CHAPTER 2

Problem Description

This thesis is part of the over-arching project of crafting an end-user-friendly client-application[1] for the Web of Needs[2] (related publications[3]), 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.

define ontologies and rdf
node = won-data/document-server

2.1 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 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.

[1] https://www.matchat.org/owner/
[2] http://www.webofneeds.org/
[3] http://sat.researchstudio.at/en/web-of-needs

```
<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. :)" .
```

Figure 2.1: Excerpt of a need description (N-Triples)

2.2 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[4]. In RDF, data is represented as a graph that can be distributed over multiple (physical) resources. Edges in the graph 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[5]. 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 the ones in figure 2.1.

As you can see, this way of specifying triples, called N-Triples, isn't exactly developer-

[4] https://en.wikipedia.org/wiki/Resource_Description_Framework
[5] https://en.wikipedia.org/wiki/Linked_data

```
{
  "@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"
     }
  }
}
```

Figure 2.2: Excerpt of a need description (JSON-LD)

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 are several other markup-languages respectively serialization-formats for better writing and serving these triples, e.g. Turtle, N3, RDF/XML and JSON-LD. The same example, but in JavaScript Object Notation for Linked Data (JSON-LD) would read as in figure 2.2.

TODO get syntax-highlighting to work in figures (see comment in .tex)

As you can see, JSON-LD allows to nest nodes 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); however, as JSON-LD also is valid JSON/JS-code, it was the natural choice for using it for the JS-based client-application.

2.3 WoN-Owner-Application

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

screens from last prototype

2.3.2 Technical Requirements

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

- Needs to be able to keep data in sync between browser-tabs running the JS-client and the Java-based server. 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 account details, which needs belong to an account, its key-pair and information on which events have been seen.
- 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.

who is we?

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:

which gave rise to the work at hand

TODO why we implemented it js-based:
* bandwith
* because it's become somewhat of a wide-spread practice, i.e. "because everybody's doing so"
* because there already was the angular prototype
* because it can run on any OS and device
status quo: angular app

- 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.

why?

* TODO image: dependency graph in angular application
* slide from FB's flux presentation?
* go through old application and do this empirically for a few components and bugs?

CHAPTER 3

State of the Art

3.1 Frameworks and Architecture

Gut wäre ein einleitender Absatz, der das Grundproblem der GUI Programmierung erklärt, das dann MVC, MVVC, angular und react lösen

3.1.1 Model-View-Controller

Das müsst du anders einleiten

~~You probably already are familiar with the classical~~ model-view-controller architecture, ~~but~~ for the sake of completeness a short overview will be given here. The pattern mainly consists of three types of building blocks (as can also be seen in figure 3.1):

TODO sources

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's 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 dependants what changed.

views are what the outside world/user's get to see. When the model changes, the view get's 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)

Satzbau Prädikate

Note that there's a wide range of different instances/interpretations of the architectural patterns can organise models/views/controllers differently. Further down, in section 3.1.3 you can find one of these (angular's MVC) described in more detail.

TODO krasner and pope 1988: http://heaveneveryw

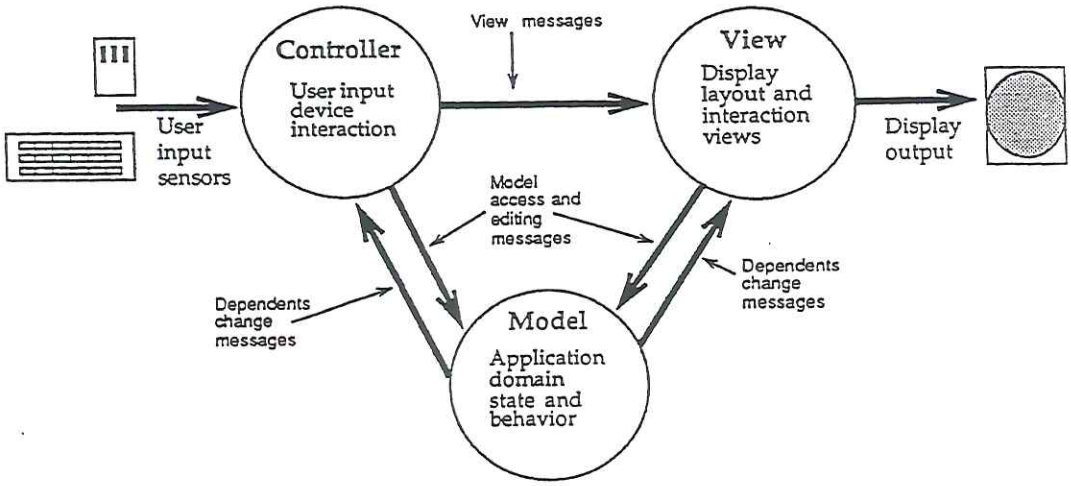

Figure 3.1: MVC-architecture (Krasner and Pope, 1988)

3.1.2 Model-View-ViewModel

This architectural pattern, also known as "Model-View-Binder", is similar to MVC but puts more emphasis on the seperation between back-end and front-end. It's parts are as follows (and can be seen in fig. 3.2):

TODO sources

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.

3.1.3 Angular 1.x MVC

Too much detail! Move a lot of these details to later chapters (e.g. "solution » ng best practices" or "solution » why we moved from ng to ng-redux")

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

On the View-side of things there's templates (see fig. 3.3 for an example from the webofneeds-codebase). These are either specified in an html-file and then later linked

[1] https://en.wikipedia.org/wiki/File:MVVMPattern.png

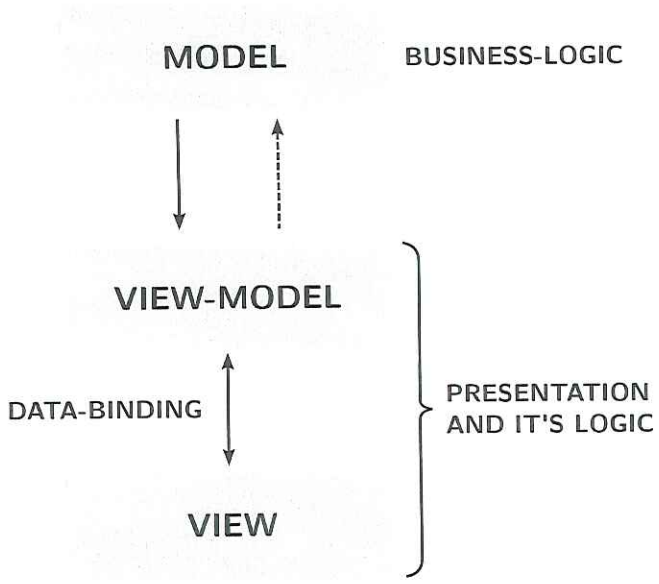

Figure 3.2: MVVM-architecture (diagram adapted from wikimedia[1])

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 example in fig. 3.3 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[2] 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`). On every "digest-cycle" 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.

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:

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

Or, similiarly, `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 scope. Additionally, templates/directives can be nested

[2] `https://facebook.github.io/immutable-js/`

```
...
<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']) }}
</p>
...
```

Figure 3.3: Excerpt of angular-template

within each other. By default, their scopes then use javascript's prototypical inheritance[3] 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. More on that later. Also, using scope inheritance reduces reusability, as the respective components won't work in other contexts any more.

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 (more on that later) or by being part of the same directive. Controllers have access to their template's scope and vice versa Theoretically it is possible to pair up / 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 they're associated controllers, ~~actually it's~~ the controllers that form the ~~prototypical~~ inheritance chain. Thus, if a variable isn't found on the controller respectively ~~it's~~ its scope, the default is to check on it's parent('s), up to the root-scope. Note that scopes can be defined as isolated in the routing config to avoid this behaviour, which I'd recommend for predicatability, and thus maintainability reasons.

~~Now~~, with models/scopes, views/templates and controllers we would have a classical MVC-framework (see section 3.1.1). However, angular also has the concept of services:

[3] https://developer.mozilla.org/en/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

TODO add reference to that subsection

move critique of bi-dir binding and inheritance to later chapter

TODO get syntax-highlighting to work in figures (see comment in .tex)

ref to routing-subsection

ref to directive-subsection

ref to controllerAs discussion

ref to routing/isolated-scope section

can be reused with different template, but that rarely happens and tends to lead to hard-to-track-down bugs.

nesting templates (not directives?)— how does it work anyway?

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

Figure 3.4: Example of a very simple controller and usage of the `$scope`-service

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; more on that later). Examples of services are `$scope` that, amongst others, allows registering custom watch-expressions with angular outside of templates (see fig. 3.4 for an example). Another example for a service would be our custom `linkeddata-service.js` that can be used to load and cache RDF-data[4].

ref to subsection

who is we? (often: we ≙ I)

It's not a blog post!

With services added to the mix, we can also view the angular framework through the lense of MVVM (see section 3.1.2), 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 the `linkeddata-service.js`).

TODO get syntax-highlighting to work in figures (see comment in .tex)

Note, that Angular 1.x uses it's 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 your scripts into one `bundle.js`[5] 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 figure 3.6.

ref number doesn't match figure's

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[6]). 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.

TODO reference ng docu

[4] see section 2.2 for more on RDF

[5] Bundling for instance helps to reduce the number of HTTP-requests on page-load and thus it's performance. It can be done by using a build-tool like browserify, webpack or jspm plus a module system like AMD, CommonJS or the recently standardized ES6-modules (see `http://www.ecma-international.org/ecma-262/6.0/#sec-imports`)

[6] `https://docs.angularjs.org/guide/concepts`

```
myApp.config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.
      when('/landingpage?:focusSignup', {
        templateUrl: 'app/components/\
          landingpage/landingpage.html',
        controller: 'LandingpageController'
      }).
      when('/post/?postUri', {
        templateUrl: 'app/components/\
          post/post.html',
        controller: 'PostController'
      }).
      otherwise({
        redirectTo: '/landingpage'
      });
  }]);
```

Figure 3.5: Example of routing configuration in Angular 1.X

3.1.4 React

React is a framework that specifically provides the view (and potentially 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 seperation of concerns and code reusability. These components are stateful 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 additional to React, for instance to handle the application-state or manage HTTP-requests and websockets. This is usually where Flux (see section 3.1.5) and Redux (see section 3.1.6) come in.

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's 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 side of things, this diff'ing-process means, that there's no need to manage DOM-state-changes

```
/* 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;
```

Figure 3.6: Example of module- and controller-declaration in `create-need.js`

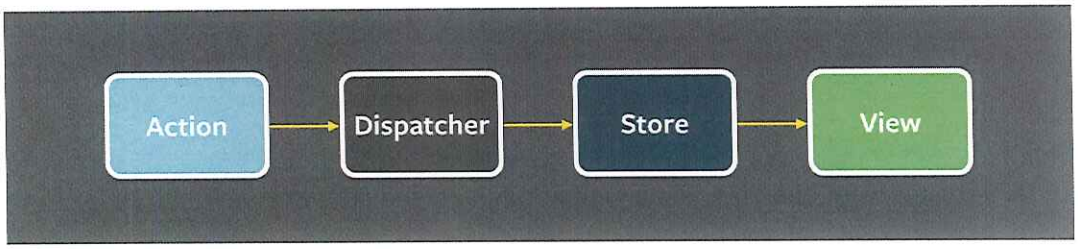

Figure 3.7: Core of the Flux-architecture (via `https://facebook.github.io/flux/img/flux-simple-f8-diagram-1300w.png`)

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 it's 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 it's 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).

3.1.5 Flux

When you start reading about React you'll probably stumple across Flux (see fig. 3.8) rather earlier than later. It is the architecture popularized alongside of React and akin to MVC in that it seperates handling input, updating the state and displaying the latter.

However, instead of having bi-directional data-flow between the architectural components, Flux' is uni-directional and puts most of it's business logic into the stores that manage the state. To give an example of a flow through this loop: Say, a user clicks on a map widget with the intend of picking a location. The widget's on-click method, would then create an object ~~that's~~ called an action, that usually contains type-field like `"PICK_LOCATION"` and any other data describing the user-interaction, ~~like~~ such as geo-coordinates. It then goes on to pass the action object to the globally available dispatcher, ~~that~~ 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 updated the geo-coordinates it holds. The stores would then go on to 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

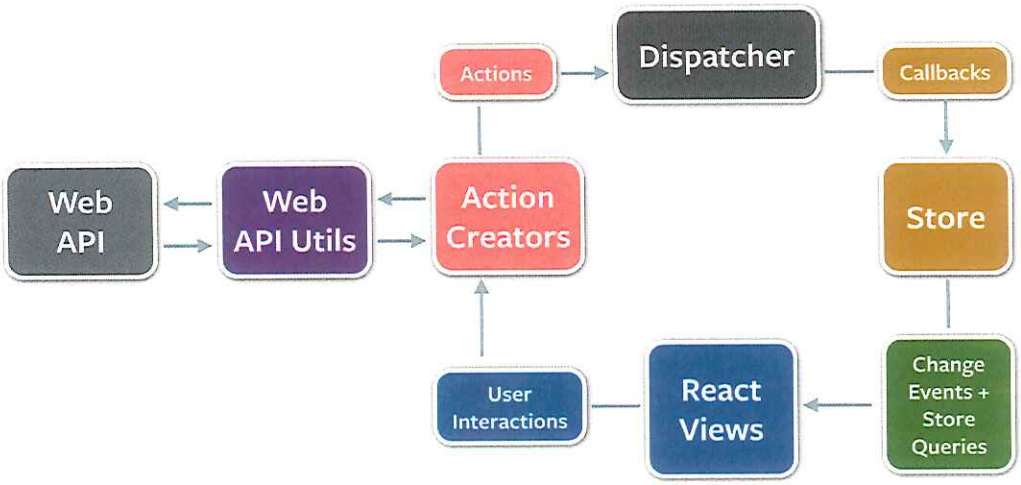

Figure 3.8: Full Flux-architecture incl. networking (via `https://facebook.github.io/react/img/blog/flux-diagram.png`)

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's 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. These are functions that do preprocessing—including HTTP-requests for instance—and then produce the action-objects and dispach 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 boilerplate code necessary to bootstrap a Flux-based application.

Stores can have dependencies amongst 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[7]). 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.

[7] `https://facebook.github.io/immutable-js/`

3.1.6 Redux

The developers/designers of Redux list the object-oriented Flux- (see above) and functional Elm-architecture (see below) as prior art[8]. It mainly differs from Flux, by 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. As such there's also formally no need for a dispatcher, as there's only a single function that's updating the state. Seperation of concerns—that Flux achieves with it's 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 shallow compared to Flux and almost flat compared to Angular's MVC.

Redux profits from immutable data-structures for the app-state almost 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 extend(nothing can prevent you from accessing the global `window`-scope in javascript though, so ideally don't do that). This property also means, that you should try to move as much busieness 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.

3.1.7 Ng-Redux

Ng-Redux[9] is a framework that's based on the Redux-architecture and is geared to be used with Angular applications. The latter then 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 which result is then bound to the controller. It also provides a plugin/middleware-system for plugins that provide convienient use of asynchronicity in action-creators (through "thunk") or keeping the routing information as part of the application state (through the "ngUiRouterMiddleware").

3.1.8 Elm-Architecture

Elm[10] is a functional language who's designers set out to create something as ac-

[8] `http://redux.js.org/docs/introduction/PriorArt.html`
[9] `https://github.com/angular-redux/ng-redux`
[10] `http://elm-lang.org/`

cessible to newcomers as Python or Javascript. It can be used to build front-end web applications (browser-less execution in node is currently being worked on). 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[11], in its basic form, requires one to define the following three functions and pass them to Elm's `Html.beginnerProgram` (that runs the app):

1. `model : Model` that initializes the app-state.
2. `update : Msg -> Model -> Model`, which performs the same role as `reduce` in Redux, with `Msg`s in Elm being the equivalent to actions in Redux.
3. And lastly, `view : Model -> Html Msg` to produce the HTML from the model.

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

1. `init : (Model, Cmd Msg)` fulfills the same role as `model`, but also defines the first `Cmd`. These 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`.
2. the function `update : Msg -> Model -> (Model, Cmd Msg)` now also returns a `Cmd` to allow triggering these 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)—arguably this is a rather neglible price.
3. `subscriptions : Model -> Sub Msg` allows to set up additional sources for `Msg`s beside user-input, things that *push* ~~if you so will~~ e.g. listening on a websocket.
4. `view : Model -> Html Msg` works the same as in the simple variant.

[11] `https://guide.elm-lang.org/architecture/`

```
import {run} from '@cycle/xstream-run';
import {div, label, input, hr, h1, makeDOMDriver} from '@cycle/dom';

function main(sources) {
  const sinks = {
    DOM: sources.DOM.select('.field').events('input')
      .map(ev => ev.target.value) // get text from field
      .startWith('') // initial value / first stream-message
      .map(name =>
        div([
          label('Name:'),
          input('.field', {attrs: {type: 'text'}}),
          hr(),
          h1('Hello ' + name),
        ])
      )
  };
  return sinks;
}

run(main, { DOM: makeDOMDriver('#app-container') });
```

Figure 3.9: CycleJS hello-world example from `https://cycle.js.org/`

3.1.9 CycleJS MVI

CycleJS is an 'functional reactive programming'-based framework, which Model-View-Intent architecture, is structured similar to the Redux- and (original) Elm-architectures.

But first: The framework itself is based on functional reactive programming (FRP) and uses observables/streams of messages for its internal data-flows. Think of them as Promises that can trigger multiple times, or even more abstract, pipes that manipulate data that flows through them and that can be composed to form a larger system. The integral part developers using the framework need to specify is a function `main(sources) => ( DOM: htmlStream)` (see fig. 3.9) 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 example of fig. 3.9 for every input-event a piece of data/message would travel down the chained functions and end up as a virtual DOM object. This `main`-function is passed to `run` to start the app.

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

1. 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.
2. The model-stage is usually implemented as a function that is `reduce`'d over the model (equivalent to how Redux deals with state-updates)
3. 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 in the first) and combining them at the end of it.

Finde den Überblick sehr gut. An der Stelle frage ich mich aber, warum ich das jetzt gelesen habe ... warum werden gerade diese Ansätze vorgestellt? Sind das alle? Sind sie irgendwie eine „Klasse" von Ansätzen?

CHAPTER 4

Methodology

4.1 Design Science in Information Systems Research

For the work preceding this thesis the methodological framework presented in "Design Science in Information Systems Research" by Hevner et al (2004) was used. I'll try to give a short overview over it in this section.

4.1.1 Design Science

The paper states that a lot of the research surrounding information systems can be described as design- and/or behavioral science.

It roughly defines **behavioral science** with regard to IS-research as concerned with the analysis of the interactions of people and technology, with the goal of uncovering "truths" and predicting or explaining phenomena surrounding these interactions.

In comparison, Hevner et al describe **design science** as concerned with problem solving and construction with the background that doing so leads to understanding the addressed "wicked" problem [1]. It diffentiates itself from **routine design** by addressing problems without existing best-practices/requisite knowledge and solves them unique/innovative ways, or improves efficiency. By doing so, new knowledge is contributed to the foundations and methodolgies. Design-science usually also produces prototypes instead of full-grown systems.

The paper also presents two **fundamental questions** of design research as "What utility does the new artifact provide?" and "What demonstrates that utility?". As all other's

[1] Here, "wicked" problems (Brooks 1987, 1996; Rittel, Webber 1984) are defined as those with unstable requirements, ill-defined environmental contexts, complex interactions among subcomponents of problem and solution, an inherent flexibility to change design processes and artifacts and a critical dependence on human cognitive abilities (e.g. creativity) and social abilities (e.g. teamwork) for effective solutions

from Hevner et al (2004) that are referenced or quoted here, they'll be addressed in the next chapter.

Habe keine Ahnung, was du mir sagen willst

4.1.2 Design Processes and Artifacts

March and Smith (1995)[2] list two processes involved in design, **build/generate** and **evaluate**, that form a cycle (see figure 4.1). They differentiate the artifacts produced as:

constructs that provide the language to define problems and solutions (e.g. programming languages)

models that abstract and represent these and allow exploring the effects of design decisions

methods that define how to solve problems or aid with searching the problem-space (e.g. algorithms, best practices)

instantiations that demonstrate feasibility and enable assessing suitability for the intended purpose

4.1.3 Design-Science Research Guidelines

Hevner et al (2004) defines seven guidelines that design-science in information systems should address (but not necessarily come-what-may adhere to), which will be done in the next chapter. They are as follows:

So einen Einschub kann man direkt in eine Fußnote packen. Außerdem ist er etwas umständlich.

Design as an Artifact "Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation." This allows to demonstrate feasibility – for cases where that wasn't a given yet – thus making it research (as opposed to routine design).

Problem Relevance "The objective of design-science research is to develop technology-based solutions to important and relevant business problems." Relevancy here is with respect to a "constituent community" (e.g. IS practitioners)

Design Evaluation "The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods." This usually requires integration into the usage context (to see if it "works" there or is "good" in it), the definition of appropriate metrics and gathering of appropriate data. Evaluation provides valuable and necessary feedback for the design iterations (see figure 4.1)

[2] TODO

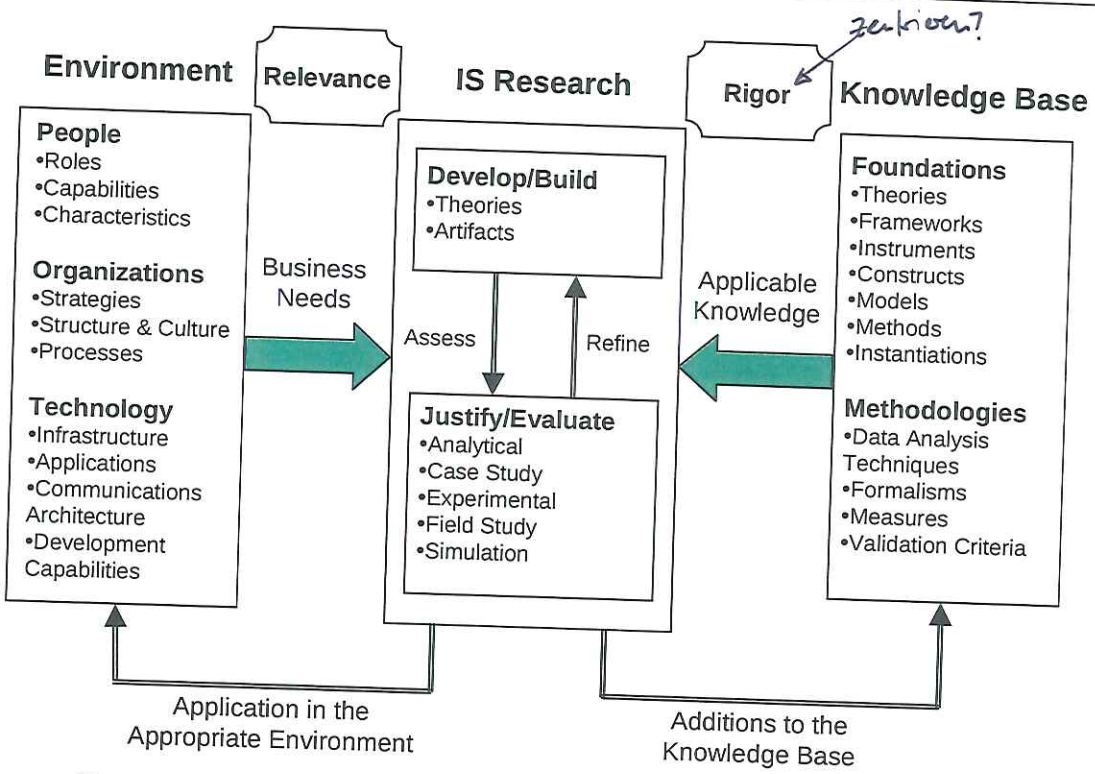

Figure 4.1: Information Systems Research Framework (Hevner et al 2004)

Research Contributions "Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies." Important here is the novelty of the artifact – by extending or innovatively (re-)applying previous knowledge – as well as its generality and significance.

Research Rigor "Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact." This means applying existing foundations and methodologies, using effective metrics and formalising. Note, however, that an overemphasis on rigor can often lead to lower relevance (Lee 1999), as many environments and artifacts defy an excessive formalism (see "wicked problems" at footnote 1).

Design as a Search Process "The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment." This entails using heuristic search stragegies (e.g. best-practices as starting point) in ght generate/test-cycle (see figure 4.1) However, again, it might not be possible to formalize or even determine any of these, due to the "wicked" (see footnote 1) nature of tackled problems. As a result it might often be necessary to only work on simpler sub-problems, giving up relevancy in turn.

Communication of Research "Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences." For the former the construction and evaluation process are important (e.g. to allow reproduction). For the latter the question boils down to "Is it worth the effort to use the artifact for my business?". This can be broken down as "What knowledge is required?" respectively "Who can use it?", "How imporant is the problem?", "How effective is the solution?" as well as some details in appendicesto appreciating the work.

4.1.4 Design Evaluation Methods

Observational methods:

Case Study "Study artifact in depth in business environment"

Field Study "Monitor use of artifact in multiple projects"

Analytical methods:

Static Analysis "Examine structure of artifact for static qualities (e.g., complexity)"

Architecture Analysis "Study fit of artifact into technical IS architecture"

Optimization "Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behavior"

Dynamic Analysis "Study artifact in use for dynamic qualities (e.g., performance)"

Experimental Methods:

Controlled Experiment "Study artifact in controlled environment for qualities (e.g., usability)"

Simulation "Execute artifact with artificial data"

Testing:

Functional (Black Box) Testing "Execute artifact interfaces to discover failures and identify defects"

Structural (White Box) Testing "Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation"

Descriptive Methods:

Informed Argument "Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact's utility"

Scenarios "Construct detailed scenarios around the artifact to demonstrate its utility"

CHAPTER 5

Suggested Solution

5.1 Architecture

Reword so it fits into the thesis. Change all links to github issues to point to other sections of the thesis.

We're using the redux-architecture for the won-owner-webapp javascript-client. The architecture is strongly based on redux[1] in general and ng-redux[2] in particular. You can find a short overview over these and their actions, reducers, the store and components in section 3.1.6. For more in-depth and hands-on documentation beyond what has been used for the work preceding this thesis, I can recommend reading their well-done documentations.

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

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.

If you want to **add new action-creators** do so by adding to the `actionHierarchy`-object in `actions.js`. From that two objects are generated at the moment:

[1] `http://redux.js.org/`
[2] `https://github.com/wbuchwalter/ng-redux`

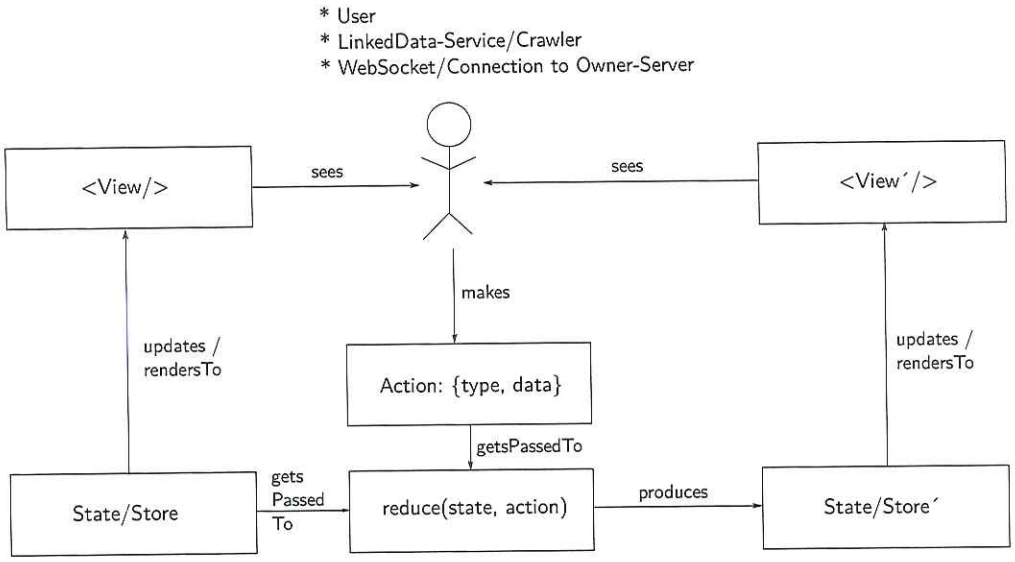

Figure 5.1: redux architecture in client-side owner-app

- `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 organised with `__` as seperator (e.g. `actionCreators.drafts__change__title('some title')`)

Btw, the easiest way for actions without sideffects is to just placing 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 be 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 suprise-factor/coupling/bug-proneness of our code and increases its maintainability.

5.1.2 Actions

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

They are objects like `{type: 'drafts.change.title', payload: 'some title'}` and serve as input for the reducers.

See: Actions/Stores and Synching — wo?

5.1.3 Reducers

Hier nochmal erklären, was dieses (state, action) => state

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.

See "Action Creators" in section 5.1.1 above for more detail

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-nee` 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.

In both cases open up the latest implemented component and use the boilerplate from these, if you want to implement your own. Once a refined/stable boilerplate version has emerged, it should be documented here.

passive voice

das wäre hier ein guter Ort ü

5.1.5 Routing

We use ui-router and in particular the redux-wrapper for it

was?

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

```
$ngRedux.getState().get('router')
/* =>
{
  currentParams: {...},
  currentState: {...},
  prevParams: {...},
  prevState: {...}
}
*/
```

Also see: Routing and Redux

5.1.6 Server-Interaction

If it's **REST**-style, just use `fetch(...).then(...dispatch...)` in an action-creator.

If it's **linked-data-related**, use the utilities in `linkeddata-service-won.js`. They'll do standard HTTP(S), but will make sure to cache as much as possible via the local triplestore.

If needs to **push to the web-socket**, add a hook for the respective *user(!)*-action in `message-reducers.js`. The `messaging-agent.js` will pick up any messages in `$ngRedux.getState().getIn(['messages', 'enqueued' ])` and push them to it's websocket. ~~This solution appears rather hacky to me (see 'high-level interactions' under 'Action Creators') and I'd be thrilled to hear any alternative solutions :)~~

If you want to **receive stuff the web-socket**, go to `actions.js` and add your handlers to the `messages__messageReceived`-actioncreator. The same I said about pushing to the web-socket also holds here.

5.2 Tooling

In 5.1 erklärst du die allgemeinen Prinzipien, nach denen die Applikation funktioniert. Jetzt beschreib die Applikation selbst:

- ~~VIEWS~~ VIEWS
- BENUTZERINTERAKTIONEN
- KOMMUNIKATION MIT SERVER
- ANBINDUNG DES RDF STORE

CHAPTER 6

<Core>

Ich bin nicht sicher, ob das als eigenes Kapitel genug Inhalt hat. Ich glaube, dann dich das obenst und belastet. Konzentriere dich erstmal auf Kapitel 5

6.1 Design as an Artifact

"Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation." This allows to demonstrate feasibility – for cases where that wasn't a given yet – thus making it research (as opposed to routine design).

passt: es kann ja ein Artefakt heraus

6.2 Problem Relevance

"The objective of design-science research is to develop technology-based solutions to important and relevant business problems."

passt: für wen ist eine Lösung wichtig

6.3 Design Evaluation

"The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods."

passt: vielfältig evaluiert: andere Programmierer haben es verwendet
User haben es ausprobiert

6.4 Research Contributions

"Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies."

nun drei eines Bakkalaureatsarbeit nicht sein (was aber okay)

6.5 Research Rigor

"Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact."

immer wieder durch andere Programmierer angewandt, schließlich auch mit Usern getestet.

6.6 Research Rigor

"Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact."

6.7 Communication of Research

"Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences."

- Dokumentation auf github
- diese Arbeit
- paper?