Masaryk University
Faculty of Informatics

A Mobile Application for the Administration of the Kentico System

Bachelor's Thesis

Linda Hansliková

Brno, Fall 2016

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 Brunno 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 safe 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 administrate 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 decided to use a technology called 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

1

2.1 Kentico CMS

Kentico CMS 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. Each tile has its own functionality. 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.

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.

---

1. Example footnote

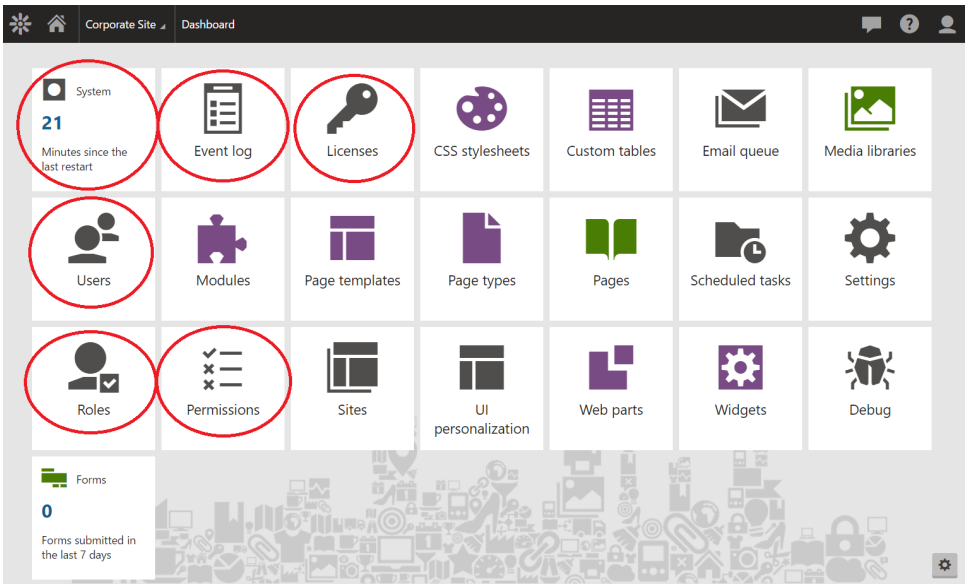

Figure 2.1: Kentico 9.0 UI. The functionality of the tiles in the red circle is implemented in the KenticoApp. This image was taken via print screen from the administration interface of the Kentico 9.0 product and was modified for illustrational purposes.

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. The next component to be described is Licenses. Its purpose 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 ready for use 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.

2.2 Web Application Interface

2.2.1 REST

[4] REST is an architecture of client-server communication. API calls built using this architecture utilise 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 call and if unsuccessful according to the reason why the call was unsuccessful. Some of the most common SCs are described below. [5]

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 given 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 fulfil the request at the moment. It is utilised when maintaining the server or overloading it.

Verbs are described using the following keywords: *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`

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 the student with the id 47. The illustration below modifies the student with the ID 47.

DELETE deletes the resource. It returns the status codes *200* and *404*. The first example URI 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 a session usually is not updated, therefore the verb *PUT* is not used with the session resource.

The server's and client's platform is not relevant. Which is the reason why REST 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 send 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 and, on initialisation, sends a file with the description of the architecture. It usually operates with Extensible Markup Language (XML). It is slower and heavier than REST.

2.3 Mobile applications

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.*[1] 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* [3] 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 objective C. 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 that 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. With these languages many programmers have the opportunity to leverage their previously acquired experience, 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.* [6] The bridge to native technologies is ensured by tools, for example in this thesis Apache Cordova 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. 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 the 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 the graphics are not that important, probably hybrid would be the best fit. If the app displays what was sent to it and the has no dynamic graphics then HTML5 should suffice.

3 Implementation

3.1 Application Overview

This thesis consists of two parts. The first of which is the CAPI backend. 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 representational state transfer (REST) architecture by using appropriate Http. For example we use POST requests for creating or GET requests for reading resources from the backend. The usage of status codes, such as 200, 403 or 503 is also a RESTful convention. Our backend 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 web 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 route to the current page.

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

3.2.1 Custom Kentico Module

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. TODO:

3.2.2 Kentico 9.0 API

3.3 Web API Application

3.3.1 Microsoft Web API 1.0 (MS API 1.0)

MS API 1.0 brings several features, one of which is the *System.Web.Http.ApiController*. All of our controllers inherit it. It enables 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 executing of a call and checks if the conditions described in the method *OnActionExecuting()* are met. Additional benefits of MS API 1.0 include making it possible to use the annotation *[Route("example.com/students")]* which indicates the location where the following method can be called. It also offers the usage of verb tags such as *HttpPost* or *HttpGet*. These annotations specify the type of the operation, e.g. whether the server should expect data. Another feature is it implements the *[FromBody]JObject postData* attribute. This attribute is sent from the web app and can contain one or more objects in the JSON format.

Since source code on the frontend can be easily read and modified by unauthorized users, security measures should be implemented in the backend 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 fil-

ter 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. Our API 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.

```
1  [Authorize]
2  [HttpPost]
3  [Route("kenticoapi/users/edit-user")]
4  public HttpResponseMessage EditUser([FromBody]JObject
       postData)
5  {
6    string username, firstName, surname;
7    try
8    {
9      username = postData["username"].ToObject<string>();
10     firstName = postData["firstName"].ToObject<string>();
11     surname = postData["surname"].ToObject<string>();
12   }
13   catch (Exception e)
14   {
15     return Request.CreateResponse(HttpStatusCode.
           ServiceUnavailable, new { errorMessage = e.Message
            });
16   }
17   try
18   {
19     UserInfo updateUser = UserInfoProvider.GetUserInfo(
           username);
20     if (updateUser != null)
21     {
22       updateUser.FirstName = firstName;
23       updateUser.LastName = surname;
24       UserInfoProvider.SetUserInfo(updateUser);
25     return Request.CreateResponse(HttpStatusCode.OK, new
           { user = updateUser });
26     }
```

```
27    } catch(Exception e)
28    {
29      return Request.CreateResponse(HttpStatusCode.
            ServiceUnavailable, new { errorMessage = e.Message
             });
30    }
31    return Request.CreateResponse(HttpStatusCode.
          ServiceUnavailable, new { errorMessage = "User␣is␣
          null" });
32  }
```

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 with 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 in JSON format. The lines 27 to 30 are similar to lines 13 to 16. If *updateUser* is *null* the response

is status code *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.

```
1  public class Token
2     {
3         [Required]
4         public int UserID { get; set; }
5         [Required][Key]
6         public string Code { get; set; }
7         [Required]
8         public DateTime Expiration { get; set; }
9     }
```

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.[2] 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

3.4.1 Apache Cordova

For the implementation of the mobile app we leveraged the Apache Cordova framework (ACF). The reason being it is less demanding to learn and supports seven platforms. As opposed to the Xamarin framework (XF) supporting only three. 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, such as Android Java or iOS SWIFT, because of their steep learning curve and the ability to deploy only to one platform. The development was divided into two stages. For creating the UI we decided to use JQuery Mobile. 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()*.

3.4.2 JQuery Mobile

Jquery Mobile is an open source HTML5-based UI framework. It allows users to design aesthetically pleasing mobile elements by utilising the languages CSS and HTML.

3.4.3 Ajax

TODO: ajax communication with web API

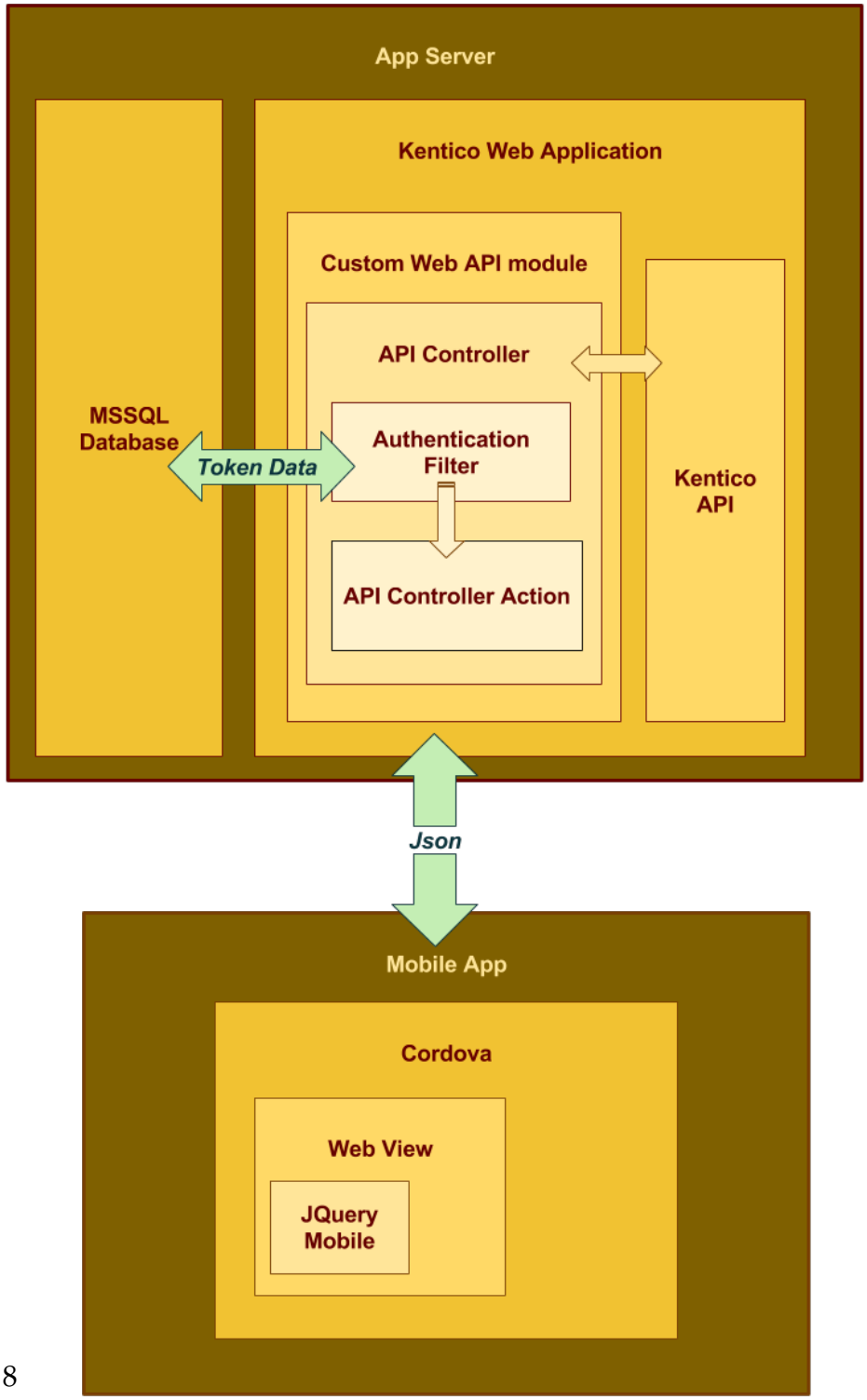

Figure 3.1: Architecture overview

4 Conclusion

4.1 Evaluation

TODO: Functionality

4.2 Future Work

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

Bibliography

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

[2] 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].

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

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

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

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