Annonseditor med AngularJS

Jimi Pirilä

Examensarbete

Informations- och medieteknik

2016

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informations- och medieteknik
Identifikationsnummer:	5852
Författare:	Jimi Pirilä
Arbetets namn:	Annonseditor med AngularJS
Handledare (Arcada):	Jonny Karlsson
Uppdragsgivare:	Sanoma Oyj

Sammandrag:

Detta examensarbete beskriver utvecklingen av en annonseditor för Oikotie Työpaikat. Annonseditorn används för att skapa och editera arbetsannonser som publiceras på tyopaikat.oikotie.fi-webbplatsen. Editorns utveckling var en del av ett större projekt där man flyttade över funktionalitet från den gamla tyonantaja.oikotie.fi till den nyare tyopaikat.oikotie.fi. Editorn består av två delar: ett formulär för att editera annonsens uppgifter, och en förhandsvisning som visar hur annonsen ser ut när den publiceras. Arbetet beskriver de tekniker som användes, och hur de tillämpades för att lösa de krav som ställts åt editorn. Tekniker som användes var bl.a. AngularJS och Drupal. Dessutom beskrivs hur Jasmine-ramverket kan användas för att testa en AngularJS-applikation.

Nyckelord:	Oikotie Työpaikat, JavaScript, AngularJS, Jasmine, PHP, Drupal, Gulp
Sidantal:	34
Språk:	Svenska
Datum för godkännande:	

DEGREE THESIS	
Arcada	
Degree Programme:	Information and Media Technology
Identification number:	5852
Author:	Jimi Pirilä
Title:	Development of an ad-editor with AngularJS
Supervisor (Arcada):	Jonny Karlsson
Commissioned by:	Sanoma Oyj

Abstract:

This thesis describes the development of an editor for job ads for the Oikotie Jobs website. The editor is used to create and edit job ads, which are published on tyopaikat.oikotie.fi. The development of the editor was part of a bigger project which goal was to migrate some of the functionality from the previous version of tyonantaja.oikotie.fi-website to the updated tyopaikat.oikotie.fi-website. The editor consists of two parts: the form used to edit the details of the ad, and a preview that shows how the ad will look like when it has been published. The technologies used to create the editor are presented in the thesis, and also how they were used to satisfy the functional requirements of the editor. E.g. AngularJS and Drupal were used to create the editor. It also has an example on how the Jasmine testing framework can be used to unit-test AngularJS applications.

Keywords:	Oikotie Jobs, JavaScript, AngularJS, Jasmine, PHP, Drupal, Gulp
Number of pages:	34
Language:	Swedish
Date of acceptance:	

OPINNÄYTE	
Arcada	
Koulutusohjelma:	Tieto- ja mediatekniikka
Tunnistenumero:	5852
Tekijä:	Jimi Pirilä
Työn nimi:	Ilmoituseditorin kehitys AngularJS-ohjelmistokehyksellä
Työn ohjaaja (Arcada):	Jonny Karlsson
Toimeksiantaja:	Sanoma Oyj

Tiivistelmä:

Tässä opinnäytetyössä kerrotaan kuinka Oikotie Työpaikat-palveluun kehitettiin ilmoituseditori, jota käytetään työpaikkailmoituksen julkaisemiseen tyopaikat.oikotie.fi-sivustolla. Editorin kehitys oli osa projektia jossa Oikotie Työpaikkojen ilmoitusliittymä siirrettiin vanhalta tyonantaja.oikotie.fi-sivustolta uudemmalle tyopaikat.oikotie.fi-sivustolle. Editori koostuu kahdesta osasta: lomakkeesta jolla muokataan ilmoituksen tietoja, sekä esikatselusta jossa näytetään miltä julkaistun ilmoituksen kohdesivu tulee näyttämään. Opinnäytetyössä kuvaillaan mitä tekniikoita editorin kehityksessä käytettiin jotta se täyttää sille esitetyt vaatimukset. AngularJS ja Drupal ovat esimerkkejä käytetyistä tekniikoista. Opinnäytetyössä esitellään myös miten Jasmine-kirjastoa voidaan käyttää AngularJS-sovelluksen yksikkötestaamiseen.

Avainsanat:	Oikotie Työpaikat, JavaScript, AngularJS, Jasmine, PHP, Drupal, Gulp
Sivumäärä:	34
Kieli:	Ruotsi
Hyväksymispäivämäärä:	

INNEHÅLL

1 Inledning **8**

2 Krav **9**

3 Teknik **10**

3.1 Drupal 7 10
3.2 AngularJS 12
3.3 Gulp 17
3.4 Implementation i editorn 17

4 Utveckling **18**

4.1 Serversidan 19
4.2 Klientsidan 20
4.3 Kommunikation mellan komponenter 26
4.4 Förhandsvisningen 26
4.5 Gulp uppgifter 27

5 Testande **28**

6 Slutsatser **31**

Källor **33**

Figurer

Figur 1. Exempel på Form API textelement. 12
Figur 2. AngularJS exempel på controller, modell/scope och vy. 14
Figur 3. Exempel direktiv som gör dess innehåll att blinka med givet mellanrum. 16
Figur 4. Flödet av annonsens data i editorn. 19
Figur 5. Editorns formulär. 21
Figur 6. Komponenten för bildhantering och förloppningsindikator. 22
Figur 7. Editorns förhandsvisning. 23
Figur 8. Mobil förhandsvisning. 24
Figur 9. Exempel på controller as syntaxen. 25
Figur 10. Ett formelements felmeddelande. 26
Figur 11. Exempel på hur en annons bild visas i förhandsvisningen. 27
Figur 12. Användning av ”replacement”-direktivet. 27
Figur 13. Gulp-arbete för att slå samman editorns JavaScript-filer. 28
Figur 14. Exempel Jasmine test suite. 30

Termer och förkortningar

PHP = PHP: Hypertext Preprocessor, skriptspråk som används främst för att skapa webbsidor med dynamiskt innehåll

HTML = HyperText Markup Language, markeringsspråk som används för att beskriva webbsidor

CSS = Cascading Style Sheets, språk som används för att beskriva hur t.ex. en HTML-sida ska se ut

JavaScript = Skriptspråk som körs i webbläddraren. Kan även användas som server programmeringsspråk m.h.a. NodeJS.

AngularJS = JavaScript-ramverk gjort för att underlätta utveckling av applikationer som körs i webbläddraren

Drupal = innehållshanteringssystem skrivet i PHP

CMS = Content Management System (sv. innehållshanteringssystem)

CMF = Content Management Framework, ramverk gjort för att underlätta utvecklande av CMS

DOM = Document Obect Model

API = Application Programming Interface

1 INLEDNING

Tyopaikat.oikotie.fi är Finlands största kommersiella webbplats för arbetsannonser. Den består av två olika delar: tyopaikat.oikotie.fi och tyonantaja.oikotie.fi. Tyopaikat.oikotie.fi är huvudsakligen riktad till arbetssökande. På webbplatsen hittar man bl.a. arbetsannonser, artiklar och information om olika företag som arbetsgivare. Tyonantaja.oikotie.fi är avsedd för arbetsgivare, och den används huvudsakligen till att publicera arbetsannonser på tyopaikat.oikotie.fi. Tyopaikat.oikotie.fi kommer att hänvisas som *arbetsportalen*, och tyonantaja.oikotie.fi kommer att hänvisas som *arbetsgivarportalen* senare i arbetet.

År 2015 inleddes ett projekt vars syfte var att överföra en stor del av arbetsgivarportalens funktionalitet till den nya arbetsportalen som förnyats år 2013.

Projektet hade många olika delområden. Jag ansvarade för att utveckla editorn som används för att skapa och editera arbetsannonser. Editorn är i grund och botten ett formulär som innehåller fält för de uppgifter som krävs för att en arbetsannons ska kunna publiceras på arbetsportalen. Sådana uppgifter är t.ex. titel, arbetsgivare, arbetsbeskrivning, läge, anställningsform, omfattning o.s.v. En av den nya editorns egenskaper är en realtidsförhandsvisning där man kan se hur annonsen kommer att se ut när den publiceras på arbetsportalen.

Editorn kommer att ha en del interaktiva egenskaper och därför är det bra att göra dem med JavaScript. AngularJS valdes som ramverk för editorn eftersom den har bra stöd för de delarna som kräver JavaScript. Dessutom hade jag tidigare också jobbat med AngularJS, så det var lätt att komma igång med projektet eftersom ramverket redan var bekant.

Syftet med arbetet är att beskriva hur en arbetsannons-editor utvecklades åt Oikotie Työpaikat. Editorn skulle fylla de krav som ställs på den, så att arbetsgivarna kan göra arbetsannonser som publiceras på arbetsportalen. I mitt arbete kommer jag endast att behandla frontend-delen av applikationen, med undantaget Drupal 7 Form API, som kan anses höra till backenden eftersom PHP (språket Drupal är skrivet med) är ett server-

programmeringsspråk. Andra backend-relaterade saker som sparande av annonser etc. kommer inte att behandlas i detta arbete.

Examensarbetet är strukturerat i följande kapitel: i kapitel två *Krav* beskrivs vilka krav som ställdes för editorn före utveckling. I kapitel tre *Teknik* berättas vilka tekniker som valdes att användas i editorn och varför. I fjärde kapitlet *Utveckling* beskrivs hur editorn utvecklades. Kapitel fem *Testande* beskriver hur editorn testats med enhetstester. I sista kapitlet *Slutsatser* kommer jag att gå igenom om editorn fyllde de krav som ställts för den och reflektera över vad som lyckats bra och vad som kunde ha gjorts på ett annat sätt.

2 KRAV

Arbetsportalens viktigaste tjänst är sökfunktionen för arbetsannonser. För att arbetsgivarna ska få sina arbetsannonser publicerade på tyopaikat.oikotie.fi behövs ett verktyg med vilket man kan skapa annonser som arbetssökande kan hitta med sökfunktionen. Verktyget för detta ändamål på arbetsgivarportalen är ett HTML-formulär med bestämda fält för uppgifter som behövs för en arbetsannons. Formuläret som även kallas för editor, används för att skapa nya samt för att editera existerande arbetsannonser.

När man hade beslutat att utveckla en ny editor ville man att den skulle fungera på samma sätt som den gamla editorn, men dessutom så skulle man lägga till några nya egenskaper som skulle göra editorn mera användarvänlig. Ett av dessa krav var en realtidsförhandsvisning av annonsen. Förhandsvisningen skulle visa hur annonsen kommer att se ut på basen av de uppgifter som finns i formuläret. Om man ändrar på t.ex. titeln så ska den samtidigt också ändras i förhandsvisningen.

Eftersom tyopaikat.oikotie.fi är en responsiv webbplats ska förhandsvisningen kunna ta detta i beaktande. Förhandsvisningen ska kunna emulera hur sidan ser ut med olika typer av datorer och skärmstorlekar, såsom med en vanlig dator (arbetsstation eller laptop), pekplatta och mobiltelefon. Vilken skärmstorlek som är aktiv ska kunna ändras av användaren medan man är i förhandsvisningsläget.

Företag kan ha färdiga modeller för sina annonser genom vilka man kan ändra utseendet på annonsen. Utseendet kan ändras t.ex. genom att ha en viss bild på någon plats i annonsen eller ändra vissa färger och teckensnitt till sådana som företaget har i sitt brand. Editorn ska kunna ta dessa modeller i beaktande i förhandsvisningen.

Eftersom publicering av en annons kräver att uppgifterna är korrekta måste editorn kunna validera uppgifterna som fyllts i formuläret. Exempelvis kan det fattas någon obligatorisk information och datumen för när annonsen ska publiceras ska vara genomförbara.

Annonsen ska också sparas automatiskt var tionde minut.

3 TEKNIK

I detta kapitel ges en översikt över de tekniker som använts för att skapa editorn. Det beskrivs i stora drag vad teknikerna används för, vilka alternativ det finns och orsakerna till att man valt att använda just den tekniken.

3.1 Drupal 7

Drupal är ett innehållshanteringssystem (CMS) med öppen källkod som är gjort med avsikt att man ska kunna utvidga dess funktionalitet. Drupal används mycket av bolag eftersom det är fritt, utvidgningsbart och har en stor community (Drupal, About). Drupals största konkurrent är WordPress som också är ett CMS med öppen källkod och skrivet med PHP. Eftersom arbetsportalen är uppbyggd med Drupal version 7 var det ett naturligt val att använda den när man utvecklar den nya webbsidan för editorn.

Drupal beskrivs ibland som ett innehållshanteringsramverk eftersom den ”out-of-the-box” inte har så mycket funktionalitet. Istället är det meningen att man bygger upp webbplatsen genom att utvidga CMS:ets funktionalitet genom att kombinera så kallade moduler för att få webbplatsen att fungera på önskat sätt. Därför hör det till Drupal-

utvecklarens kompetens att veta vilka moduler som kan användas för olika ändamål. (Drupal Documentation 2016a)

En modul i Drupal är en samling funktioner som utvidgar CMS:ets funktionalitet. Det finns tre typer av dem:

- Core-modulerna är de moduler som kommer färdigt med Drupal,
- Contrib-moduler är GPL-licenserade moduler som vem som helst kan ladda ner och använda på sin sida,
- Custom-moduler är sådana man utvecklat själv, ofta för ett specifikt behov på webbplatsen.

(Drupal Community Documentation 2016)

Moduler kommunicerar med Drupal med s.k. *hooks* (sv. krokar). En *hook* är en funktion som anropas vid ett visst skede under en Drupal-applikations körning. En *hook* kan tänkas vara lite som ett event i program som använder sig av sådana. Då man syftar till en *hook* i Drupal (då man t.ex. söker dess dokumentation) så beskrivs den som hook_{hooknamn}. *Hook* berättar att det är frågan om en *hook*, medan {hooknamn} identifierar namnet på den. (Drupal Documentation 2016b)

En *hook* implementeras som en global PHP-funktion. För att Drupal kan kalla på den så måste den vara namngiven på ett exakt sätt. Formeln för en *hook*-funktion är {modulnamn}_{hooknamn}. Modulnamn syftar till den modul som funktionen deklarerats i, medan hooknamn syftar på den *hook* man vill implementera (Drupal API, Hooks).

Ett exempel på en *hook* är *hook_menu* som används av moduler för att registrera en eller flera *paths*, d.v.s. adresser i Drupal (Drupal API, function hook_menu). Moduler kan också skapa egna *hooks* som andra moduler kan implementera i sin kod. Eftersom varje modul i en Drupal-installation ska ha ett unikt namn så kan flera moduler implementera samma *hook* för sina egna behov.

Form API (FAPI) är ett applikationsprogrammeringsgränssnitt (API) inbyggt i Drupal och den används för att skapa HTML-formulär på ett strukturerat sätt. Formulär i Drupal har alltid ett unikt namn med vilken den kan hänvisas till. Ett formulär görs med att en modul deklarerar en s.k. *form builder* funktion. I funktionen definierar man formuläret som en *associativ array* (andra namn för samma datastruktur är bl.a. *dictionary* och *hash*) där varje nyckel i sin tur definierar ett element i formuläret. Exempel på ett Form API element visas i Figur 1. (Drupal Documentation 2016c)

```
// FAPI
$form['name'] = array(
  '#type' => 'textfield',
  '#maxlength' => 128,
  '#required' => TRUE,
);
```

Figur 1. Exempel på Form API textelement.

Ett formulär hämtas med funktionen *drupal_get_form* (Drupal API, function drupal_get_form) och den returnerar en s.k. *render array* vilken i sin tur kan skrivas ut på sidan. *Render array* är en datastruktur som ges åt funktionen *drupal_render* vilket ändrar den till en HTML-teckensträng som kan printas på sidan. Nyttan med *render arrays* i Drupal är att de gör det enkelt för utvecklaren att ändra dess struktur före den ändras till HTML. (Melancon m.fl. 2011 s.321)

3.2 AngularJS

Redan då man planerade hur editorn tekniskt skulle fungera så var det klart att det skulle behövas en hel del JavaScript-kod för de interaktiva delarna i editorn. AngularJS valdes som ramverk eftersom den stöder de egenskaper som man visste att skulle behöva i editorn. AngularJS är utvecklat av Google och är MIT-licenserat vilket gör det möjligt att använda det i kommersiella projekt med sluten källkod. Eftersom AngularJS 2 var ännu i beta-skede då projektet påbörjades så valde man att använda version 1 av AngularJS. Utvecklaren hade också använt AngularJS tidigare i andra projekt så ramverket var redan bekant.

Alternativa bibliotek som man kunnat använda för editorn är t.ex. React. React som är utvecklat av Facebook är ett av de populäraste JavaScript-ramverken för tillfället. AngularJS är ett fullfjädrat ramverk med stöd för Model-View-Controller-arkitekturmönstret (MVC) (Green & Seshandri 2013 s.12) medan React kan tänkas vara ett bibliotek med vilket man skapar komponenter för en JavaScript-applikations användargränssnitt (React, Tutorial: Intro To React). Man kan oftast använda AngularJS utan att behöva andra bibliotek, medan med React väljer man ofta ett antal andra bibliotek som vanligtvis behövs i en JavaScript-applikation.
AngularJS har ett antal egenskaper som gör det lämpligt för utveckling. Till dem hör bl.a. *dependency injection* och testbarhet. Det är möjligt att följa MVC-arkitekturmönstret vilket många programmerare är bekanta med, men ramverket tvingar inte utvecklaren att använda det utan det är möjligt att strukturera applikationer även på andra sätt.

Oberoende hur man väljer att strukturera sin applikation så gäller vissa koncept. En *controller* är en klass vars huvudsakliga uppgift är att definiera modellen för vyn. Det görs i AngularJS med en mekanism som kallas för *scope*. *Scope* är en tjänst (eng. service) som en *controller* kan använda för att definiera modellen som vyn ska kunna använda. En *controller* är vanligtvis också ansvarig för att tolka vad som ska göras då användaren gjort något på sidan, t.ex. tryckt på en knapp. I vyn kommer man åt värden i modellen genom s.k. interpolationer. En interpolation är ett uttryck som evalueras mot en *scope*. Man kan säga att en *scope* är en mekanismen för hur modellen och dess ändringar görs bemärkta för applikationen. (Green & Seshandri 2013 s.28) Figur 2 illustrerar kopplingen mellan *controller*, modell/*scope* och vyn i en AngularJS applikation.

```
// JavaScript applikationen
var app = angular.module('mymodule', []);

app.controller('FooController', function($scope) {
  $scope.message = {
    text: 'Foo'
  };
});
app.controller('BarController', function($scope) {
  $scope.message = {
    text: 'Bar'
  };
});

// HTML källkod
<div ng-app="mymodule">
  <div ng-controller="FooController">
    <p>{{message.text}}</p>
  </div>
  <div ng-controller="BarController">
    <p>{{message.text}}</p>
  </div>
</div>

// Resulterande HTML
<body ng-app="mymodule">
  <div ng-controller="FooController">
    <p>Foo</p>
  </div>
  <div ng-controller="BarController">
    <p>Bar</p>
  </div>
</div>
```

Figur 2. AngularJS exempel på controller, modell/scope och vy.

I kodsnutten i Figur 2 finns vissa saker som är värda att notera. För det första skapas en modul med namnet ”mymodule” och sparas i variabeln ”app”. Efter det anropas modulens *controller* funktion två gånger med olika argument. Funktionen accepterar två argument, ett namn och implementationen av en *controller*. Till båda injiceras ett *$scope*-argument som kan användas för att definiera modellen. Med hjälp av *ng-app* och *ng-controller* attributerna berättar man åt AngularJS vilken modul och *controller* som används för olika delarna av HTML-snutten. Vanligtvis har man endast en modul på HTML-sidan, medan *controller*-antalet varierar. Uttrycket *{{message.text}}* används för att evaluera ”message.text”-attributen mot den *scope* som uttrycket är bundet med.

Dependency injection används på många ställen i AngularJS-applikationer. Det används för att en komponent ska kunna berätta åt ramverket vilka tjänster den behöver i sin implementation. Den inbyggda $injector-tjänsten i är ansvarig för att skapa och injicera

tjänster i AngularJS. Det finns flera sätt att berätta åt $injector vilka tjänster som ska injiceras. Ett sätt är att lägga till en ”$inject”-attribut, som är en räcka med namnen på det som ska injiceras, till funktionen som ska ha tjänsterna injicerade. Den andra metoden är att byta ut funktionen till en räcka där alla utom sista värdet är namn på tjänster som ska injiceras, det sista elementet i räckan är funktionen som skapar komponenten. Det sista sättet är det enklaste att använda för programmeraren, $injector ser på namnen på de argument som funktionens har och injicerar dem. Det sista sättet går ändå inte att använda om man minimerar koden, eftersom verktyg som gör det brukar byta namnen på en funktions argument. (AngularJS API Reference, $injector) En tjänst i AngularJS instansieras först då den krävs för första gången i en applikation och det finns alltid endast en instans av tjänsten (AngularJS Developer Guide, Services).

Tjänster är objekt eller värden som går att injicera i olika delar i kod. Orsaker varför man vill använda dem är bl.a. organisering av kod samt minimering av duplikat. Man kan också använda dem för att dela data mellan olika delar av applikationen. Tjänster går att skapa med en moduls *constant*, *value*, *factory*, *service* och *provider* metoder. *Constant* och *value* används för vanliga värden och funktioner. *Factory*, *service* och *provider* skapar alla likadana objekt, men skillnaden på dem är hur objektet instansieras. Eftersom tjänsterna går att injicera så har det den fördelen att de lätt kan *mockas* (sv. imiteras) för test. Exempel på inbyggda tjänster är t.ex. *$http* som används för att göra HTTP-anrop till en server. (Lerner 2015 s. 159-161) Modulen har dessutom en *decorator* metod som kan användas för att ändra de publika egenskaperna på vilken som helst tjänst. Det kan vara nyttigt för att lägga till någon funktionalitet till en tjänst som endast behövs i tester. Exempel på det är *$interval*-servicens ”flush” metod som läggs till med AngularJS eget enhetstestbibliotek *ngMocks*.

Ett av AngularJS viktigaste och mest komplicerade egenskaper är s.k. direktiv. Direktiv är ett sätt att lägga till mera funktionalitet till ett HTML-element än vad den vanligtvis stöder. (Lerner 2015 s.104) Ramverket har många färdigt inbyggda direktiv, men det går också att skapa själv sådana för olika behov. Ett av de mest använda direktiven är det inbyggda *ngModel* som binder värdet i en formkontroll, t.ex. ett input-element med en modells värde i en *scope*. Med hjälp av det direktivet så kan modellens värde bytas vid interaktion av input-elementet. (AngularJS API Reference, ngModel) Andra in-

byggda direktiv är t.ex. *ngApp* och *ngController* som visades i Figur 2. Exempel på ett eget direktiv kunde vara t.ex. en bildkarusell.

Direktiv registreras med modulens *directive* metod. Metoden tar som argument direktivets namn, och en funktion ska returnera ett objekt vilken definierar hur direktivet fungerar. Med objektet kan man kontrollera mycket noggrant hur direktivet ska fungera, men för de enklare direktiven så räcker det oftast med att man specificerar endast det nödvändigaste. De vanligaste inställningarna man brukar välja för ett direktiv är hur de skapas, vilken *scope* som används och hur direktivet kommunicerar med DOM:en. Ett exempel på ett direktiv som gör dess innehåll att blinka med en viss intervall visas i Figur 3.

```
// JavaScript
angular.module('mymodule')
.directive('blinking', function($interval) {
  return {
    restrict: 'AE',
    link: function(scope, element, attrs) {
      var intervalMs = parseInt(attrs.blinkingInterval);
      if (!intervalMs) {
        return;
      }

      element.css('visibility', 'visible');

      var promise = $interval(function() {
        var isHidden = element.css('visibility') === 'hidden';
        element.css('visibility', isHidden ? 'visible' : 'hidden');
      }, intervalMs);

      scope.$on('$destroy', function() {
        $interval.cancel(promise);
      });
    }
  };
});

// HTML
<blinking blinking-interval="1000">Blinking text</blinking>
```

Figur 3. Exempel direktiv som gör dess innehåll att blinka med givet mellanrum.

Direktivet i Figur 3 har några saker som kan tas upp. Eftersom direktivets mening att blinka med ett jämnt mellanrum så behövs den inbyggda *$interval*-tjänsten. *$interval* är en *wrapper* för webbläddrarens *window.setInterval* metoden som används för att kalla på en funktion med jämna intervaller. Eftersom man valt att göra det som en tjänst går den också lätt använda i test på ett synkront sätt, något man inte kan göra med

window.setInterval. Man berättar mera om testande i AngularJS i kapitel 5 Testande. *Restrict "AE"* betyder att direktivet kan användas som ett element eller som ett attribut. En annan sak som är värd att notera är att "blinking-interval" attributen blivit normaliserad i *attrs*-objektet eftersom HTML inte är skiftlägeskänsligt. Sedan definierar man i *link*-metoden intervallfunktionen som ska köras med jämna mellanrum. *Scope.$on* är en metod för att registrera en funktion som anropas vid en viss händelse. När denna *scope* förstörs så avregistreras intervallfunktionen för att inte skapa minnesläckage i applikationen. (AngularJS Developer Guide, Creating Custom Directives)

3.3 Gulp

Då man utvecklar en JavaScript-applikation kan man märka att man håller på att göra samma saker om och om igen. Det är vanligt att man splittrat koden i flera olika filer, men det rekommenderas att man slår dem alla ihop till en fil så att webbläsaren behöver ladda ner endast en fil istället för många olika. När filarna är slagna ihop så vill man kanske ännu minimera filen så att den ska bli så liten som möjligt. Om man följer Test Driven Development-metodologin (TDD) då man utvecklar kan det hända att man vill köra enhetstester varje gång som koden ändras för att se om man introducerat buggar i koden. Alla dessa är exempel på saker som kan automatiseras med en s.k. uppgiftsautomatiserare. Exempel på sådana är Gulp och Grunt, och de är båda program som körs på NodeJS. Skillnader mellan dem är att Grunt använder temporära filer för att dela data mellan uppgifter, medan Gulp använder strömmar (eng. *stream*). Grunts sätt att definiera uppgifter är deklarativ medan man i Gulp skriver allting med vanliga JavaScript funktioner. Gulp-uppgifterna konfigureras i en JavaScript-fil som heter "gulpfile.js", och den placeras vanligen i projektets rotkatalog. Utvecklaren valde att använda Gulp eftersom det verkade enklare att använda än Grunt.

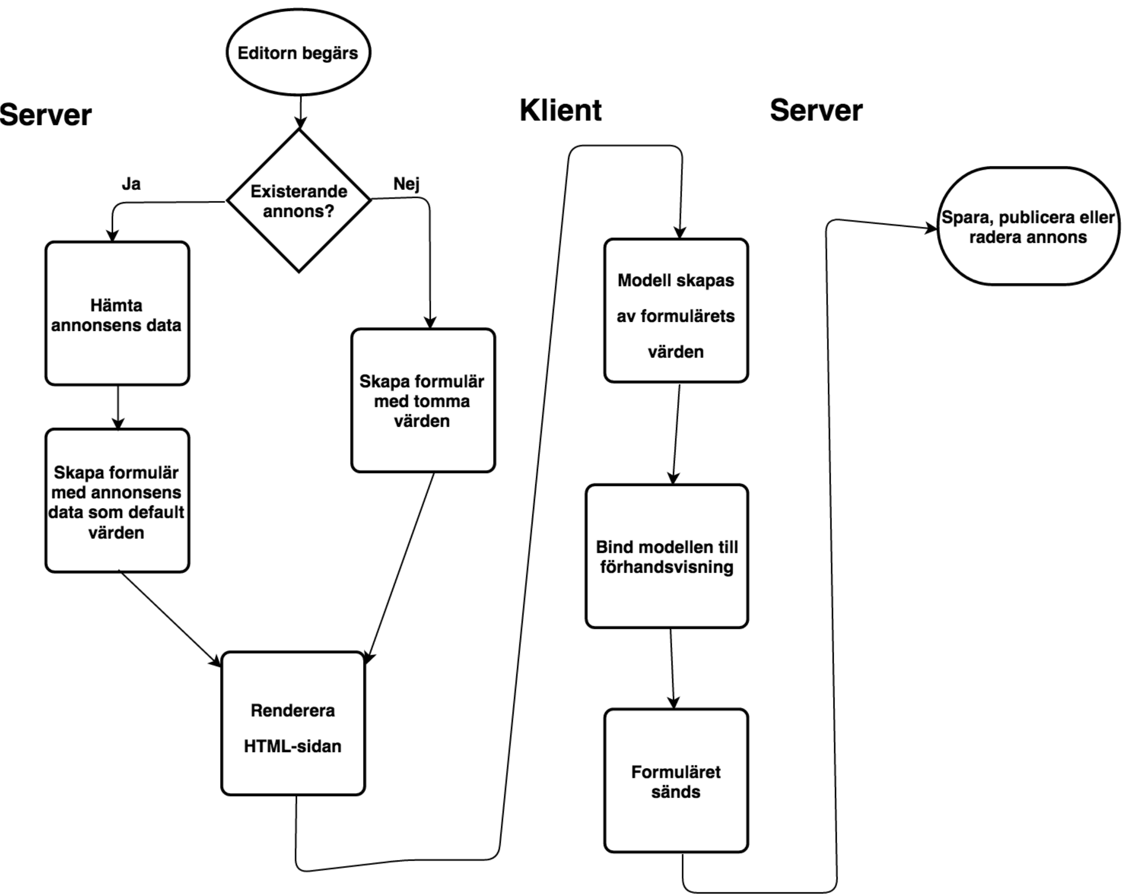
3.4 Implementation i editorn

De tekniker som berättats om i detta kapitel kommer att användas i nästa kapitel där man går igenom hur editorn är uppbyggd. Drupal Form API användes för att skapa formuläret, AngularJS användes huvudsakligen för förhandsvisningen, och eftersom den redan användes för det så valde man att implementera en del andra saker också med

AngularJS, t.ex. annonsens validering. Annonsen valideras redan på klientsidan eftersom det är vänligare för användaren än att validera först på servern, men eftersom det är en säkerhetsrisk att lita endast på klienten så valideras uppgifterna en gång till på servern. Gulp användes för att bygga applikationens JavaScript-del, även det kommer att behandlas noggrannare i nästa kapitel.

4 UTVECKLING

Editorn består som sagt av ett formulär där man editerar annonsens uppgifter, och förhandsvisningen som visar annonsen på basen av de uppgifter som finns i formuläret. Största delen av editorns HTML skapas på serversidan med PHP, exklusive det vissa direktiv skapar i klienten. Servern renderar formuläret, skelettet för förhandsvisningen och knapparna man använder för att kontrollera editorn. Med skelettet av förhandsvisningen menas att annonsens HTML skapas redan på serversidan, men platserna i förhandsvisningen där det borde finnas data från annonsen så hänvisas istället till värden i en AngularJS modell. När JavaScript-applikationen initialiserats så kommer modellen att få värden från annonsens motsvarande fält i formuläret. För att få JavaScript-kod anknutet med HTML-delar används *ngController* och andra direktiv. I editorn skapas annons-modellen så att formuläret har en egen *controller* (instansierat med *ngController*) som skapar modellen av formulärets värden, och ger modellen vidare i applikationen med en tjänst som andra kan använda. Noggrannare detaljerna på hur editorn tekniskt sett är uppbyggd beskrivs i detta kapitel.



Figur 4. Flödet av annonsens data i editorn.

4.1 Serversidan

Formuläret gjordes med Drupal 7 Form API (FAPI) p.g.a. några olika orsaker. En av orsakerna var att man med FAPI har åtkomst till vissa behändiga egenskaper som hjälper utvecklandet, t.ex. funktioner för validering av uppgifterna i formuläret. Då formuläret skickas kan man välja att köra en viss PHP-funktion som kan varieras beroende på vilken knapp som formuläret skickats med. Editorn har tre olika knappar som skickar formuläret till servern. Annonsen kan antingen raderas, sparas, eller publiceras. Om användaren väljer att radera annonsen så anropas PHP-funktionen som raderar annonsen. Likaså anropas särskilda funktioner om användaren väljer att spara eller publicera annonsen. Skillnaden mellan en sparad och publicerad annons är att en sparad annons endast syns för användaren, medan en publicerad annons syns på arbetsportalen så att alla kan se den.

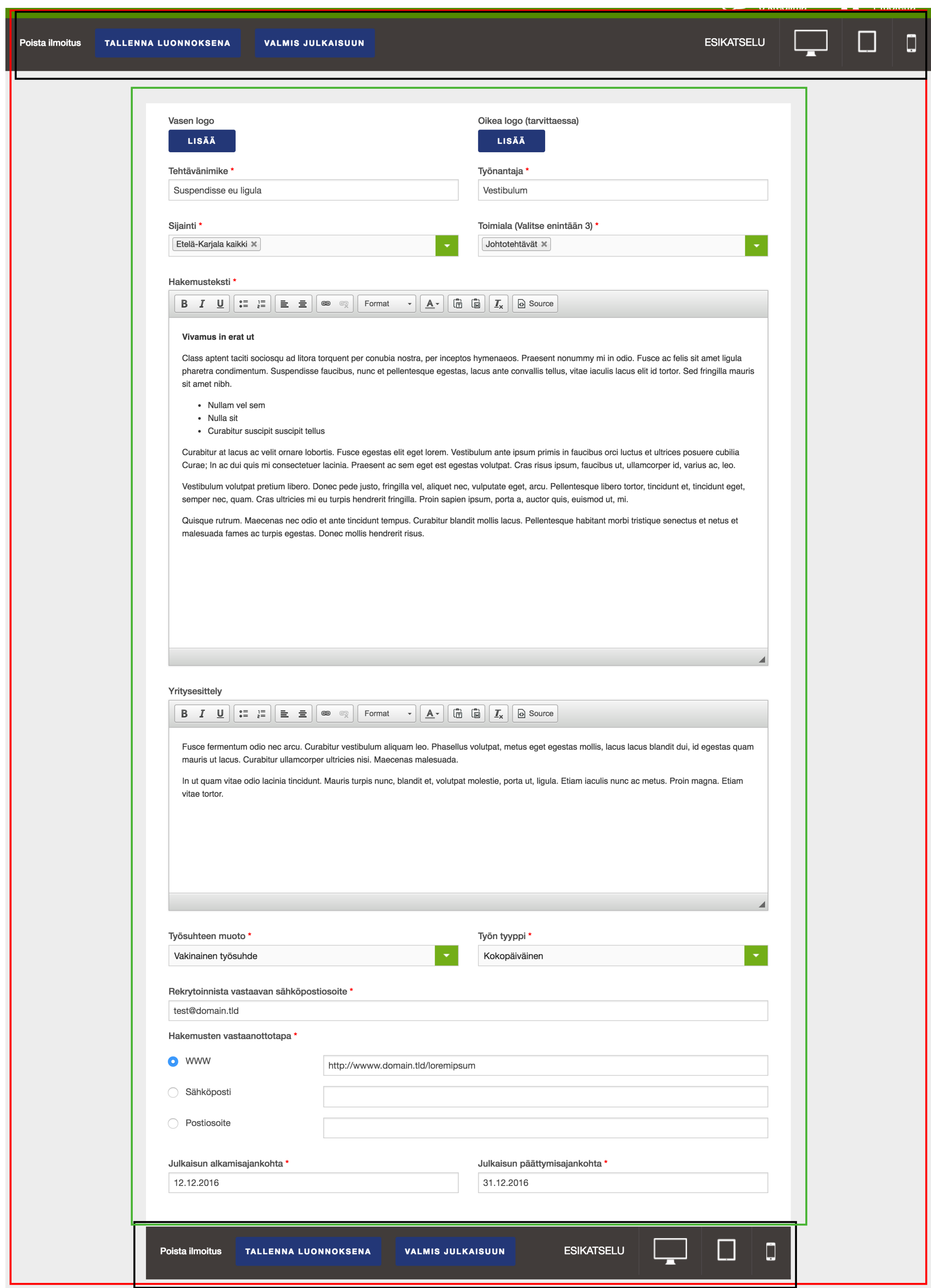
Arbetsflödet för hur editorn skapas på server-sidan är följande:

- Webbläddraren ber editor-sidan av Drupal.
- Om sidbegäran innehåller en annons ID så hämtas annonsens data och försäkrar att användaren faktiskt har åtkomst till annonsen.
- Skapar formuläret med FAPI. Om det är en existerande annons som ska editeras så skrivs annonsens uppgifter färdigt i fälten för dem.
- Skapa HTML-sidan med formuläret, förhandsvisning etc. Här bestäms vilken JavaScript-kod som binds med olika delarna av HTML-sidan m.h.a. direktiv som används på passande ställen, t.ex. *ngController* för formuläret och förhandsvisningen.
- Drupal svarar till sidbegäran med HTML-sidan som innehåller editorns formulär, förhandsvisningen o.s.v.

4.2 Klientsidan

Efter att klienten fått svaret från servern och sidans DOM är färdigt så anropas *angular.bootstrap*-metoden för att starta editorns JavaScript. Editorns AngularJS-applikation är strukturerad så att en *controller* är ansvarig för en specifik del av det som syns i webbläddraren.

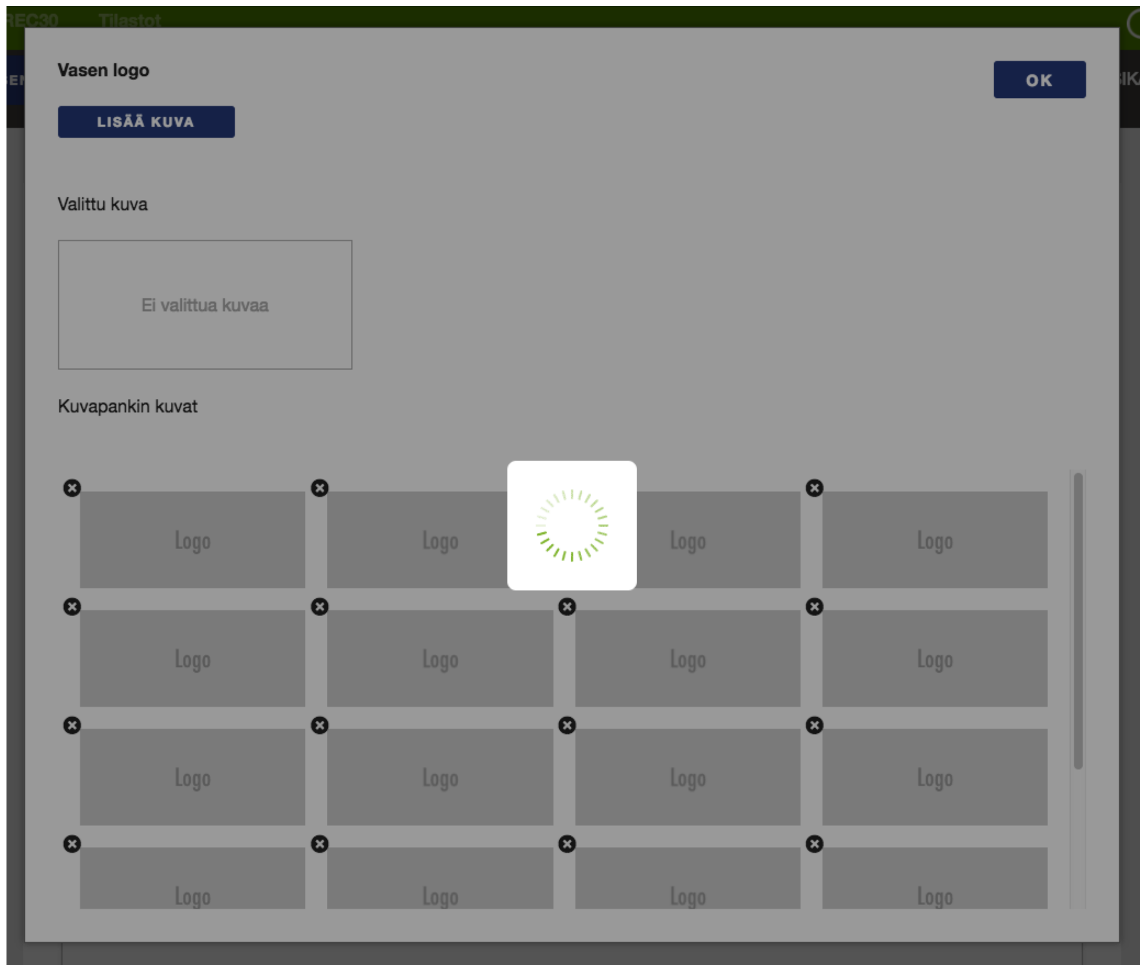
Figur 5 som visar hur editorns formulär ser ut har olikfärgade rektanglar ritade runt de delar som är kontrollerade av en *controller*. Innanför den gröna rektangeln är editorns formulär. I den fyller man i de uppgifter som behövs för en arbetsannons. Den *controllerns* primära uppgift är att hämta annonsens värden från formuläret och publicera dem till en tjänst som resten av applikationen kan använda för sina ändamål. De delar som är omgivna med svart låda i kallas för verktygsfältet och är kontrollerade av den *controller* som har knappar för att ändra vad som syns i editorn, d.v.s. formuläret eller förhandsvisningen. Där finns det också knappar för raderande, sparande och publicerande av annonsen. Knapparna på datorer av olika skärmstorlek används för att ändra storleken på förhandsvisningen. Den röda rutan visar att det finns en *controller* som man kan använda var som helst i HTML-dokumentet.



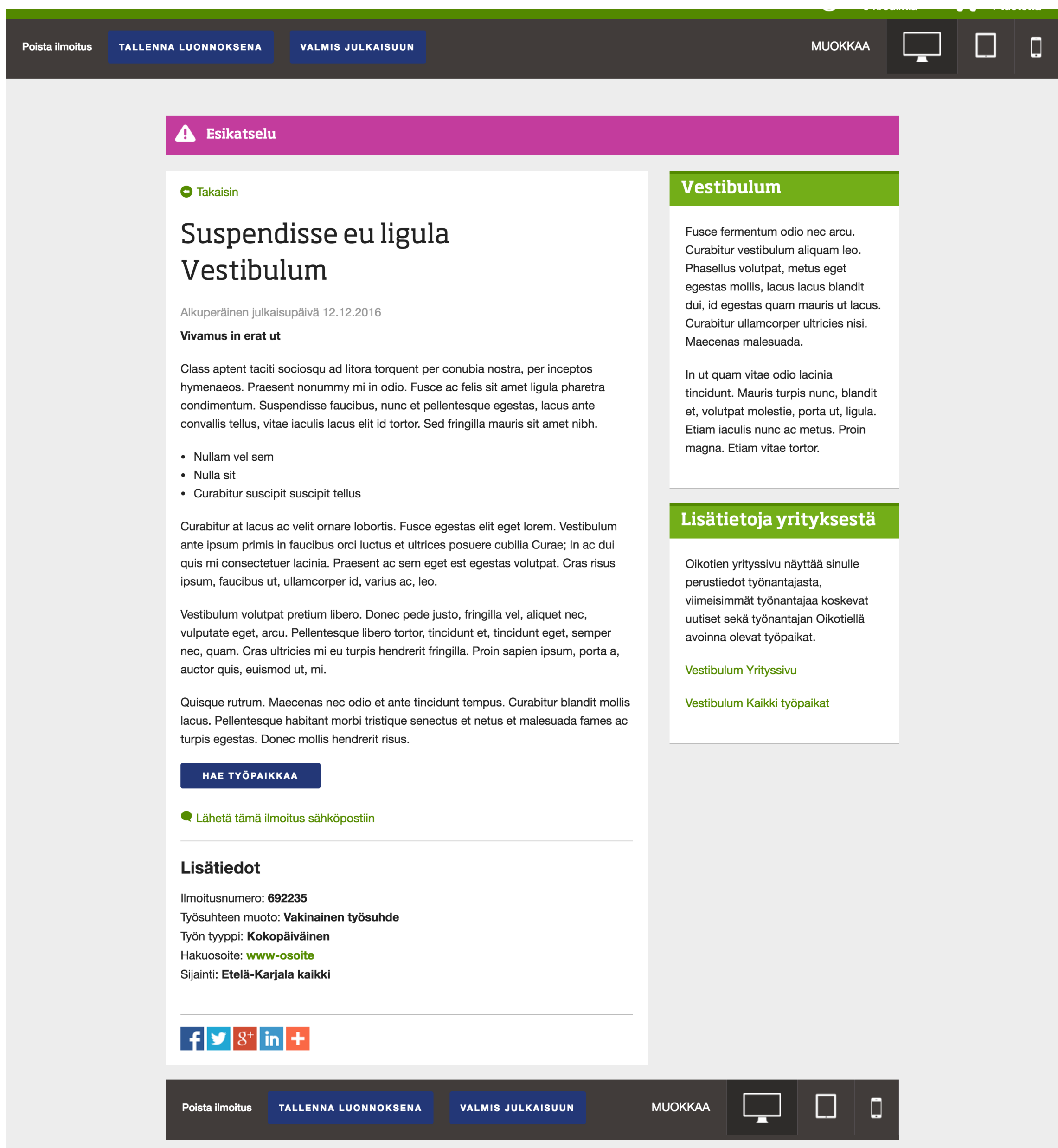
Figur 5. Editorns formulär.

Behovet för en global *controller* uppstod då man behövde komma åt vissa funktioner från s.k. modalfönster. Problemet med modalfönster i editorn är att de placeras i slutet

av HTML-dokumentets ”body”-element, vilket är utanför varenda en annan *controller* i DOM:en. Ett exempel på en sådan funktion är att visa en förloppningsindikator då något är under arbete, t.ex. då man laddar upp en bild till servern från modalfönstret med komponenten som används för att välja bilder till annonsen. Fast knappen man öppnar modalfönstret med är innanför formulärets *controller*, så kommer själva modalfönstret inte längre vara det.



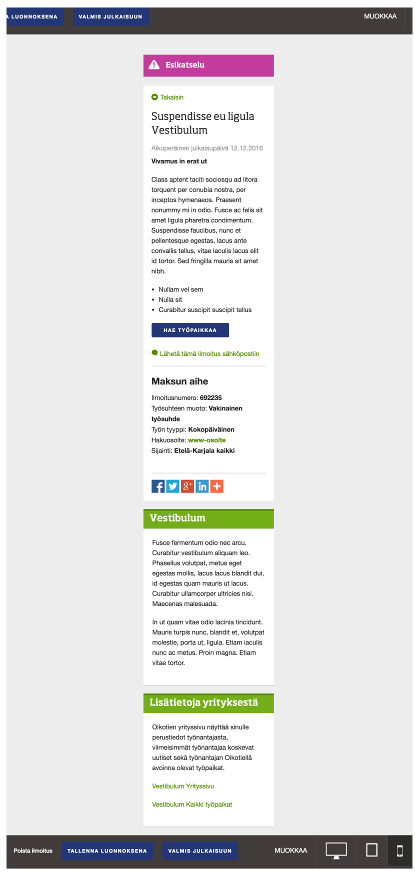
Figur 6. Komponenten för bildhantering och förloppningsindikator.



Figur 7. Editorns förhandsvisning.

I Figur 7 visas förhandsvisningen av annonsen. De uppgifter som syns på olika ställen i förhandsvisningen är exakt de samma som finns i formuläret. T.ex. texten i annonsens rubrik kommer från formulärets fält för titel och arbetsgivare. Vilken skärmstorlek som emuleras väljs med knapparna i verktygsfältet. Oberoende vilken skärmstorlek som valts så syns endast förhandsvisningen. Om formuläret valts så syns inte förhandsvisningen. För att välja vilken skärmstorlek som är aktiv så lägger man till en HTML-klass till div-element som omringar förhandsvisningen. Vilken klass som är aktiv beror på den skärmstorleken som valts. Beroende på vilken klass som är aktiv så finns det CSS-

kod som motsvarar den klassen och specificerar hur bred förhandsvisningen är. ”Desktop”-bredden har inga restriktioner för hur bred sidan får vara. ”Tablet” sätter bredden till en sådan pixelmängd som kan anses vara vanlig för pekplattor. Likaså sätter ”mobil” förhandsvisningens bredd till sådan som används ofta med mobiltelefoner. Dessutom ändras sidans layout till att ha endast en kolumn istället för två, såsom det görs för annonser på arbetsportalen.



Figur 8. Mobil förhandsvisning.

Som man såg i Figur 5 så omringar den s.k. globala *controllern* de andra. För att skilja mellan dem så har man gett ett eget namn åt varje *controller* som de kan hänvisas med i HTML-dokumentet. AngularJS *ngController*-direktivet har stöd för s.k. *controller as* syntax, med vilket man kan ge specifika namn åt dem. Exempel på hur *controller as* syntaxen används för att skriva om koden i Figur 2 visas i Figur 9. Då man använder *controller as* syntaxen så hänvisar man till värden definierade i *controllerns* ”this”, istället för värden i $scope-tjänsten (AngularJS API Reference, ngController).

```
// JavaScript
var app = angular.module('mymodule', []);

app.controller('FooController', function() {
  this.message = {
    text: 'Foo'
  };
});
app.controller('BarController', function() {
  this.message = {
    text: 'Bar'
  };
});

// HTML
<div ng-app="mymodule">
  <div ng-controller="FooController as foo">
   <p>{{foo.message.text}}</p>
  </div>
  <div ng-controller="BarController as bar">
   <p>{{bar.message.text}}</p>
  </div>
</div>
```

Figur 9. Exempel på controller as syntaxen.

AngularJS har bra egenskaper för att skapa egna valideringsfunktioner på klientsidan med hjälp av det inbyggda *ngModel*-direktivet vilket ofta användas i samband med formelement som ”input”. AngularJS har skapat färdigt inbyggda valideringar, bl.a. *required* och *pattern* för ”input”-elementet, men om de inbyggda valideringsfunktionerna inte räcker till så går det bra att skapa egna också. *NgModel*-direktivet skapar en instans av *NgModelController* för elementet, och den har en API som man kan använda bl.a. för att skapa egna funktioner som validerar att informationen i elementet är korrekt varje gång värdet i elementet ändras. För editorn användes både inbyggda och skräddarsydda valideringar. Dessutom gjordes det ett särskilt direktiv för att visa felmeddelanden vid formelementet.

Figur 10. Ett formelements felmeddelande.

4.3 Kommunikation mellan komponenter

Komponenterna i applikationen, t.ex. formuläret och förhandsvisningen kommunicerar med varandra. Den tydligaste av dessa krav av kommunikation var att uppgifterna från formuläret måste visas i förhandsvisningen på specifika ställen. Det fanns olika alternativ för hur kommunikationen kunde skötas. Man kan ha den gemensamma datan i en enda *controller* som formuläret och förhandsvisningen delar på. Eftersom man försökte hålla varje *controller* enkel valde man att använda en tjänst för att dela med sig uppgifterna från formuläret till resten av applikationen. Dessutom kan det hända att man behöver annonsens uppgifter annanstans också än i förhandsvisningen, och då är det praktiskt att man får uppgifterna via en tjänst. Om kommunikationen valts att sköta med en *scope* så är implementationen beroende på HTML-strukturen, vilket den inte är då man använder en tjänst. Det underlättar också testandet eftersom man då kan *mocka* tjänsten som erbjuder annonsens uppgifter, istället för att vara tvungen att skapa *controllern* som skapar annonsens uppgifter. Eftersom objekt i JavaScript är passade som referenser så betyder det att då annonsens data givits åt förhandsvisningen så behöver man inte skicka objektet på nytt varje gång ett värde i den ändrats. Då någon uppgift i formuläret ändras syns ändringen direkt i förhandsvisningen också eftersom det objekt som den använder för annonsens uppgifter hänvisar till samma objekt.

4.4 Förhandsvisningen

Förhandsvisningen består av HTML som syns då editorn är i förhandsvisningsläge, och en *controller* som erbjuder HTML-koden uppgifter från annonsen. Förhandsvisningens

controller får annonsens data som sagt från en tjänst. Exempel på hur en bild från annonsen visas i förhandsvisningen visas i Figur 11.

```
<img ng-src="{{previewVm.ad.logopath}}"/>
```

Figur 11. Exempel på hur en annons bild visas i förhandsvisningen.

Ifall arbetsannonsmallar är i bruk används ett direktiv speciellt gjort för dem på flera ställen i förhandsvisningen. Direktivet för mallarna fungerar så att de tar den mall som används för annonsen som input, och den egenskap av modellen som ska användas. Med hjälp av direktivet kan man sätta in en viss förhandsinställd HTML-snutt i annonsen på bestämda ställen i förhandsvisningen. Exempel på hur det direktivet används visas i Figur 12. Eftersom ”replacement”-direktivets ”replacements”-attribut är specificerat som en interpolation i direktivet, kommer direktivet att se det värde som objektet ”previewVm.replacements” har i *scope* istället för en teckensträng.

```
<replacement replacements="previewVm.replacements" name="header"></replacements>
```

Figur 12. Användning av ”replacement”-direktivet.

4.5 Gulp uppgifter

Gulp användes för några olika ändamål i projektet, varav ett var att sammanslå JavaScript-filerna. Eftersom editorn består av många AngularJS-tjänster och komponenter så var det bra att ha dem i egna filer för att hålla en bra struktur på projektet. Istället för att webbläddraren skulle vara tvungen att begära varje JavaScript-fil i editorn skilt så har de slagits samman till endast några filer. Ena filen innehåller ramverket och andra bibliotek som behövs för editorn, medan den andra innehåller den kod som skrivits själv för editorn. Koden för Gulp-uppgiften som slår samman filerna visas i Figur 13 och den körs med kommandot ”gulp build” i kommandotolken. *Gulp* hänvisar till programmet som ska köras, medan argumentet *build* berättar åt Gulp vilket arbete som ska utföras. Den hämtar först alla editorns JavaScript-filer definierade i *scriptPaths* variabeln, kör koden genom funktionerna *ngAnnotate* och *iife* och slår dem samman i en ny fil *app.js*

och skriver ut den resulterande filen till *dist*-katalogen. NgAnnotate-funktionen är en plugin för Gulp som skriver om alla ”dependency”-injicerade funktioner till en sådan form att de kan användas även när JavaScript-filen blivit minimerad. Iife står för ”immediately-invoked function expression” och den slår in all resulterande kod innanför en anonym JavaScript-funktion som anropas direkt av sig själv. Det är en teknik som JavaScript-utvecklare använder för att variabler och funktioner inte ska bli globala.

```
// Gulpfile.js

var gulp = require('gulp');
var iife = require('gulp-iife');
var concat = require('gulp-concat');
var ngAnnotate = require('gulp-ng-annotate');

gulp.task('build', function() {
  return gulp.src(scriptPaths)
    .pipe(concat('app.js'))
    .pipe(ngAnnotate())
    .pipe(iife())
    .pipe(gulp.dest('dist'));
});
```

Figur 13. Gulp-arbete för att slå samman editorns JavaScript-filer.

Gulp användes också för att slå samman vissa CSS-filer som man hamnade använda för att några biblioteks element skulle se rätt ut. Dessutom används Gulp för att köra enhetstester. En mycket använd Gulp-egenskap var *watch.* Det är en process som körs konstant i kommandotolken. Alltid när den märker att någon fil i editorn ändrats så sammanslår den alla filer vilket snabbade utvecklandet. Man kunde även köra enhetstesterna varje gång en fil ändrats så att man får direkt respons på om man söndrat något, men till vidare körs testerna skilt med ett eget arbete i gulp.

5 TESTANDE

Det finns olika testmetodologier, varav de vanligaste är *black box* och *white box* testande. Med *black box* testande menas att man betraktar applikationen som en låda vars inre struktur och design man inte vet någonting om. Programmet tar in data, behandlar den och ger sedan ett resultat. Ett test anses vara lyckat då programmet kan både hantera den data den fått och resultatet är som förväntat. I *white box* testande så krävs det däre-

mot att man känner till hur inre delarna av programmet är uppbyggt. Testfallen ska utses noggrant för att täcka de viktiga delarna av koden. (Riley & Goucher s.270)

Enhetstester är ett exempel på *white box* testande, och sådana valdes att göra till editorn. Det rekommenderas att man använder Karma, Jasmine och ngMock för att underlätta enhetstestandet av AngularJS-applikationer (AngularJS Developer Guide, Unit Testing). Karma är ett program som används för att köra enhetstesterna i en webbläddrare (Karma, How It Works). Jasmine är ramverk för att testa JavaScript kod. Den ger utvecklaren bl.a. funktioner som används för att hävda att ett en del av programmet, t.ex. en funktion fungerat rätt. *NgMock* är en modul som hjälper med enhetstestandet av AngularJS applikationer. Den ger utvecklaren verktyg för att injicera och *mocka* tjänster i enhetstesterna. Dessutom utvidgar modulen vissa ramverkets tjänster så att testerna går att användas i testerna på ett synkront sätt (AngularJS API Reference, ngMock).

Ett Gulp-arbete gjordes för att köra testerna. Detta arbete startar Karma, som sedan kör alla editorns enhetstester. Webbläddraren som Karma kör testerna i är PhantomJS som är en *headless* webbläddrare vilket betyder att den inte har ett grafiskt användargränssnitt. Man är ändå inte tvungen att använda PhantomJS utan man kan använda t.ex. Google Chrome eller Mozilla Firefox istället. När testerna körts så syns testresultaten i kommandotolken. Med Karma är det även möjligt att köra testerna som en del av kontinuerlig integration, och det finns en plugin till det för bl.a. Jenkins build-systemet.

Jasmine valdes som testramverk eftersom den verkade ha stöd för allt som skulle behövas för att göra enhetstesterna till editorn. Ett liknande ramverk som kan användas istället för Jasmine är Mocha.

Enhetstesterna för editorn är strukturerade i filer som innehåller tester för en komponent. Testerna i Jasmine skrivs med JavaScript och är strukturerade i block som kallas för "test suite", och de innehåller tester för komponenten. En *test suite* deklareras i Jasmine med *describe* funktionen som accepterar två argument: en teckensträng (vanligtvis namnet på det som testas) och en funktion som implementerar en *test suite*. Själva testerna skapas med *it* funktionen som också accepterar två argument: en teckensträng och en funktion. Teckensträngen är titeln på det som testas och funktionen implementerar

testen. Ett test kan ha en eller flera påståenden (eng. assertion) som skapas med funktionen *expect*. Den används för att se till om enheten man testar fungerar på förväntat sätt. Om ett påstående inte stämmer så har testet misslyckats. Exempel på en *test suite* vilken testar ”blinking”-direktivet från Figur 3 visas i Figur 14 nedan.

```
// blinkingSpec.js
describe('blinking directive', function() {

  var $interval, $compile, $rootScope;

  beforeEach(function() {
    module('mymodule');
    inject(function(_$interval_, _$compile_, _$rootScope_) {
      $interval = _$interval_;
      $compile = _$compile_;
      $rootScope = _$rootScope_;
    });
  });

  it('should toggle visibility by X seconds', function() {
    var tpl = '<blinking blinking-interval="5000">Blinking text</blinking>';
    var scope = $rootScope.$new();
    var element = $compile(tpl)(scope);
    expect(element.css('visibility')).toBe('visible');
    $interval.flush(5000);
    expect(element.css('visibility')).toBe('hidden');
    $interval.flush(5000);
    expect(element.css('visibility')).toBe('visible');
  });

});
```

Figur 14. Exempel Jasmine test suite.

I Figur 14 finns några saker som inte tagits upp tidigare. ”BeforeEach” är en funktion som körs före varje test i en *test suite* och kan användas för att göra saker som man annars skulle hamna repetera i varje test. ”Module” och ”inject” är funktioner från *ngMock* biblioteket och de används för att ladda moduler och injicera tjänster till den *test suite* de körs i. Som man ser så kan tjänsterna i ”inject”-funktionen injiceras så att de har ”_”-tecken i början och slut på deras namn. Det har gjorts så att man kan spara dem med deras riktiga namn i testen. Då *$interval* används i tester med *ngMock* biblioteket så har den fått en ”flush”-metod vilken kan anropas för att ”hoppa framåt” i tiden och på det sättet kalla på de intervallfunktioner som registrerats med $interval-tjänsten.

6 SLUTSATSER

Editorn publicerades samtidigt med resten av projektet i början av november 2016, och då konstaterade man att editorn uppfyllde de krav som hade ställts för den. Användarna av editorn har dock gett ganska lite feedback som kunde tas upp här.

Tekniskt sett var det nyttigt att använda ett ramverk för editorns JavaScript-del. Under editorns utveckling hade jag knappt alls några problem med att förstå hur något i ramverket skulle användas eller varför något man gjorde inte fungerade. Fast AngularJS är ett komplicerat ramverk så har den mycket bra dokumentation där man hittar alla tekniska detaljer på hur delarna av ramverket fungerar och ska användas. Dessutom finns det mycket skrivet om AngularJS i bloggar och i andra publikationer. Däremot kan kvaliteten på tredje partens moduler vara ganska varierande. Vissa moduler som prövades var vi tvungna att avslå för att de inte fungerade tillräckligt bra i editorn. Till slut använde vi ganska få tredje partens moduler, och de som valdes var ofta större projekt med många användare.

Enhetstesterna gjordes först efter att editorn annars var klar, så de kunde tänkas som ett slags verktyg för att testa hur bra designen på koden är. Vissa enheter var enkla att testa medan vissa var lite mera krävande. I något skede av utvecklingen blev jag missnöjd med hur mycket kod som fanns i vissa komponenter, t.ex. den *controller* som är ansvarig för formuläret hade ganska mycket kod och gjorde många olika saker, vilket också gjorde det svårt att få en bra överblick av koden. Som en följd av detta så omstrukturerades editorns JavaScript-kod till flera tjänster och direktiv, med mycket lite egen kod kvar i en *controller*. Som en följd av detta hade man komponenter och tjänster som är lättare att resonera med, vilket också borde leda till helheter som är lättare att testa. Med tanke på detta så är det inte så överraskande att den komponenten som var svårast att testa hade överlägset mest kod. Jasmine kändes som ett bra ramverk att skriva enhetstester med och var ganska lätt att komma igång med.

Eftersom arbetsportalen är gjord med Drupal 7 så är det till slut ganska naturligt att man använde Drupal Form API för att skapa formuläret. Då utvecklingen av editorn inleddes försökte jag göra editorns formulär helt och hållet på klientsidan med AngularJS och

vanlig HTML, medan endast en del gjordes på server-sidan. När projektet framskred blev editorns server-sida mera komplicerad p.g.a. nya egenskaper som behövdes, och till slut valde man att göra formuläret helt på server-sidan med Form API för att få koden bättre strukturerad. Ändringarna i formuläret gjordes i samband med omstrukturerandet av editorns JavaScript-kod, och personligen är jag ganska nöjd med resultatet.

Före jag skrev enhetstesterna för editorn så hade jag gjort endast s.k. *acceptance* test. En av skillnaderna som jag märkte var att enhetstesterna utförs mycket snabbare. För följande JavaScript-projekt kunde jag tänka mig skriva enhetstester redan då man utvecklar programmet.

KÄLLOR

AngularJS API Reference, ngController. Tillgänglig: https://docs.angularjs.org/api/ng/directive/ngController. Hämtad: 11.12.2016

AngularJS API Reference, ngMock. Tillgänglig: https://docs.angularjs.org/api/ngMock. Hämtad: 7.12.2016

AngularJS API Reference, ngModel. Tillgänglig: https://docs.angularjs.org/api/ng/directive/ngModel. Hämtad: 30.11.2016

AngularJS API Reference, $injector. Tillgänglig: https://docs.angularjs.org/api/auto/service/$injector. Hämtad 30.11.2016

AngularJS Developer Guide, Creating Custom Directives. Tillgänglig https://docs.angularjs.org/guide/directive. Hämtad 30.11.2016

AngularJS Developer Guide, Services. Tillgänglig https://docs.angularjs.org/guide/services. Hämtad 30.11.2016

AngularJS Developer Guide, Unit Testing. Tillgänglig https://docs.angularjs.org/guide/unit-testing. Hämtad 7.12.2016

Drupal, About. Tillgänglig: https://www.drupal.org/about. Hämtad: 11.12.2016

Drupal API, function hook_menu. Tillgänglig: https://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/hook_menu/7.x. Hämtad 30.11.2016

Drupal API, function drupal_get_form. Tillgänglig: https://api.drupal.org/api/drupal/includes%21form.inc/function/drupal_get_form/7.x. Hämtad: 30.11.2016

Drupal API, Hooks. Tillgänglig: https://api.drupal.org/api/drupal/includes!module.inc/group/hooks/7.x. Hämtad: 30.11.2016

Drupal Community Documentation. 2016, Module developer’s guide. Tillgänglig: https://www.drupal.org/developing/modules. Hämtad: 30.11.2016

Drupal Documentation. 2016a, Overview. Tillgänglig: https://www.drupal.org/node/265726. Hämtad: 30.11.2016

Drupal Documentation. 2016b, Understanding the hook system for Drupal modules. Tillgänglig: https://www.drupal.org/docs/7/creating-custom-modules/understanding-the-hook-system-for-drupal-modules. Hämtad 30.11.2016

Drupal Documentation. 2016c, Creating a builder function to generate a form. Tillgänglig: https://www.drupal.org/docs/7/api/form-api/creating-a-builder-function-to-generate-a-form. Hämtad: 30.11.2016

Green, Brad & Seshadri, Shyam. 2013, AngularJS, O'Reilly Media, 183 s.

Karma, How It Works. Tillgänglig: https://karma-runner.github.io/1.0/intro/how-it-works.html. Hämtad: 7.12.2016

Lerner, Ari. 2015, ng-book: The Complete Book on AngularJS, Fullstack.io, 605 s.

Melancon, Benjamin m.fl. 2011, The Definitive Guide to Drupal 7, Apress, 1047 s.

React, Tutorial: Intro To React. Tillgänglig: https://facebook.github.io/react/tutorial/tutorial.html. Hämtad: 11.12.2016

Riley, Tim & Goucher Adam. 2009, Beautiful Testing, O'Reilly Media, 329 s.