ANGULAR.JS EN VOYAGE PART DEUX

Christian Ulbrich, Zalari UG

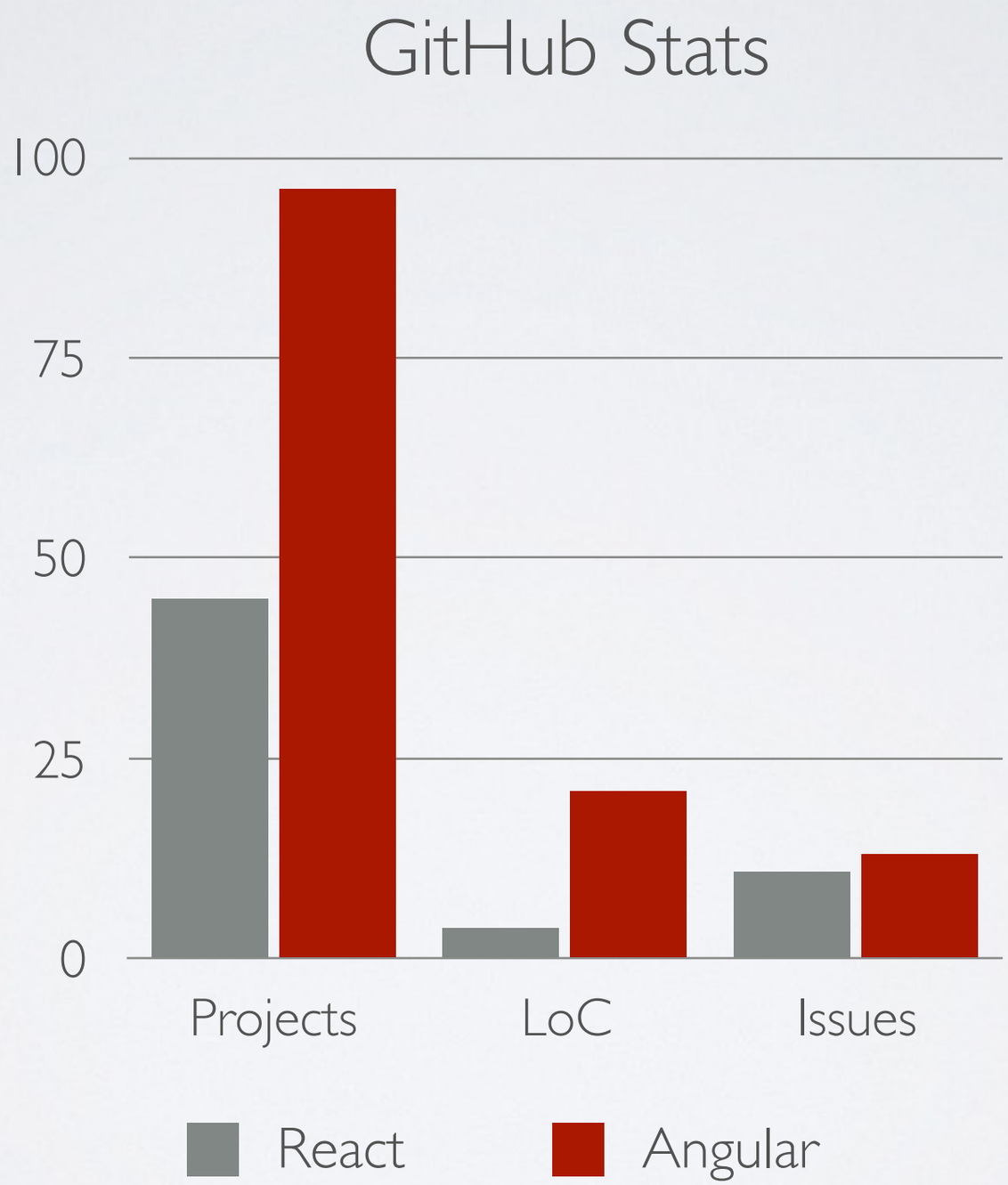
AGENDA PROPAGANDA

- Error Handling
- Form processing
- Company Culture
- Performance
- Testing
- Links

MOTIVATION

- What has Angular - that the other uber frameworks do not have?
  - Opinion
  - Completeness
  - Scope
  - Maturity

MOTIVATION


GitHub Stats
100
75
50
25
0
Projects
LoC
Issues
React
Angular

MOTIVATION


GitHub Stats
160
120
80
40
0
Projects
LoC
Issues
React
Angular

CREDIBILITY

- 15k LoC JS, 5k CSS
- 100+ services
- 65 directives
- 50+ controllers
- German-wide deployment on-site
- roughly 30 man months

ERROR HANDLING GOALS

- errors should be handled in a consistent way
- classification or errors
- error reporting
- central error handling

ERROR HANDLING ADVISES

- AngularJS is ultimately running everything inside a huuuuuge try /catch block
- exposes *$exceptionHandler*
- decorating $exceptionHandler allows to plugin own logic for *any* errors
- central error handling

ERROR HANDLING ADVISES

- prominently exposing any errors improves code quality big time
- errors can be classified, certain errors might occur and are expected
- expected errors can be handled
- unexpected errors should reload the app

ERROR HANDLING ADVISES

app.decorators.js

```
angular.module('[redacted]')
  .config(function($provide) {

    //hook up $exceptionHandler with our custom exceptionHandler
    $provide.decorator('$exceptionHandler', function($delegate, appExceptionHandler) {
      return function(exception, cause) {
        //for the time being, use the standard behaviour
        $delegate(exception, cause);
        //call our own implementation
        appExceptionHandler.handleException(exception, cause);
      };
    });
  });
```

ERROR HANDLING ADVISES

- classification of errors into at least 2 categories - recoverable and non-recoverable
- hierarchies of errors are good candidates for some inheritance meddling
- combined with a $httpInterceptor one can map arbitrary http errors to specific application errors

ERROR HANDLING ADVISES

```
this.handleException = function(exception, cause) {
  $log.error('Exception occured: %0', exception);

 if (exception instanceof appExceptions.Exception) {
    //those are our own, known exceptions...

    //handle recoverable exceptions
    if (exception instanceof appExceptions.RecoverableException) {
      _processRecoverableException(exception);
    } else if (exception instanceof appExceptions.NonRecoverableException) {
      _processNonRecoverableException(exception);
    } else {
      //this must be an instanceof Exception, that is not recoverable and not non-recoverable
      //thus it should never be reached; process as unknown exception
      $log.error('Unknown exception type!');
      _processUnknownException(exception);
    }
  }
  else {
    //all un-handled exceptions end up here... they are by definition non-recoverable
    $log.error('Unknown exception!');
   _processUnknownException(exception);
  }

};
```

FORM PROCESSING - GOALS

- we need to validate form inputs against custom logic
- forms should have global states (valid, invalid, pristine, dirty)
- error messages should be re-usable and related to input
- how are forms built - via html / via code
- nested forms…

FORM PROCESSING - ADVISES

- either AngularJS built-in or custom / third party

FORM PROCESSING - ADVISES

- the *gretchenfrage*: programmatic or declarative?
- programmatic -> https://github.com/formly-js/angular-formly
- declarative -> either user custom directives or AngularJS
- beware: usually multiple form inputs are in relation with each other!

FORM PROCESSING - PROGRAMMATIC

- pros:
  - allow for highly-dynamic forms
  - consistent markup generation across whole application
- cons:
  - form customization has to be done in the code
  - customization of individual elements is complicated
  - composing forms has its limits
  - using global form state is more difficult

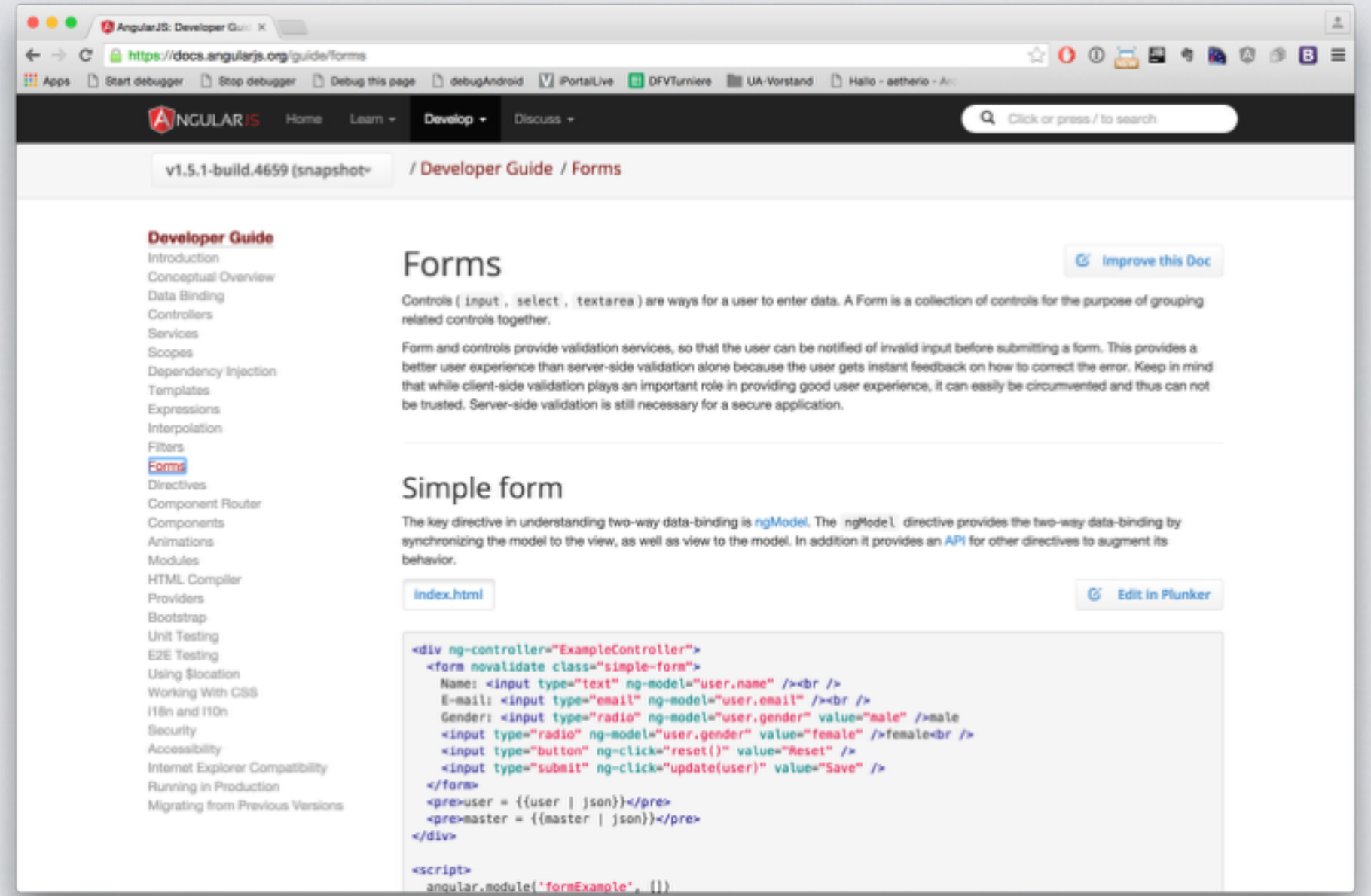
FORM PROCESSING - DECLARATIVE

- pros:
  - either use built-in functionality or roll your own AngularJS-aware form inputs
  - leverage of powerful built-in functionality for easy declarative enforcing of rules
- cons:
  - without custom inputs, enforcing global unique style is cumbersome
  - enforcing logic is hidden in view template and might need to be duplicated in controllers as well
  - interacting with ngModelController is complicated

FORM PROCESSING - ADVISES

- static forms -> AngularJS functionality
- dynamic forms -> third party
- custom input directives are in between

FORM PROCESSING - ADVISES



FORM PROCESSING - ADVISES

- <forms> need to have a name attribute, e.g. *loginForm*

```
<form ng-if="!isAuthenticated()" name="loginForm">
```

- all child input elements need to use ngModel
- custom input elements need to require ngModelController or pass ngModel along
- -> AngularJS will (automagically) expose *testForm* in the current *$scope*

FORM PROCESSING - ADVISES

- $scope.loginForm has properties: *$pristine, $dirty, $valid, $invalid, $pending, $submitted, $error*

- these can be used in view and controller

```
<div class="form-group" ng-class="{'has-error': loginForm.$invalid && !loginForm.$pristine}">
```

```
<button type="button" class="btn btn-primary btn-login" ng-disabled="loginForm.$invalid" ng-click="login()">
  Sign In
</button>
```

- retaining this behaviour with your own controls is only possible if you stick to ngModelController

FORM PROCESSING - ADVISES

- ngModel is a complex beast
- *validator, parser & formatter* pipelines
- can be tamed with ngModelOptions
- for many use cases, simply passing ngModel along as a reference is enough
- online example for usage in combination with the pickadate date picker -> https://github.com/zalari/angular-pickadate.js

FORM PROCESSING - ADVISES

- $scope.loginForm.$error can have the following keys (*validation tokens* in AngularJS parlance) set:
  - email, max, maxLength, min, minLength, number, required…
  - -> https://docs.angularjs.org/api/ng/type/form.FormController
- ngMessage is constantly evolving
- boils down to evaluating an *object hash* and showing whenever a key is set to true and guest what *$scope.loginForm.$error* is?
- allows for application-wide template for error messages using *ng-messages-include* attribute

FORM PROCESSING - ADVISES

formInputNumber.view.html

div.nm-input.nm-input-text

```
1   <div class="nm-input nm-input-text"
2        ng-class="{disabled: ngDisabled}">
3       <ng-form name="nestedForm">
4       <label for="input-{{name}}">{{label ? label : ' '}}</label>
5       <input id="input-{{name}}" step="any"
6           ng-disabled="ngDisabled" type="number" name="{{name}}"
7           ng-pattern="ngPattern" ng-maxlength="ngMaxlength" ng-minlength="ngMinlength"
8           ng-required="ngRequired" ng-change="ngChange" ng-model="ngModel"/>
9       <!--error messaging-->
10      <div class="form-input-warning" ng-messages="nestedForm[name].$error"
11           ng-messages-include="scripts/modules/errorHandling/views/defaultValidationErrorMessages.view.html">
12      </div>
13      </ng-form>
14  </div>
```

FORM PROCESSING - ADVISES

defaultValidationErrorMessages.view.html

```
1   <!--TODO: this is language specific and should be using a filter for localization-->
2   <div class="error-messages">
3     <div ng-message="required">Pflichtfeld</div>
4     <div ng-message="minlength">Eingabe zu kurz</div>
5     <div ng-message="maxlength">Eingabe zu lang</div>
6     <div ng-message="email">E-mail-Adresse ist falsch</div>
7     <div ng-message="number">Feld muss eine Zahl sein</div>
8     <div ng-message="pattern">Eingabeformat ungültig</div>
9   </div>
10
```

PERFORMANCE

- AngularJS 1 revolutionzed the usage of two-way-databinding
- Performance is a typical non-functional software requirement
- Performance should be measured

PERFORMANCE

- AngularJS' implementation of two-data-binding is magic™
  - because you can use POJV
  - i.e. AngularJS detects changes on primitives and on Objects and it resolves not only references but actually determines which change results in a DOM change
- boils down to having a *watcher* on a variable
- hard-limit of watchers used to be ~ 2.000
- although a beefy machine can take 4.000 without any big problems :)

PERFORMANCE

- especially data-driven views accumulate watchers very fast
  - e.g. dataTables, each row might incur a number of watchers
- if you don't have paging / use infinite scrolling, this can easily explode the watchers
- -> https://github.com/kentcdodds/ng-stats

PERFORMANCE

- AngularJS 1.3 introduced one-way data binding
- very often you only need one-way data binding, especially in dataTables
- -> syntax is {{*::model.value*}}
- AngularJS' two-way-data-binding allows for *easy solutions* of problems, however these might come at a cost
- -> think about alternate strategies / solutions for certain implementations, especially when they operate on bigger data sets

PERFORMANCE

- AngularJS 1 is not ReactJS
- ReactJS has a fundamental different architecture and overall approach
- AngularJS is a complete framework for Single Page Apps
- ReactJS is about components and does not enforce any style of application architecture
- AngularJS 2 won't be a silver bullet either

COMPANY CULTURE

- it is broader topic, concerning Agile development and stuff like Extreme Programming
- establish processes for reviewing your code
- establish processes, that individual developers with their varying experience can benefit from each other

COMPANY CULTURE

- use a good IDE tooling
- use good commenting
- use something like JSDoc
- use linting / hinting of code

COMPANY CULTURE

- think about DevOps™
- DevOps is way, to speed up the on boarding of new developers by at least one order
  - the tools are plentiful: Vagrant, Docker, …
- use and foster open source software

TESTING

- sorry no time for that any more

TESTING

- different kinds of tests smoke tests, unit test, end-to-end tests
- AngularJS is only the front-end
- testing needs to be automated
- testing only makes sense in an continuous integration environment

TESTING

- unit tests can be run with karma and numerous testing frameworks (jasmine, mocha, chai, mocha +chai, …)
- test-driven-development is expensive up-front
- end-to-end tests in AngularJS are run with protractor

TESTING - PROTRACTOR

- having meaningful and re-usable tests is a big project on its own
- you need a framework / file layout / concept for your e2e tests!
  - -> pageObjects, mockData, …
- e2e test should be run on real browsers
- phantomJS is a headless browser and does not behave consistently like a real browser