Input and Validation

Mendel Rosenblum
Early web app input: HTTP form tag

```html
<form action="/product/update" method="post">
    Product: <input type="text" name="product"/>
    Deluxe: <input type="checkbox" name="delux" />
    <input type="submit" value="Submit"/>
</form>
```

- **method="get"** - Encode form properties as query params
  
  HTTP GET product/update?product=foobar&delux=on

- **method="post"** - Encode form properties as query params in message body
  
  HTTP POST product/update
  
  Content-Type: application/x-www-form-urlencoded
  
  product=foobar&delux=on
Rails input pattern using form POST

- GET Page containing form
  - Contains a method="post" form to a POST Page

- POST Page - Validate and perform operation (typically create or update)
  - If successful, redirect to a "done "page (possibly another GET Page) if successful
  - If failed validation, redirect page to the GET Page with incorrect fields highlighted
  - If error, redirect to some oops page
Validation requirements in web applications

● Protect integrity of storage (required fields, organization, security, etc.)
  ○ Can let HTTP request either from web app or generated out the web app damage us
  ○ Need to enforce at web server API

● Provide a good user experience
  ○ Don't let users make mistakes or warn them as soon as possible
  ○ Pushing validation closer to the user is helpful

● Validation in JavaScript frameworks (AngularJS/ReactJS)
  ○ Rule #1: Still need server-side validation to protect storage system integrity
  ○ Rule #2: Let user know about validity problems as early as possible
  ○ Both frameworks followed familiar HTML form/input model
Validation with AngularJS

- Angular reuses the HTML form tag
  ```html
  <form name="myForm">
    <input type="text" name="myName" ng-model="name" required
           ng-minlength="3" ng-maxlength="20" />
  </form>
  ```

- Generates a scope object property under form name (myForm)
  ```javascript
  $scope.myForm.myName has validation information
  ```
  - Two-way binding - Controller can do anything with JavaScript object
Angular validation information

$\text{scope.myForm.myName}$

Status: $\text{untouched, touched, pristine, dirty, valid, invalid}$

Error: $\text{error.required}$
- $\text{minlength}$
- $\text{maxlength}$

- Also updates classes on input tag (e.g. ng-invalid-maxlength)
- Can provide instant feedback on errors
Angular Material: md-input-container pattern

<form name="userForm" ...>
  <md-input-container>
    <label>Last Name</label>
    <input name="lastName" ng-model="lastName" required
             md-maxlength="10" minlength="4">
    <div ng-messages="userForm.lastName.$error" ng-show="userForm.lastName.$dirty">
      <div ng-message="required">This is required!</div>
      <div ng-message="md-maxlength">That's too long!</div>
      <div ng-message="minlength">That's too short!</div>
    </div>
  </md-input-container>
</form>
Asynchronous validation

- Can in background communicate with web server to validate input
  - Example: username already taken
- Example: states search with md-autocomplete

```html
<md-autocomplete md-selected-item="ctrl1.selectedItem"
    md-search-text="ctrl1.searchText"
    md-items="item in ctrl1.querySearch(ctrl1.searchText)"
    md-item-text="item.display" placeholder="What is your favorite US state?">
    <span md-highlight-text="ctrl1.searchText">{{item.display}}</span>
</md-autocomplete>
```

- Trend towards using recommendation systems for input guidance
Single Page App Input

- Rather than POST with redirect you can do a XMLHttpRequest POST/PUT
- Angular supports two interfaces to XMLHttpRequest ($http and $resource)

```javascript
function FetchModel(url, doneCallback) {
  $http.get(url).then(function(response) {
    var ok = (response.status === 200);
    doneCallback(ok ? response.data : undefined);
  }, function(response) {
    doneCallback(undefined);
  });
}
```
Minor Digression - Promises
Callbacks have haters

- Pyramid of Doom

```javascript
fs.readFile(fileName, function(error, fileData) {
    doSomethingOnData(fileData, function(tempData1) {
        doSomethingMoreOnData(tempData1, function(tempData2) {
            finalizeData(tempData2, function(result) {
                // Called **Pyramid of Doom**
                doneCallback(result);
            });
        });
    });
});
```

- An alternative to pyramid: Have each callback be an individual function
  - Sequential execution flow jumps from function to function - not ideal
Idea behind promises

- Rather than specifying a done callback
  
  ```javascript
  doSomething(args, doneCallback);
  ```

- Return a promise that will be filled in when done
  
  ```javascript
  var donePromise = doSomething(args);
  
  donePromise will be filled in when operation completes
  ```

- Doesn't need to wait until you need the promise to be filled in
then() - Waiting on a promise

- Get the value of a promise (waiting if need be) with then

```javascript
donePromise.then(function (value) {
  // value is the promised result when successful
}, function (error) {
  // Error case
});
```
Example of Promise usage

- `$http.get()` returns a promise

```
$http.get(url).then(function(response) {
    var ok = (response.status === 200);
    doneCallback(ok ? response.data : undefined);
}, function(response) {
    doneCallback(undefined);
});
```
Promises

```javascript
var myFile = myReadFile(fileName);
var tempData1 = myFile.then(function (fileData) {
    return doSomethingOnData(fileData);
});
var finalData = tempData1.then(function (tempData2) {
    return finalizeData(tempData2);
});
return finalData;
```

- **Note no Pyramid of Doom**
- Every variable is a promise
  - A standard usage: Every variable - If `thenable` call `then()` on it otherwise just use the variable as is.
Chaining promises

return myReadFile(fileName)
    .then(function (fileData) { return doSomethingOnData(fileData); })
    .then(function (data) { return finalizeData(data); })
    .catch(errorHandlingFunc);

- Add in ES6 JavaScript arrow functions:

  return myReadFile(fileName)
      .then((fileData) => doSomethingOnData(fileData))
      .then((data) => finalizeData(data))
      .catch(errorHandlingFunc);
From loadDatabase.js

- Mongoose returns promises so instead of async

```javascript
var removePromises = [User.remove({}), Photo.remove({}),
                     SchemaInfo.remove({})];

Promise.all(removePromises).then(...
```

-- and --

```javascript
var userPromises = userModels.map(function (user) {
    return User.create({ ...

Promise.all(userPromises).then(...
```
Creating your own promise

- Create a promise with `new Promise()`

```javascript
var donePromise = new Promise(function (fulfill, reject) {
  // calls `fulfill(value)` to have promise return value
  // calls `reject(err)` to have promise signal error
});
```
Converting callbacks to Promises

```javascript
function myReadFile(filename) {
    return new Promise(function (fulfill, reject) {
        fs.readFile(filename, function (err, res) {
            if (err)
                reject(err);
            else
                fulfill(res);
        });
    });
}
```
JavaScript and Promise

- Lots of slightly different JavaScript promise libraries
  Q, Bluebird, RSVP

- Used in many software packages
  ○ jquery, Angular, Protractor, ...

- JavaScript ES6 specification defines a Promise API
End Digression - Back to $\text{http API}$
Uploading models using $http.post

```
$http.post(url, modelObj).then(function successCallback(response) {
    // response.status     --- HTTP status code
    // response.data       --- POST response if successful (decoded)
    // response.headers    --- HTTP response headers
    }, function errorCallback(response) {
    // Network Error case (webServer or network down?)
    
});
```

- App must wait for reply since errors may occur on server
  - Need some user interface way of communicating this to the user
$resource - RESTful server access

- In REST APIs you have resources named as URLs
  
  var resource = $resource(resourceURLTemplate, paramDefaults);

- And operations on resources:
  
  resource.get(params, doneCback) - {method:'GET'}
  resource.save(params, doneCback) - {method:'POST'},
  resource.query(params, doneCback) - {method:'GET', isArray: true}
  resource.remove(params, doneCback) - {method:'DELETE'},
  resource.delete(params, doneCback) - {method:'DELETE'};
$resource examples

```javascript
var testRes = $resource("/test/info");
    var infoModel = testRes.get({}, function () {
        console.log('infoModel', infoModel);
    }, function errorHandling(err) {
        // Any error or non-OK status
    });

var userRes = $resource("/user");
    userRes.save({user: 'mendel', password: 'pwd'}, function () {
        // Success
    }, function errorHandling(err) {
        // Any error or non-OK status
    });
```
Server-side validation

- Regardless of validation in browser, server needs to check everything
  - Easy to directly access server API bypassing all browser validation checks

- Mongoose allows validator functions
  ```javascript
  var userSchema = new Schema({
    phone: { type: String,
      validate: {
        validator: function(v) {
          return /d{3}-d{3}-d{4}/.test(v);
        },
        message: '{VALUE} is not a valid phone number!'  
      }
    }
  });
  ```
Some integrity enforcement requires special code

- Maintaining relationship between objects
- Resource quotas
- Examples related to our Photo App
  - Only author and admin user can delete a photo comment.
  - A user can only upload 50 photos unless they have a premium account.