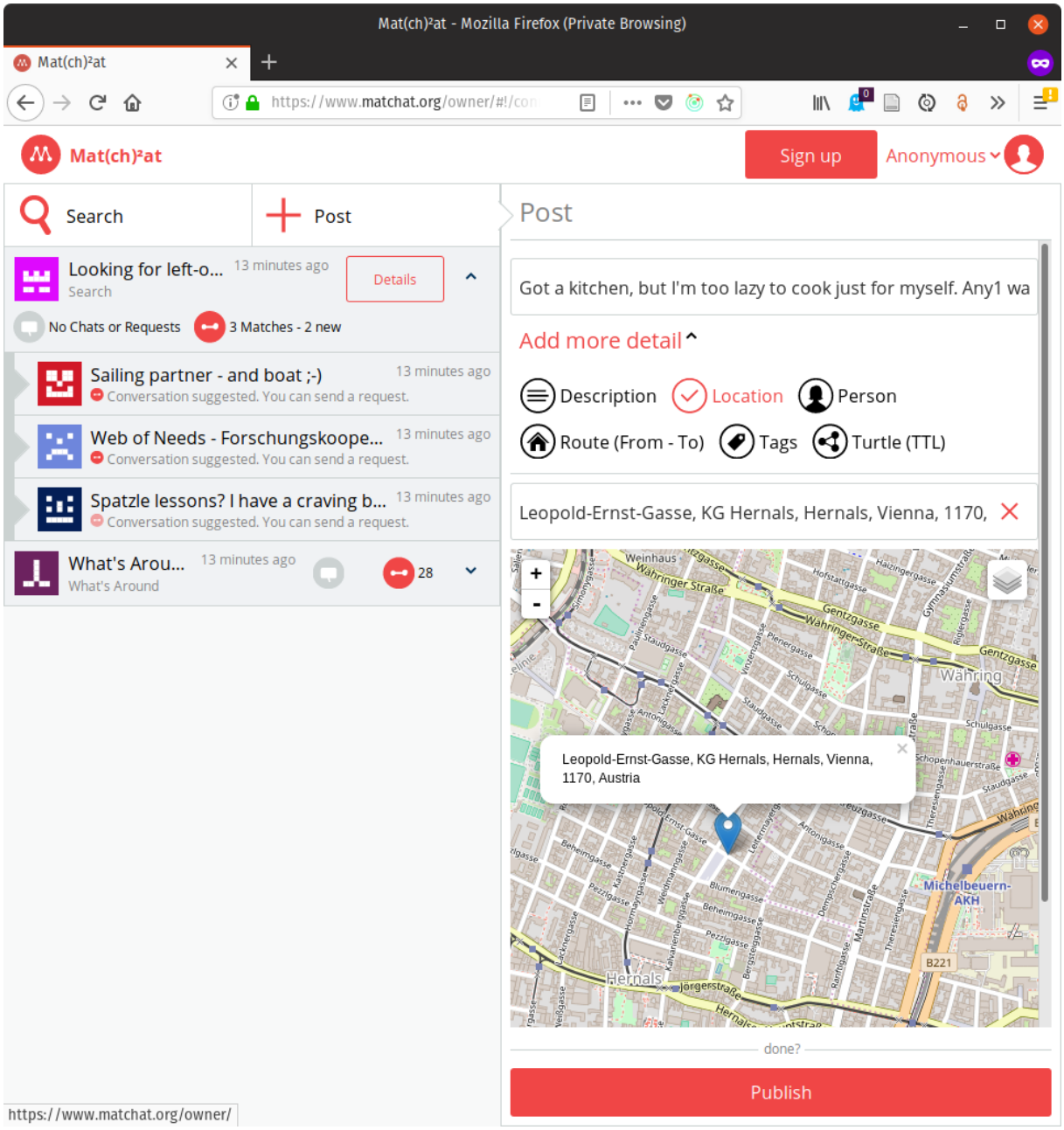

Figure 5.2: Authoring a need (right half of the screen) with an anonymous account (top-right) and some previously created needs (left half).

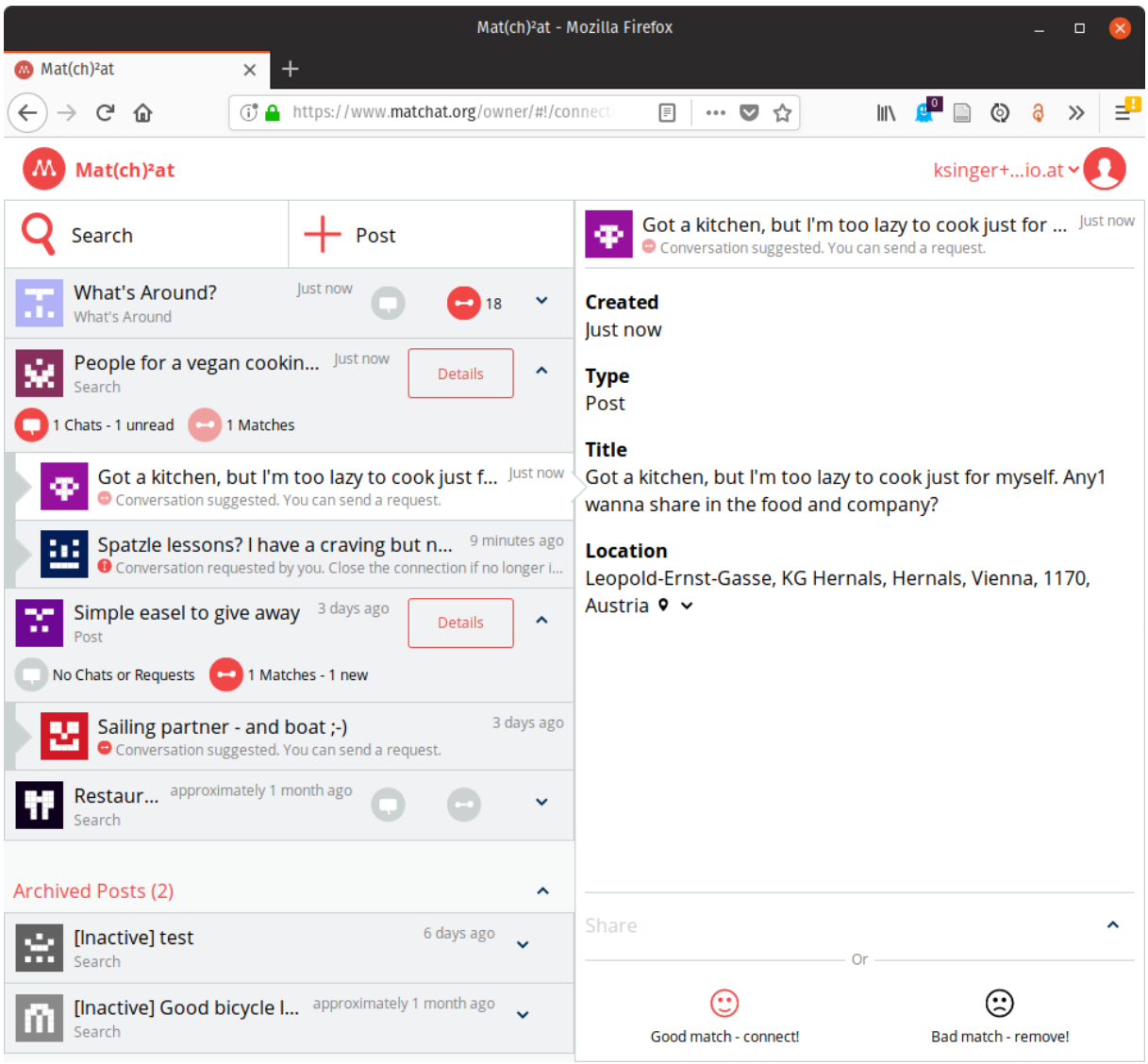

Figure 5.3: Got the match (white card on the left) on another account (top-right). Viewing its details (right half) with option to send a contact request (bottom-right). There's also another connection with a send contact request to the owner of "Spatzle lessons" on the left and some archived posts on the bottom left.

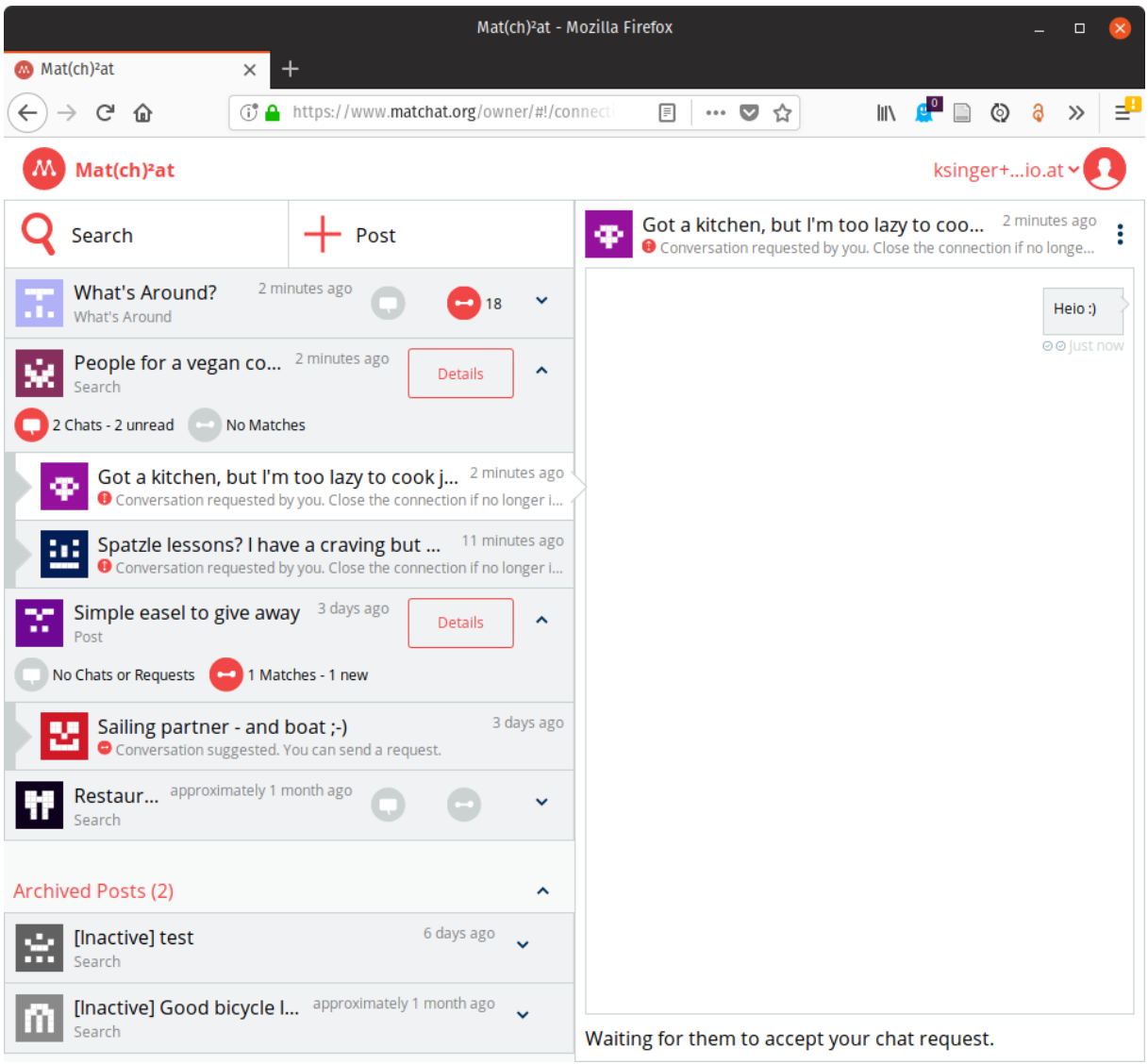

Figure 5.4: Right-side after making a contact request: waiting for the other person (the anonymous account) to accept (or decline) the contact request.

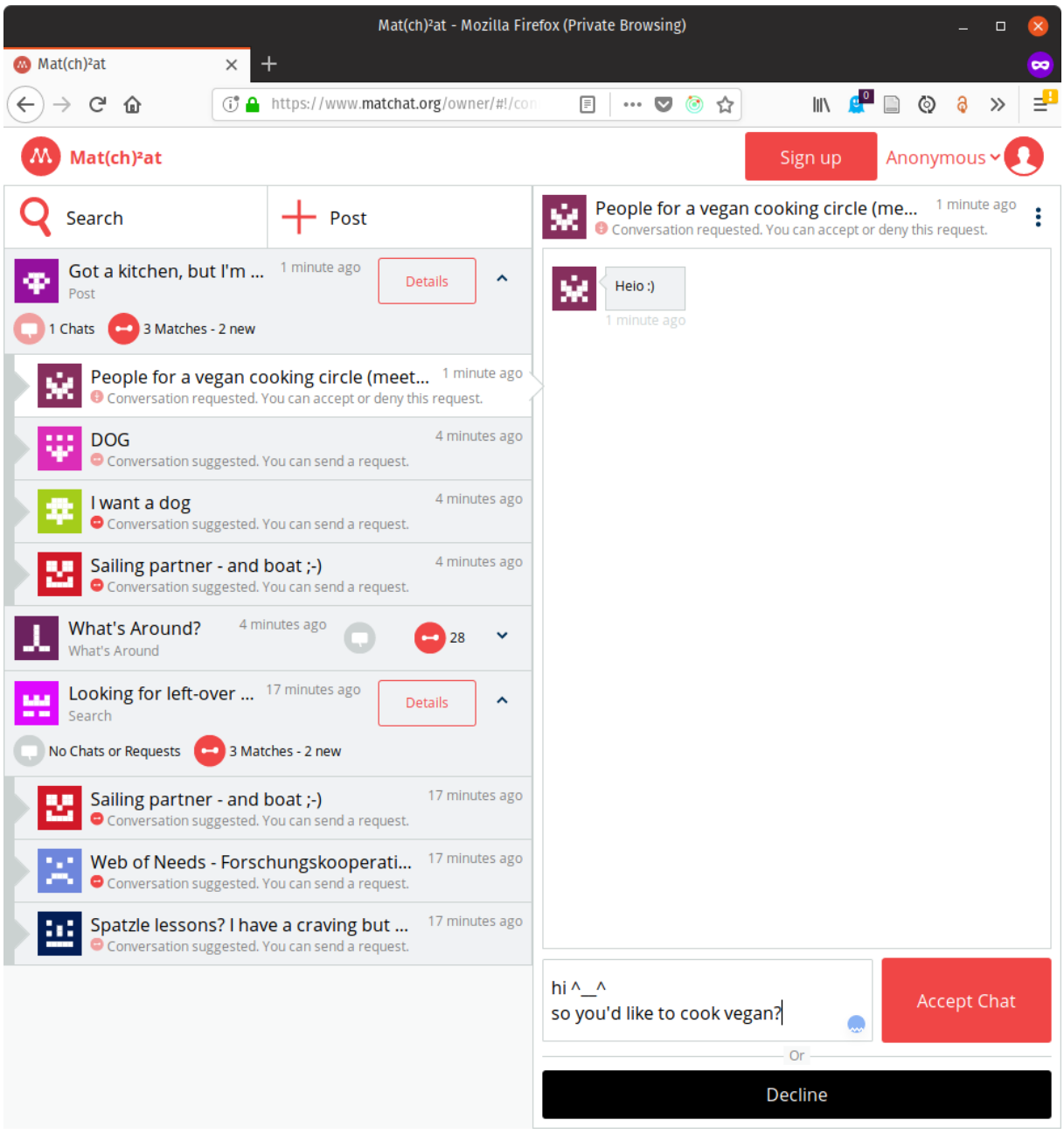

Figure 5.5: Right-side of the person receiving the contact request: they can eiher accept or decline the contact request. They're in the progress of writing a message to send along with the accept.

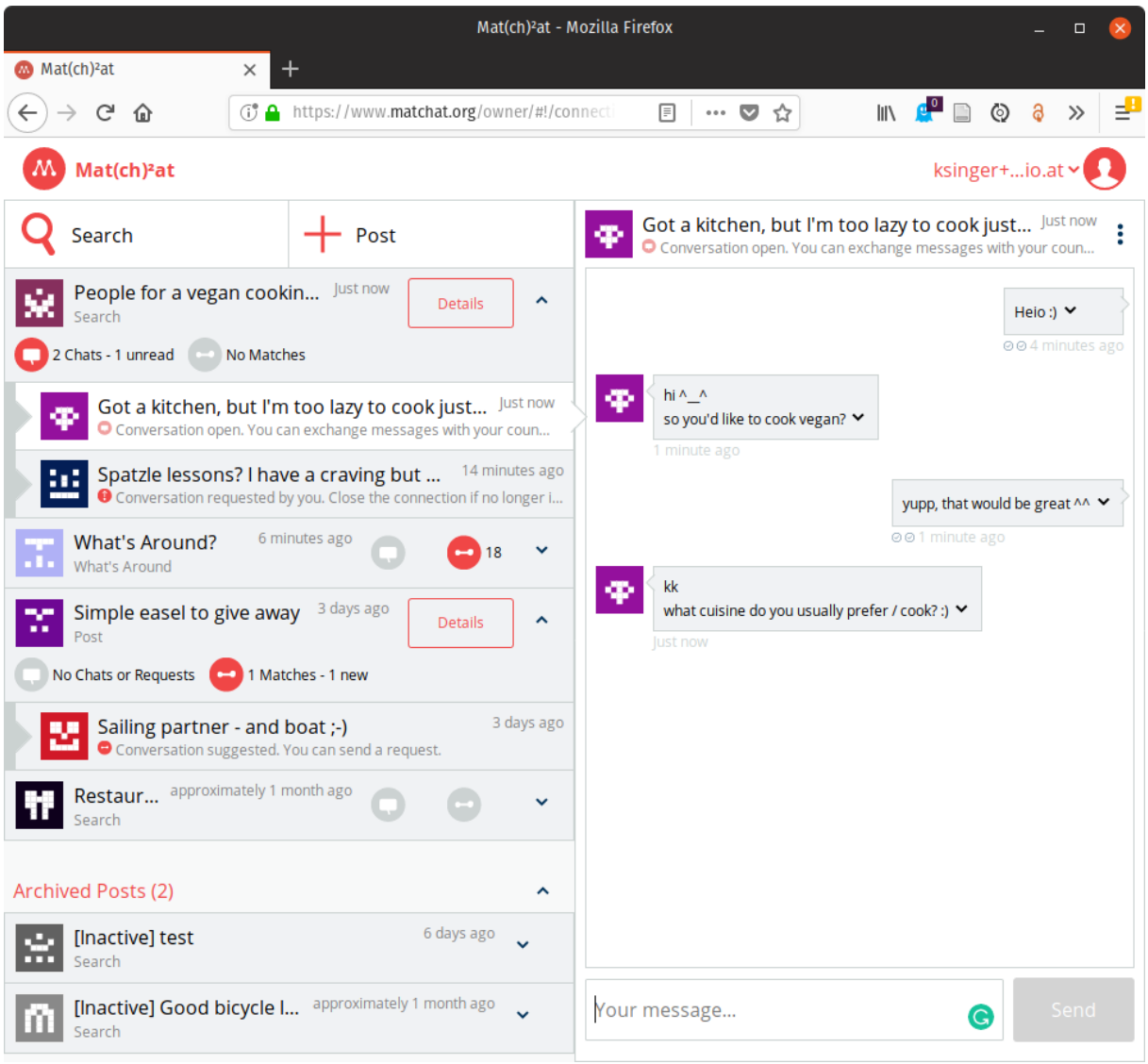

Figure 5.6: Right-side: a bit later in the conversation – several messages have been sent.

5.3 Tooling

5.3.1 ES6

As mentioned in section 3.3.2, one of the goals was to improve the quality of the code, its readability and authoring support, especially regarding expressiveness, robustness, conciseness and bug prevention. For this it seemed natural to start using features from the latest JavaScript standard (at the time of writing ES6, also known as ES2015, optionally plus experimental features). Amongst others, this

would give us access to:

- ES6-style variable declarations (e.g. `const x = 2; let y = 3; y = 1`) to prevent accidental variable overwriting through redeclaration.
- Native promises (e.g. `asynchronousFunction().then(x => /*...*/)`) to handle asynchronicity.
- Async-await (e.g. `const x = await asynchronousFunction()`) to write asynchronous code as if it were not.
- Arrow-functions (e.g. `x => 2*x` instead of `function(x){ return 2*x }`) as more concise syntax.
- Destructuring assignment (e.g. `const { a, b } = someObj`) to get fields of objects, especially when having multiple return arguments from a functions.
- Spread operators (e.g. `[1, 2, ...anotherArray, 4]` or `{a: 1, ...anotherObject}`) to concisely copy properties.
- ES6-Modules (e.g. `import { someFn } from './moduleA.js`) as a standardized module syntax (instead of CommonJS and ASM)
- etc

As some of these features weren't fully supported by all browsers cross-compilation to older JavaScript versions was necessary. Also, the module-syntax required a bundler, that combines the JavaScript modules into one file, that can be included via a `<script>`-tag.

5.3.1.1 ES6-style Variable Declarations

ES6 also gives us `const`-variables, that throw errors when trying to accidentally reassigning to them, and `let`-variables that behave like variable-declarations in other C-style-languages would. In contrast, `var`-declarations are always scoped to the parent-function not the containing scope, e.g. in an if, and can be silently re-declared, potentially causing bugs in unsuspecting developer's hands.

5.3.1.2 Promises

These are a fix to the so-called callback hell, i.e. code like the following:

Listing 5.7: Callback hell.

```
won.login(credentials, function(error, userInfo) {
  if(!error) {
    won.getOwnedNeedUris(function(error, needUris) {
      if(!error) {
        for(int i = 0; i < needUris.length; i++) {
```

```
6             var needs = [];
7             won.getNeed(needUris[i], function(error, need) {
8               if(!error) {
9                 needs.push(need);
10                if(needs.length === needUris.length) {
11                  // all callbacks have resolved, pass the result on to
                        redux
12                  dispatch(actionCreators.initialPageLoad({
13                    needs: needs,
14                    userInfo: userInfo,
15                  }));
16                }
17              }
18              else {
19                handlePageLoadError(error);
20              }
21            }
22          }
23        } else {
24          handlePageLoadError(error);
25        }
26      })
27    } else {
28      handlePageLoadError(error);
29    }
30  })
```

With promises, arrow-functions[6] and the enhanced object literals[7] this looks like:

Listing 5.8: Same example but using promises

```
won.login(credentials)
.then(userInfo =>
  won.getOwnedNeedUris()
  .then(needUris =>
    Promise.all(
      needUris.map(needUri => won.getNeed(needUri))
    )
  )
  .then(needs =>
```

[6] a conciser function syntax with slightly different behavior regarding the `this`-keyword, i.e. it doesn't rebind it to the local scope, making them good for use within methods of ES6-style classes (see refs. "Arrow Functions" n.d.; and "ECMAScript 2015 Language Specification – ECMA-262 6th Edition" 2015, sec. 14.2 Arrow Function Definitions).

[7] `{needs, userInfo}` as syntactic-sugar for `{needs: needs, userInfo: userInfo}`

```
    dispatch(actionCreators.initialPageLoad({needs, userInfo}))
  )
)
.catch(error => handlePageLoadError(error))
```

This is already a lot conciser and more expressive. If error occurs at any point the control-flow will jump to the next catch in the promise-chain and `Promise.all` makes sure all needs finish loading before continuing. However, notice that the later access to `userInfo` requires nesting the Promises again.

Before the rework, the code-base was already, occasionally using Angular's `$q` as polyfill that was providing the same functionality in different places. However, as angular-service, `$q` required to keep all code, even asynchronous utility-functions, within angular-services.

5.3.1.3 Async-Await

While promises are a great way of managing asynchronicity in our code, async-await, a form of syntactic sugar for promises, allows further simplifications. The promise-based code-example from above (fig. 5.8) can be written using async-await as follows:

Listing 5.9: Same example but using async-await

```
try {
  const userInfo = await won.login(credentials);
  const needUris = await won.getOwnedNeedUris();
  const needs = await Promise.all(
    needUris.map(needUri => won.getNeed(needUri))
  );
  dispatch(actionCreators.initialPageLoad({needs, userInfo}))
} catch (error) {
  handlePageLoadError(error);
}
```

As you can see, this looks somewhat conciser and saves us the nesting caused due to the later use of `userInfo`. In this example this is a rather small difference, but the code-base had contained some three to five layer nesting that could be significantly simplified using async-await.

5.3.1.4 ES6-Modules and Bundling

Previously we'd been including the JS-files via `<script>`-tags in `index.html` which was very fragile as dependency information wasn't solely managed by the scripts themselves but also redundantly

managed via this include list. Also, it depended heavily on Angular's dependency-injection mechanism, thus even utility-modules had to use that or expose themselves to global scope (and then be included in right order, lest they crash during startup). A less standardized variant here would have been to use the AMD-[8](see ref. "Why AMD?" n.d.) or CommonJS (see ref. "CommonJS Notes" n.d.) syntaxes. A small caveat here, is that we still have to use the AngularJS dependency-injection mechanism, thus causing redundant dependency management, but now the duplication is contained in the same file (once as `import`-statement at the top of a view- or component-script and once in the AngularJS-module-declaration at the bottom).

As browsers can't directly load these modules, however, it's necessary to use a script that loads them on-demand at runtime, like SystemJS (see ref. *Systemjs: Dynamic ES Module Loader* [2013] 2018), or a bundler, that compiles all JavaScript-module together into a single JavaScript-file during the build-process. Such a bundle can that can then be included via a `<script>`-tag. We started of with the "JavasScript Package Manager" (see ref. "Jspm.io - Native ES Modules CDN" n.d.), short JSPM, that provides a convenient command-line-utility for installing packages (`jspm install npm:<pkgname>`) and handles the SystemJS-integration. Including it in a page is as simple as running `npm install jspm && jspm init` and adding the following to one's `index.html`:

Listing 5.10: SystemJS startup.

```
<script src="jspm_packages/system.js"></script>
<script src="jspm_config.js"></script>
<script>
  System.import('app/app.js');
</script>
```

The downside of this approach is that every script file will be loaded separately and cross-compiled (see below in section 5.3.1.5), i.e. turning every page-load into a full build – with a build-times of 1-5 minutes for a codebase with >16k lines of JavaScript and ~20 dependencies (translating into >800 indirect-dependencies, and – more representatively – 5MB of unminified and 1.5MB of minified code as of 2017/09[9]).

A solution there, which is necessary for production anyway, is to bundle the modules into one JavaScript-file via `jspm bundle lib/main --inject`, by using `gulp-jspm` (see ref. "Gulp-Jspm" n.d.) in our Gulp-based build-setup (see section 5.3.5) or the Webpack-build triggered via `npm build` in our latest Webpack-based build see 5.3.6. Additionally, the resulting bundle was minified (e.g. by shortening variable names, dropping non-essential white-space-characters, etc). Together these reduced the all-important page-load times to – still excessive – 16 seconds on a simulated 3G

[8] Asynchronous Module Definition

[9] Owner-webapp in September 2017: https://github.com/researchstudio-sat/webofneeds/tree/69de16c1c7bc8495d915696665ae73b4dd1fd8f6/webofneeds/won-owner-webapp/src/main/webapp

connection (see ref. “Page-Load Performance Optimisation · Issue #546 - Comment 327556409 · Researchstudio-Sat/Webofneeds” n.d.). Further **page-load-optimizations** pushed this down to 4.5s (see section 5.3.7)

5.3.1.5 Cross-compilation

During the build-process all JavaScript files are run through BabelJS, that converts all newer feature, to equivalent but more verbose code, that supports older browsers. See the `.babelrc`-file for details on the configuration. Additionally, as not all features can be straight-out converted, a library providing some code that polyfills those features is added, i.e. adds them in the form of JavaScript where native implementation is lacking.

5.3.2 SCSS

Sassy CSS is a CSS pre-processor that allows features like:

- compile-time variables
- nesting of style-blocks (that generate nested selectors)
- code-reuse via mixins/`@include` (that copy code) and `@extend` (that append css-selectors)
- modularization via `@import`
- conditionals
- mathematical expressions

During the build-process it gets converted to CSS.

5.3.3 BEM

For the CSS-classes the naming convention BEM (short for “Block Element Modifier”) was used that helps with avoiding name-collisions between css-classes in different components. It distinguishes between “blocks”, i.e. stand-alone components, “elements” that only make sense within the context of a single block and “modifier” that model a sort of state of the block or element (e.g. “disabled”). The naming scheme for the class-names then looks as following: `<block>((__<element>)*--<modifier>)*`, e.g. `won-button--disabled__icon` for styling the icon in a disabled `won-button`-custom-component/-tag, i.e. “block” in the BEM-sense.

5.3.4 SVG-Spritemaps

To optimize page-load, instead of having every small icon in a separate file, the build-process puts them all into a single big SVG with defined `<symbol>`-tags around the markup from each source file

and a generated IDs corresponding to its file name. These icons can then be used via an inline-SVG containing a `<use>`-tag, as can be seen in fig. 5.11. Note that `xlink:href` and `href` would per se be redundant, but by declaring them both all browsers are supported.

Listing 5.11: Usage of the icon placed in the file ico36person.svg. A color is set to the css-variable local-primary that can be used inside the SVG to enable icon-reuse.

```
<svg class="…" style="--local-primary:var(--won-primary-color);">
    <use xlink:href="#ico36person" href="#ico36person"></use>
</svg>
```

Using the new Webpack build see 5.3.6, the SVGs are only included in the sprite-map when they are "imported" in any JavaScript-modules, in particular AngularJS-components. Also, in the SVGs the css-variables (most oftenly `--local-primary`) are used for colors, allowing to send the icon once and use it in multiple colors. Both of these enhancements further reduce the transmission-overhead.

5.3.5 Gulp

Gulp (see ref "Gulp.js" n.d.; respectively "Gulp" n.d.) is a build-tool that allowed us to define tasks for transpiling our JavaScript (using JSPM at the time) from ES6 (sec. 5.3.1) to older versions, our SCSS (sec. 5.3.2) to minified CSS, SVGs into a Sprite-Map (sec. 5.3.4) and copy around any static resources. It allows defining watch-tasks where file-changes to any of these trigger a corresponding rebuild, which makes development a lot smoother. However, it's been dropped out of the project by our recent switch from JSPM and Gulp to Webpack (sec. 5.3.6).

5.3.6 Webpack

Webpack is a bundler, that allows us to take all ressources (in particular JavaScript-modules) in the project and put them in one file that can be fetched in a single server round-trip. The related build-config can be found in `webpack.config.ts`. Through a set of plugins other necessary build-steps, like JS and (S)CSS transpilation and minification are handled and SVGs required by any Angular-components marked for inclusion into the SVG-Spritemap (see sec. 5.3.4). A handy watch-task (the target for `npm watch`) makes sure the bundle is rebuild as soon as any ressource changes.

5.3.7 Other Page-Load Optmizations

Back in September 2017[10] the code-bundle was 5MB of unminified and 1.5MB of minified code, which took 16 seconds on a simulated 3G connection (see ref. “Page-Load Performance Optimisation · Issue #546 - Comment 327556409 · Researchstudio-Sat/Webofneeds” n.d.) to load. A set of small adjustements allowed to push this down to 4.5s:

- Minifying the CSS
- Placing a `<link rel=”preload”href=”bundle.js”>`-tag in the header to make sure bundle-loading starts before the `<body>`-HTML is parsed.
- Enabling `gzip`-compression on the server for all served files
- Removing unused font-weights
- Non-blocking font-loading by adding `font-display: swap;` to the `@font-face`-declarations. Fallback-fonts declared as part of the `font-family`-rules are displayed until the proper fonts have loaded.
- Using `woff`/`woff2` as font-format, as it’s about a tenth of the size of `otf` and `ttf`-fonts
- Making sure *all* JavaScript dependencies are part of the bundle.

[10] owner-webapp in September 2017: https://github.com/researchstudio-sat/webofneeds/tree/69de16c1c7bc8495d915696665ae73b4dd1fd8f6/webofneeds/won-owner-webapp/src/main/webapp (accessed 18.06.2018).

References

"AngularJS: Developer Guide: Conceptual Overview." n.d. Accessed June 18, 2018. https://docs.angularjs.org/guide/concepts.

"Arrow Functions." n.d. MDN Web Docs. Accessed June 18, 2018. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions.

Buchwalter, William, and Antonin Januska. 2018. *Ng-Redux: Angular Bindings for Redux* (version 3.5.2). JavaScript. Angular Redux. https://github.com/angular-redux/ng-redux.

"CommonJS Notes." n.d. Accessed June 18, 2018. http://requirejs.org/docs/commonjs.html.

"ECMAScript 2015 Language Specification – ECMA-262 6th Edition." 2015. June 2015. https://www.ecma-international.org/ecma-262/6.0/.

"Elm Language." n.d. Accessed June 18, 2018. http://elm-lang.org/.

Fenton, Neil. (2015) 2018. *Redux-Ui-Router: ngRedux Bindings for Angular UI Router* (version 0.7.3). JavaScript. https://github.com/neilff/redux-ui-router.

"Gulp." n.d. Npm. Accessed June 18, 2018. https://www.npmjs.com/package/gulp.

"Gulp.js." n.d. Accessed June 18, 2018. https://gulpjs.com/.

"Gulp-Jspm." n.d. Npm. Accessed June 18, 2018. https://www.npmjs.com/package/gulp-jspm.

"Immutable.js." n.d. Accessed June 18, 2018. https://facebook.github.io/immutable-js/.

"Inheritance and the Prototype Chain." n.d. MDN Web Docs. Accessed June 18, 2018. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

"Ionic Framework." n.d. Ionic Framework. Accessed June 18, 2018. https://ionicframework.com/.

"Jspm.io - Native ES Modules CDN." n.d. Accessed June 18, 2018. https://jspm.io/.

Krasner, Glenn E., and Stephen T. Pope. 1988a. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System." *Journal of Object Oriented Programming* 1 (3): 26–49.

———. 1988b. "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80." *J. Object Oriented Program.* 1 (3): 26–49. http://dl.acm.org/citation.cfm?id=50757.50759.

"Linked Data." n.d. *Wikipedia*. Accessed June 18, 2018. https://en.wikipedia.org/w/index.php?title=Linked_data&oldid=846403008.

"Mat(ch)²at." n.d. Accessed June 18, 2018. https://www.matchat.org/owner/.

"Page-Load Performance Optimisation · Issue #546 - Comment 327556409 · Researchstudio-Sat/Webofneeds." n.d. GitHub. Accessed June 18, 2018. https://github.com/researchstudio-sat/webofneeds/issues/546.

"PhoneGap." n.d. Accessed June 18, 2018. https://phonegap.com/.

"Prior Art - Redux." n.d. Accessed June 18, 2018. https://redux.js.org/introduction/prior-art.

Reenskaug, Trygve. 1979. "Thing-Model-View-Editor." http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf.

"Resource Description Framework." n.d. *Wikipedia*. Accessed June 18, 2018. https://en.wikipedia.org/w/index.php?title=Resource_Description_Framework&oldid=836549456.

Systemjs: Dynamic ES Module Loader. (2013) 2018. JavaScript. systemjs. https://github.com/systemjs/systemjs.

"The Elm Architecture · an Introduction to Elm." n.d. Accessed June 18, 2018. https://guide.elm-lang.org/architecture/.

Ui-Router. n.d. TypeScript. AngularUI. Accessed June 18, 2018. https://github.com/angular-ui/ui-router.

"Webofneeds." n.d. Accessed June 18, 2018. http://www.webofneeds.org/.

"Web of Needs Publications List." 2013. Research Studios Austria Forschungsgesellschaft mbH. February 20, 2013. https://sat.researchstudio.at/en/web-of-needs.

"Why AMD?" n.d. Accessed June 18, 2018. http://requirejs.org/docs/whyamd.html.