Masaryk University
Faculty of Informatics

A Mobile Application for the Administration of the Kentico System

Bachelor's Thesis

Linda Hansliková

Brno, Spring 2017

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Linda Hansliková

Advisor: Bruno Rossi, Ph.D

Acknowledgement

My thanks go to Bruno Rossi and Marek Fešar my advisors for allowing me to proceed with this topic and their advice and patience. And also to my friends Matej Pavla, Jozef Vilkolák and Roman Mačor for their support and their help with proofreading this text.

Abstract

In a time where time is more precious than money it is crucial for people to accomplish a task as quick as possible. When creating various web-sites, the Kentico Enterprise Marketing Solution (KEMS) is a helpful tool to save time and therefore money. It is a content management system (CMS) which allows clients to create and manage their web-sites using a single user interface (UI). This thesis is about adding an extension to the said system which allows administrators to manage their site from their smartphones. The functionality implemented should reflect the basic needs of an administrator of the KEMS. The extension consists of two parts: the custom web application programming interface (API) and the mobile application (app). The custom web API (CAPI) was leveraged to call the Kentico API (KAPI) and retrieve data and the mobile app was used as a gateway for the user and the CAPI.

Keywords

Mobile, Mobile Application, Kentico, Javascript, JQuery, WebAPI, Apache Cordova

Contents

List of Figures

1 Introduction

KEMS is a content management system (CMS) which allows clients to form and manage their web-sites using a single user interface (UI). In this thesis we created a mobile app called KenticoApp which calls an API that we also developed by extending the API of KEMS. An API is a collection of functionality which a programmer is able to utilise in a third party app. The KenticoApp makes it possible for clients to manage their site from their smartphones. It consists of two parts: the CAPI backend, which stores and retrieves data from and to the database, and the mobile client app, which allows the user to communicate with the system. The functionality is divided into three main categories. The first category represents the system tasks such as restarting the server, cleaning unused memory or cache and reading the event-log or general system information. The second one operates with the users and their roles. It offers the editing of the user's first and last name and adding or removing their roles. The third and last category makes it possible to create or delete roles and edit them by adding or removing permissions. To be able to perform all of the above actions the user has to be authenticated and authorized first. The authentication credentials are checked against the KEMS database using KAPI. Only global administrators are authorized.

The backend was implemented in C# .NET and communicates with the KAPI. The mobile client app is a Cordova app written in JavaScript (JS), HyperText Markup Language (HTML) and Cascading Style Sheets (CSS). The communication is ensured by asynchronous JS and Extensible Markup Language (Ajax) in the format JS Object Notation (JSON). For the purpose of version control and backup we used Git. Our Git project was hosted on the web-based Git repository hosting service called GitHub. GitHub is an industry standard for hosting open-source software source code.

Chapter one introduces KEMS, web API and hybrid mobile applications. In the second chapter we describe the application architecture and the implementation of the extension of the KEMS in more detail. Finally, we valorise the achieved result and suggest other potential extensions or solutions.

2 Analysis

The purpose of this thesis was to create a mobile app for managing key features of the Kentico CMS system. We created our CAPI using the KAPI which is implemented in the .NET framework. This is why the CAPI is created in it. For faster development we decided to utilise the active server pages (ASP) .NET's library Web API 1.0. The implementing of the CAPI architecture was influenced by the representational state transfer (REST) architecture. The goal of the mobile app itself was to support as many platforms as possible. It would not have to implement difficult graphics. We concluded the Apache Cordova framework (ACF) to be the most appropriate one. As for the functionality of the mobile app the JS library JS Query (JQuery) was used and the for asynchrony Ajax was utilised. To achieve better results in the presentation aspect another JS library called JQuery Mobile(JQM) was used. The communication between the CAPI and the KenticoApp in ensured by the JSON format.

In this chapter we will describe Kentico CMS which is the system extended by this thesis. Next are web APIs and their architectures, in particular REST architecture will be introduced. Later we will talk about mobile apps, the ACF and the languages used for the development in this framework, which are CSS, HTML and JS. Lastly Ajax and JSON will be detailed.

2.1 Kentico CMS

Kentico CMS [5] is a content management system (CMS) which allows clients to create and manage their web-sites using a single user interface (UI) which is made of tiles, a layout and an edit button as can be seen in the image 2.1. Each tile has its own functionality and the functionality of the tiles in the red circle is implemented in the KenticoApp. This image was created via print screen from the administration interface of the Kentico 9.0 product.

The client can rearrange them either by simply dragging them or by pressing the edit button. Pressing the button leads to the tiles having an *X* in the upper-right corner for removing the tile. If place on the dashboard is available, a blank rectangle with a plus in the position of the future tile enables the client to add a new tile from the menu.

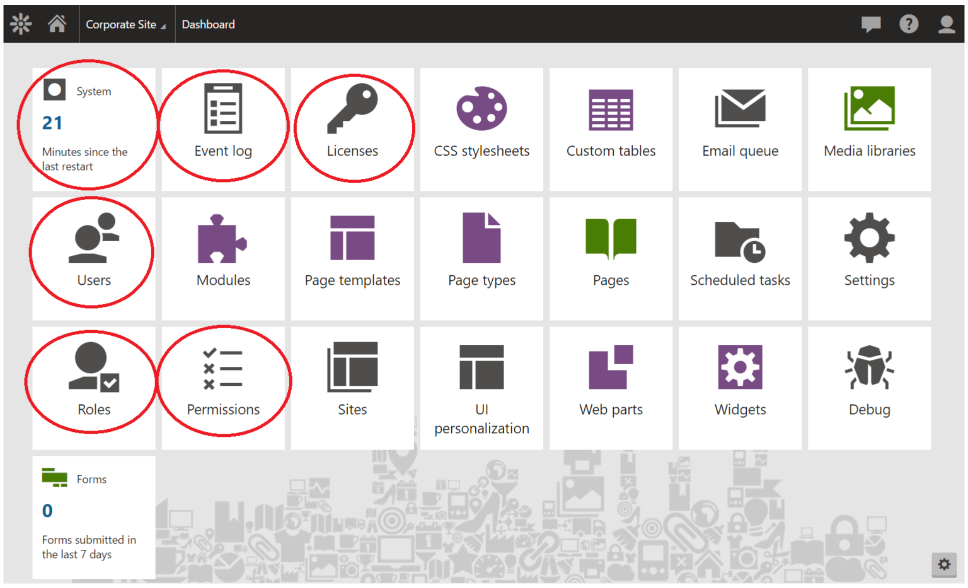

Figure 2.1: Kentico 9.0 UI. Modified for illustrational purposes.

The functionality in the menu is divided into six categories, namely: Content Management, On-line Marketing, E-Commerce, Social & Community, Development and Configuration.

Content Management sees to the contents of the client's site such as pages, tables, polls, etc.

On-line Marketing enables the client to handle marketing elements. Visitor's behaviour and reactions are taken into consideration. The tiles to be chosen are Email marketing, MVT Tests, Personas and others.

E-commerce offers actions which lead to motivating the visitor's behaviour to resemble the client's wished one, managing products and to track sales. These action are for example Buy X get Y discounts, Products and Store reports.

Social & Community makes it possible for the client to maintain the community around the site and its communication. Some of these tiles are for instance Avatars, Chat, Events.

Development 's task is to empower the client to administer sources of functionality and programmable elements. This section consists of tiles such as CSS stylesheets, Email templates, Web Part Containers, etc.

Configuration The last, in this thesis most important category. This category mostly oversees the overall configuration of the Kentico server. It contains the key requirements of KenticoApp.

- **System** One of those requirements is the System tile. Part of the System are several subcategories. The one of interest, however, is the one called General. It shows general information about the system and system time, the database and statistics of memory, garbage collection, cache and page view. The default value of the refresh interval is 1 second. It can be changed to up to 60 seconds. Other services General provides are Restart application, Clear cache, Clear performance counters and Clear unused memory.
- **Eventlog** is another key feature. It offers a dropdown list of available sites, a list of events, a filter to view specific events and a button to clear the log.
- **Licenses** the purpose of tile is to show and add licenses of the client and their details. It also allows the client to Export list of domains.
- **Users** grants the ability to view, add and edit the users, monitor the on-line ones and send mass emails. A filter tool is at service for searching users.
- **Roles** Users are assigned with roles and this is where these roles are administered. The overview displays all of the site's roles end their details. The client is able to add, edit and delete roles. A dropdown with sites to be chosen is present.

Permissions Roles authorize users to execute certain actions. Permissions define what these actions are. They are managed in this tile. Again, filter options and a dropdown with site names are available.

More functionality can be added to the Kentico application by using already created modules or by developing new ones from scratch since it is an extensible system.

2.2 Web API

An API is a collection of functionality which a programmer is able to utilise in a third party app. A web API is an API intended for utilisation by a web server or browser. APIs can be implemented following certain architectures, e.g. REST.

2.2.1 REST architecture

[9] REST is an architecture of client-server communication. We decided to be inspired by this architecture since most modern APIs are supported by it. It offers constraints and conventions and the programmer has to decide whether they will be followed or not.

[2] Hypermedia as the engine of application state (HATEOAS) is one of those restrictions. It demands the API is completely hypermedia driven which means a change of state is achieved using hyperlinks. After sending a request, the returned file contains the wanted commodity but also information about how it can be handled. An example of a HATEOAS response using JSON format is below. JSON will be described later 2.4.

```
{
  "id":42, "links":[{
         "self":{"href":"/students/42"},
         "timetable":{"href":"/timetables/42"}}]
}
```

The response contains the student with the ID 42 and links to the student itself and to their timetable.

APIs built using REST leverage most commonly JSON format and are lightweight. This means they operate with simple data representation and therefore the delay between sending and delivering is comparatively small. The client needs no knowledge of how the server is built. It depends on the resources (nouns) and operations (verbs). Hyper text transfer protocols (Http) status codes (SC) can be returned after executing the verbs. The nouns are identified with Universal Resource Identifiers (URIs). Each URI represents only one noun. The system returns a SC depending on the success of the request. If it is unsuccessful the SC returned gives away the reason why the call failed and might carry additional information. Some of the most common usage of the most utilised SCs is described below[10].

200 OK SC means the request was successful and the data have been returned. SC *204 No Content* represents the same meaning but returns nothing.

201 Created is used when the request was successful and the resource was created. It should return a link to the resource created.

400 Bad Request is returned when the given parameters were invalid. A reason might be added in the error message. The call should be repeated with different parameters.

401 Unauthorized it transmits the information the user should try and sign in again. It is meant to be returned if the client is not signed in or does not have the needed permissions. However, it mostly is returned if the user is not authenticated.

403 Forbidden its purpose is to let the user know not to repeat the request. It is meant to be returned if the client is authorized and authenticated but the system refuses to execute the call but it is most often applied in case the client is not authorized to execute a specific call.

404 Not Found is shown if a resource is not found with the given URI and it is unknown how long this condition will remain. It can be used if the reason for the failure remain unknown or a resource has to be hidden. Also this is the SC which is applied if no other code is suitable.

503 Service Unavailable SC means the server is not able to fulfill the request at the moment. It is utilised when maintaining the server or overloading it.

The most applied verbs are described using the following keywords: *OPTIONS*, *GET*, *POST*, *PUT*, *PATCH* and *DELETE*. For the description of them we provided two example URIs:

1. `http://example.com/students/47`

2. `http://example.com/students`

OPTIONS offers information about what verbs can be invoked upon a noun. The first URI example with the prefix *OPTIONS* returns all of the bellow verbs.

GET is used to read data from the server. It is read only, data must not be changed. The SCs it returns are *200*, *400* and *404*. The first URI example with the prefix *GET*returns the student with the ID 47.

POST is mostly utilised to create data. It returns SCs such as *201*, *400* or *404*. The second URI with the prefix *POST* creates a new student and returns the resource's location.

PUT is most commonly leveraged for updating resources with the sent data. The data are a representation of the complete resource. It returns SCs *200*, *204* and *404*. The first URI edits the student with the id 47.

PATCH is mostly used for modifying resources. The difference between PATCH and PUT is the sent data can be described by only a part of the resource that needs to be changed. It returns the same SCs as *PUT*. The first URI with the prefix *PATCH* modifies the student with the ID 47.

DELETE deletes the resource. It returns the status codes *200* and *404*. The first example URI with the prefix *DELETE* deletes the student with the id 47.

Each verb can be shared among multiple nouns. However, each noun might not be able to use each verb. For example an AT usually is not updated, therefore the verb *PUT* is not used with the AT resource.

The platform is not relevant in the implementation when using API's with REST principles. The client and server can even be built on different platforms. This is the reason why it is widely used in web apps. The session between the client and server is stateless, which means the server does not store the client's state. Instead it generates an AT which is sent to the client. This is an advantage since the client is enabled to communicate with different servers or machines in one session. Another benefit is if the server has to be restarted no data of the client's state are lost. It also makes the system highly scalable. REST is an alternative to Simple Object Access Protocol (SOAP), which is more complicated. It has a steeper learning curve. When communicating with SOAP services the client app reads the web service definition language (WSDL) file which contains the description of the functionality supported by the service. REST relies on conventions, therefore this step is omitted. SOAP usually operates with Extensible Markup Language (XML). It has its own advantages in case of large enterprise systems as opposed to REST which is mostly used when building web apps, as already stated.

2.3 Mobile applications

The client app we are developing in this thesis is running on mobile devices. Mobile apps are more important every day since *no other technology has impacted us like the mobile phone. It's the fastest growing manmade phenomenon ever – from zero to 7.2 billion in three decades*[6]. The platforms, on which these apps run on, can be divided into three main categories: Android, iPhone OS (iOS) and the Windows family of operating systems for mobile devices (WM). The most used operating system (OS) with a share of 68.67% of all mobile devices as of November 2016, according to *netmarketshare.com* [8] is Android. It was released in September 2008 and its native language is Java. With a percentage of 25.71%, iOS is the second most sold mobile OS and was released in June 2007. Its native language is Swift. The third platform is WM with 1.75% share. It was first released in November 2010 and the native language is C#.

Mobile apps can be divided into three types: Native, HTML5 and Hybrid.

Native This type of app is written in the native language of the platform the developer wants to target. It is fast and the behaviour is most intuitive for its users because the developer focuses on one particular OS and its features. The drawback is that for the development of a native app the learning curve is steep which is why an experienced team of programmers is needed and it targets only one OS. If the app has to run on other platforms, it has to be built in their native languages. This is time costly and therefore expensive.

HTML5 These apps run in a devices browser. They are usually implemented in CSS, HTML and JS. Many programmers have the opportunity to leverage their previously acquired experience with these languages, alternatively the skills gained here in web programming. HTML5 based apps run slower than native ones but work on multiple OS. This saves time and money. The disadvantage of cross-platform apps is they are less intuitive. For example an app with android features would be uncommon for iOS users and vice versa. Another flaw of this approach is the inability to use the device's hardware, such as its camera or microphone.

Hybrid This is a combination of the two above. It is primarily built utilising CSS, HTML5 and JS and is *hosted inside a native application that utilizes a mobile platform's WebView.* [11] The bridge to native technologies is ensured by tools, for example in this thesis the ACF is used. A WebView is a headless browser, without any buttons and without higher level functionality such as tabs, navigation, etc. The perks of this approach is the small learning curve. Thanks to HTML5 it is cross-platform and the native wrapper allows it to use the device's hardware and native API. It still runs slower than Native but is cheaper when more OSs are targeted.

When creating an app the developer has to consider what approach will be the most effective one. If smooth graphical performance is crucial, the app should be native. If it has to run on multiple platforms, make use of a device's hardware component and native API and the graphics are not that important, probably hybrid would be the best fit. If the app displays what was sent to it and has no dynamic graphics then HTML5 should suffice.

The development of the KenticoApp was divided into two stages. For the implementation of the mobile app we leveraged the ACF, as already stated. The reason being it is less demanding to learn and supports seven platforms. For creating the UI we decided to use JQM. It is an HTML5-based UI framework which allows users to design aesthetically pleasing mobile elements by utilising the languages CSS and HTML. Document object model (DOM) elements are individual parts of a web page described by tags such as div, span, input or others. These tags assign styling and properties to elements. For the DOM manipulation we used the JQuery library which has a small learning curve and offers a fast way to add, modify, style and delete elements or change their behaviour. It also offers a set o handy helper functions which provide easy to use interface for frequently used operations in web development, e.g. *ajax()*.

2.3.1 Apache Cordova

ACF is a framework used for creating hybrid mobile apps that are compatible with seven platforms. As opposed to the Xamarin framework (XF) supporting only three. It allows the programmer to utilise the languages CSS, HTTP and JS. A native wrapper offers the simulation of a native app, meaning the app is able to operate native API through libraries. For example it is able to access the gyroscope information or open the native calendar. XF is used to build native apps in the language C# on WM, Android and iOS. When launched on Android or iOS its code is translated into Android Java or iOS Swift. Even though XF should be faster than ACF, and therefore offer a smoother user experience, the difference between execution times of non performance sensitive apps on today's devices is negligible. We did not consider development in native languages because of their steep learning curve and the ability to deploy only to one platform.

2.3.2 HTML, CSS, JS

These languages are the most common ones in front-end web programming. HTML is a markup language and is used to define the structure of a page. CSS is leveraged to add styles, such as colors, fonts or alignment. JS adds functionality to a page. Because there are nuances between different browsers and browser versions it can be time consuming to create a page using only these languages. Therefore a framework is typically used to simplify these tasks, providing its own set of styles, tags and functionalities which are tuned to work in different environments.

2.4 Ajax and JSON

Ajax offers the execution of asynchronous operations. It is able to update a web page without reloading the page. After a page has loaded it can request or receive data from a server and it also is able to send data to a server in the background [1]. This saves time when using an app because the user does not have to wait for a whole page to load again. Instead, only data of the call have to be awaited. The user can leverage the page before data which take longer loading time, e.g. images are loaded. It is used in browser to server communication.

JSON format is a simple *string*. Attributes are represented with their name, a colon following and the attribute's value. For example *"id":"47"*. If an object is described, the attributes are put into braces. A comma separating them: *{"id":"47", "type":"student"}*. If multiple objects are displayed in a *array* the objects are bordered with brackets and again separated by a comma. *[{"id":"47", "type":"student"}, {"id":"007", "type":"teacher"}]*. This format is not dependent on any platform. JS can manipulate with a JSON file without any conversion which is why it is widely used by front-end web developers.

3 Implementation

In this chapter we will describe the application overview. Next the extending of the Kentico system will be explained and its API will be introduced. Later we will talk about the library ASP.NET API 1.0 which was utilised in our CAPI and how we managed ATs. Lastly the KenticoApp will be detailed. The languages CSS, HTML and JS leveraged with JQM will be described along with the usage og AJAX.

3.1 Application Overview

This thesis consists of two parts. The first of which is the CAPI back-end. It stores and retrieves data from and to the database via calls to the KAPI. It itself is called by the second part - the mobile client app, called KenticoApp, through which the user is able to communicate with the system and manage his site.

The CAPI partially follows the REST architecture by using appropriate HTTP methods. For example we use POST requests for creating or GET requests for reading resources from the back-end. The usage of status codes, such as 200, 403 or 503 is also a RESTful convention. Our back-end is stateless. This is achieved by using ATs instead of storing the user session across multiple HTTP requests. One of the reasons why we cannot call this application RESTful is it does not follow the fundamental concept of identifying all resources and relationships between them. For example our *System* "resource" contains the method *ClearCache()* and *ShowEventlog()*. These should be identified in separate resources *CacheClearer* and *Eventlog*. Further description of the CAPI can be found below.

The KenticoApp is a mobile app which can be used by global admins. It offers a welcome page where the user has to sign in. If the authentication is successful and the user has the proper authorization, the user is redirected to a menu page with three buttons and their descriptions, each one representing a controller in the CAPI. From there on the user can choose what action to conduct. A layout with options such as view the current user or logout is available. It also contains the breadcrumbs to the current page.

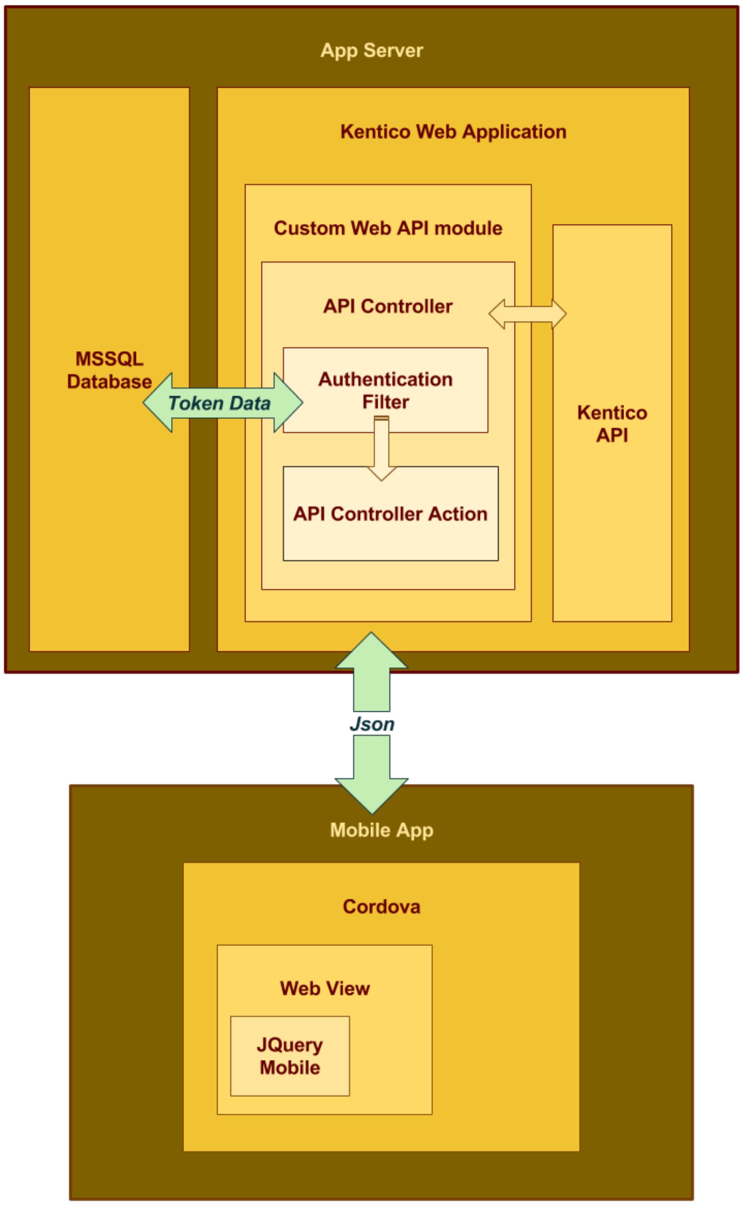

Figure 3.1: Architecture overview

The communication between the CAPI and KenticoApp is ensured by AJAX using JSON format. It is an effective way to broadcast information via a simple string.

3.2 Extending Kentico

To be able to add functionality to Kentico CMS a new project we called CAPI had to be created in the already existing solution which was installed with the installation of Kentico CMS. The project then was registered using a Kentico module ASP.NET API 1.0. This will be described in the subsections 3.2.1 and 3.3.1 The CAPI was created using the .NET framework. It uses KAPI calls and is called by the KenticoApp. For executing an API call, the user has to be signed into the system and have the proper authorization.

3.2.1 Custom Kentico Module

A file with the *[assembly: RegisterModule(typeof(CustomWebApiModule))]* annotation was added into CAPI. It must inherit from the *CMS.DataEngine.Module* class. We added an empty constructor and overrode the *CustomWebApiModule.OnInit()* method. We will discuss the content of this method later 3.3.1.

3.2.2 Kentico 9.0 API

3.3 Web API Application

3.3.1 ASP.NET API 1.0

To allow a module to be registered to Kentico CMS the method *Configure(WebApiConfig.Register)* method of the class *System.Web.Http.GlobalConfiguration* must be called in the overridden *CustomWebApiModuleOnInit()* method. ASP.NET API 1.0 provides several components which make it easier for us to create an API for our application, one of which is the *System.Web.Http.ApiController*. All of our controllers inherit from it. One of its features is it enables the programmer to return the SC and possibly a JSON representation of the data we want to return. Another component is the *System.Web.Http.Filters.ActionFilterAttribute* which is inherited by the filter in the CAPI. It allows to implement the method *OnActionExecuting()*. This method is called before the execution of an API action. It is usually used when lading data from a database which are then used by big numbers of different actions. If these data are loaded into the *actionContext* they can be used by all those actions and time is saved. Also the execution of an action can be prohibited. The filter we use in CAPI checks if the user attempting to carry out a task provided a valid AT. It stores the user information if the AT is valid and if it is not it prevents the task to be performed. Additional benefits of ASP.NET Web API 1.0 include making it possible to use the annotation *[Route("myapi/students")]* which indicates the location no the web server from where the following method can be called. It also offers the usage of HTTP method tags such as *HttpPost* or *HttpGet*. These annotations specify the type of the operation, e.g. whether the server should create a new resource or just return data. Calling the action with a different method than specified is not possible. Another feature it provides is it parses a request and the programmer can then read it as a *[FromBody]JObject postData* attribute. This attribute represents the JSON data sent in the body of the request from a third party application.

Since source code on the front-end can be easily read and modified by unauthorized users, security measures should be implemented in the back-end app. To secure our system we use access tokens (ATs) and a filter. Filters are used to prevent unauthorized users from executing operations. They are noted through annotation, e.g. *[Authorizator]*, either in front of a particular method, or in front of a whole controller so that all its methods are affected. The filter we use is called *Authorizator* and, as already stated, it inherits from the *ActionFilterAttribute*. It was implemented as our custom authorization filter and checks if the user is authenticated and if he is a global admin since only global admins are permitted to use the KenticoApp. If not, the SC 403 is returned. CAPI uses controller classes to divide functionality into section for better overview and security. Each controller contains methods mainly affecting resources represented by it. In this thesis four controllers were implemented: *AuthenticationController*, *AuthorizationController*, *UserController* and *SystemController*. Each one of these controllers symbolizes a group of related functionality. For example the *AuthorizationController* contains methods for managing roles and permissions. The API call structure is demonstrated in the illustrated code below. This specific call edits a user.

```
[Authorize]
[HttpPost]
[Route("kenticoapi/users/edit-user")]
public HttpResponseMessage EditUser([FromBody]JObject postData)
{
  string username, firstName, surname;
  try
  {
    username = postData["username"].ToObject<string>();
    firstName = postData["firstName"].ToObject<string>();
    surname = postData["surname"].ToObject<string>();
  }
  catch (Exception e)
  {
    return Request.CreateResponse(HttpStatusCode.ServiceUnavailable, new { errorMessage = e.Message });
  }
  try
  {
    UserInfo updateUser = UserInfoProvider.GetUserInfo(username);
    if (updateUser != null)
    {
      updateUser.FirstName = firstName;
      updateUser.LastName = surname;
      UserInfoProvider.SetUserInfo(updateUser);
    return Request.CreateResponse(HttpStatusCode.OK, new { user = updateUser });
    }
  } catch(Exception e)
  {
    return Request.CreateResponse(HttpStatusCode.ServiceUnavailable, new { errorMessage = e.Message });
  }
  return Request.CreateResponse(HttpStatusCode.ServiceUnavailable, new { errorMessage = "User is null" });
}
```

The annotation from the 1st line is our custom *AuthenticationFilter* and checks if the user is authenticated so the call can be executed. If successfully authorized, the user is stored into the request properties from where he can be retrieved with the following command:

```
UserInfo user = (UserInfo) Request.Properties["
   LoggedUserInfo"]
```

as it is done in the method *GetCurrentUser()*. Line 2 ensures that only POST requests are handled by the method. POST requests send data from the client to the server as opposed to GET requests which demand data from the server. In this example the system stores updated user information from the KenticoApp into the database. The 3rd line represents the route where the call can be accessed through the client app. The 4th line is the head of the method. Its return type enables the client to receive a *StatusCode* and a value, which is the content of the HTTP response message. The parameters are passed on from the client as one object in the JSON format. On the lines 6 to 12 the JSON object is parsed into separate parameters as *strings*. This is done in a *try-catch* block to handle possible exceptions and return the proper response message on the line 15. The *CreateResponse()* method is of the class *Request* and its parameters are the status code *503* and an object with the error message of the caught exception from line 13. The line 19 gets the user using the parsed *username* and stores it in the variable called *updateUser* of the type *UserInfo*. This type is defined in the KAPI documentation and has attributes such as *username*, *user ID*, *user first* and *last name*, etc. Line 20 checks if the *updateUser* is not *null*. Lines 22 and 23 change the *updateUser*'s first and last name. On the line 24 the *updateUser* is inserted in the database. Line 25 returns the status code *200* and the *updateUser* object, which is later converted into JSON format. The lines 27 to 30 are similar to lines 13 to 16. If *updateUser* is *null* the response status code is *503*, the same as on line 15, and the error message *"User is null"*.

3.3.2 CAPI Token Management

For user authentication we decided to use ATs. ATs are leveraged to secure the communication between a user and the system. After signing in the user is given a random generated unique AT by the system which stores it in its database. Before every API call, the system requires the user's AT and then checks it against the database. For the call to be executed the AT has to exist in the database with the corresponding user ID and must not be expired. If this is not the case the user is redirected to the welcome page, where he has to sign in. To represent and store the ATs in the database in our project we were inspired by the layered application design pattern, more specifically by its data access layer (DAL). This pattern is used to ensure security and scalability of an application by partitioning it into three layers. The first and lowest layer is needed to operate the database. It is called DAL and contains entities which are depictions of objects. The next layer is the business logic layer which contains the logic of the system. And the last one is the presentation layer, utilised to display the application through a UI to users. For the purpose of this thesis we decided to represent the ATs as an entity using the Entity Framework. The entity contains the user identification (ID), a unique pseudo-random code and an expiration date and time (expiration) as can be seen in the following example code.

```
public class Token
   {
       [Required]
       public int UserID { get; set; }
       [Required][Key]
       public string Code { get; set; }
       [Required]
       public DateTime Expiration { get; set; }
   }
```

The ID is of the type *int* and is equal to the user's ID who "owns" the AT. The code is type *string* and is generated with the pseudo-random number generator *Random. The chosen numbers are not completely random because a mathematical algorithm is used to select them, but they are sufficiently random for practical purposes.*[7] Right after generating the code is tested against the database if no AT with the same one exists. If the code is already taken, another one is generated and tested. If not, the token entity is assigned the code, user ID and date and time 10 minutes from the assignment. The expiration is of the type *DateTime*. After every executed API call the AT's expiration is set to 10 minutes from calling. Before every API call the system searches its database for expired ATs and deletes them.

3.4 Cordova Mobile Application

To enable a user to use the CAPI a front-end mobile app had to be created. We already specified that for this purpose ACF with JQM was used.

3.4.1 HTML, CSS, JS

As mentioned earlier, due to browser differences it can be time consuming to use these languages and the soulution can be to use a framework such as JQM. It is an open source HTML5-based UI framework and it allows users to design aesthetically pleasing mobile elements by utilising the languages CSS and HTML. It also assigns properties to these elements.

```
<div role="main" id="welcomePageBody" class="ui-content">
  <span data-position-to="window">
    <img src="Kentico_logo.png" id="kentico_logo" alt="
        kentico_logo" style="width:280px">
  </span>
  <span class="push-down">
    <label for="usrname-input">Username:</label>
    <input id="usrname-input" placeholder="Username"
        required />
    <label for="passwrd-input">Password:</label>
    <input id="passwrd-input" input type="password"
        required />
    <button id="login-btn" data-role="button" class='ui-
        btn ui-btn-inline ui-corner-all ui-shadow pull-
        right'>Log in</button>
  </span>
  <div id="error-msg" class="not-displayed"></div>
</div>
```

The sample code above creates a tag with an image, two text fields, their labels and a button. The visualisation is shown in figure 3.2. The content in the red quadrilateral in the image is the graphical representation of the code sample above. It was created as a print screen of the KenticoApp. On the first line a *div* element contains a part of the page. *Div* tags divide the page into smaller fractions to diverse the styles, alignments or to create groups for selection. The *role="main"* attribute of the JQM library binds predefined behaviour to the con-

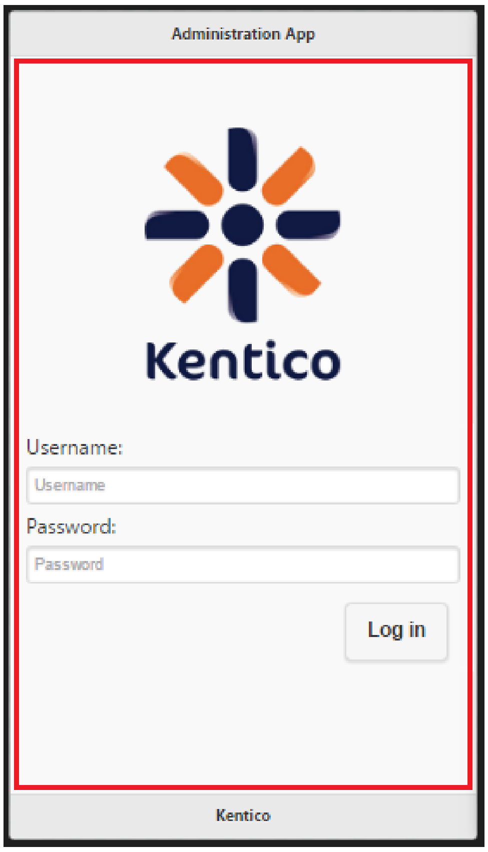

Figure 3.2: KenticoApp Welcome Page UI.

tent of this *div*, for example it will be placed between the header and footer, it defines the paddings[1] and more. Up next are *id* and *class* HTML attributes. They are used for identifying elements for further use. Two equal IDs cannot exist in one HTML document. Classes can be assigned to multiple elements. Both classes and IDs are selectors and are leveraged for CSS formatting or for adding behaviour in JS. In this case, the value of the class is *ui-content* which adds styles to all the elements int this *div*. Tags with the keyword *ui-* are JQM tags and are styled so the user utilising a mobile app feels comfortable using it. On line 2 a *span* element is present. It is used for assigning IDs or classes inside other parent elements, for example if not all children of the parents have the same classes. The *data-position-to* keyword establishes the position of the element. The value *window* makes the position to be the center of the page. The *img* tag on line 3 represents an image element. The value of *src* represents the relative path to the physical location of the image file on the web server. The content of *alt* is featured if the image was not. *Style="width:280px"* defines the width of the displayed image, which will be 280 px. The tag on line 4 ends this *span* element. Line 5 holds another *span* element with a styling class demonstrated below:

```
.push-down { padding-bottom:50px }
```

This class sets the value of the distance between this and the next element to 50 pixels(px). The keyword *label* on line 6 ensures the element will behave as a label. Labels are used to describe other elements, e.g. inputs. The *for* keyword determines the described element and should match the *id* in that element. The text between the *label* keywords will be displayed in the UI. The *input* tag on line 7 represents a text field element. Text fields are used to gather inputs from users. The *id* keyword is a selector. A *placeholder* is text in the text field. It disappears after the cursor is placed into the field and is not considered as input. Because of the *required* HTML5 attribute the field is compulsory and must be filled out. Line 8 is similar to line 6. Line 9 is similar to line 7 with an extra attribute *type*. Its default value is *text*, the user will see what is typing in the field and the input is not restricted. Other types are for example *email*, *number* or *password*. For

1. Padding determines the distance between the border of an element and display of a device

the *passwrd-input* we used the *password type*. The characters the user is typing into this field cannot be seen, they are displayed as dots.

3.4.2 AJAX

AJAX calls are used in the KenticoApp to communicate with the API back-end.

```
1  function editUserUsersApiCall(username, firstName,
      surname, success_callback) {
2  showCustomLoadingMessage();
3  $.ajax({
4    url:"http://localhost:8080/kenticoapi/users/edit-user",
5    type: 'POST',
6    data: {
7      username: username,
8      firstName: firstName,
9      surname: surname,
10   },
11   success: function (data) {
12     if (success_callback) success_callback(data);
13   },
14   error: function (jqXHR, textStatus, errorThrown) {
15     showAjaxError(jqXHR);
16   },
17   complete: function () {
18     hideCustomLoadingMessage();
19   }
20 });
```

An illustration of the structure of an AJAX call can be seen above. The first line represents the header of the function which represents the call. The keyword *function* marks the following code to be a method, the name and its parameters follow. Not all parameters have to be given when calling the function. In this example the *success_callback* can be omitted. If the omitted parameter is the last one, it simply can be left out and the value *undefined* will automatically be assigned. But if it is not on the last position and any parameter following it will be defined, the *null* type must represent it. In JS, the types of variables are defined after assigning them with values. Therefore their names are important to be chosen carefully. There are different conventions to be found, one widely used was published by Google [3]. Line 2 ensures the user cannot interact with the app while executing an API call. To ensure this, a dark shadow overlay is added above the UI and a gray loader with a clockwise rotating line will appear. On line 3 the *$* sign stands for *jQuery*. It's an alias used to make the code more readable. *ajax()* is a method of the *jQuery* library. It has several attributes predefined allowing the programmer to fully control the asynchronous method. On line 4 the attribute's *url* value represents the address where the API method can be found. Line 5 defines the HTTP method which will be executed. Line 6 - 10 holds the data which are assigned with the parameters from line 1 and are then sent as JSON format to the server. The attribute on line 11 defines what will happen if the call has been successful, the so called *callback* function. In this case it's a *function* whose first parameter will be filled with the response *data* from the server. Its body is implemented on line 12. If the *success_callback* from the header present it will be called with the *data*. Line 14 carries the *error* attribute which describes the callback for an unsuccessful execution. Again a function is invoked. One of its attributes is *jqXHR*, which stands for jQuery XMLHttpRequest. It is a *string describing the type of error that occurred and an optional exception object, if one occurred. Possible values for the second argument (besides null) are "timeout", "error", "abort", and "parsererror". When an HTTP error occurs, errorThrown receives the textual portion of the HTTP status, such as "Not Found" or "Internal Server Error"* [4]. The function on line 15 was implemented by us and sets the text according to the information from the *jqXHR* object in an error popup. The attribute on line 17 defines what will happen disregarding if the request has been successful or has failed. The body of the following *function* is on line 18. It enables the user to interact with the page once again.

4 Conclusion

4.1 Evaluation

We created a mobile app with a CAPI.

4.2 Evaluation

TODO: Functionality

4.3 Future Work

TODO: Ability to choose between available sites on Kentico server, Access control, Security Token, Forgotten Password, polished UI

Bibliography

[1] Ajax introduction. http://www.w3schools.com/xml/ajax_intro.asp. [cit. 2016-12-22].

[2] Hateoas. https://spring.io/understanding/HATEOAS.

[3] Javascript conventions. https://google.github.io/styleguide/jsguide.html. [cit. 2016-12-22].

[4] jquery.ajax documentation. http://api.jquery.com/jquery.ajax/.

[5] Kentico cms. http://www.kentico.com/product/overview. [cit. 2016-12-18].

[6] More gadgets than people. https://www.cnet.com/news/there-are-now-more-gadgets-on-earth-than-people/. [cit. 2016-12-13].

[7] Msdn documentation - random. https://msdn.microsoft.com/en-us/library/system.random%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396. [cit. 2016-12-09].

[8] Operating systen market share. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=1. [cit. 2016-12-13].

[9] Rest. http://rest.elkstein.org/. [cit. 2016-12-16].

[10] Rfc 2616. http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html/. [cit. 2016-12-16].

[11] What is a hybrid mobile app. http://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/. [cit. 2016-12-14].