|  |  |  |
| --- | --- | --- |
|  | FACULTY OF GERMAN ENGINEERING EDUCATION AND INDUSTRIAL MANAGEMENT |  |

**Development of a system for the automation of processes in a modern gym**

Prepared by:

Martin Vladimirov Gebov

201314009

A thesis submitted to the Faculty of

**GERMAN ENGINEERING AND INDUSTRIAL MANAGEMENT**

In fulfillment of the requirements for the

MASTER’S DEGREE

in Computer Science

Technical University of Sofia

Sofia 1000

Bulgaria

March 2017

# Statement by the author

I hereby declare that this submission is my own work and to the best of my knowledge, it contains no material previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at any educational institution, except where due acknowledgement is made in the thesis.

|  |  |
| --- | --- |
| Martin Gebov | Date |
| Student |

|  |  |
| --- | --- |
| Doz.Dr.Ing.Adelina Aleksieva-Petrova | Date |
| Thesis Advisor |

|  |  |
| --- | --- |
| Prof. Dr. Ing. Stefan Stefanov | Date |
| Dean |

# Abstract

© Copyright 2014

by Martin Gebov

All rights reserved

# ACKNOWLEDGEMENTS

I would like to thank all of the people who helped me in the development of this thesis. Their gentle but firm direction has been most appreciated. Doz. Dr. Ing. Adelina Aleksieva-Petrova was particularly helpful in guiding me on how to conduct the needed research, how to gather the useful materials and knowledge to formulate my thesis and in the technical implementation of the project, in which she supplied me with the needed materials and source code samples to get me started on the project at hand.

# TABLE OF CONTENTS

[STATEMENT BY THE AUTHOR 2](#_Toc474078036)

[ABSTRACT 3](#_Toc474078037)

[ACKNOWLEDGEMENTS 5](#_Toc474078038)

[TABLE OF CONTENTS 6](#_Toc474078039)

[INTRODUCTION 9](#_Toc474078040)

[PRACTICAL APPLICATIONS 10](#_Toc474078041)

[CONTRIBUTIONS 10](#_Toc474078042)

[PREVIOIS WORK 10](#_Toc474078043)

[REQUIREMENTS 11](#_Toc474078044)

[OVERVIEW 11](#_Toc474078045)

[GENERAL 11](#_Toc474078046)

[SECURITY 11](#_Toc474078047)

[USER INTERFACE 12](#_Toc474078048)

[ADMINISTRATION 13](#_Toc474078049)

[ARCHITECTURE 16](#_Toc474078050)

[OVERVIEW 16](#_Toc474078051)

[CLIENT SIDE 16](#_Toc474078052)

[STATE 16](#_Toc474078053)

[SERVER SIDE 17](#_Toc474078054)

[REQUEST FLOW 18](#_Toc474078055)

[CRUD 18](#_Toc474078056)

[CREATING ITEMS 20](#_Toc474078057)

[UPDATING ITEMS 21](#_Toc474078058)

[DELETING ITEMS 21](#_Toc474078059)

[RETRIEVING ITEMS 22](#_Toc474078060)

[TECHNOLOGIES 24](#_Toc474078061)

[CLIENT SIDE 24](#_Toc474078062)

[JAVASCRIPT & AJAX 24](#_Toc474078063)

[SPA APPLICATIONS 24](#_Toc474078064)

[ANGULAR 2 26](#_Toc474078065)

[DATA BINDING 26](#_Toc474078066)

[ROUTING 26](#_Toc474078067)

[MODULARITY 27](#_Toc474078068)

[NPM 27](#_Toc474078069)

[TYPESCRIPT 27](#_Toc474078070)

[WEBPACK 28](#_Toc474078071)

[SERVER SIDE 28](#_Toc474078072)

[ASP.NET CORE 28](#_Toc474078073)

[ASP.NET WEB API 28](#_Toc474078074)

[EF CORE 29](#_Toc474078075)

[IMPLEMENTATION 30](#_Toc474078076)

[SECURITY 30](#_Toc474078077)

[USER MODEL 30](#_Toc474078078)

[AUTHENTICATION 31](#_Toc474078079)

[LOGIN 31](#_Toc474078080)

[LOGOUT 32](#_Toc474078081)

[REGISTRATION 33](#_Toc474078082)

[FUTURE WORK 33](#_Toc474078083)

[USER METADATA 34](#_Toc474078084)

[AUTHORIZATION 34](#_Toc474078085)

[CORS 34](#_Toc474078086)

[ADMINISTRATTION 36](#_Toc474078087)

[USER MANAGEMENT 36](#_Toc474078088)

[CARD MANAGEMENT 37](#_Toc474078089)

[REFERENCES 39](#_Toc474078090)

INTRODUCTION

TODO

## practical applications

The main advantage of this project is that it is entirely web-based and requires no installation on local computers. Users will be intuitively guided through the interface and thus not required to enter the data onto plain sheets of paper. Owners (Admin) tracks this information in a central place and not have to resort to constantly require reports from their Employees (Teachers). The system will is implemented with validation and proper error handling in mind. It as well exposes limited functionality based on the current role of the user. The users of the application may be all kinds of studios, who wish to digitize their business.

## Contributions

The above mentioned web application has been successfully build by leveraging on the capabilities of Angular2 and Asp.net Core and was later successfully tested in a live scenario of a yoga studio, where the users were able to track their visits online.

## Previois work

TODO

requirements

## overview

Build a fast and responsive web application that takes care of automating all of the work required to be done by hand by the participants in the gym. The requirements are described via the MoSCoW method.

### GENERAL

[MUST HAVE]

The system must be hosted on a dedicated server and must have a 99% uptime.

[SHOULD HAVE]

The web interface must be able to be easily converted to a mobile app and the code must be at least 90% reused in a mobile app context.

### security

[MUST HAVE]

The system must be protected via an authentication mechanism that restricts the access to maleficent e ntities such as bots and hackers or entities that do not have an account.

#### Login

[MUST HAVE]

A login form must be present to allow the users to log in to the system. The login must be performed by providing an email and a password. The login form must allow only users that are authenticated inside the system. Every unsuccessful login attempt must not hint the user on whether he has mistaken his username or password, but provide a general error message.

[SHOULD HAVE]

Optionally the login form may include a Facebook or Google login to ease the registration of the users and allow for a SSO experience.

#### Registration

[MUST HAVE]

A registration form that allows users entering their credentials to register with the site.

* The registration form must comply with the specifications for the password length and the presence of special characters.
* The system must detect duplicate emails and not allow two users with the same email to register.
* Passwords must be stored in a one-way hash pattern that cannot be reversed engineered
* Upon a valid registration the user must be automatically redirected to the login screen.

[SHOULD HAVE]

The system may additionally require users to verify their account by sending them an email with a confirmation link. This link activates the account and logs the users in the system.

#### roles

[MUST HAVE]

The system must have a minimum of three user roles to distinguish the different personas that will be using the site – Administrator, Instructor and Student.

* There must be always one administrator present in the system and the system must never be left without users.

### USER INTERFACE

[MUST HAVE]

The user interface components must be coded such that they can be easily reused across the pages of the web application.

There must be a general grid that lists specific sets of data that allows to be configured by metadata.

* The grid must allow to be configured via metadata to allow maximum code reuse.
* The grid should support filtering and sorting of the data [SHOULD HAVE]
* The grid should have a “load more” functionality to allow to scroll to all of the sets of data

There must be a general create/edit form that allows to be configured only by metadata.

There must be a navigation that allows for the users to navigate through the site

* The navigation should hide pages that the user does not have rights for

### administration

#### users

[MUST HAVE]

Administration must be visible to users that are part of the Admin role.

The system must have an administration page where the users are managed.

* There must be a grid or a list that displays all of the users in the system.
* The grid must display the user and his roles.
* The grid must not display the users` password hash.
* The current user must not be listed in that grid, or if listed he must not be able to delete himself
* The grid must allow the editing and deleting of users

[SHOULD HAVE]

The grid should support searching of a user by email to ease the finding of a specific user that needs to be edited or deleted.

Admin user should be able to create a user manually (without email confirmation)

#### card management

#### Types of cards

[MUST HAVE]

The system must provide the administrator to define new card types targeting different types of personas – adult, child or family.

Admin must be able to define his own types of cards and be able to declare the following properties of a card.

* Title, that will be used to identify the card
* Number of visits
* Validity of the card (in months)
* Price of the card

[SHOULD HAVE]

Admin should be able to archive cards that are no longer used.

Admin should not be able to delete card types that have issued cards

#### cards

[MUST HAVE]

Admins and Teachers must be able to issue cards to student users based on a selected type of card

* Admins must be able to change the validity of cards
* Admins can override the price of the card [SHOULD HAVE]
* Admins can override the validity of cards [SHOULD HAVE]
* Admins can override the amount of visits for a card.
* Cards can be marked as not-paid and be paid on a later stage in time [SHOULD HAVE]
* Admins and Teachers must be able to issue one day cards.

Admins and Teachers must be able to enter visits for a student user – this way using the amount of visits in a card.

Student users must be able to view their visits.

#### Money management

[MUST HAVE]

Admins are able to track the daily income of sold items.

* Admins can query the system and view all of the income for a specific period of time.

#### Workout management

[SHOULD HAVE]

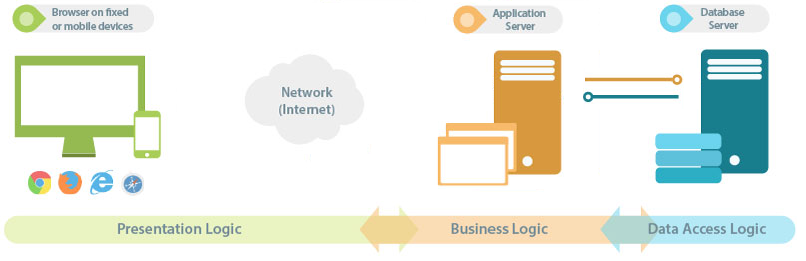
Admins must be able to define a schedule with workouts and when they occur.

Student users can reserve a spot for a specific workout.

Architecture

## overview

In order to fulfil the above requirements and provide a fast performing application, the end system is separated into two separated smaller apps – client-side app and server-side app. This makes the system loosely coupled and being able to easily scale if needed. Following is a diagram showing the general architecture of the system.

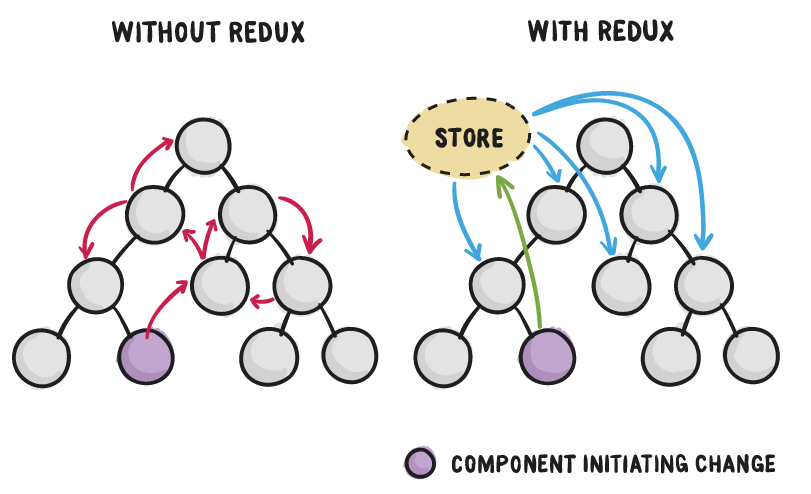


## Client side

The client is built with JavaScript and is hosted on browsers supporting HTML5 on desktop, mobile and tablet devices. The client is responsible for interacting with the user and interpreting user input that will be forwarded to the server. It communicates with the server using AJAX requests that are executed using the HTTP protocol with data passed in JSON format. The client is built with the Angular2 framework and leverages the concepts of the framework. Such concepts include Components, Modules, Services and Routing.

### State

State managements is taken into account in the application using the ngrx library. The library leverages on the concept of the Redux architecture and provides a centralized place to store the current application state (data). The biggest advantage here is that every part of the application that wants to subscribe to the changes of data can subscribe to a single source. This way whenever a Component or Service updates the central state every Component that is subscribed to it, will receive the notification and have the ability to update its view.

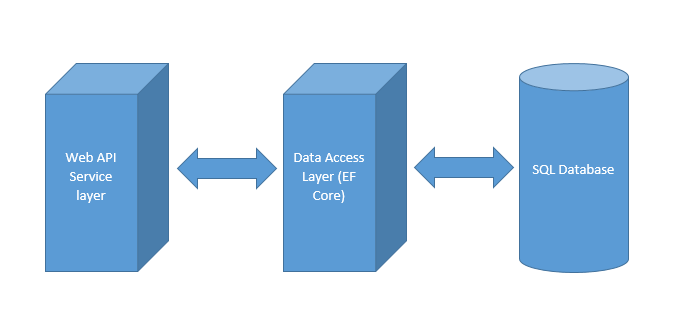


## Server side

The server app is separated into three layers – Service layer, Data Access Layer and the Database itself. The first layer is responsible for intercepting all of the http calls. The flow for processing the http calls is as follows:

1. Authorize or authenticate the request (if endpoint requires authentication)
2. Deserialize the JSON data (if there is a payload present)
3. Validate the deserialized data according to the model. E.g. check validations for each field – min value, max value, regular expressions. (if there is a payload present)
4. Forward the request to the appropriate handler (Web API Controller) that will execute the business logic.

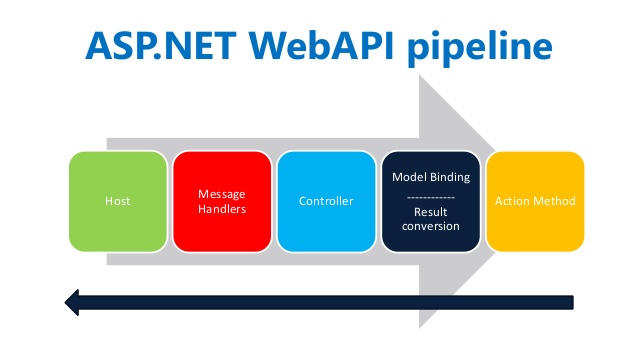
The data access layer (DAL) is responsible for making the connection to the database. The ORM is responsible for generating the SQL from the provided user model and LINQ statements. Once the SQL is constructed and depending on the request – Insert, Update, Delete or Get, different SQL is generated and forwarded to the database. Once dispatched the call awaits a callback from the database to signal if and how the request completed. Depending on the response from the database the server Web API service layer is responsible for returning a proper HTTP status code to the client.



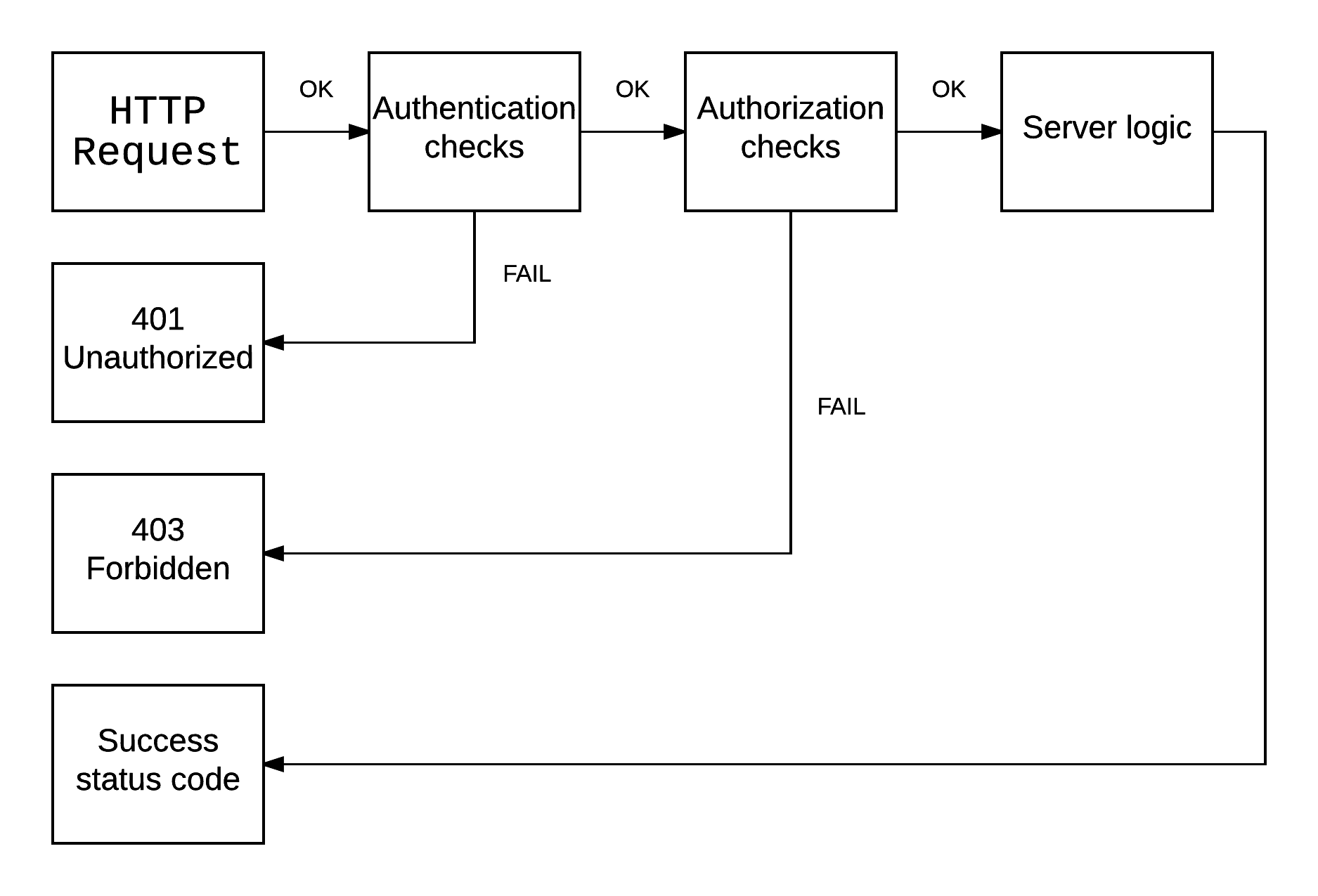
## Request flow

## CRUD

Implementing the REST protocol for each entity requires a server side framework – Web API (Asp.Net Core). The framework handles the deserialization of the request, the model validation and the returning proper errors status codes. It requires that each separate entity has its own Controller that handles all of the CRUD operations related to it. Some of this logic is specific for each type, but the general implementation for retrieving, deleting, updating and creating is the same. Thus a general CrudController class can be extracted that handles the boilerplate logic. Each Controller will have the opportunity to override the 5 methods that the base class has and place its own logic in them.



Some endpoints are protected and do not allow the retrieval of items. When the request reaches the server, the cookie is deserialized and the user object is set into the current HTTP context. If there is no cookie present or it has expired the user receives a status code of 401. If the user is valid but is not allowed the requested operation, the status code of the request will be 403.



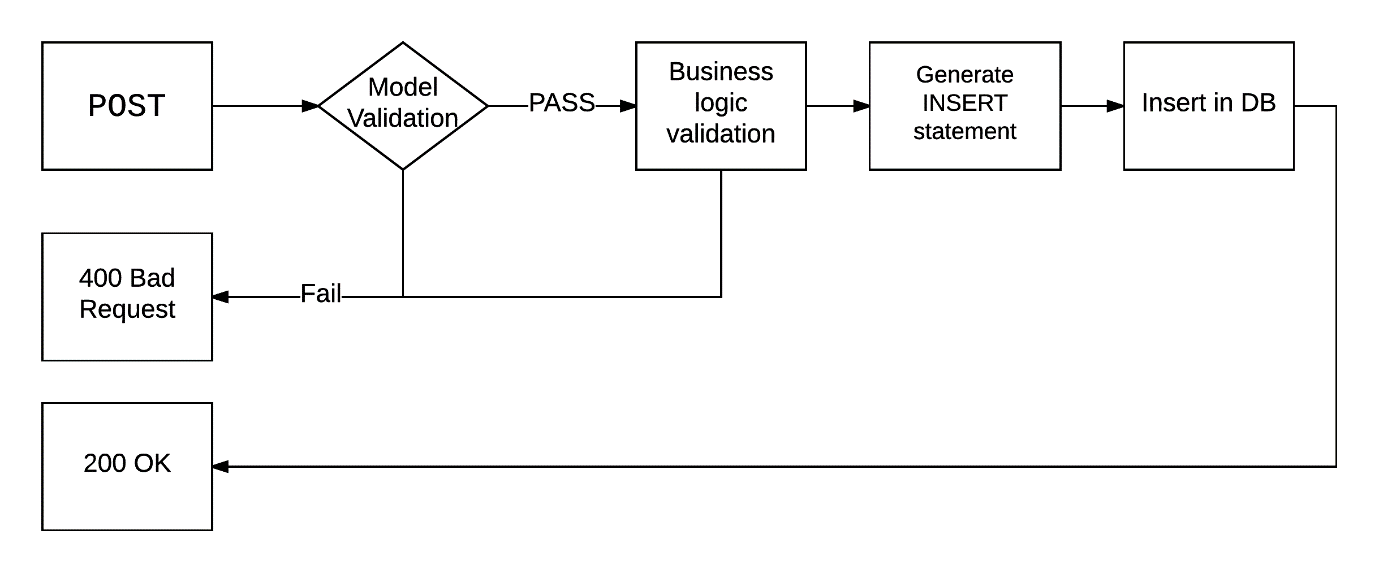
## Creating items

In order to create an entity the client must submit an HTTP POST request that contains the data to be inserted into the database. The data is submitted in JSON format. When the request body is deserialized on the server, the data goes through validation to see if the supplied entity data complies with database restrictions or business logic restrictions. This step is called model validation. If the model turns out invalid the server returns a status code of 400 with an error message explaining the reason why the server call failed. The message is again in JSON format and the client is responsible for parsing the response message and displaying it to the user.

If the validation passes, specific business logic is executed in each controller- e.g. validating the duplicate email of a newly created user. After the business logic passes and everything is validated, the entity will be inserted into the database.

Following the HTTP Protocol, after a successful insert into the database the server returns status code 200 and a Location header containing the unique URL of the entity. Using this URL each client is able to make a GET request and retrieve the full item data in a separate request.

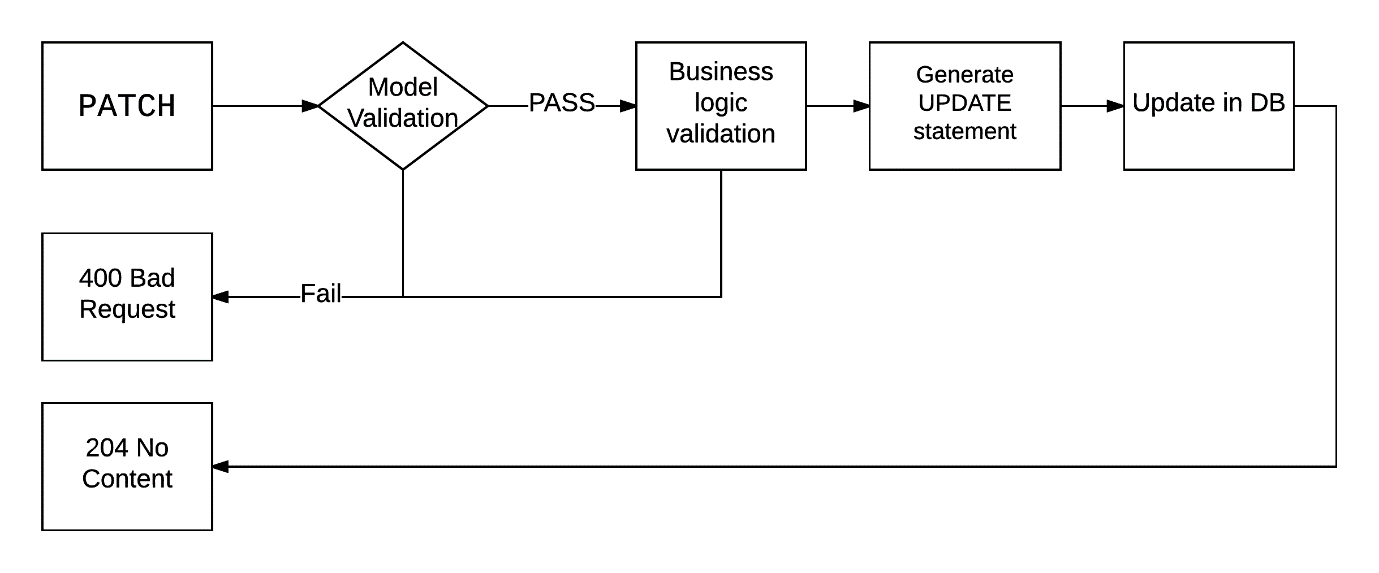
Aside from the location header the POST request returns the item data as well. The returned data might be different from the one passed to the server, because of server calculated properties. Such properties are e.g. the DateCreated property that can only be set on the server, because it is calculated in UTC.



## updating items

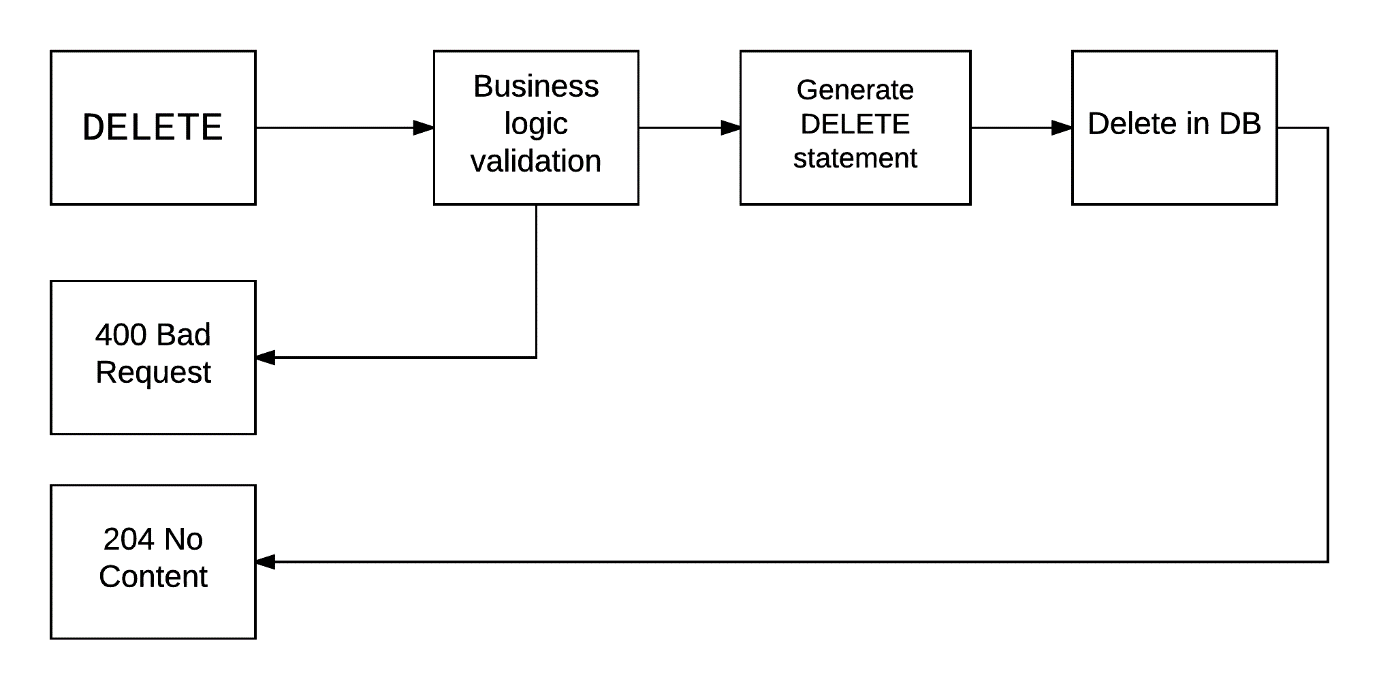
In order to update an entity the client must submit an HTTP PATCH request that contains the data to be modified. The data must be submitted in JSON format. As it is with the POST request, the validation kicks in after the request has been deserialized and every field that has validation undergoes integrity checks. When everything is ok, the ORM builds an update statement and dispatches it to the database.

Once the ORM gives back control to the server, the server returns a response with a status code of 204 No Content



## deleting items

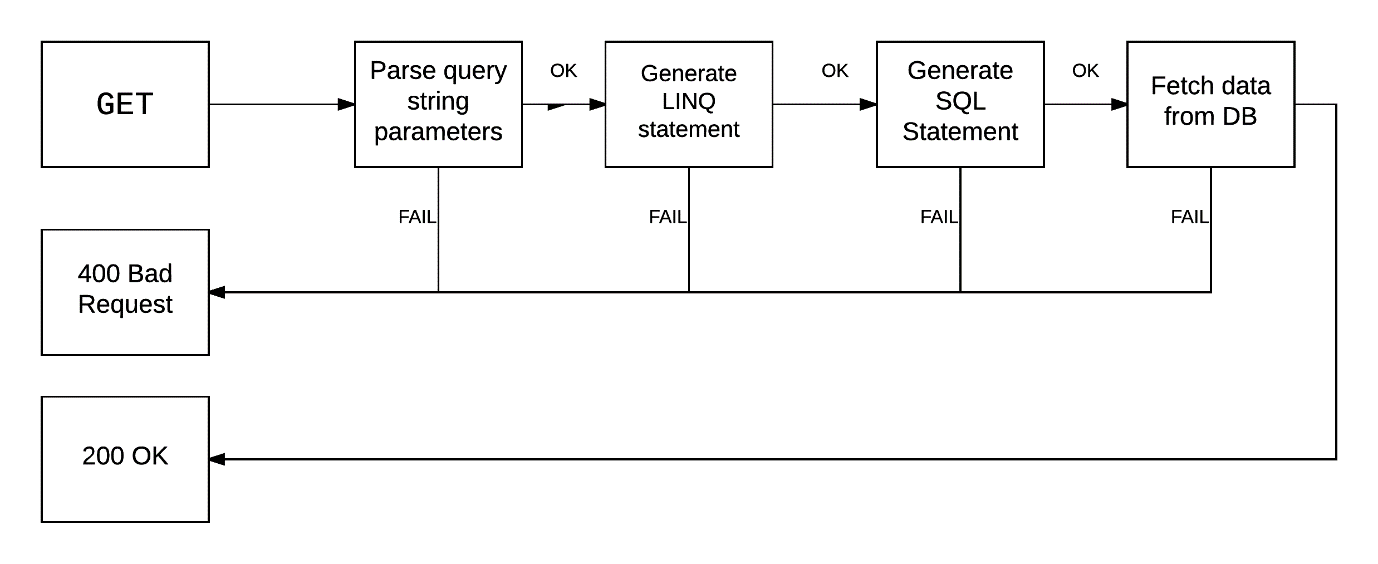
In order to delete an entity the client must submit an HTTP DELETE request to the URL of the item that needs to be deleted. Once again here validation checks are made, because deleting some items may result in system failures as other entities may depend on them. E.g. deleting a user will result in cards that don’t belong to anyone and are left as orphaned records in the database. There are two options here – either forbid the deletion of the item if DB integrity can be guaranteed or do a cascade delete that removes all of the related entities of this item - although sometimes this operation may not work as well due to other conflicts. If the server denies the deletion or a foreign key in the DB prevents this item from being deleted, the client will receive a status code of 400. If the request is successful, he receives a status code of 204 No Content as the protocol states.



## Retrieving items

In order to retrieve a single item the client must send an HTTP GET request with the item URL. The server goes to the database and retrieves these records. He gets back to the client with the item data returning a status code of 200.

Retrieving a collection of items requires the client to send an HTTP GET request to the URL of the entity without additional segments. If the client wishes to retrieve a subset of the items, for example the first 20 items, he should pass query string parameters in the URL. This will be translated to a LINQ statement and later on interpreted by the ORM and translated into a SQL statement. Supported query string parameters are skip, take, filter and sort.



## Error handling

The application includes error-handling code that allows it to recover gracefully from unexpected errors. When an error occurs, the application can need to request user intervention, or it may be able to recover on its own.

On the server side this is implemented via a global exception handler plugged into the Web API pipeline. Once an error occurs it will be intercepted and logged. For the moment errors are logged into a console window at runtime. Ideally this would be replaced with either a file logger or an external logging system like LogStash. Once the error is handled the application responds with a proper status code and reason.

On the client side error handling is done by logging in the browser console. For the moment all http calls that are 400+ or 500+ are logged to it. This may be replaced in the future with a client side logging system lime TrackJs

Technologies

## Client side

### Javascript & ajax

JavaScript is a high-level, dynamic, untyped, and interpreted programming language. It has been standardized in the ECMAScript language specification. Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web content production. The majority of websites employ it, and all modern Web browsers support it without the need for plug-ins. JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. It has an API for working with text, arrays, dates and regular expressions, but does not include any I/O, such as networking, storage, or graphics facilities, relying for these upon the host environment in which it is embedded.

Ajax (short for asynchronous JavaScript and XML) is a set of web development techniques using many web technologies on the client-side to create asynchronous Web applications. With Ajax, web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behaviour of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows for web pages, and by extension web applications, to change content dynamically without the need to reload the entire page. In practice, modern implementations commonly substitute JSON for XML due to the advantages of being native to JavaScript.

### SPA applications

SPA stands for single page application and is a technique used for building web applications. It is a commonly used and widespread method that is beginning to outshine the traditional server side rendering frameworks like Asp.Net or JSP.

In a SPA app all the resources, required to load the page are either retrieved on the first page load or are lazy loaded. Lazy-loading means that resources are loaded on demand when needed. This is the preferable method as this way the resources that are required for the initial page load are smaller. This helps for faster loading of resources and improves the overall performance of the web app not only on start-up, but during the whole life-cycle of the application. Resources are retrieved in small optimized chunks from the server and can vary in size and content. For instance, a resource can contain JavaScript logic for customizing the UI elements` behaviour or it can contain .html file templates for data-binding to data. Such resources may even contain a mix of the latter two resource types combined into a "bundle". Using this approach, a module loader framework such as webpack is required to help retrieve the files from the bundle. One of the differences between SPA apps and other frameworks such as MVC is that at no time does the client go to the server to do a full page render. Resources are retrieved, modified and consumed by the client framework. This helps achieve a seamless transition between the different sections of the site.

Another significant difference is that all of the UI logic happens on the client. The server is unaware of the client and knows only how to provide the data to it (provided that the current user has permission to access it). The benefit of this approach is that this solution is scalable and loosely coupled. It's scalable, because whenever the load increases the servers can be increased to handle more client requests. It's loosely coupled because the UI can connect to any of the servers that provide the needed APIs.

All of the data is retrieved via web services using the http protocol. The data is in the form of JSON as it is lightweight data structure and can be consumed by a wide variety of clients including JavaScript clients like browsers.

The responsibilities of the server consist of serving the data via web APIs and the HTTP REST protocol.

Building a SPA app requires a framework that can manage all of the data-binding, navigation, security restrictions and user interaction. The two most popular frameworks that are mostly adopted by the community and have support for building large enterprise applications are Angular2 and React. These frameworks are built from start with performance and memory management in consideration. For instance, Angular2 supports Ahead of time compilation (AOT) of html templates. For this project Angular2 will be used as it has a larger community build around it and is a rewrite from AngularJS (the v1 version of the framework). Using Angular is preferable, because it was built from scratch to address all of the issues the v1 (AngularJS) version had.

### Angular 2

Angular2 is a modern framework for building scalable, testable, memory and CPU performing client web applications. Its official release data was on 15.09.2016 and is relatively stable for use. It provides the necessary functionalities which can be used to quickly build a web application, some of which are described below.



### Data binding

Since the data and html are separated, there needs to be a mechanism for providing the data to the templates, so they can be populated with data. Angular provides declarative data binding with the html template syntax, that lets developers use directives and curly braces or even write JavaScript code in the templates.

### Routing

The browser is a familiar model of application navigation. We enter a URL in the address bar and the browser navigates to a corresponding page. We click links on the page and the browser navigates to a new page. We click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages we've seen.

The Angular Router ("the router") borrows from this model. It can interpret a browser URL as an instruction to navigate to a client-side page and pass optional parameters along to the supporting page to help it decide what specific content to present. We can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link. We can even navigate imperatively when the user clicks a button, selects from a drop box, or in response to some other stimulus from any source. And the router logs activity in the browser's history journal so the back and forward buttons work as well.

Routing enables the user to navigate to different parts of the app, without doing a full page reload. Behind each route lies an entry component that renders the html output on the browser. This component may be composed of child components, which may have child components of their own. This way a component tree structure is build that in the end outputs html to the browser.

### Modularity

Angular Modules help organize an application into cohesive blocks of functionality. They aim to group components, directives, pipes and services with similar or dependent functionality. Each is focused on a feature area, application business domain, workflow, or common collection of utilities. Modules can also add services to the application. Such services might be internally-developed such as the application logger. They can come from outside sources such as the Angular router and Http client. Modules can be loaded eagerly when the application starts. They can also be lazy loaded asynchronously by the router.

### NPM

Client-side web applications are usually made using the standard approach with writing the js logic in scripts (or using third-party libraries) and including them in some of the pages that require them. The dependency management of such dependencies can be difficult and error prone. Moreover, it's not easily maintainable. Systems that require external dependencies are using packages-manager systems to help with the management of such dependencies. Examples of such systems are - apt-get (Linux), nuget (.NET). For client-side applications this is [NPM][8]. NPM will be used for the development of the library as it supports both dev dependencies and application dependencies.

### Typescript

JavaScript is the language of choice for writing the client-side logic of web applications. However it lacks type support and the security that languages such as C# and Java offer. Typescript is a superset of JavaScript and provides types and OOP to JavaScript. It’s very similar to C# and makes use of interfaces and abstract classes. Everything is then compile-time checked, before being transpiled to JavaScript. Moreover the transpiled JavaScript is optimized and validated. The community around the project is very large and has helped to develop a very good tool to writing large enterprise web applications.

### Webpack

Client-side applications must be fast and small in size. With eternal dependencies this gets difficult to achieve as most of them come pre-bundled and cannot be shrunk. [Webpack][2] takes care of this by traversing through the required and using the method 'tree-shaking' excludes the code that is not needed. It can be integrated with typescript to provide minification and bundling of the client-side code.

## Server side

### asp.net core

ASP.NET Core is a significant redesign of ASP.NET. It is a new open-source and cross-platform framework for building modern cloud based internet connected applications, such as web apps, IoT apps and mobile backends. ASP.NET Core apps can run on .NET Core or on the full .NET Framework. It was architected to provide an optimized development framework for apps that are deployed to the cloud or run on-premises. It consists of modular components with minimal overhead, so it retains flexibility during the constructing of solutions. Development of ASP.NET Core apps cross-platform is available for Windows, Mac and Linux.

### asp.net web api

HTTP is not just for serving up web pages. It is also a powerful platform for building APIs that expose services and data. HTTP is simple, flexible, and ubiquitous. ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices

### ef core

Entity Framework (EF) is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write like writing direct SQL statements.

It also provides an abstraction over the underlying database, meaning that the database is transparent to the user and. This provides a huge advantage as the application can use different databases in different scenarios. And all of this can be achieved without making any code changes at all, but with just a simple configuration change.

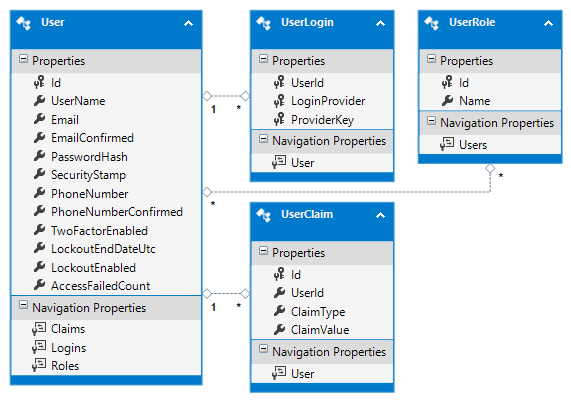
Implementation

security

As this application requires working with sensitive data, there must be some restrictions placed on the users not to access different parts of the applications thus restricting their access. Therefore the application houses functionalities that prevent users from tampering with restricted data such as Data and View access restrictions. Furthermore the security concept is split into two parts - Authentication and Authorization.

## User model

Asp.net Core provides an out of the box identity management library named Microsoft Identify v3. The library has integration with EntityFramework Core and has the ability to automatically set up the database schema for roles and users. The library provides functionality such as managing users, roles, custom claims and even two factor authentication. Moreover, this library can plug into the Asp.net Core request pipeline and automatically login users based on the cookies that they provide. Another big advantage is that it is very well tested.



## authentication

With authentication a user can be recognized as an existing registered user. The user has already passed through the registration process (which may even include email confirmation) and the system can confirm that he is a valid user. If the system does not find such a user present, all requests coming from this user will return with a status code of 401 (Unauthorized).

There are only two publicly visible server endpoints - /login and /register. Every other endpoint requires the user to be authenticated. On the client there are only two routes that the user can access - /login and /register. They have the same name as the corresponding server endpoints that they communicate with for the same purpose. These routes need to be available to anonymous users, so that they can register and login.

### login

Users are able to login to the system using the login interface, located at the "/login" route. The user is able to enter an email and password and attempts to authenticate with the system. If the user provides invalid credentials then the server should respond with a status code of 400 (Bad request) and optionally return a response message that something is wrong with either the username or password, but not specifying which. By limiting the error information, hacker bots will be more troubled to guess the exact combination required for a successful login. The login screen requires a valid email to be entered using a regular expression to match the valid sequence of characters.

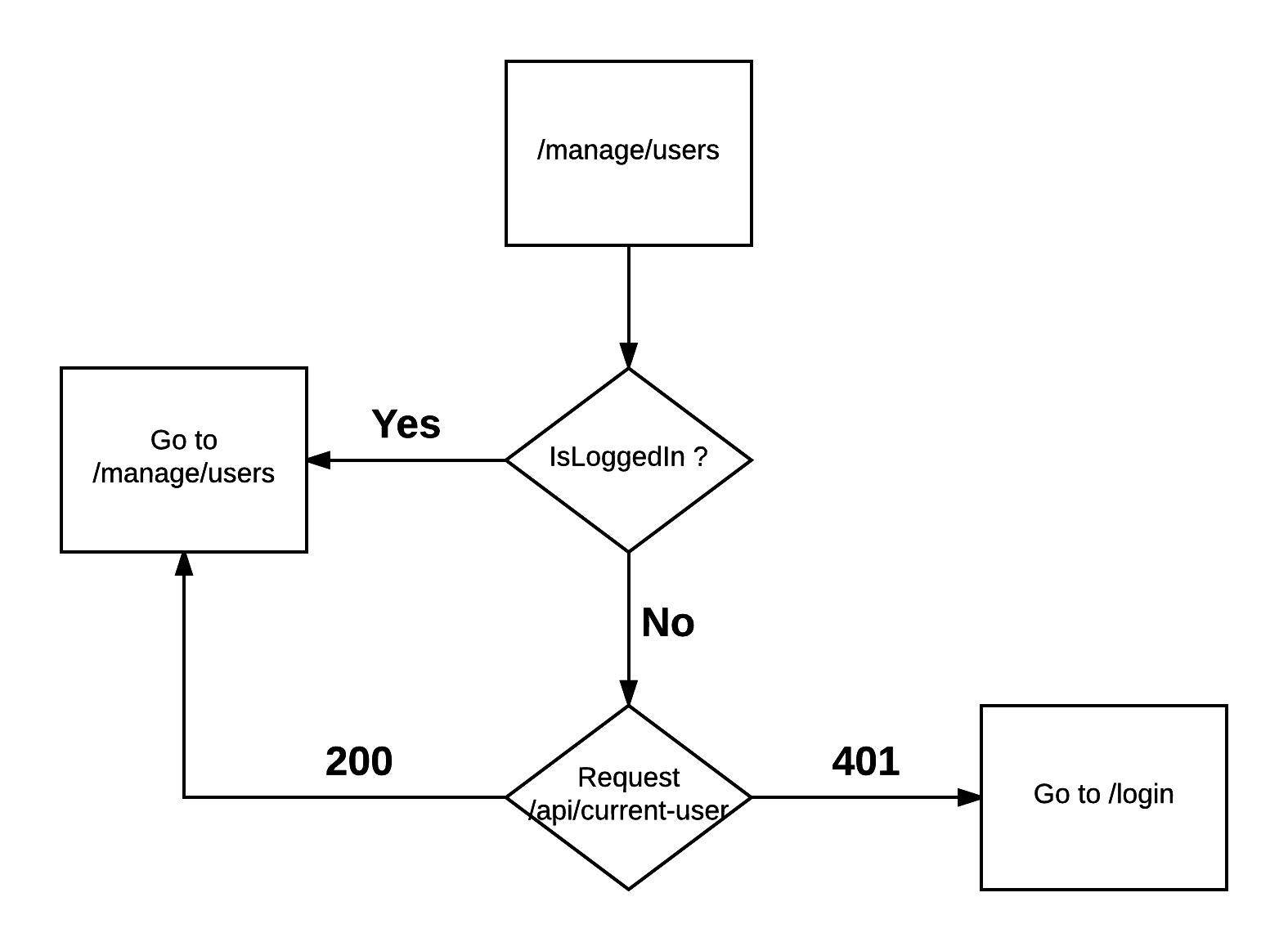
Users can check the "keep me logged in checkbox" when logging in the system. This checkbox instructs the server to issue a "session" cookie that lives even when the browser is closed. This way, when opening the browser again in the future, this cookie may still be valid and can be reused for authentication. However, the cookie has an expiration date to limit its lifetime.

The client application keeps track of the state of the current user (whether the user is logged in). Upon a successful authentication, a flag is set to true indicating the success. This flag is checked on every route request to the client application. It is set to false when the cookie expires or the user initiates a logout (here the whole user state is destroyed).

Upon the first request when the client application loads, the app does a "sniffing" call to check if the user is logged in by requesting the user's meta-data from the server. This sniffing call sends the cookie data with it. The sending of the cookie happens automatically and is performed by the browser. If the cookie is not present or it has expired, the response of the status code of this service call is 401. This automatically redirects the user to the login screen. Every request that returns with a status code of 401, automatically redirects the user to the /login route and presents a challenge (in the form of username and password prompt).

The /login route is used as the default route to which the user is navigated if he is not authenticated. This is possible by using route guards that are invoked before each route is triggered. This way the app ensures that no unauthenticated user is able to access views that he is not allowed to.

Future development of the application might include Facebook and Google login, which would be a lot easier for the user. He would just need to confirm that the application is a trusted source and he will have an account set-up for him without needing to provide his credentials again. Another feature would be to actually send an email confirmation letter that contains a verification code only for this user. Doing so prevents the system from being flooded by bots.



### logout

Upon a successful login, the server issues a cookie which identifies the user. The actual data that the cookie contains is encrypted with a private key and only the server has the ability do decrypt it as he is the owner of the key used for encryption/decryption. This cookie is sent with each request to the server (handled automatically by the browser), so that the server can identify the user and authenticate him without requiring his email and password again.

In order to initiate a "logout", a request to the server endpoint "/logout" must be made which returns a response that instructs the browser to dispose the cookie. This can be achieved through the Logout button that is available as part of every screen of the client side app. Although the logout endpoint requires no parameters, it still requires the HTTP POST verb, as the post verb implies a change on the server - namely logging out the user.

Alternatively the user can directly communicate with the server API by sending an HTTP POST request to /login containing the user credentials.

## Registration

Users may want to register with the system if they do not have an account. This can be achieved by requesting /register or navigating to the register page from the login page. The register screen requires the user to enter data such as an email, password, confirm password and username. All of these input fields are checked on the client side to provide a quick feedback to whether the data that the user entered is valid. This is done using HTML5 and Angular2 validators. The submit button is enabled only when all of the required data is valid. Because not all of the validation can happen on the client, some validation checks need to be performed on the server. Such validation checks include if the username or email is already taken. If one of these is, the server returns a status code of 400 (Bad Request) with a response message indicating the exact problem, so that the app can provide feedback to the user on what went wrong.

Additionally all of the data is further validated on the server, because the user can always bypass the client web application and issue requests directly.

### Future work

Future developments may introduce an ability to block a user or even resend an email conformation if the user was not able to receive it in the first place.

### user metadata

The meta-data for the current user is retrieved via a /current endpoint that is accessible by every authenticated user. Each time the application starts, this endpoint is requested to check if the user is logged in or not. If the response from the server is 401 (Unauthorized), then the user is redirected to the /login screen. If he is however, he is redirected to the initial requested route.

## authorization

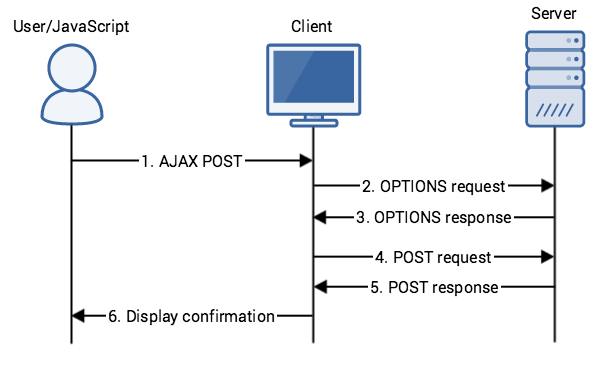
Authorization limits the rights the user has to specific resources - for instance sensitive data. In order to have a proper Authorization mechanism, the system provides the ability to add specific roles to users. Using this approach on the server, requests are restricted to specific roles and deny access to whomever does not have such. Leveraging the HTTP protocol, all requests coming from a user that doesn't have the proper permissions will have a response of 403 (Forbidden).

In order to guard routes on the client that the user does not have access to, a guard will be used to validate the current user. The current user's roles are compared to the one the route allows and if the user does not satisfy the route roles, he is redirected to a 404 page. Another requirement of the system is to have specific portions of html hidden for some roles. This is implemented by using an angular directive that takes the current user meta-data (roles) and validates it against a predefined set. If the condition is false no html is rendered.

## cors

The Cross-Origin Resource Sharing (CORS) mechanism gives web servers cross-domain access controls, which enable secure cross-domain data transfers. Modern browsers use CORS in an API container - such as XMLHttpRequest or Fetch - to mitigate risks of cross-origin HTTP requests. By default the site's api will not be accessible from other domains. Thus there will be a specific handler for requests coming from other sites and the server will cut off such requests returning a status code of 400 (Bad Request).

A resource makes a cross-origin HTTP request when it requests a resource from a different domain than the one which the first resource itself serves. For example, an HTML page served from http://domain-a.com makes an <img> src request for http://domain-b.com/image.jpg. Many pages on the web today load resources like CSS stylesheets, images and scripts from separate domains.



Administrattion

The administration is located under the tab “Administration” in the navigation menu. The administration is still accessible by direct URL via /admin. The pure admin URL (without additional segments following it) will navigate to the users management screen. This page is accessible by users having the Admin role only.

Under the administration there are two child pages – user management and card type management. Every page located under the root admin page is accessible by users having the Admin role only.

## user management

In order to manage the users and their roles, an interface is needed where the Admin(s) can have the ability to change user access or resolve user conflicts. For instance - assign a role to a user or delete a user. The functionality of the grid is limited to changing the roles of users, adding, deleting and browsing users. Additionally the current admin that is editing the users is not able to see himself in the grid so that he cannot delete himself.

Upon initial navigation the users will be retrieved from the server in order to populate the grid. This is achieved via dispatching an action that states “Fetch the users from the remote data store”. Prior to that the Component for managing users on that page (GridComponent) has already subscribed itself to state changes. This is because if we subscribe after firing the event, we can miss out on the callback that fetches the data. Although this is nearly impossible it is better to have it this way to minimize any possible bugs in the future.

The grid includes two buttons for each user entry in the system – one for delete and one for edit.

Pressing the delete button triggers an action that sends an HTTP request to delete the user. Provided the request is successful, the callback that will be invoked will check the state and find the user via email. If the email is present the state is modified and the user is removed from it. This in turn triggers a state change and all of the subscribers are notified. The Grid is then updated according to the state change – deletion of the user.

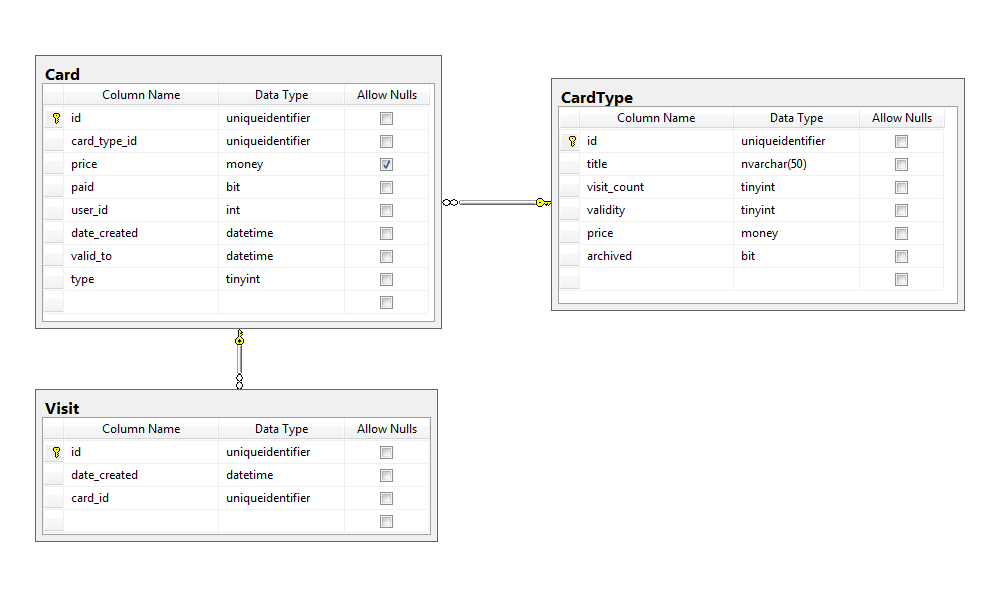
Pressing the edit button reveals a section of the page that is initially not visible. This section contains the editable information for the user. As the email is kept unique and used as a means of logging into the application, it is not available for editing. The only thing that is available for editing is the role of the user. As the user may be in more than one role, the user is provided with an HTML multiselect option that enables him to select more than one role. Additionally it highlights the currently selected role and detects changes. Upon pressing the done button, if there are no changes, no action is performed to minimize to traffic to the server. If there are however, an action is triggered that dispatches an HTTP request to the server. There the request is again validated for changes and if there are any, the DB is updated accordingly.

## Card management

Visitors of the facility must have a way to buy cards and monitor them. Admins on the other hand must have a way to manage these cards and define templates that provide previously defined cards that will be offered to the users. Thus a special section in the administration is created to help admins deal with cards.

### Database schema

The database structure of the card management functionality consists of three tables CardTypes, Cards and Visits. The CardTypes table is used for storing card templates. It has a relationship of one-to-many with the Card table. The cards table holds the cards and has a reference to the Visits table that keeps track of the visits made by the visitors of the facility. The functionality has the following schema.



### User interface

Card templates (card types) are used to manage the cards provided to the users to use. This helps ease the selling of the cards. A grid is used to manage the card types. In the grid all of the fields are displayed except the Id of the card, which is irrelevant to the user. However the system underneath keeps a reference to the Id of the item, so it can be used later on for other operations. Other than the fields, there is another column in the grid that defines the available actions for the user – archive and delete.

Because of the relationship Card-CardTemplate a card template cannot be simply deleted. This will cause inconsistency in the database as there will be card templates that are no longer present. Moreover if there is a history of the used cards based on certain templates that are no longer available, the system will stop working properly. In the end there are two options that guarantee database integrity – provide the user with a delete option that deletes the card template and all of the cards belonging to it, or provide an option to archive the card and remove it from the grid where it will be no longer visible, but still available for future references.

The final solution is based upon the previous two. The user is provided with both functionalities – archive and delete. Delete is available only when there are no cards associated with the selected card template. This is calculated on the server by running a query for each card that goes to the database and queries all the cards that are associated with a particular card template. The server response contains a field that has the outcome of this result – true or false. If the response is true, then the Delete button is greyed out and not available. However if the server returns false, then there are no cards associated and the card template can be safely deleted.

Once the user presses the delete button, an action is dispatched that executes an http call. If the http call is successful, the client side state is modified and the card is removed from the grid.

The button archive does the same thing as the delete button only the card template is not removed from the database, but only from the grid to allow the user to focus only on the important cards.

The administrator is able to add new cards by pressing the Add button on the screen. Here a section is expanded that contains all of the fields of the card – Title, VisitCount, Validity and Price. Each of the fields has validation enabled for it using Angular Form validation. Once all of the fields are valid the done button is enabled and the user is able to add the new card template.

For the same reason as with the delete functionality, the system will not allow Card Templates to be edited. Editing a Card Template will mix up the reports for incomes of each month that the administrator wishes to view. E.g. if a Card Template has a price of 10 and there are 5 sold cards for month January, which means an income of 50. But if the admin changes the price of the card Template the income will be 60, leading to confusion. An alternative to disabling the editing of a card would be to do a copy of the card and archive the previous one.

## Card management

Alongside the management of templates a screen is provided where the administrator will be able to manage the card instances for all of the users. This screen is used by both administrators and teachers. The users in Teacher role will use it to issue new cards to users, but will not have the ability to edit them. The administrator role will have the ability to make adjustments if something is not OK with the card. Such adjustments are for example – the price of the card (some clients may be more privileged than others and have discounts for them).

By default the grid will show only cards that have not expired, but there is an option to show those that have as well. This is made for the convenience of the user, so he can view every card in the system. However even if the expired cards ca be viewed, they will not support adding of new visits to them. The system restricts such actions to ensure that everything is following the workflow properly.

Alongside the expiration filter, there is an option to filter by a specific user. This again is made for convenience as to track the progress of the student – how often he has attended the exercises.

### Visits

There is an option in the grid that shows the visits for the current card – how much visits have been made. They are shown in a list on the same page alongside the dates that they were made on.

Visits can be added to a card by pressing the “Add Visit” button. This opens up a section of the screen and the user can select the amount of visits to add and on which date. The amount will default to one, but in the case of family cards or bringing friends that share the card, the user will be able to select more than one. This amount will be displayed in a prepopulated dropdown that will not allow to add more visits than there are left for the current card.

Editing a visit is not included as there is only a single field to edit – the date on which the visit occurred. This is reserved for future implementations.

Deleting a visit is possible and there is a delete button for that purpose. Admins are able to use the delete button only.

### Adding a Card

When adding a new card, the user is presented with a form that enables him to fill in the properties of the card – date from which the card is valid, the price of the card (if different from the template), the user that the card belongs to and if the card is payed or not. It is sometimes convenient for students coming to the facility to pay on a later stage in the process, but they will still be able to have their visits tracked. Therefore one of the options in the create screen is to check if the card is payed or not.

By default the option will be checked, but if the user decides he can disabled the checkbox and mark the card as unpaid. It will not show up in the monthly reports as no amount has been paid for this purchase. It can be edited on a later stage by administrators and once it is paid for, it will appear in the current month’s report. Again this field will support filtering to help administrators track abusers and better handle their money management.

The user is able to preselect a card template. Once the card template is selected, the field price is automatically populated. Here if the Teacher/Admin decides he can change the price to a lower value.

### Editing a card

Administrators are able to edit a card and change the user, price date of creation. Additionally they are able to change the card template in case an error occurred. Here it is important to take some precautions as the database integrity may be compromised. Changing the card template is allowed if the newly selected template’s allowed visits does not exceed the amount of current visits. Otherwise there is no point of changing it as the card will be marked as expired. Once the template is changed the price (if not initially overridden) can be changed as well. Otherwise the price should be changed accordingly.

Another use case for the editing is to check if the card is payed or not. If the payed checkbox is not marked, then the price field will not be available as well, but will preserve its value if present as to not lose data.

## Money management

To help administrators manage the cash flow of the facility, there is a money management screen available. This screen serves the purpose of listing all of the sold cards within a time box. Alongside the date filters that will be used to define the time box, there will be a total amount displayed to show the income for the time box. By default the time box will be based on months and will start from the beginning of the month till the end of the month.

No editing or deleting is allowed here. This screen is used for generating reports only.

## Workout management

Just as it was with the card management screen, here the users of the system (students) are able to track their own progress and cards. This view will be identical the one shown in the administration part except it will be viewed in read-only format and it will be always filtered for the current user.

# References

*Angular 2*. (n.d.). Retrieved from https://angular.io

*Angular component decorators*. (n.d.). Retrieved from https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html

*Angular guards*. (n.d.). Retrieved from https://angular.io/docs/ts/latest/guide/router.html#!#can-activate-guard

*Attribute directives*. (n.d.). Retrieved from https://angular.io/docs/ts/latest/guide/attribute-directives.html

*Entity framework core*. (n.d.). Retrieved from https://docs.microsoft.com/en-us/ef

*Microsoft identity v3*. (n.d.). Retrieved from https://github.com/aspnet/Identity

*Moscow method*. (n.d.). Retrieved from https://en.wikipedia.org/wiki/MoSCoW\_method

*React*. (n.d.). Retrieved from https://facebook.github.io/react

*Spa applciations*. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Single-page\_application

*Typescript*. (n.d.). Retrieved from https://www.typescriptlang.org