Teemu Suvinen

Angular migration

Case: File Manager

Metropolia University of Applied Sciences

Bachelor of Engineering

Media Technology

Thesis

4.5.2017

Author Title Number of Pages Date	Teemu Suvinen Angular migration 32 pages + 3 appendices 4 May 2017
Degree	Bachelor of engineering
Degree Programme	Media technology
Specialisation option	Digital media
Instructor	Ilkka Kylmäniemi, Lecturer

The purpose of this thesis was to compare the differences between AngularJS and Angular frameworks and to migrate a file managing -tool, developed and used by a Finnish media company, to utilize the new Angular framework.

Single page applications are more popular than ever as they can provide a native application like user experience. AngularJS and Angular are frameworks designed for single page applications. They provide a robust set of tools to create data-driven, rich applications.

As the web and web development have become more advanced, many of the AngularJS features are now outdated. Angular is a rewrite of AngularJS, written in TypeScript and ES6. It takes some of the concepts from its predecessor and improves the stability and performance of the framework.

The project was to migrate a file manager application from using AngularJS to Angular. Some of the functionalities stayed somewhat unchanged, but as the application was split into multiple smaller components a shared service had to be created to handle the communication across the application. New features were added to help with the workflow process.

The project was successful and no significant problems were faced. The most challenging part of the migration was to get into the Angular mindset of building an application with small components. The results show that for a small-to-medium sized application, an Angular migration can be done without significant problems. A large application can be migrated to Angular in stages.

Keywords	Angular, AngularJS, TypeScript, ES6, programming, web application

Tekijä Otsikko Sivumäärä Aika	Teemu Suvinen Angular-migraatio 32 sivua + 3 liitettä 4.5.2017
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaaja	Lehtori Ilkka Kylmäniemi

Insinöörityön tarkoituksena oli tutkia AngularJS- ja Angular-sovelluskehyksien eroja sekä siirtää suomalaisessa media-alan yrityksessä käytettävän tiedostonhallintatyökalun käyttöliittymä uuden Angular-sovelluskehyksen päälle.

Yksisivuiset verkkosovellukset ovat suositumpia kuin koskaan. Tämä johtuu siitä, että ne voivat tarjota natiivisovellusten kaltaisen käyttökokemuksen. AngularJS ja Angular ovat sovelluskehyksiä, jotka on suunniteltu juuri yksisivuisten verkkosovellusten toteuttamiseen. Ne antavat vakaat työkalut datan ohjaamien, rikkaiden sovellusten kehitykseen.

Verkon ja web-ohjelmoinnin kehittymisen myötä AngularJS-sovelluskehyksen ominaisuudet ovat nykyään vanhentuneita. Angular on täysin uudelleenkirjoitettu AngularJS:stä. Sen kirjoittamiseen on käytetty TypeScript- ja ES6-ohjelmointikieliä. Sen kehityksessä on hyödynnetty AngularJS:n konsepteja, ja sovelluskehyksen vakautta ja toimintaa on parannettu.

Insinöörityössä siirrettiin AngularJS-pohjainen tiedostonhallintatyökalu käyttämään uutta Angular sovelluskehystä. Osa toiminnoista pysyi lähes muuttumattomana, mutta koska sovellus koostettiin useasta pienestä komponentista, sovelluksen komponenttien kommunikointia varten piti luoda jaettu palvelu. Samalla sovellukseen lisättiin uusia toimintoja helpottamaan työnkulkua.

Projekti onnistui ilman merkittäviä ongelmia. Haastavin osa siirrossa oli oppia Angular-tyyppinen ajattelutapa, jossa sovellus rakennetaan pienistä komponenteista. Insinöörityön tulokset osoittavat, että Angular-sovelluskehyksen käyttöönotto tai vaihto voidaan tehdä pienelle tai keskisuurelle sovellukselle ilman merkittäviä ongelmia. Suuret sovellukset voidaan siirtää vaiheittain käyttämään Angular-sovelluskehystä.

Avainsanat	Angular, AngularJS, TypeScript, ES6, ohjelmointi, verkkosovellus

Table of contents

List of abbreviations

List of abbreviations

AJAX	Asynchronous JavaScript and XML. A method used to request data from a server after the page has loaded.
API	Application Programming Interface. A set of clearly defined methods of communication between software components.
CDN	Content Delivery Network. Globally distributed network of servers.
CSS	Cascading Style Sheets. A language used to describe the presentation of a document
DOM	Document Object Model. The logical structure of documents.
ES6	ECMAScript 6. A trademarked scripting language specification standardized by European Computer Manufacturers Association.
HTML	Hypertext Markup Language. A markup language to define the structure of a web application.
JavaScript	A programming language used to add interactions and behavior to websites.
JSON	JavaScript Object Notation. A lightweight data-interchange format.
JSONP	JSON with Padding. Used to request data from a server residing in a different domain.
MVC	Model-View-Controller. A programming architecture design model.

NPM	Node Package Manager. A tool to handle packages and dependencies in modern web development environment
SPA	Single Page Application. A web application that is presented to a user through a single HTML page.
UI	User Interface. A visual part of a computer application through which a user interacts with the software.
URL	Uniform Resource Locator. A reference to a resource on the Internet
UX	User Experience. Internal experience a user has as they interact with every aspect of an application.

1 Introduction

Single page applications, or SPAs, are applications built for web. They have become increasingly popular as they can offer a native application like experience and usually provide more dynamic interactions than conventional websites. Despite being called single page applications, they can have multiple views and states. Usually, only a part of the page is updated when a user navigates inside the application.

In this thesis AngularJS, a popular framework to create single page applications, is introduced and compared to its successor, Angular. The goal is to demonstrate the main differences in the Angular frameworks. First, some of the basic features and functionalities of the AngularJS framework are presented. Afterwards, similar features are examined in the new Angular framework and lastly the two frameworks are compared.

The practical portion of the thesis demonstrates how an application can be migrated from using AngularJS to Angular. The initial state of the application and the plan for the migration process is explained. Then some of the decisions in the rewrite process will be presented. Lastly a look over the finished product with some of the improvements is taken.

AngularJS and Angular frameworks provide a robust set of tools to create dynamic and data-driven web applications. They extend the native HTML elements with custom attributes, *directives*, and allow writing expressive, custom elements with complex functionality. Both Angular versions utilize the MVC (Model-View-Controller) programming architecture design. Angular takes many of the concepts from its predecessor, but improves the performance and stability of the framework. Angular was built for modern browsers and mobile devices.

2 Angular frameworks

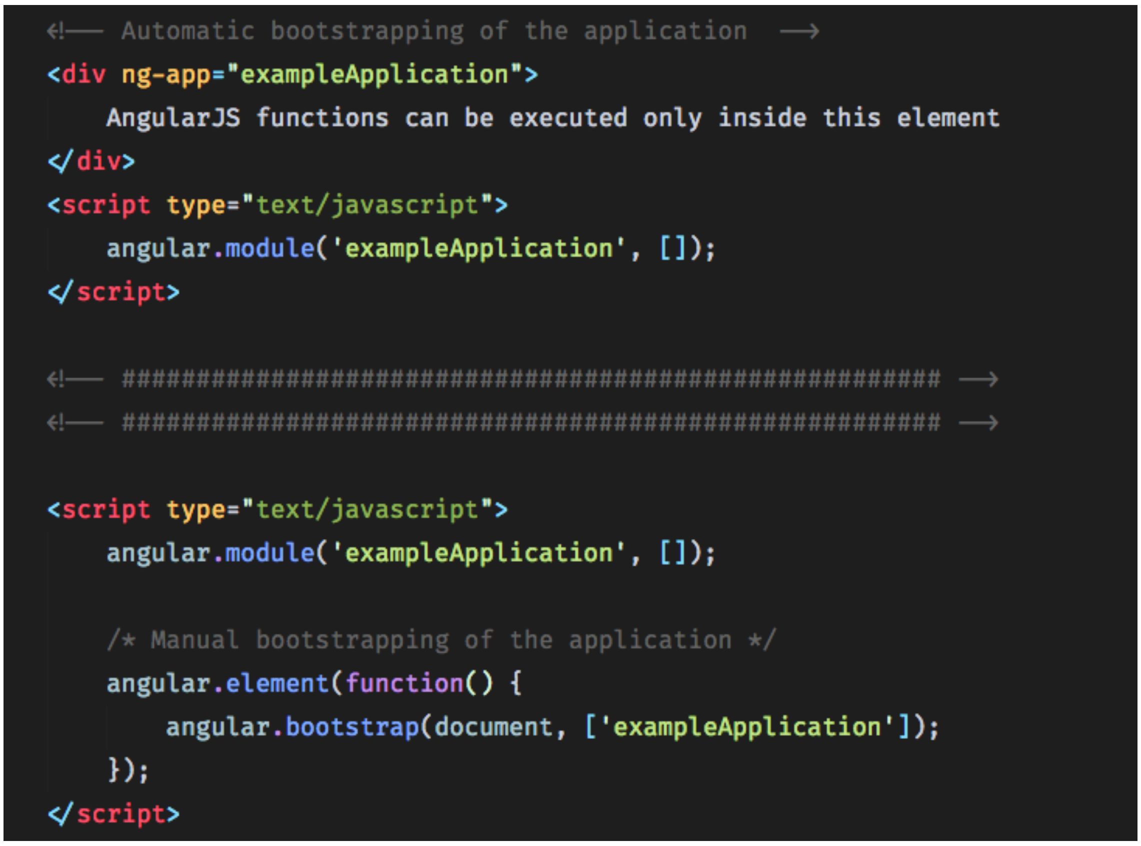
2.1 AngularJS

AngularJS is a client-side, structural framework written entirely in JavaScript. It is developed and managed by Google and later in collaboration with a community of individual developers and corporations. AngularJS was initially released on October 21, 2010 as a version “angular-0.9.0 dragon-breath” and is distributed under open source MIT license. (1; 2).

The main purpose of AngularJS is to give a developer a robust set of tools to build a variety of dynamic, client centric applications while using HTML as a template language and further extending the HTML syntax to express web application’s components clearly and concisely. AngularJS takes care of advanced features that developers have become accustomed to in modern web applications, such as separation of application logic, data models and views, AJAX services, dependency injection, testing and more. As AngularJS is written in JavaScript, it runs in the web browser and can therefore be used with any server technology. (1; 3).

In order to execute any AngularJS functions, a source file of the AngularJS framework must be linked in the HTML template. The source file can be downloaded from the official AngularJS website or with a *package manager* like Bower or NPM (Node Package Manager) and then hosted on the same server as the application, or it can be loaded remotely from a content delivery network (CDN). (4). The latter method can reduce bandwidth costs and improve page load times. However, including any file necessary for the functionality of an application, via CDN is more prone to errors. A content delivery network can be unavailable in certain countries of the World or the server could go down possibly leaving an application unusable for that time. Therefore, any essential file for an application should be hosted on the same server. (3; 5)

When the framework file is linked to a template, the application must be initialized, “bootstrapped”, with the “ng-app” *directive*. This defines the root element of the AngularJS application and initializes the application automatically when a web document is loaded. Any scripts written in AngularJS can be executed inside of this element only. The application can also be bootstrapped manually in a JavaScript file if more control of the initialization process is desired, for example if the application requires script loaders or needs to perform operations before AngularJS compiles a page. (6; 7). Figure 1 shows the two possible ways to initialize an AngularJS application.



```
<!-- Automatic bootstrapping of the application  -->
<div ng-app="exampleApplication">
    AngularJS functions can be executed only inside this element
</div>
<script type="text/javascript">
    angular.module('exampleApplication', []);
</script>

<!-- ####################################################### -->
<!-- ####################################################### -->

<script type="text/javascript">
    angular.module('exampleApplication', []);

    /* Manual bootstrapping of the application */
    angular.element(function() {
        angular.bootstrap(document, ['exampleApplication']);
    });
</script>
```

Figure 1. AngularJS application can be bootstrapped automatically, using *ng-app* directive, or manually with JavaScript. Note that all controllers and modules must be defined before bootstrapping the application manually.

The application should only be bootstrapped once and only one method of initialization can be used as multiple initializations or root elements may lead to runtime errors in the application. If there is no *ng-app* directive found in the DOM, and the application is not manually bootstrapped, AngularJS will not run.

Both initialization methods require a definition of an application – a *module*. A module is a single core unit, a “container”, that encapsulates all of the application code. An application can contain several modules, each one containing code that pertains to specific functionality. A module is declared with the “*angular.module( )*” method. The method needs two parameters: a name of the module and a “*requires”* array of names of external modules that the declared module can use. If no external modules are needed, the requirement array can be left empty. The name of a module is then passed to the bootstrapping process to tell AngularJS which module to initialize when an application is loaded in the browser. (3; 7).

AngularJS utilizes the MVC (Model-View-Controller) software design pattern. MVC is an architecture principle where the data model, user interface, and the application logic are isolated from each other. It allows writing more maintainable and reusable code by organizing it into smaller components. (8).

Model is where the application’s data is stored. It is the lowest level of the MVC-pattern. The model doesn’t need to know how to interact with the views or controllers. It only contains and handles the data and methods to manipulate the view. If a model changes, it will notify its observers that a change has occurred.

View is the whole or part of an interface that is presented to the users. The view usually consists of the HTML markup, templates and CSS attached to the DOM elements. It does not need to know how to manipulate data objects, only how to display them. The view should be aware of the models via controllers in order to observe data changes, but do not directly communicate with the models.

Controllers are the functions that handle and validate the user input like clicks or typing in the view and perform interactions on the data model objects when necessary. Controllers act as an interface between the models and views and process the data before it is rendered to the view. Controllers perform asynchronous API calls to the server, handle the received responses and update the data models or views accordingly.

The MVC model is visualized in figure 2.

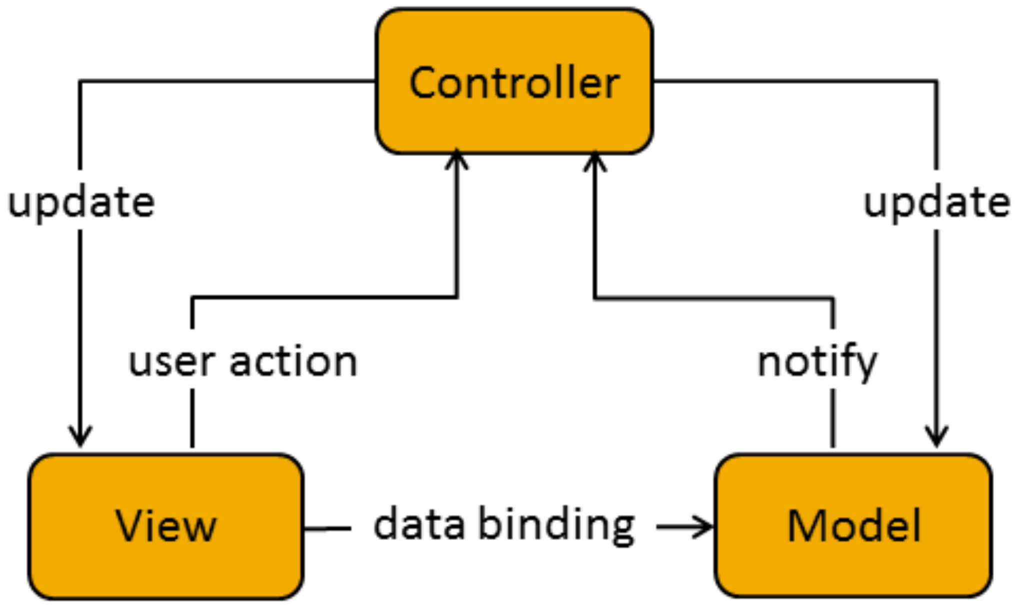
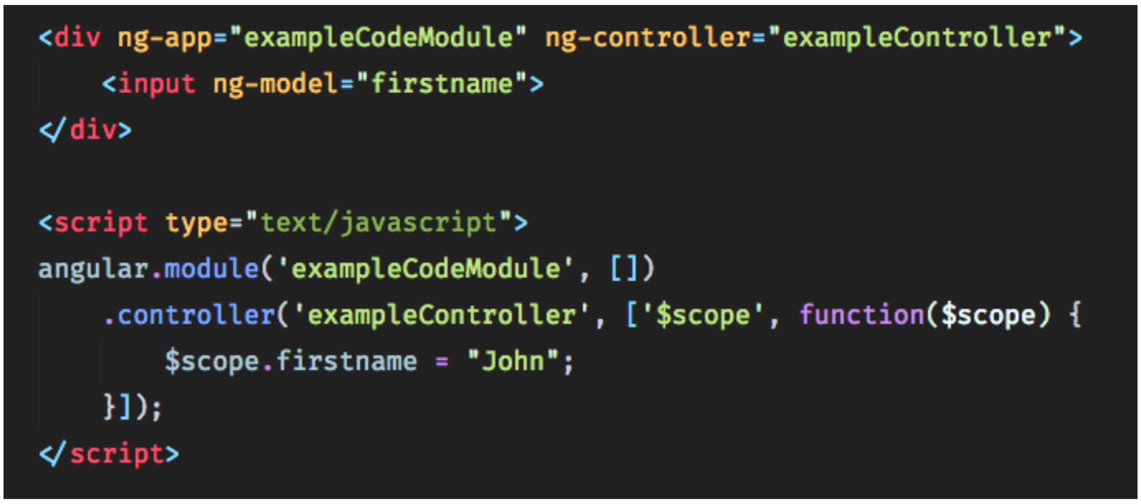


Figure 2. Visualization of the traditional MVC model (10.). Controllers handle the state of the models and views and update them if necessary.

AngularJS handles the MVC model via two-way data binding. It is an automatic and more efficient way (compared to the traditional “manual” way) to update the view whenever the data model changes, as well as updating the model if the view changes. (3; 8; 9).

Two-way data binding eliminates the need for a developer to manually write synchronization code to keep track of data model and user interface thus keeping the codebase smaller, cleaner and easier to maintain. Two-way bound values are mostly used in form input fields where the data model has to change constantly based on the user input. Two-way bound variable values can be pre-defined based on the data provided by the data model or they can be defined later based on the user input. A pre-defined value will change as soon as the user interacts with it. Pre-defining a two-way data bound variable value is done in the *scope* of a *controller.* (3; 9). Figure 3 shows how two-way data binding is marked in the HTML template with AngularJS *directive* “ng-model”.



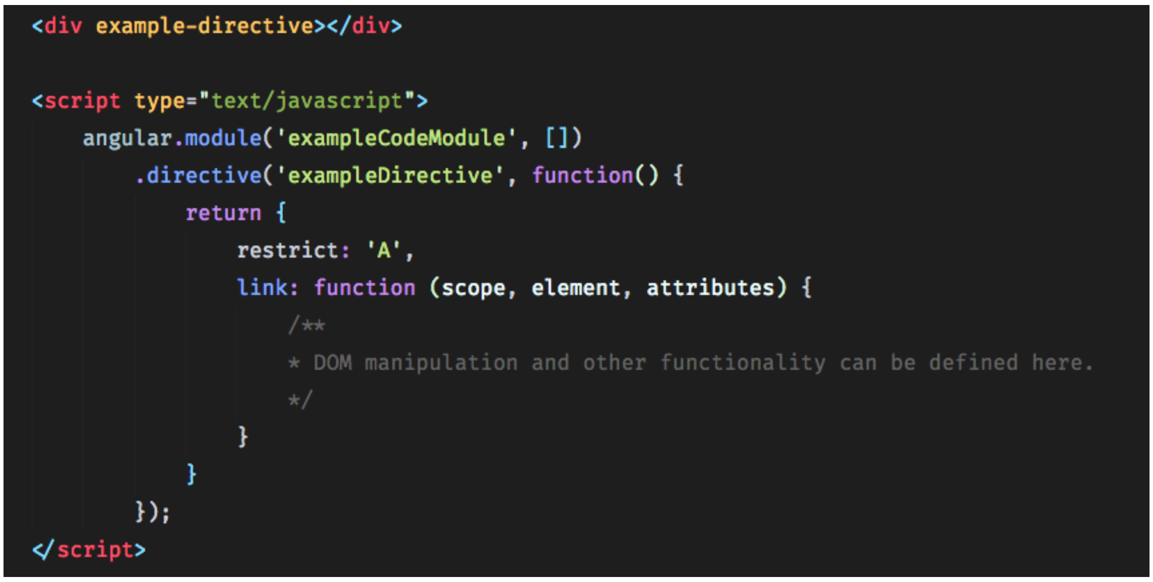
```
<div ng-app="exampleCodeModule" ng-controller="exampleController">
    <input ng-model="firstname">
</div>

<script type="text/javascript">
angular.module('exampleCodeModule', [])
    .controller('exampleController', ['$scope', function($scope) {
        $scope.firstname = "John";
    }]);
</script>
```

Figure 3. Two-way binding of a variable is defined in the HTML markup with “ng-model” *directive*. In this case the value for the variable “firstname” is pre-defined as “John” in the *controller* named exampleController

Ng-model is a built-in *directive* in AngularJS. Directives are extended HTML attributes, element names, comments or CSS-classes that offer additional functionality to the web application. Directives tell AngularJS’s HTML compiler to attach a specified behavior to that DOM element or to transform the DOM element and/or its child elements.

In addition to the built-in directives a developer can define and use their own custom-built directives. All built-in directives in AngularJS have “ng” prefix to help the developer distinguish the built-in directives from the HTML native attributes or the developer’s own custom directives. A developer should never prefix their custom directives with “ng” in order to avoid namespace collision.

Custom directives are defined with “*angular.directive()*” method, shown in Figure 4. The method takes two arguments: a name of the directive, as a string, that will be used to refer to inside of the views, and a factory function that returns an object defining the behavior of a directive. It is expected to return an object providing options that tell the AngularJS’s compiler service how the directive should behave when it is invoked in the DOM. The link function of a directive has control over live data-bound DOM. This means that the directive link function gets executed after the template has been cloned and, thus, it can be used to listen for DOM events or update the DOM. Use of directives is recommended if manipulation of a DOM element is desired. (3; 6; 12;)



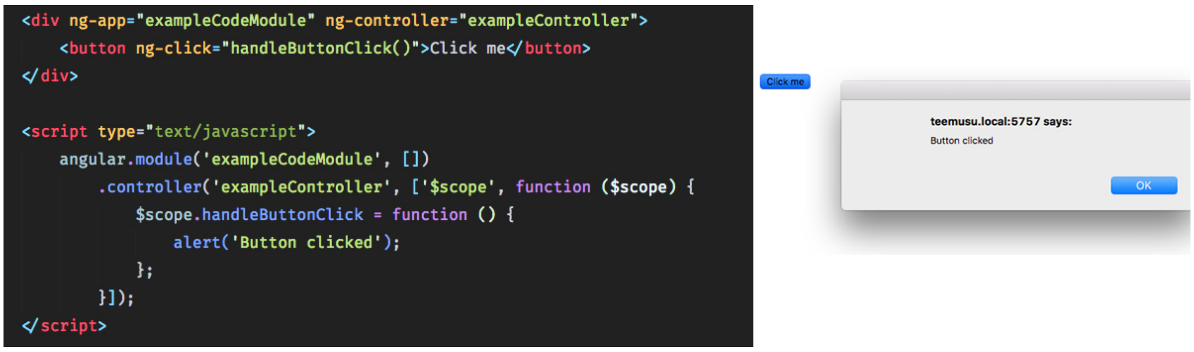
```
<div example-directive></div>

<script type="text/javascript">
    angular.module('exampleCodeModule', [])
        .directive('exampleDirective', function() {
            return {
                restrict: 'A',
                link: function (scope, element, attributes) {
                    /**
                    * DOM manipulation and other functionality can be defined here.
                    */
                }
            }
        });
</script>
```

Figure 4. Custom directives are defined with the *directive()* method. Link functions can manipulate the DOM, or add other functionality to the element.

It is possible to tell AngularJS in which format a directive can be declared in the DOM. This is done with a *restrict* argument, as shown in Figure 4. There are four possible ways to declare directives: as an element (restrict E), as an attribute (restrict A), as a comment (restrict M) and as a class (restrict C). It is recommended to use the element or attribute declaration methods as they will work with most browsers. (12).

Controllers in AngularJS handle the "business logic" of an application. They are defined by a JavaScript constructor function and are then used to augment the *scope* of the view in an application. When a new controller is created on a page, AngularJS passes it a new scope where initial state and custom behavior to the *scope* object can be set up. In addition to declaring initial values of data bound variables used in the views, controllers are used to handle the logic of a single view in a single container. All of the scope properties are available to the template at the point in the DOM where the controller is registered. (3; 6; 13).

Controller functions can be executed based on the user actions in the view. A typical practice is to execute scope functions when a user clicks a button or when a data model changes. Figure 5 shows an example of how a controller scope function is defined and executed when a user clicks a button in the view. (3).



```
<div ng-app="exampleCodeModule" ng-controller="exampleController">
    <button ng-click="handleButtonClick()">Click me</button>
</div>

<script type="text/javascript">
    angular.module('exampleCodeModule', [])
        .controller('exampleController', ['$scope', function ($scope) {
            $scope.handleButtonClick = function () {
                alert('Button clicked');
            };
        }]);
</script>
```

teemusu.local:5757 says:
Button clicked

Figure 5. Scope functions can be executed when a user clicks a button in the view. $scope is passed to the controller via *dependency injection* – a method of providing dependencies to components as requested. A click is registered with a built in *ng-click* directive.

A new controller is defined with the "*angular.controller()*" method. It takes two arguments: a name of the controller, and a constructor function. The constructor function is then *injected* with the necessary dependencies for that function. (13).

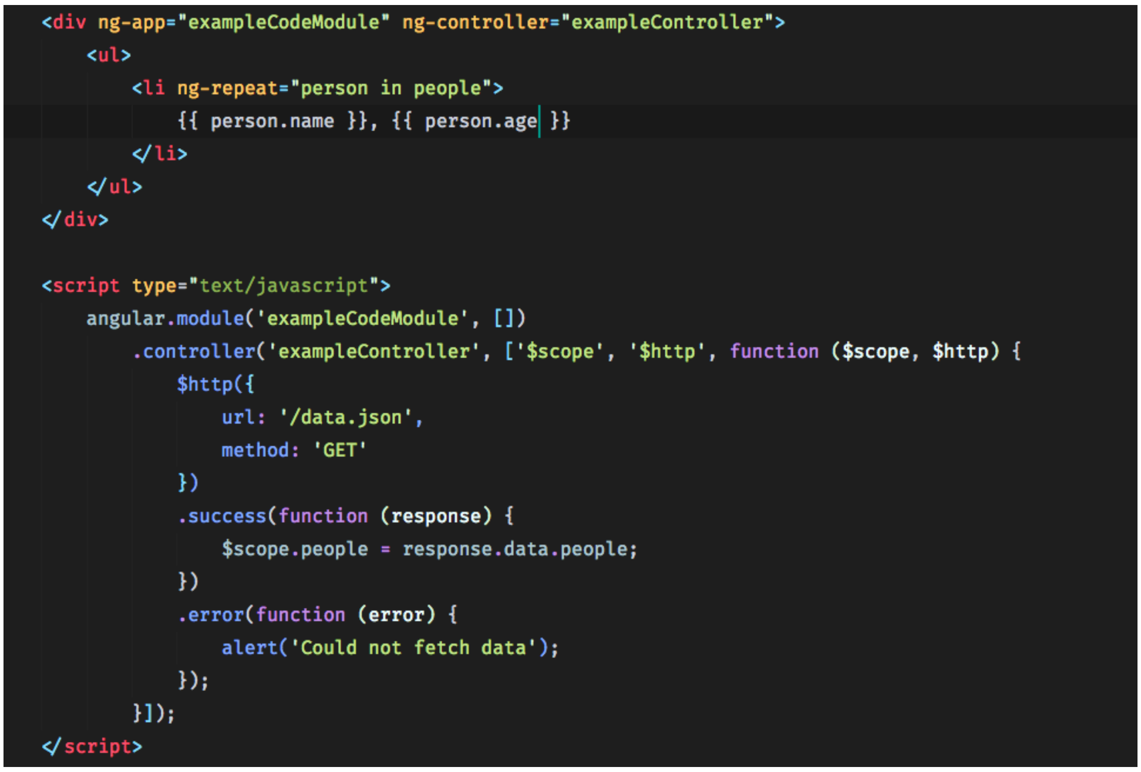
A major distinction between AngularJS and other JavaScript frameworks is that the controller is not the appropriate place to do any direct DOM manipulation or formatting, data manipulation or fetching, or manage the life-cycle of other components. It is best practice to do these kind of actions via directives or *services*.

For memory and performance purposes, controllers are instantiated only when they are needed and discarded when they are not. It means that every time a *route* is switched or a view is reloaded, the current controller is cleaned, or reset by AngularJS. (3; 6; 13).

Services provide a method of keeping data around for the lifetime of an application, and communicate across multiple controllers in a persistent manner. Services are singleton objects instantiated only once and created only when necessary. AngularJS provides multiple built in services. One of the most common used service is the *$http* service that makes requests to the server via the browser's XMLHttpRequest object or via JSONP, and lets an application handle the response.

The $http service is based on the deferred/promise API. It takes a single argument, a configuration object, that is used to generate an HTTP request and returns a promise. A promise is a method of resolving or rejecting a value in an asynchronous manner.

The promise returned by the $http service contains two helper methods – *success* and *error*, that can be used to handle the logic of what happens when a request to a server is complete. (3; 14; 15). A basic usage of $http service is shown in Figure 6. Note that the $http service is only used in the controller in order to keep the example short.



```
<div ng-app="exampleCodeModule" ng-controller="exampleController">
    <ul>
        <li ng-repeat="person in people">
            {{ person.name }}, {{ person.age }}
        </li>
    </ul>
</div>

<script type="text/javascript">
    angular.module('exampleCodeModule', [])
        .controller('exampleController', ['$scope', '$http', function ($scope, $http) {
            $http({
                url: '/data.json',
                method: 'GET'
            })
            .success(function (response) {
                $scope.people = response.data.people;
            })
            .error(function (error) {
                alert('Could not fetch data');
            });
        }]);
</script>
```

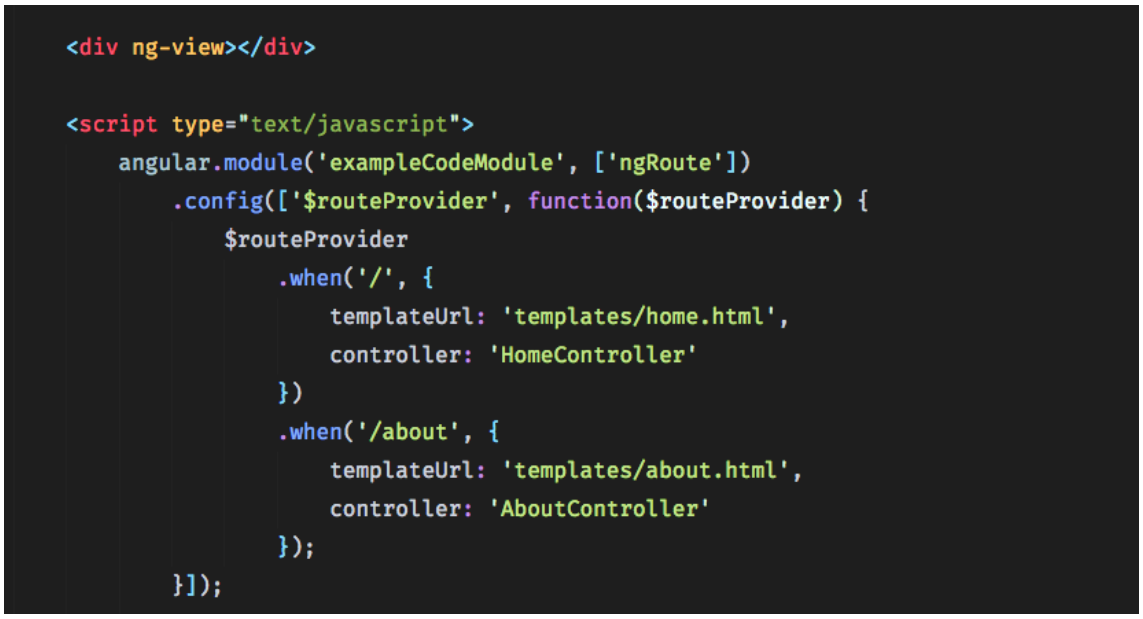
Figure 6. Basic usage of the $http service. Data is fetched from a server and stored in a scope variable *people*. This data is then used to iterate over data objects in the template using AngularJS *ng-repeat* directive. Data values are bound to the template using curly brace notation.

In addition to using built-in services, a developer can create custom services by registering the service's name and service factory function with an AngularJS module API. It is considered as the best practice to use services when an application uses same functions in multiple controllers. Using services minimizes writing repetitive code and keeps the codebase cleaner by centralizing most needed functions in one place. (3; 6; 15).

Single page applications have become increasingly popular as they are able to provide the feel of a phone or tablet application. The most notable difference between a regular website and a single page application is the reduced amount of page refreshes. A single page application utilizes AJAX requests to communicate with a server without doing a full page refresh, and only a part of the view is updated with fresh data. Single page applications can have multiple views on one page. (16). AngularJS provides a *router* to transition between these views. The views are split into templates and displayed depending on the requested route. (17.)

As of version 1.2.2, AngularJS router has been separated from the core of Angular into its own module. To make an AngularJS application to work as a single page application, a router module must be included to the base template and injected as a dependency to the application’s main module. After the router module is injected to the main module, routes are defined within the application configuration block using the route provider.

Each route has its own template (view), controller, and a url that can be used to implement deep-linking services and directives, and bookmarking a page or view. The views are rendered inside a main view specified with a *ng-view* directive from the router module. This tells AngularJS compiler where a template should be placed in the DOM when a route is requested. Figure 7 shows a basic routing example.



```
<div ng-view></div>

<script type="text/javascript">
    angular.module('exampleCodeModule', ['ngRoute'])
        .config(['$routeProvider', function($routeProvider) {
            $routeProvider
                .when('/', {
                    templateUrl: 'templates/home.html',
                    controller: 'HomeController'
                })
                .when('/about', {
                    templateUrl: 'templates/about.html',
                    controller: 'AboutController'
                });
        }]);
```

Figure 7. After the router module, *ngRoute*, has been injected to the main module, routes are defined using *$routeProvider* in the configuration block of the application.

The route provider uses “*when*” method to inspect the url in the browser and provides “components” to the view accordingly. The default AngularJS router is somewhat limited in functionality and, hence, many developers choose third-party router modules, like Angular UI router, which offers multiple views on a same page and nested views. (3; 17; 18).

As with any single page application, search engine optimization can be a difficult task. Search engines use automated robots, called “crawlers” or “spiders”, to decipher and index web pages. As single page application views are rendered in the browser using JavaScript, and AngularJS adds a hash to the url before the last fragment (name of the view), and nothing after the hash gets sent to the server, search engine crawlers have a hard time indexing the page. One option is to pre-render every view of the application and store them as static HTML files saved on the server. This allows the crawlers to index pages easily. (19; 20).

While AngularJS offers great tools for building and prototyping data-driven, dynamic and interactive web applications quickly and expressively, it has some notable drawbacks. Learning AngularJS basics are quite easy, but when the application grows and more advanced techniques are needed, the learning curve becomes very steep making it hard to learn. AngularJS scopes are a great way to handle user interaction, but they are hard to debug when errors occur as the error messages are not as expressive as in other JavaScript frameworks. A major con in AngularJS is the lack of complete, up to par, documentation, and the difficulty of search engine optimization. (20; 21; 22).

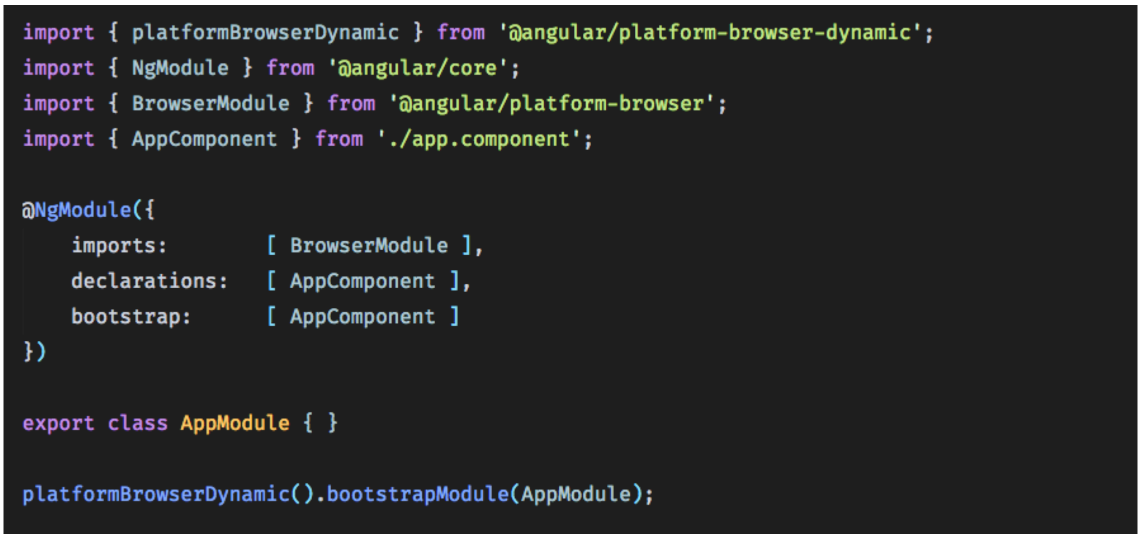
2.2 Angular 2

The web and web development have evolved a lot since the first release of AngularJS in 2010 and, thus, a lot of the concepts within AngularJS have become outdated. Updating and modernizing the existing AngularJS framework would have been impossible without breaking much of the original implementation. In 2014 the Angular team announced that they are going to build a complete rewrite over the popular AngularJS framework. In September, 2016, the final stable release version of Angular 2 was published. (23; 24).

Angular 2 – or just Angular, is a complete rewrite of the older AngularJS framework. It was built as an open source project from the beginning. Angular was designed for the modern web browsers as well as mobile platforms. It introduces a new syntax for writing code, and modular approach to building applications with components. Angular is written in TypeScript, a superset of JavaScript ES6 (ECMAScript 6) that compiles to plain JavaScript that runs in any browser, host or operating system. TypeScript enables optional static typing and class-based object-oriented programming. Angular is still, much like AngularJS, used to create single page applications. (23; 24; 25;)

Because Angular applications are written in TypeScript and ES6, there are more prerequisites for the development environment. TypeScript and ES6 files need to be compiled to plain JavaScript before they can run in the browser. NodeJS and NPM are essential to Angular development. Using NodeJS *task runners*, TypeScript and ES6 syntax is transpiled into plain JavaScript. NodeJS *task runners* can also, among other tasks, transpile SCSS, a CSS pre-processing language, into CSS. NodeJS compiler watches the files and recompiles an application when changes to the files are saved in the editor. (23; 26; 27).

Like AngularJS applications, Angular applications must have at least one root module that is bootstrapped, initialized, to launch the application. An Angular module class describes how the different parts of the application fit together. An Angular module is a class with an *@NgModule decorator.* Decorators are functions that modify JavaScript classes and add metadata to those classes. Metadata tells Angular how the classes should work. (23; 28). Figure 8 shows the initialization of an Angular application.



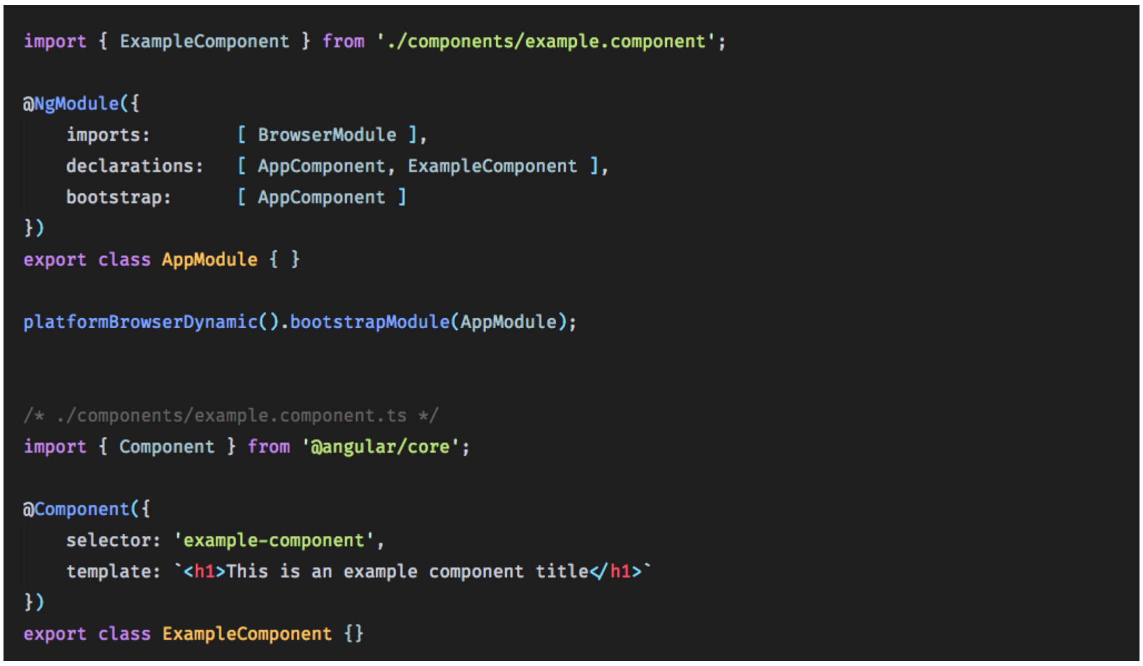
```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
    imports:        [ BrowserModule ],
    declarations:   [ AppComponent ],
    bootstrap:      [ AppComponent ]
})

export class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Figure 8. Bootstrapping an Angular application is done inside the *@NgModule decorator* function that takes a single metadata object whose properties describe the module.

Modules needed to write Angular code are imported from Angular library. The ‘from “*@angular/ ...*”’ portion tells the TypeScript compiler where to find the dependencies that are imported. The bootstrap method needs to know which *component* is set up as the root element of the application. (23; 28).

In Angular, applications are built using components. Components are much like native HTML elements. They are a combination of an HTML template and a component class that controls a view – a portion of the screen. Components are the main way to build and specify elements and logic on the Angular application. An Angular application is a “tree structure” of components.

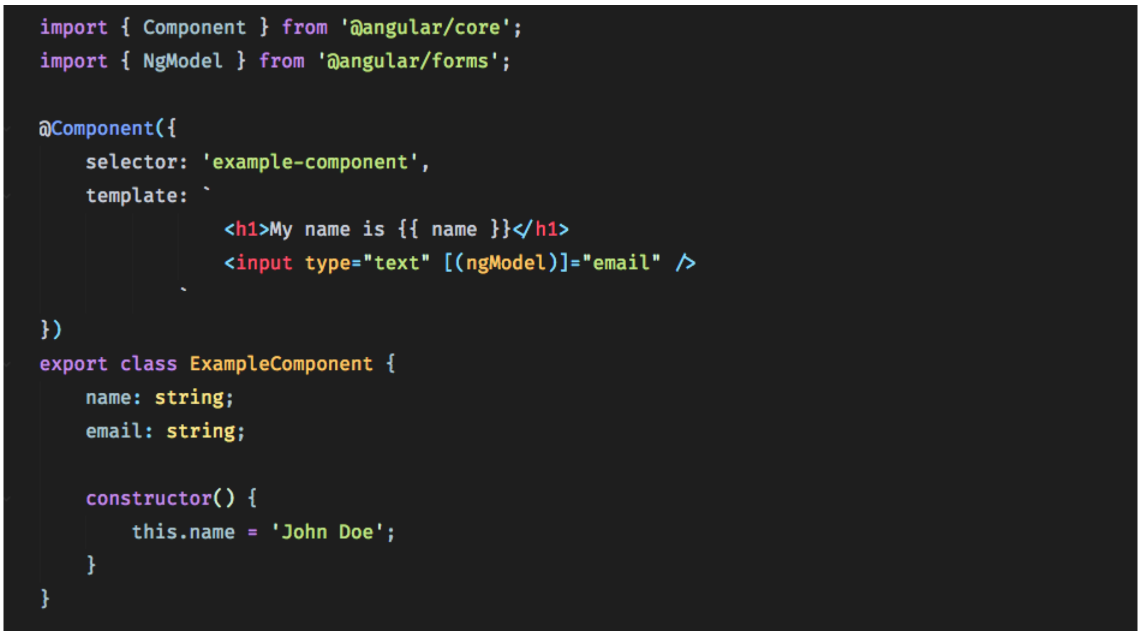
A class is declared as a component using @*Component* decorator and metadata provided to it. Component metadata determines how the component should be processed, instantiated and used at runtime. A component decorator needs at least two arguments: a *selector*, that is used in the DOM to render the component, and a *template* as a path to a template file or inline HTML directly inside the component decorator metadata. A component class must be then exported and imported, and injected, to the declarations array of a module. (29; 30; 31). A declaration of a new component is demonstrated in Figure 9.



```
import { ExampleComponent } from './components/example.component';

@NgModule({
    imports:        [ BrowserModule ],
    declarations:   [ AppComponent, ExampleComponent ],
    bootstrap:      [ AppComponent ]
})
export class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);


/* ./components/example.component.ts */
import { Component } from '@angular/core';

@Component({
    selector: 'example-component',
    template: `<h1>This is an example component title</h1>`
})
export class ExampleComponent {}
```

Figure 9. A new component class is declared using a @*Component* decorator. The component is then exported, imported to the main module file, and injected to module declarations as a dependency.

Angular components replace the controllers and template directives from AngularJS. Component's application logic and initial data values are defined inside a class constructor function. Two-way data binding was removed from Angular, but it is available via external *FormsModule*, and can be used similarly to AngularJS. (31). Figure 10 demonstrates the definition of initial values inside a constructor function.

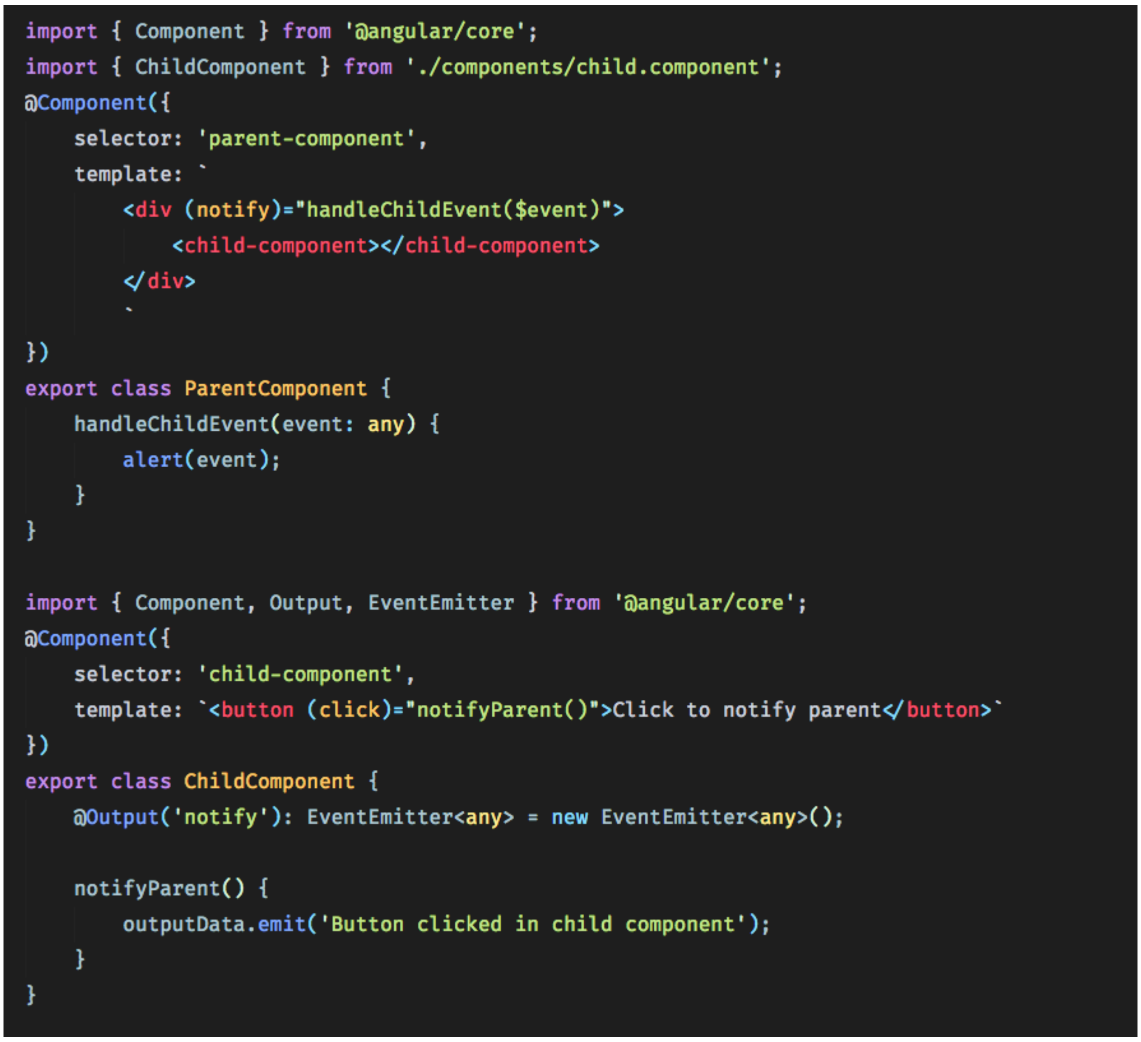


```
import { Component } from '@angular/core';
import { NgModel } from '@angular/forms';

@Component({
    selector: 'example-component',
    template: `
                <h1>My name is {{ name }}</h1>
                <input type="text" [(ngModel)]="email" />
              `
})
export class ExampleComponent {
    name: string;
    email: string;

    constructor() {
        this.name = 'John Doe';
    }
}
```

Figure 10. Initializing component values is done in the constructor function. Two-way data binding requires importing an external module, and is used using “banana-in-a-box” notation. Multi-line template can be written using back ticks inside of the component decorator.

TypeScript enables pre-defining the *types* of the variable values. In Figure 10, values need to be of type “string” in order to compile the file successfully. If the values are of any other type (e.g. Boolean or numbers) the TypeScript compiler will not process the file and throws an error. Types help to find potential errors before an application is run in the browser, reducing the risk of runtime errors. (23; 25; 27).

When a component is instantiated, Angular creates an encapsulated instance data object (much like isolated scopes in AngularJS), and this data is only available for the component. Components can receive data from, or output data to their parent components. Angular also enables creating custom events that allow the component to communicate with other components in the view. Custom events can be used to notify parent components when a data model of a child component changes or a child component completes a certain action that the parent needs to know about. Custom events are emitted out of the component with the Angular @*Output* decorator and a built-in EventEmitter class. (23; 30). Figure 11 shows a basic example of emitting data from a child component to a parent.

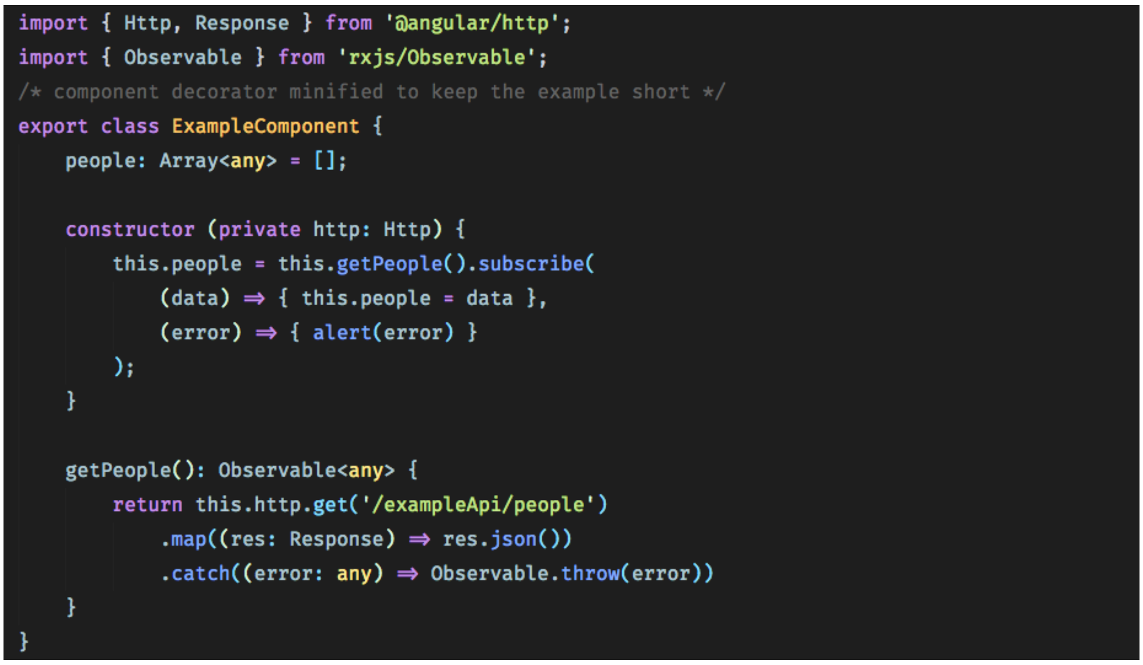


```
import { Component } from '@angular/core';
import { ChildComponent } from './components/child.component';
@Component({
    selector: 'parent-component',
    template: `
        <div (notify)="handleChildEvent($event)">
            <child-component></child-component>
        </div>
        `
})
export class ParentComponent {
    handleChildEvent(event: any) {
        alert(event);
    }
}

import { Component, Output, EventEmitter } from '@angular/core';
@Component({
    selector: 'child-component',
    template: `<button (click)="notifyParent()">Click to notify parent</button>`
})
export class ChildComponent {
    @Output('notify'): EventEmitter<any> = new EventEmitter<any>();

    notifyParent() {
        outputData.emit('Button clicked in child component');
    }
}
```

Figure 11. Custom events or a data change in a child component can be emitted to parent components using an Event Emitter and an @*Output* decorator.

Services in Angular are similar to AngularJS. Services are used to eliminate repetitive code and provide methods and data to different parts of the application. In Angular, services are used to create asynchronous data flow to a component. Single page applications utilize AJAX requests to fetch data from, or post data to a server. To help developers manage the requests needed to handle data, Angular comes with its own HTTP library which can be used to call out to external APIs. The Http module is separated from the core of Angular. In order to make Http requests to a server using Angular, the Http module must be imported to a service using it. After importing, the imported module must be injected as a dependency to the component (or service) constructor function. (23; 32).

In Angular, the preferred method of dealing with asynchronous data requests is using *Observables.* Using observables to structure the data is called *Reactive Programming.* Observables are objects or functions that represent a push based collection. This means that every time a change in the data is detected, an observable object notifies about this change to its consumers. Observables open up a continuous channel of communication in which multiple values of data can be emitted over time. Figure 12 shows how an API request can be performed and how the data fetched from a server is handled using Observables.



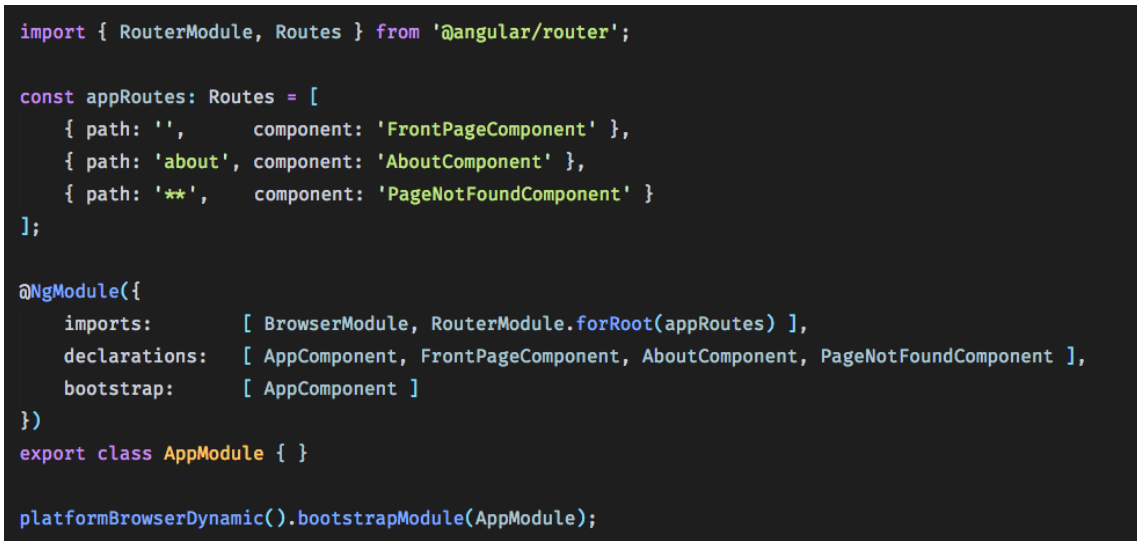
```
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
/* component decorator minified to keep the example short */
export class ExampleComponent {
    people: Array<any> = [];

    constructor (private http: Http) {
        this.people = this.getPeople().subscribe(
            (data) => { this.people = data },
            (error) => { alert(error) }
        );
    }

    getPeople(): Observable<any> {
        return this.http.get('/exampleApi/people')
            .map((res: Response) => res.json())
            .catch((error: any) => Observable.throw(error))
    }
}
```

Figure 12. In order to use the Angular Http module, it must be imported to a component or service using it. Here, the get*People* method is set up as an observable and the stream of data is mapped to json format. Subscribing to a data stream allows asynchronous data to flow into the *people* array.

Http.get method returns an Observable of HTTP Responses. These responses can then be used to determine how the application should handle the completed request. After a successful request, the received data can be displayed in the view using data binding methods provided by Angular. Data binding methods are similar to the AngularJS framework. (23; 32; 33).

As single page applications usually consist of multiple views, routing is needed to navigate between them. Managing state transitions is one of the hardest parts of building web applications where the state is reflected in the browser's URL. The Angular router resolves this problem with its own Component Router module. It is separated from the core but can be imported to the application from an external module. The Angular router can interpret a browser url as an instruction to navigate to a client-generated view. The router logs the application view states to the browser's history journal using the *history.pushState* method which enables the native back and forwards buttons to work with the application.

When using the Angular Component router, a base URL must be set to tell the router how to compose navigation URLs. In Angular, routes are configured by mapping paths to the component that will handle them. There are three main components that are used to configure routing in Angular: *Routes*, that describe the routes an application supports, *RouterOutlet* which defines where each route's content is rendered, and *RouterLink*, a directive used to link to routes. Figure 13 shows a basic routing setup using Angular Component router.



```
import { RouterModule, Routes } from '@angular/router';

const appRoutes: Routes = [
    { path: '',      component: 'FrontPageComponent' },
    { path: 'about', component: 'AboutComponent' },
    { path: '**',    component: 'PageNotFoundComponent' }
];

@NgModule({
    imports:        [ BrowserModule, RouterModule.forRoot(appRoutes) ],
    declarations:   [ AppComponent, FrontPageComponent, AboutComponent, PageNotFoundComponent ],
    bootstrap:      [ AppComponent ]
})
export class AppModule { }

platformBrowserDynamic().bootstrapModule(AppModule);
```

Figure 13. Application routes are set up using Angular RouterModule.

The routes need at least a path and a component as arguments. Routing is then initialized in the imports array of the main module by using *RouterModule.forRoot* method and passing the route objects to it. If no route matches the url in the browser, the router can redirect to a pre-defined component using double asterisk symbols. (23; 34; 35)

Because single page applications use the browser to render content and views using JavaScript, search engine optimization is traditionally difficult. The Angular team has come up with a tool to render Angular applications on the server.

Angular Universal is a library, or a server side plugin, that can be used to to render an Angular application on the server. It generates static HTML files at build time that can be deployed to a web host. It also enables to run Angular application code on the server with each request to generate a server side view on the fly. The static HTML pages are displayed to a user while Angular loads and after loading, the dynamic version of the application is switched to the client view. The pre-rendered pages are what search engine crawlers see and, thus, the application can be indexed without problems. (36).

Angular is a fantastic framework for building modular, highly scalable, high performance applications for web and native platforms. It adds many improvements to the innovations introduced in AngularJS and takes the concepts even further. While Angular is still relatively new framework, major drawbacks or cons to it have not been documented yet. (23).

2.3 AngularJS and Angular in comparison

When comparing Angular to AngularJS, the most noticeable difference for a developer is the new syntax and the development process. Switching from AngularJS to developing with Angular requires a completely new mindset. When learning AngularJS, a developer needs to learn only the framework and its features, assuming they have a basic knowledge of HTML, CSS and JavaScript. Although the learning curve for learning AngularJS is quite steep, starting the application build process needs very little preparation and no pre-processing scripts or other perquisites to get started is needed other than including the framework source file to the application. (37; 38).

With Angular, a developer needs to learn not only the framework, but also ES6 syntax and TypeScript features, not forgetting the reactive extensions and observables used with API calls for example. As Angular is written in ES6 and TypeScript, pre-processing is needed to generate JavaScript files that run in the browsers. If a developer has no earlier experience with NodeJS, NPM, Webpack, Yarn, or other JavaScript package managers, bundlers and task runners, learning those is also required in order to build applications with Angular. This may add a large learning curve to the framework. However, many developers say learning Angular is still easier than learning AngularJS. The community support is still quite poor in Angular as it is such a new framework. AngularJS has a much larger community and many of the solutions to common problems can be found easily by searching Stack Overflow or other development related websites. (37; 39).

AngularJS applications were built around templates and controllers attached to them. The controllers had a *$scope* where all the functions and variables needed in a specific part of the application were placed to access them in the view. In Angular, the controllers and *$scope* are gone and replaced with components. This allows writing much cleaner and reusable code that is also easier to maintain and scale when needed.

Speed and performance are improved in Angular. The component router delivers automatic code-splitting that loads only the code required to render the view. Many of the modules have been removed from the Angular core package, and can be imported to the application when and if needed. The digest cycle from AngularJS is replaced with a “Change detection” system which yields up to five times better performance compared to AngularJS. Mobile performance is also greatly improved as Angular was designed for the mobile devices from the ground up.

Search engine optimization is easier with Angular. Angular Universal plugin enables Angular applications to be pre-rendered or rendered in real time on the server. It helps search engine crawlers to index a page without having to execute JavaScript functions. (23; 36; 38; 39).

3 The project: Angular migration – Case file manager

Valve is a Finnish creative agency, based in Helsinki, that combines creative marketing, technology, and communications. They employ close to 140 software development, branding, and communications professionals including a 40-person unit specializing in video production and motion graphics. Valve has worked with hundreds of domestic and multinational brands since 2000. Their work has gained international recognition including honors from The Webby Awards and Cannes Lions.

Wake is a content management system developed by Valve. It is a centralized content editing platform which enables publishing rich and dynamic content to multiple media platforms including websites, mobile devices, public screens, and television. Wake Core uses PHP programming language, Slim Framework, Twig template system, Redis cache and MongoDB database. The front-end of Wake is built with HTML, CSS and AngularJS.

The idea for the migration project came from the product development team as the file manager of Wake has for long been planned to get a “facelift” and new functionalities. The admin side of Wake will get migrated from using AngularJS to Angular, and as the file manager tool is separated from the core application, it was a perfect place to start the process. Despite being a separated application, the file manager used to load a new instance of the Wake platform every time the file manager was opened in order to get the required data for example the current *context* and user information of the application. The product development team wanted to completely separate the file manager from the admin application and provide the data using alternative methods. While the main focus in the project was the migration process, an update to the user interface and experience was planned to be done alongside.

3.1 Initial state and planning phase

The file manager was built with HTML, CSS, and AngularJS. It is a separated application that communicates with the Wake admin via custom APIs. It opens in a new browser window when opened from the admin view. The file manager had some basic CRUD functionalities but many of the actions needed to perform tasks were difficult to use and required a lot of clicks and knowledge of the application. For example, renaming a file or moving it to another folder was done by selecting the item, clicking an edit link and from the edit view, a cog icon had to be clicked to display an edit modal. The file could only be moved to another folder by editing its path in the editor.

The codebase of the file manager was relatively cluttered and difficult to maintain as all of the functions were placed inside one AngularJS controller and a service containing the API request functions. The UI consisted of a single template and an upload modal.

The project started with a planning phase. First, the desired improvements to the interface were discussed with a UX designer and some of the product development team members. In addition, customer feedback was used to map out some of the main problems with the file manager. It came out clear that many of the main actions needed to organize files and edit file metadata had to be made easier for the users. A perfect scenario would be to have the application work as close to a native file browser, on the computer, as possible. The UX designer then created a simplified layout guide with some of the components and user interaction maps.

After the UI and UX improvements were mapped out, the codebase was then inspected for the main functions needed for the application. It turned out that some of the external libraries, written in AngularJS, and used to handle some of the functions didn't have an Angular version so finding alternative methods to include those or creating them oneself was required.

The migration process was then planned. There were two options to approach the migration: using Angular Upgrade module to allow writing Angular code alongside with AngularJS code, or a complete rewrite of the application. While it is possible to use the Angular Upgrade module to bootstrap and manage an application that supports both Angular and AngularJS code, a complete rewrite of the file manager was selected as the migration method. It seemed to be a better option because the code was desired to be split into multiple components and no AngularJS code was needed in the application.

To enable easier maintenance and implementation of new functionality, the application was planned to be composed of multiple smaller components instead of one single component. How the various components would communicate with each other, and how the data would be handled across the application was initially planned to be done by using multiple Angular services, but some of the functionality had to be planned and researched alongside the development process as the Angular framework was completely new to me at the time.

After the codebase inspection and examining the UI layout provided by the UX designer, a project plan and a ballpark time estimate was created. Initially the migration and UX process seemed to be a relatively straightforward task. However, I needed to learn the Angular framework while developing the application and, thus, the time estimate to execute the project was set higher than in “normal” projects.

3.2 Development process

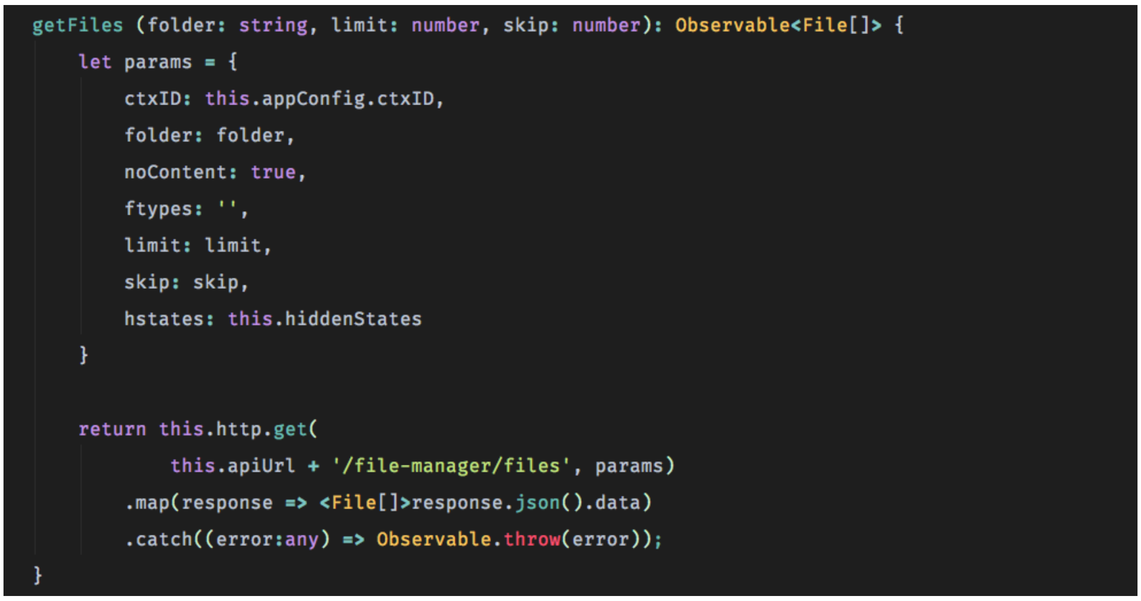
Wake platform is developed by deploying the code files to a server where it is then processed and transpiled with NPM task runners and bundlers. Because the new file manager was to be developed with Angular, a new task runner and bundler script had to be written. Webpack was chosen for this process as it offers a great module bundler and *loaders* that transform TypeScript and ES6 code into JavaScript.

Three different Webpack script files were created. One of them has all the module loaders needed for the script transpiling and HTML and SCSS processing. Debugging errors in the development process is easier if source maps of the transpiled JavaScript files are present in the error messages, and the code can be inspected in the browser when no minification on the script files is done. A development version of Webpack script was written to include all source maps of the TypeScript and SCSS files. For the production version, the files were minified and source maps were removed to improve the speed of the application.

Before creating any of the functions for the application, the UI of the file manager was split into small Angular components and the HTML template was built using those components and mockup data. For a modern look, the new UI utilizes material design pattern. Angular has a material design library with a selection of components like buttons and dropdown elements, but many of the features were at the time still in beta and, thus, many components had to be custom made.

Every Wake application has a unique context id that is used in the database queries. In order to get the context id of the current application to the file manager an existing Post Message service was modified and written in Angular. The service creates a *handshake* with the admin application to fetch data from it. A *handshake* method utilizes web socket protocol and allows multiple browser windows or tabs to communicate with each other. When the handshake is successful, user data and the admin application context id is placed to an application configuration object in file manager. This data is used to allow permissions and to create correct API requests to fetch data from the server.

To create multiple API requests using Angular Http methods, a service was created. The service allows these methods to be called anywhere in the application and handles all the CRUD operations needed in the file manager. The file manager used the AngularJS *$resource* factory to define a resource object containing all API request methods. Angular doesn't have the *$resource* factory so the methods needed to be created separately. Wake has multiple API endpoints that can be used to perform various requests. Figure 14 shows the method to fetch the file data.

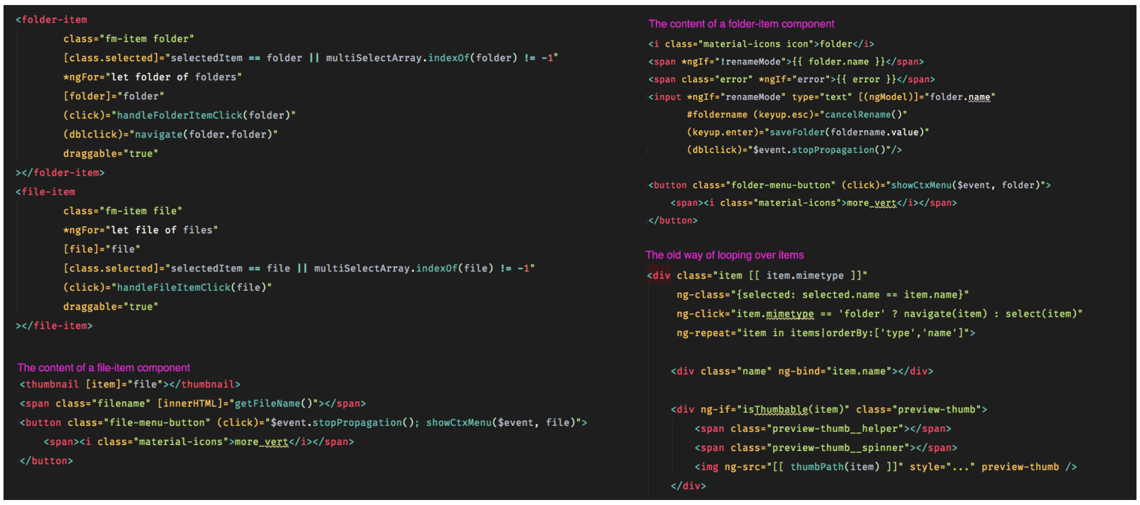


```
getFiles (folder: string, limit: number, skip: number): Observable<File[]> {
    let params = {
        ctxID: this.appConfig.ctxID,
        folder: folder,
        noContent: true,
        ftypes: '',
        limit: limit,
        skip: skip,
        hstates: this.hiddenStates
    }

    return this.http.get(
            this.apiUrl + '/file-manager/files', params)
        .map(response => <File[]>response.json().data)
        .catch((error:any) => Observable.throw(error));
}
```

Figure 14. The *getFiles* method is used to fetch file data. Several parameters are passed to the API call, of which the most important are the context id and the name of the folder where the files are located.

All methods in File Service are written similarly to the *getFiles* method in Figure 14, have a type of an Observable, which allows subscribing to the response in the components using the data and performing tasks based on the type of the response. Responses are mapped to JSON format for an easy data manipulation and looping over it in the templates. Comparing the Angular way of fetching file data to the AngularJS, Angular service is much cleaner and easier to understand than the AngularJS resource factory (see appendix 1).

Looping through the arrays of fetched data in Angular is much more similar to AngularJS. Angular has a built-in structural directive, *ngFor*, to loop a part of the template and bind data to it. In AngularJS version of the file manager, looping folder and file objects to the template was done by creating a single element and adding conditionals to it to detect the type of the object. More control over the different types of items was desired so folders and files were turned into separate components and data was passed to them with the component input method. Figure 15 shows the new way of separation of files and folders compared to the old version.



```
<folder-item
        class="fm-item folder"
        [class.selected]="selectedItem == folder || multiSelectArray.indexOf(folder) != -1"
        *ngFor="let folder of folders"
        [folder]="folder"
        (click)="handleFolderItemClick(folder)"
        (dblclick)="navigate(folder.folder)"
        draggable="true"
></folder-item>
<file-item
        class="fm-item file"
        *ngFor="let file of files"
        [file]="file"
        [class.selected]="selectedItem == file || multiSelectArray.indexOf(file) != -1"
        (click)="handleFileItemClick(file)"
        draggable="true"
></file-item>
```

The content of a file-item component

```
<thumbnail [item]="file"></thumbnail>
<span class="filename" [innerHTML]="getFileName()"></span>
<button class="file-menu-button" (click)="$event.stopPropagation(); showCtxMenu($event, file)">
    <span><i class="material-icons">more_vert</i></span>
</button>
```

The content of a folder-item component

```
<i class="material-icons icon">folder</i>
<span *ngIf="!renameMode">{{ folder.name }}</span>
<span class="error" *ngIf="error">{{ error }}</span>
<input *ngIf="renameMode" type="text" [(ngModel)]="folder.name"
       #foldername (keyup.esc)="cancelRename()"
       (keyup.enter)="saveFolder(foldername.value)"
       (dblclick)="$event.stopPropagation()"/>

<button class="folder-menu-button" (click)="showCtxMenu($event, folder)">
    <span><i class="material-icons">more_vert</i></span>
</button>
```

The old way of looping over items

```
<div class="item [[ item.mimetype ]]"
     ng-class="{selected: selected.name == item.name}"
     ng-click="item.mimetype == 'folder' ? navigate(item) : select(item)"
     ng-repeat="item in items|orderBy:['type','name']">

    <div class="name" ng-bind="item.name"></div>

    <div ng-if="isThumbable(item)" class="preview-thumb">
        <span class="preview-thumb__helper"></span>
        <span class="preview-thumb__spinner"></span>
        <img ng-src="[[ thumbPath(item) ]]" style="..." preview-thumb />
    </div>
```

Figure 15. Folder and file elements were separated from each other to gain more control over them. Data is passed to each item using the component input method, *[property]="value"*.

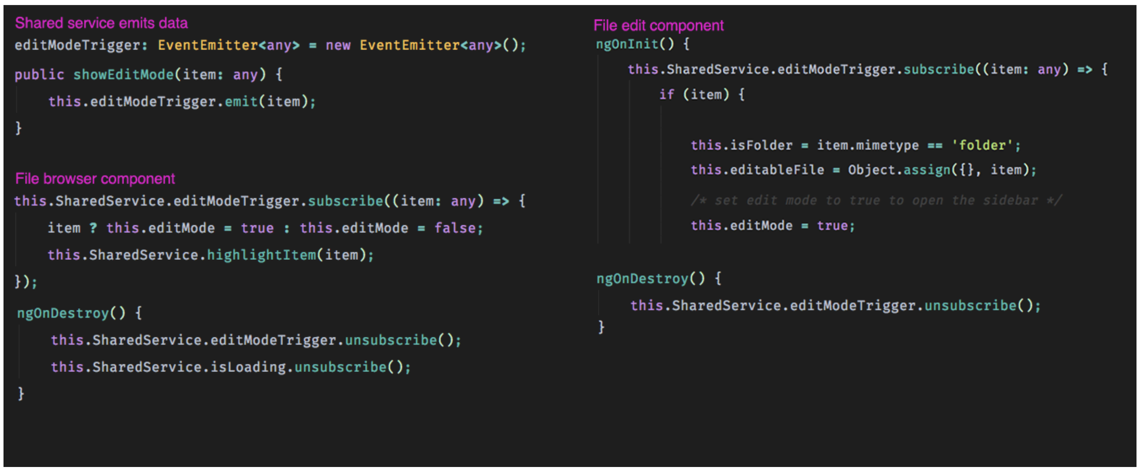
While the new method grows the template in size, it allows a greater distinction between different item types. This is useful when different kinds of items have different actions and structure.

To upload files to the server, the file manager used the AngularJS version of FlowJS library, which allows uploading multiple files simultaneously and larger files in smaller chunks to introduce fault-tolerance. If the upload of a chunk fails, the upload process is retired until the procedure completes. It allows to automatically resume uploading after a network connection is lost either locally on to the server. FlowJS doesn't come with a version that supports Angular, and therefore the Vanilla JavaScript version had to be implemented. Angular supports writing application code in JavaScript and importing the FlowJS was easy by adding it to the imports array of the main application module.

In the older file manager uploading files required to click an upload button that opened a modal where another button had to be clicked in order to select files from a computer. After selecting files, another button was clicked to start the upload. This process was simplified by using FlowJS's *assignBrowse* method assigned to a single button that, when clicked, opens the computer's native file browser and when files are selected, the upload starts automatically. Files are uploaded to the currently selected folder in the view and a small popup is shown to indicate the upload progress.

Because the older file manager was built using a single AngularJS controller to handle the state of the application, detecting data changes and executing functions were easy by inspecting the scope values. Re-writing the scope functions in the AngularJS controllers was a straightforward task. Just by removing the *$scope* keyword before function declarations was in most cases all that was needed. But as the new file manager consists of multiple components separated from each other handling data across the application needed some research and planning. The data and custom events would have to be sent to other components in the application. The solution was to build a service with observable methods where data could be passed from the component and the service would emit the data to the other components.

The data receiving components need to subscribe to the observable stream in order to have access to the data objects. Figure 16 shows the data emitting and subscription methods used to emit a file editing mode status.



Shared service emits data
```
editModeTrigger: EventEmitter<any> = new EventEmitter<any>();

public showEditMode(item: any) {
    this.editModeTrigger.emit(item);
}
```

File browser component
```
this.SharedService.editModeTrigger.subscribe((item: any) => {
    item ? this.editMode = true : this.editMode = false;
    this.SharedService.highlightItem(item);
});

ngOnDestroy() {
    this.SharedService.editModeTrigger.unsubscribe();
    this.SharedService.isLoading.unsubscribe();
}
```

File edit component
```
ngOnInit() {
    this.SharedService.editModeTrigger.subscribe((item: any) => {
        if (item) {

            this.isFolder = item.mimetype == 'folder';
            this.editableFile = Object.assign({}, item);
            /* set edit mode to true to open the sidebar */
            this.editMode = true;

ngOnDestroy() {
    this.SharedService.editModeTrigger.unsubscribe();
}
```

Figure 16. A shared service was created to share data with multiple components. In this code example, two components subscribe to *editModeTrigger* event emitter that is used to show a file edit sidebar in the application. *NgOnDestroy* life-cycle hook is used to unsubscribe from the data stream to avoid memory leakage.

Because multiple data streams could cause some memory leakage, the components need to unsubscribe from the observation when the component life-cycle ends. Angular has a built-in life-cycle hook, *OnDestroy,* that can be used to detect the end of a component life-cycle.

Using a shared service turned out to be an efficient and easy way to trigger events and broadcasting data in the application, while simultaneously allowing flexibility to the component placement in the templates.

The project was finished successfully, new features were added, and the migration process wasn’t as difficult as at first thought. The biggest challenge in the migration was to get into the different mindset needed to build Angular applications. While some of the concepts familiar from AngularJS are still present in Angular, the different syntax and component based architecture required some time to get used to. The application improved in speed and performance, partly due to the optimization and improvements in the Angular framework.

3.3 Finished product overview

In addition to the migration process, new features were added to improve the user experience and usability of the file manager. The most noticeable difference is the new visual layout. Material design gives the elements a modern look and all buttons and other clickable elements give a visual feedback on interaction. Figure 17 shows the landing view of the new file manager. The original version of the file manager landing view can be seen in appendix 3.

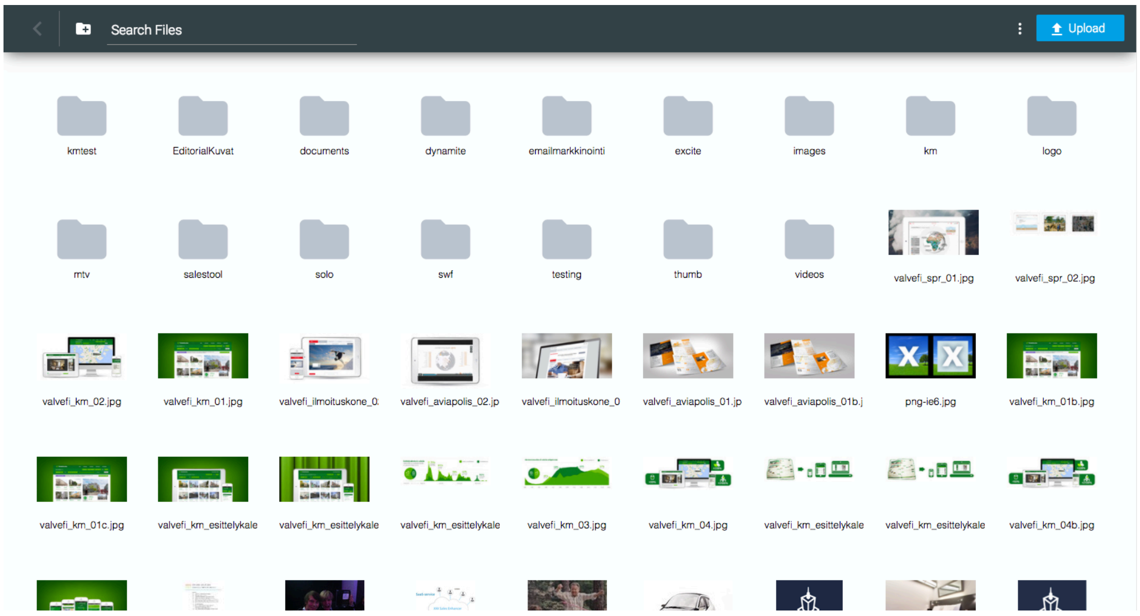


Figure 17. File manager landing view.

Compared to the older version, the upload process is much simpler and faster in the new file manager (see appendix 2). Files can be uploaded by clicking a button in the top right corner, or as a new feature, by dragging and dropping files to the view. Uploading starts as soon as the files are selected or dropped.

Moving files to another folder was a difficult and time consuming task, if multiple files should be moved. In the old version, moving a folder or multiple files at once was not possible, and to move a single file would have to be done by editing its path under the settings (appendix 5). A new feature was built to simplify the process and mimic the computer's native file browser behavior. A user can now drag a selection over files and move them to a folder by dragging the selected files and dropping them over a folder, or by opening a context menu by right-clicking on top of a file and selecting a target folder from the tree-view, shown in appendix 6. Figure 18 shows the selection of files by dragging over them.

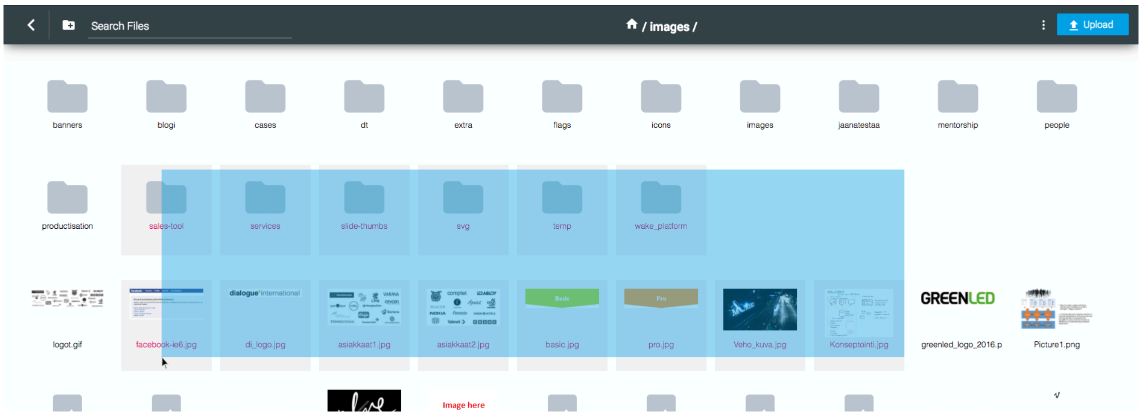


Figure 18. Files and folders can now be selected by drawing a selection area with a mouse and then moved by dragging and dropping the selected items on top of a target folder. Selected items are highlighted with a background and text color change.

Navigating to a folder happens by double-clicking a folder item. Folders can now be renamed like files, and is done by right-clicking on top of a folder item, or clicking its settings icon and selecting “Rename” option from the context menu that opens where the mouse event happened (Figure 19). Some validation is done when renaming items, but more thorough validation will be done later.

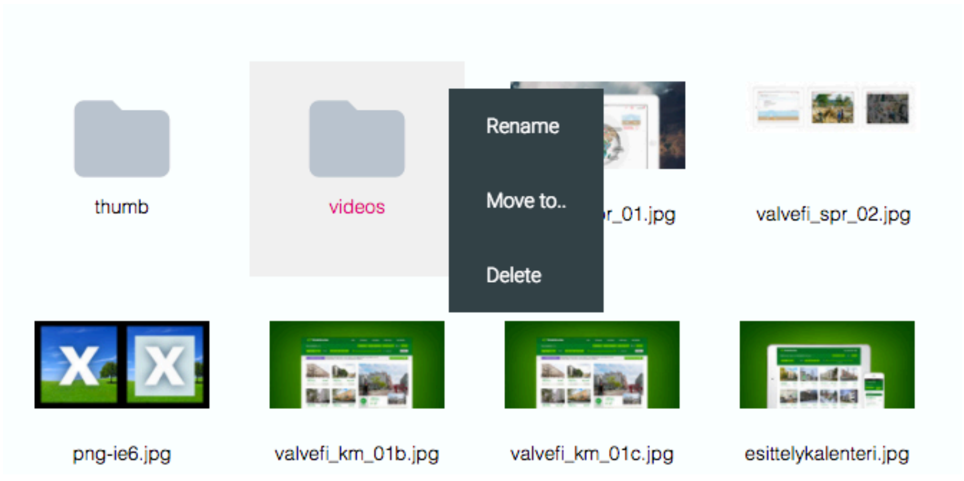


Figure 19. A context menu with some of the main functionalities can be opened by right-clicking on top of an item.

Many of the functionalities in the file manager were simplified and optimized for an easier and more user-friendly experience. All the functions in the components were built with error handling in mind, and the user is notified if any errors occur during the operation. The application will be available for all Wake users after complete testing and possible bug fixes.

4 Summary

AngularJS framework introduced a robust set of tools to build a variety of dynamic, client centric applications while using HTML as a template language and further extending the HTML syntax to express web application's components clearly and concisely. It utilizes the MVC programming architecture design. Two-way data binding eliminates the need for a developer to manually write synchronization code to keep track of data model and user interface thus keeping the codebase smaller, cleaner and easier to maintain.

As the web and web development has advanced tremendously since the release of AngularJS, many of the concepts within AngularJS have become outdated. Updating and modernizing the existing AngularJS framework would have been impossible without breaking much of the original implementation. New version of AngularJS was written and is called *Angular*.

In Angular, controllers and scopes are gone and applications are built with components. They are a combination of an HTML template and a component class that controls a portion of the view. Components are the main way to build and specify elements and logic on the Angular application.

If an application is desired to be migrated from using AngularJS to Angular, there are two options at the moment. The migration can be done using the Angular Upgrade module that allows running both AngularJS and Angular code simultaneously and the application can be migrated small piece at a time. If more complete migration is wanted, and more modifications are done, the application should be re-written from the start.

The re-write, or migration process is not extremely difficult after learning the basic features of Angular and getting into the mindset of creating an application with components instead of templates and controllers. Hands-on experience shows that Angular can be learned faster than AngularJS. Many of the functions to perform tasks in the project application were simpler to create in Angular. The speed and performance improvement from Angular was noticeable, especially with slower internet connections.

Resources

1 Google. What is AngularJS? Web document. <https://docs.angularjs.org/guide/introduction>. Read 10.3.2017.

2 Releases. 2010. GitHub. Web document. <https://github.com/angular/angular.js/releases?after=v0.9.7>. Updated 6.3.2017. Read 10.3.2017.

3 Lerner, Ari. 2013. NG-Book: The Complete Book on AngularJS. Fullstack.io.

4 The Seed for AngularJS apps. GitHub. Web document. <https://github.com/angular/angular-seed>. Read 11.3.2017.

5 What is a CDN. Imperva Incapsula. Web document. <https://www.incapsula.com/cdn-guide/what-is-cdn-how-it-works.html>. Read 15.3.2017.

6 Google. Conceptual Overview. Web document. <https://docs.angularjs.org/guide/concepts>. Read 16.3.2017.

7 Google. Bootstrap. Web document. <https://docs.angularjs.org/guide/bootstrap>. Read 16.3.2017.

8 Milner, Matt. 2015. AngularJS: MVC Implementation. Pluralsight. Web document. <https://www.pluralsight.com/blog/software-development/tutorial-angularjs-mvc-implementation>. Read 17.3.2017.

9 Google. MVC Architecture. Web document. <https://developer.chrome.com/apps/app_frameworks>. Read 15.3.2017.

10 SAP. Model-View-Controller. 2015. Web document. <https://help.sap.com/erp_hcm_ias2_2015_01/helpdata/en/91/f233476f4d1014b6dd926db0e91070/content.htm>. Read 14.3.2017.

11 WebSystique. AngularJS Custom directives link-function guide. 20.12.2015. Web document. <http://websystique.com/angularjs/angularjs-custom-directives-link-function-guide/>. Updated 21.12.2015. Read 15.3.2017.

12 Google. Creating cutom directives. Web document. <https://docs.angularjs.org/guide/directive>. Read 18.3.2017.

13 Google. Understanding controllers. Web document. <https://docs.angularjs.org/guide/controller>. Read 18.3.2017.

14 Google. $http. Web document. <https://docs.angularjs.org/api/ng/service/$http>. Read 20.3.2017.

15 W3Schools. AngularJS Services. Web document. <https://www.w3schools.com/angular/angular_services.asp>. Read 19.3.2017.

16 Code School. Single-page Applications. Web document. <https://www.codeschool.com/beginners-guide-to-web-development/single-page-applications>. Read 22.3.2017.

17 Google. Component Router. Web document. <https://docs.angularjs.org/guide/component-router>. Read 24.3.2017.

18 McKeachie, Graig. 2014. UI-Router: Why many developers don't use AngularJS's built-in router. Web document. <http://www.funnyant.com/angularjs-ui-router/>. Read 25.3.2017.

19 Baxter, Richard. The Basics of JavaScript framework SEO in AngularJS. Built-visible. Web document. <https://builtvisible.com/javascript-framework-seo/>. Read 25.3.2017.

20 Hooper, Todd. 2014. AngularJS SEO. Prerender.io. Web document. <https://builtvisible.com/javascript-framework-seo/>. Read 25.3.2017.

21 Rajput, Mehul. 2016. The pros and cons of choosing AngularJS. Web document. <https://jaxenter.com/the-pros-and-cons-of-choosing-angularjs-124850.html>. Read 26.3.2017.

22 Chemel, Renee. 2015. Pros and cons of AngularJS. Web document. <http://blog.backand.com/pros-and-cons-of-angularjs/>. Read 28.3.2017.

23 Lerner, Coury, Murray, Taborda. 2016. NG-Book 2: The Complete Book on Angular. Fullstack.io.

24 Google. Features & benefits. Angular. Web document. <https://angular.io/features.html>. Read 28.3.2017.

25 Microsoft. TypeScript. Github. Web document. <https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md>. Read 24.3.2017.

26 TypeScript. Web document. <http://www.typescriptlang.org/>. Read 25.3.2017.

27 Savkin, Victor. 2016. Angular: Why TypeScript. Web document. <https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8>. Read 27.3.2017.

28 Google. NGModules. Angular. Web document. <https://angular.io/docs/ts/latest/guide/ngmodule.html>. Read 25.3.2017.

29 Google. Quickstart. Angular. Web document. <https://angular.io/docs/ts/latest/quickstart.html>. Read 24.3.2017.

30 Google. Architecture overview. Angular. Web document. <https://angular.io/docs/ts/latest/guide/architecture.html>. Read 24.3.2017.

31 Precht, Pascal. 2016. Two-way data binding in Angular. Web document. <https://blog.thoughtram.io/angular/2016/10/13/two-way-data-binding-in-angular-2.html>. Read 26.3.2017.

32 Google. Services. Angular. Web document. <https://angular.io/docs/ts/latest/tutorial/toh-pt4.html>. Read 26.3.2017.

33 Reactive Extensions. Observable Object. Web document. <https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/observable.md>. Read 28.3.2017.

34 Savkin, Victor. 2016. Angular Router. Web document. <https://vsavkin.com/angular-2-router-d9e30599f9ea>. Read 26.3.2017.

35 Basic Routing in Angular 2. Ng-book blog. Web document. <http://blog.ng-book.com/basic-routing-in-angular-2/>. Read 28.3.2017.

36 Google. Overview. Angular Universal. Web document. <https://universal.angular.io/overview/>. Read 29.3.2017.

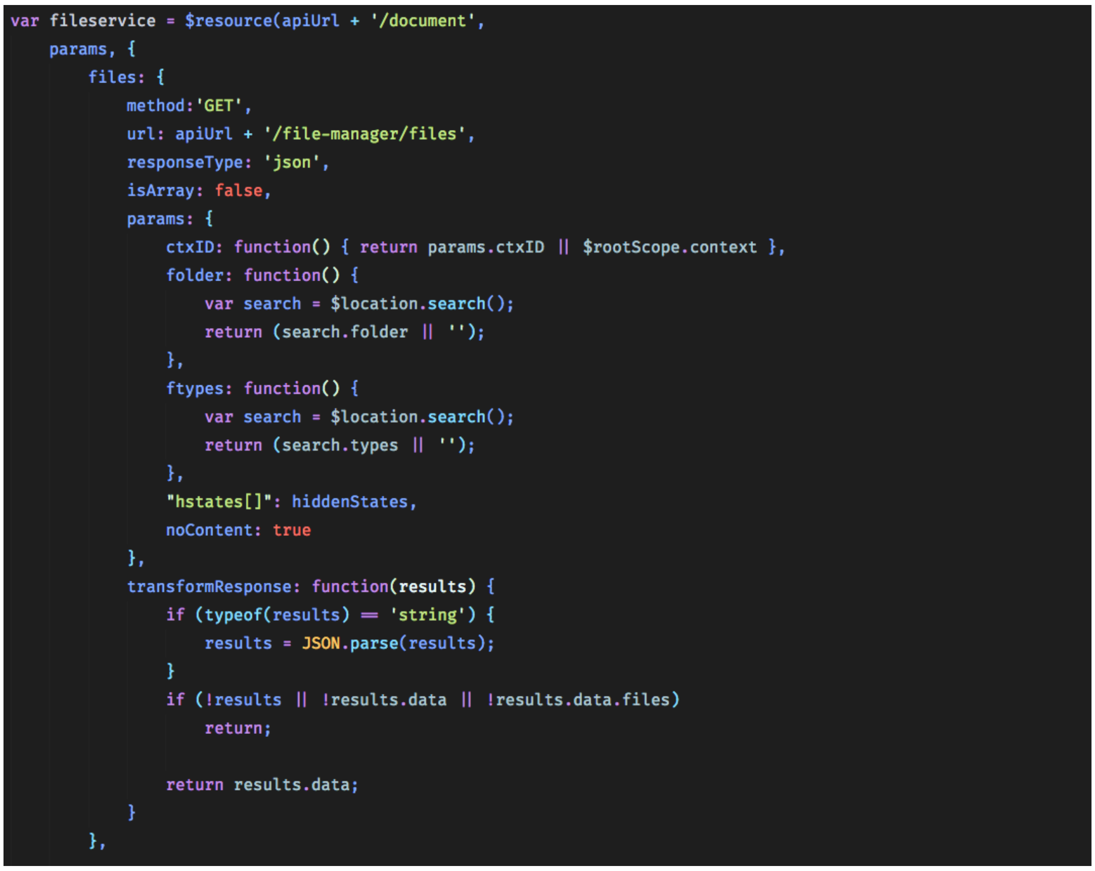
37 Muller, Elco. 2015. Angular 2 vs Angular1: Key differences. Web document. <https://dzone.com/articles/typed-front-end-with-angular-2>. Read 30.3.2017.

38 Devblast. 2016. The differences between Angular 1.x and Angular 2. Web document. <https://devblast.com/b/differences-angular-1-x-angular-2>. Read 29.3.2017.

39 Bandi, Jonas. 2016. Angular 2 vs Angular 1: A teacher's perspective. Medium. Web document. <https://medium.jonasbandi.net/angular-2-vs-angularjs-a-teachers-perspective-d7e10ba29ede>. Read 30.3.2017.

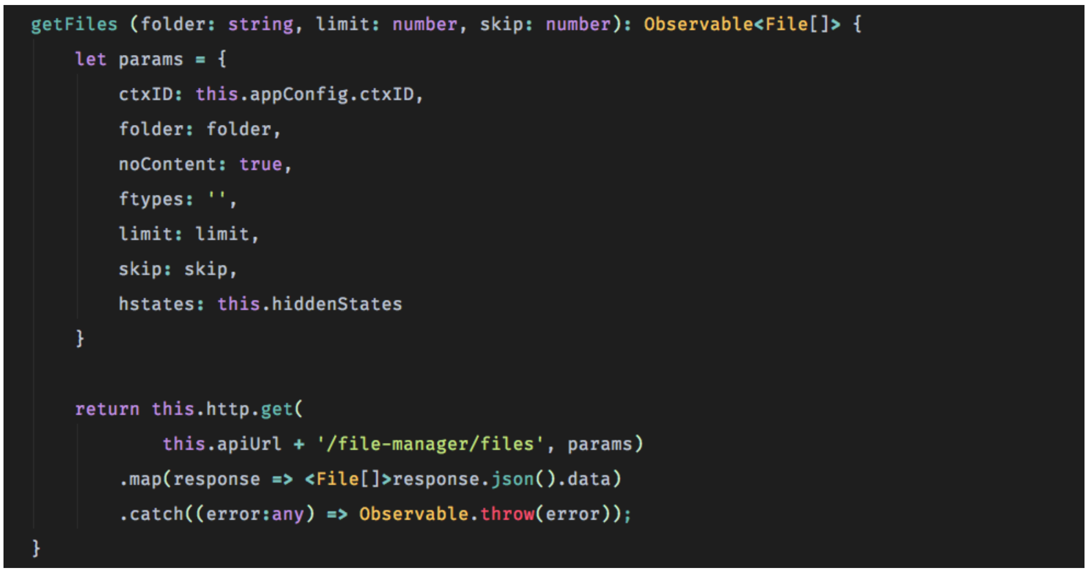
Getting files from the API: AngularJS vs Angular

AngularJS



```
var fileservice = $resource(apiUrl + '/document',
    params, {
        files: {
            method:'GET',
            url: apiUrl + '/file-manager/files',
            responseType: 'json',
            isArray: false,
            params: {
                ctxID: function() { return params.ctxID || $rootScope.context },
                folder: function() {
                    var search = $location.search();
                    return (search.folder || '');
                },
                ftypes: function() {
                    var search = $location.search();
                    return (search.types || '');
                },
                "hstates[]": hiddenStates,
                noContent: true
            },
            transformResponse: function(results) {
                if (typeof(results) == 'string') {
                    results = JSON.parse(results);
                }
                if (!results || !results.data || !results.data.files)
                    return;

                return results.data;
            }
        },
```

Angular

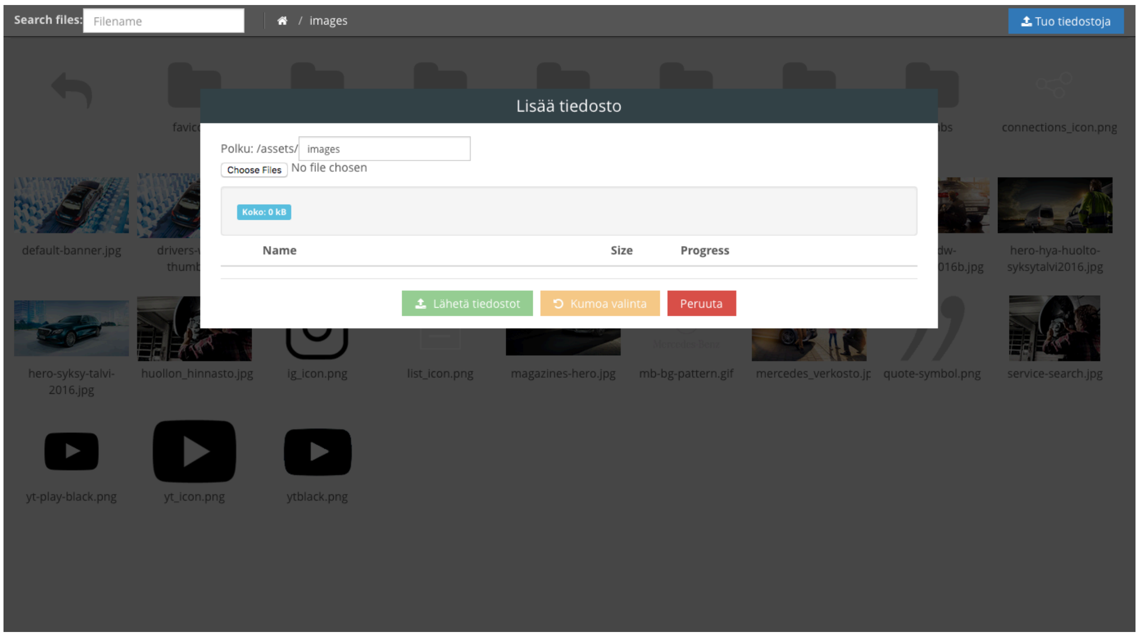


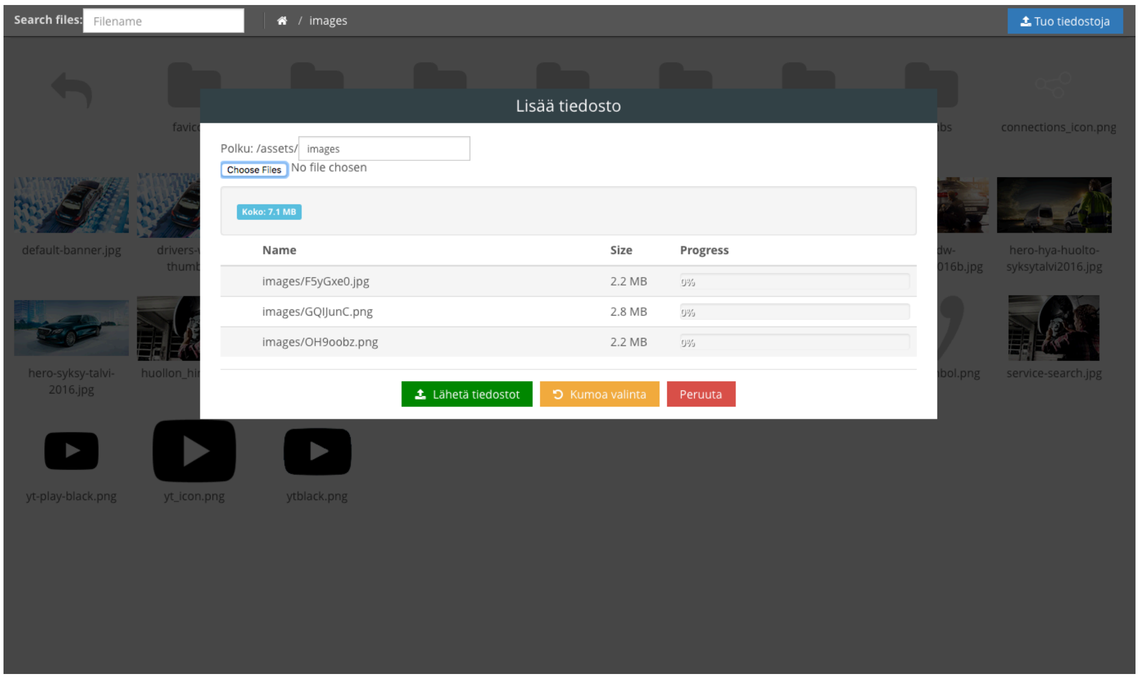
```
getFiles (folder: string, limit: number, skip: number): Observable<File[]> {
    let params = {
        ctxID: this.appConfig.ctxID,
        folder: folder,
        noContent: true,
        ftypes: '',
        limit: limit,
        skip: skip,
        hstates: this.hiddenStates
    }

    return this.http.get(
            this.apiUrl + '/file-manager/files', params)
        .map(response => <File[]>response.json().data)
        .catch((error:any) => Observable.throw(error));
}
```

Uploading files

Old file manager

Adding files required clicking the blue button at top right, then a button “Choose files”, and lastly, after selecting files, clicking an upload button.

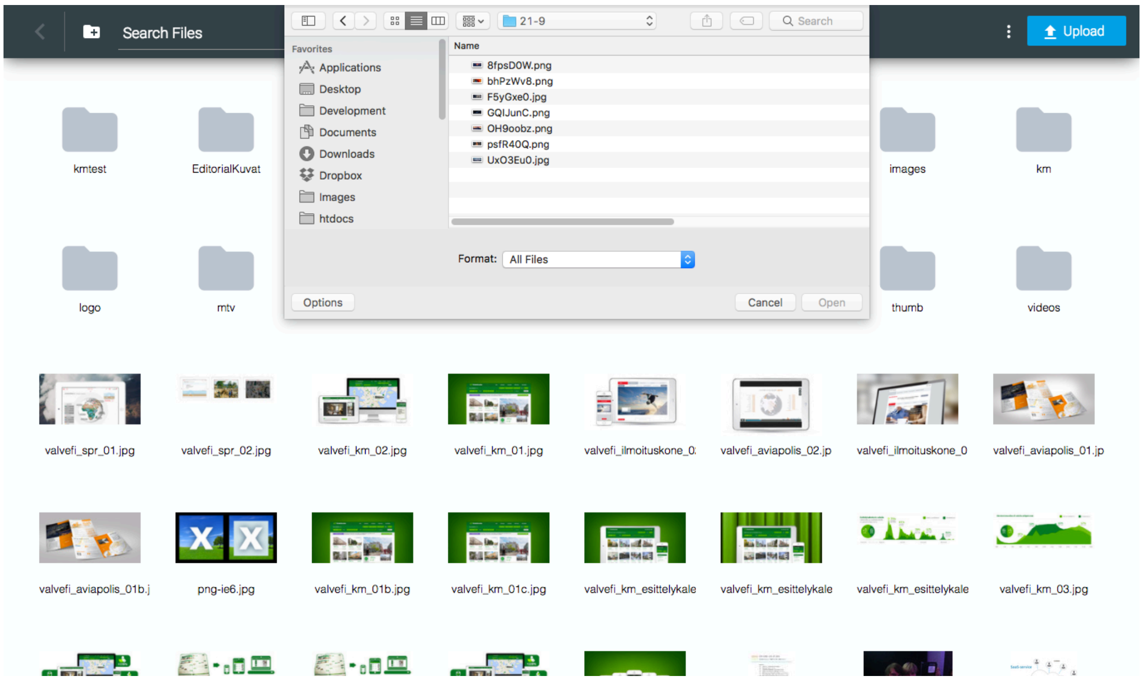


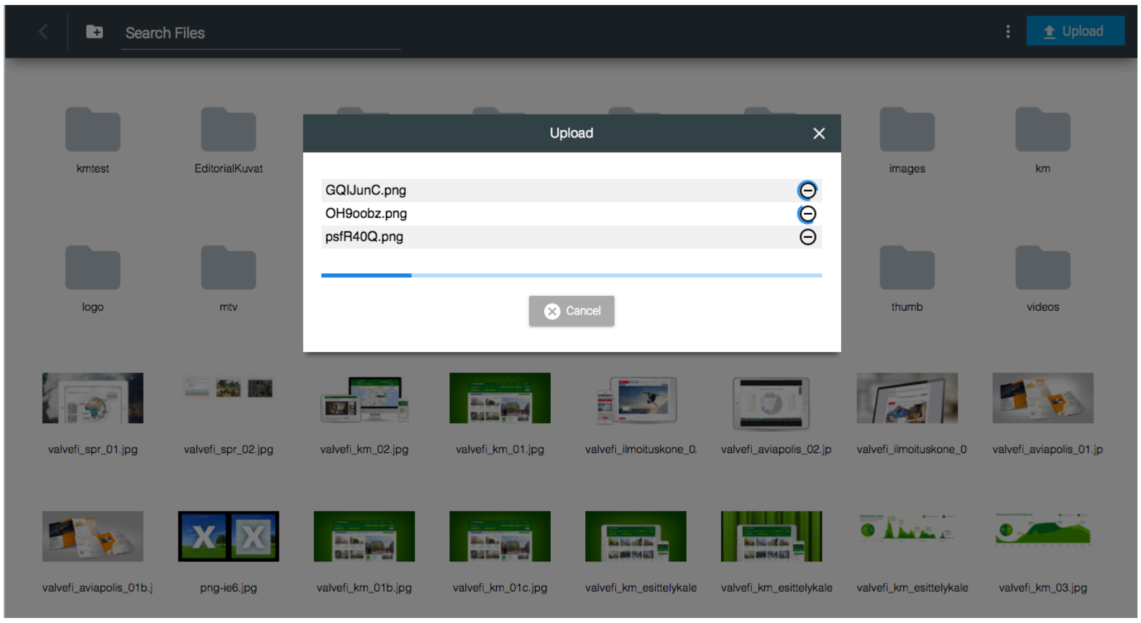


Name	Size	Progress
images/F5yGxe0.jpg	2.2 MB	0%
images/GQIJunC.png	2.8 MB	0%
images/OH9oobz.png	2.2 MB	0%

The new file manager

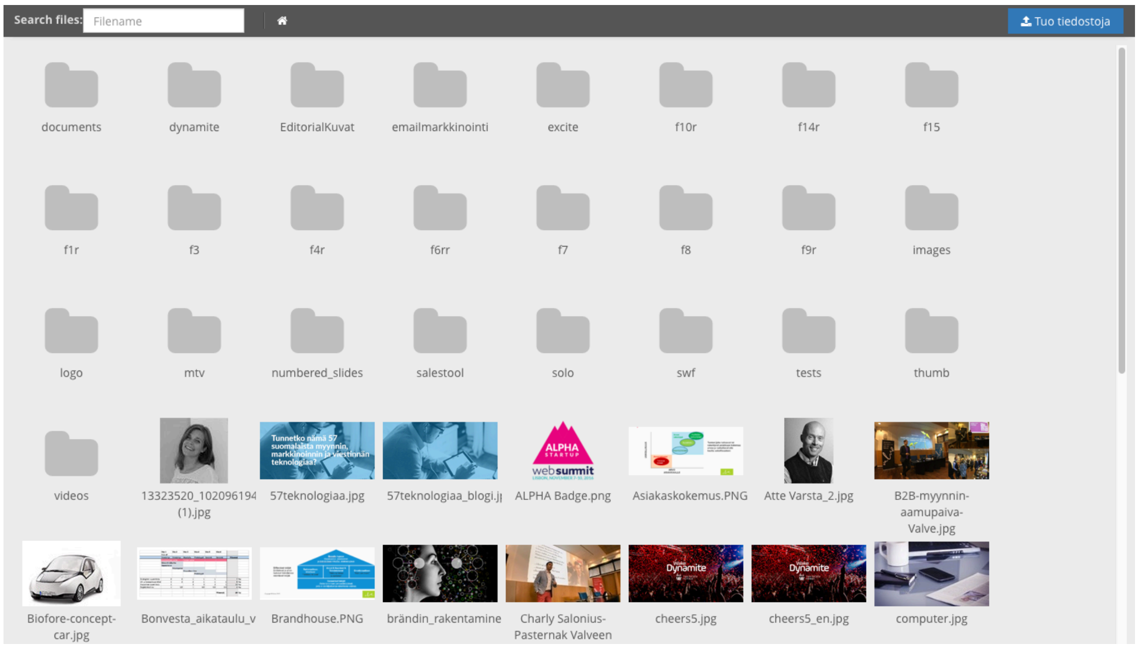
In the new file manager, uploading files is easier. Files can be dragged and dropped to the view, or by clicking a button at top right and choosing files with the native file browser. Upload process starts immediately after files have been dropped or chosen.





A popup is shown while files are being uploaded.

Original file manager landing view


Search files:
Filename
Tuo tiedostoja
documents
dynamite
EditorialKuvat
emailmarkkinointi
excite
f10r
f14r
f15
f1r
f3
f4r
f6rr
f7
f8
f9r
images
logo
mtv
numbered_slides
salestool
solo
swf
tests
thumb
videos
13323520_102096194 (1).jpg
57teknologiaa.jpg
57teknologiaa_blogi.j
ALPHA Badge.png
Asiakaskokemus.PNG
Atte Varsta_2.jpg
B2B-myynnin-aamupaiva-Valve.jpg
Biofore-concept-car.jpg
Bonvesta_aikataulu_v
Brandhouse.PNG
brändin_rakentamine
Charly Salonius-Pasternak Valveen
cheers5.jpg
cheers5_en.jpg
computer.jpg