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" /><br />
  Deluxe: <input type="checkbox" name="delux" /><br />
  <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 not 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
  <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)
  $scope.myForm.myName has validation information
    - Two-way binding - Controller can do anything with JavaScript object
Angular validation information

`$scope.myForm.myName`

**Status:** `$untouched`, `$touched`, `$pristine`, `$dirty`, `$valid`, `$invalid`

**Error:** `$error.required`
  - `minlength`
  - `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
React form input pattern - JSX

```jsx
<form onSubmit={this.formSubmit}>
  <span>
    Input1: <input type="text" name="input1" onChange={this.updateInfo} />
  </span>
  <span>
    Input2: <input type="text" name="input2" onChange={this.updateInfo} />
  </span>
  <span>
    <button type="submit">Submit</button>
  </span>
</form>
```
JSX and this handling - No ideal way

- Specifying a method as DOM event callback doesn't work:
  ```jsx
  <form onSubmit={this.formSubmit}> ... // Wrong! Calls with this undefined
  ```

- Arrow function embedded in JSX render: Can call class method
  ```jsx
  <form onSubmit={(event) => this.formSubmit(event)}> ... 
  ```

- Redefine method function in instance to have correct this in constructor:
  ```jsx
  this.formSubmit = this.formSubmit.bind(this); // In component constructor
  ```

- Use new JavaScript class `fields-class`:
  ```jsx
  class Foo {
    fieldName = value;
  }
  ```
React form input pattern - JSX

class AppComponent extends React.Component {
    ...
    updateInfo = (event) => {
        this.setState({ [event.target.name]: event.target.value });
        // Could do validate feedback here.
    }

    formSubmit = (e) => {
        e.preventDefault(); // Need to prevent browser form submit
        // Validate and submit - this.state.input1, this.state.input2
    }
}
React input validation

- Unopinionated: meaning lots of unofficial options
  - Including some similar to the AngularJS support

- Roll your own. Example: Check inputs and set error message in state
  
  ```
  { this.state.validationErrorMsg ?
      <span style={{color:'red'}}> {this.state.validationErrorMsg}
      </span>
    : null }
  ```

- Style-guide driven classes are common. Example MaterializeCSS:
  
  ```
  <input className={!this.state.input1Invalid ? "invalid" : "valid"} ...}
  ```
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
Same code without pyramid: Control jumps around

```javascript
fs.ReadFile(fileName, readDone);

function readDone(error, fileData) {
  doSomethingOnData(fileData, doSomeDone);
}

function doSomeDone (someData) {
  doSomethingMoreOnData(someData, doSomeMoreDone);
}

function doSomeMoreDone (someMoreData) {
  finalizeData(someMoreData, doneCallback);
}
```
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 (as does axios)

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

```
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 --

  var userPromises = userModels.map(function (user) {
    return User.create({ ...
  
  Promise.all(userPromises).then(...
  ```
Creating your own promise

- Create a promise with \texttt{new Promise()}

  ```javascript
  var donePromise = new Promise(function (fulfill, reject) {
    // calls \texttt{fulfill(value)} to have promise return value
    // calls \texttt{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 $http API
Uploading models using $http.post

```javascript
$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
ReactJS Model uploading

- Unopinionated - Many options to choose from.
  - Axios - Promise based HTTP client for the browser and node.js

```javascript
axios.post(URLpath, objectWithParameters)
  .then((response) => {
    // response.status - HTTP response status (eg 200)
    // response.statusText - HTTP response status text (eg OK)
    // response.data - Response body object (JSON parsed)
  })
  .catch((err) => {
    // error.response.{status, data, headers} - Non-200x HTTP response
    // if !error.response - No reply, can look at error.request
  });
```
$resource - AngularJS RESTful server access

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

- And operations on resources:
  
  ```javascript
  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.