TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Cross-plattform development using the Electron Framework, Angular and ASP.NET Core

Bachelorarbeit

im Studiengang Informatik

vorgelegt von:	Jan Morawietz
Matrikelnummer:	2731582
Abgabedatum:	02.09.2018
Erstkorrektor:	Prof. Dr. M. Teßmann
Zweitkorrektor:	Prof. Dr. T. Voit

Contents

List of Figures

List of Listings

List of Abbreviations

API	Application Programming Interface
ASP.NET Core	Active Server Pages .Net Core
CLI	Command Line Interface
COM	Component Object Model
CSS	Cascading Style Sheets
DI	Dependency Injection
DLL	Dynamically Linked Library
DOM	Document Object Model
DTO	Data Transfer Object
ECDSA	Elliptic Curve Digital Signature Algorithm
EF Core	Entity Framework Core
ER Diagram	Entity Relationship Diagram
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HMAC	keyed-Hashing for Message Authentication
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environments
IIS	Internet Information Services
IPC	Inter-process Communication
JS	JavaScript
JSON	JavaScript Object Notation

JWT	JSON Web Token
MSSQL	Microsoft Structured Query Language
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
MySQL	My Structured Query Language
NPM	Node Package Manager
ORM	Object Relational Mapper
OWIN	Open Web Interface for .NET
REST	Representational State Transfer
RSA	Rivest-Shamir-Adleman
SPA	Single Page Application
SQL	Structured Query Language
TPL	Task Parallel Library
TS	TypeScript
URL	Uniform Resource Locator
UTC	Universal Time Coordinated
UWP	Universal Windows Platform
WDC	Windows Dev Center
WPF	Windows Presentation Foundation
XML	Extensible Markup Language

1 Introduction

When developing a new application, certain aspects are to be considered. A fundamental decision to be made is whether the application is going to be a web or a desktop application. Many applications, like the entire Microsoft Office Suite, which used to be pure desktop applications now either have web counterparts or have been moved to the web environment completely. The web environment gives the developer access to all desktop platforms as well as mobile platforms. Frameworks for those types of applications include Angular, jQuery, React, Vue, Ember, or Meteor for the front-end as well as Active Server Pages .Net Core (ASP.NET Core), Django, Express, or Ruby on Rails for the back-end.

Although web applications are becoming more and more popular, desktop applications have not yet ceased to exist. This may be due to the fact that web applications are subject to many restrictions. Many tasks like editing files, communicating with special hardware or querying databases are not possible. Yet these basic tasks are essential for many applications. Other reasons why users might prefer desktop applications, is the fact that they can be used offline and that a desktop application is what most people are used to.

The release of the Electron Framework in 2013 opened up a way to combine the flexibility of web applications with the strength of desktop applications. Electron makes it possible to write one single code base consisting of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript (JS), and still access native Application Programming Interfaces (APIs), files and databases. Electron also offers APIs for notifications or a system tray icon, which are commonly used elements in desktop applications.

The thesis will discuss how to write an application using the Electron framework in combination with the Angular front-end framework and the ASP.NET Core back-end framework. Its goal is to determine if there are features which do not work at all and what adaptations are needed to get other features working given this specific combination of frameworks. Furthermore, the advantages and disadvantages compared to a pure desktop or web application will be discussed.

The Jukebox Project

The Jukebox project is the proof-of-concept application developed for this thesis. The application is a music player. The music files are read from disk

by the ASP.NET Core back-end, and then transmitted to the Angular web front-end for playback. The code parts which are responsible for handling the playback are referred to as the „player“. A requirement was that the player can be remote-controlled. For this purpose, the ASP.NET Core server also hosts the application as a pure web page. A second client may use this web page to control the player created by the first client. Websockets are used for this feature. The application should have a system tray icon when run as an Electron desktop application. Any noteworthy event should be communicated to the user via notifications.

Architecture Overview

The entry point for the Jukebox application is in the Electron main.js file. From this file, the ASP.NET Core server is started as a separate process using node's `child_process.spawn()` method [Nod18d]. Then the `BrowserWindow` is created and loads the index.html file. At this point, the Angular framework is initialized. The startup sequence is visualized in Figure 1.1 on this page.

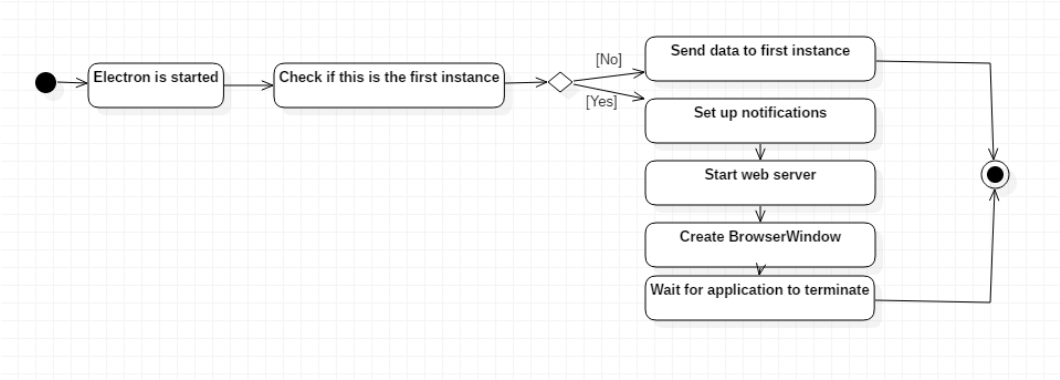

Figure 1.1: Startup sequence of the Jukebox application.

Communication between the frameworks is usually initiated from within the Angular code. The „ngx-electron“ package is used for communication with Electron's APIs (subsubsection 3.2 ngx-electron on page 22). For the communication with the ASP.NET Core, Hypertext Transfer Protocol (HTTP) calls and websockets are used.

Deployment

Each of the three frameworks comes with some sort of Command Line Interface (CLI), build tools, or required steps for the packaging process. For Electron, the Squirrel [Git18c] installer is recommended, but during testing many problems were encountered and therefore, no installer is used. Instead, Gulp scripts [Fra18] are used to create a completely packaged version of the

application, which then can be shipped with any installer framework or simply copied to the target system. Electron offers pre-built binaries for each operating system, these can be downloaded from the Electron site. Those binaries and all required Node Package Manager (NPM) packages are copied to the target packaging location. Then, the `electron-rebuild` NPM package is used to compile all NPM packages to target the current Electron version. After this step has been completed, the Angular CLI is used to build the Angular part of the application. This step is done twice with varying parameters. The first build targets the Electron version, and the second build targets the pure web version of the Angular part. The Angular files are then placed in the `/resources/app/dist` folder. Next, the ASP.NET Core application is published. Publishing an ASP.NET Core application first compiles the source code for a given platform and then copies the binaries including all .NET Core framework libraries to the target folder. For the Jukebox project, this folder is located under `/resources/app/api/win/` for the windows build. Finally, the „electron.exe“ is renamed into „Jukebox.exe“.

General Notes

This thesis discusses all three frameworks in a separate chapter each. All descriptions of framework methods are based on the manufacturers documentations. For the sake of simplicity and readability the text uses the male form „he“ instead of „he or she“ when talking about developers, this is deemed to include „she“.

2 Angular

Angular is an open-source front-end web application framework which is written in TypeScript (TS). It is developed by Google and contributors. When talking about Angular it is important to distinguish between AngularJS, also called Angular 1.x, and Angular 2+. For the Jukebox project Angular 2+ was used. In the following text Angular always refers to Angular 2+.

2.1 Structure

When using the Angular framework, a developer will implement his view logic using Components, Modules, Services and many more pre-defined Angular classes. This section describes which elements are offered by the Angular framework and how they should be used.

Components

An Angular component can be compared to a View and ViewModel combination from the Model-View-ViewModel (MVVM) pattern. It is composed of three files. There is one HTML and one CSS file which make up the View. The ViewModel is a TS file which contains the essential logic for the View. The component is always declared in the TS file with the `@Component` decorator, see Listing 2.1 on the following page.

The `@Component` decorator can take many arguments, but the commonly used ones are: „selector“, „templateUrl“, and „styleUrls“ [Goo18a]. The selector is a string which starts by convention with „app“ and is then followed by the component's name, see Listing 2.1 on the next page, Line 5. The selector is used to include this component in the HTML of another component.

The „templateUrl“ and „styleUrls“ arguments reference the corresponding HTML and CSS files, respectively (Listing 2.1 on the following page, Line 6 and 7). Both arguments take a relative path, „templateUrl“ is always related to a single file, „styleUrls“ may be related to multiple files in the project. Inline variants may be used for both arguments, though for reasons of readability concerns they should be avoided when possible.

```
1  import {Component, EventEmitter, Input, OnInit, Output} from '@angular/core';
2  import {AngularMenuItem} from "../models/angular-menu-item";
3
4  @Component({
5    selector: 'app-nav-item',
6    templateUrl: './nav-item.component.html',
7    styleUrls: ['./nav-item.component.css']
8  })
9  export class NavItemComponent implements OnInit {
10
11   @Input()
12   CurrentItem: AngularMenuItem;
13   @Output()
14   ItemClicked = new EventEmitter();
15
16   constructor() { }
17   ngOnInit() { }
18 }
```

Listing 2.1: Example of an angular component typescript file.

Lifecycle Hooks

The Angular framework manages the lifecycle of each component. Throughout this lifecycle Angular offers the possibility to respond to certain events e. g. `ngOnDestroy()` where the developer could run a cleanup routine. In order to respond to any of these events, the developer has to implement one method per event. The method signatures are defined by the Angular framework and exposed as interfaces. Since the executed code at runtime is JS code, interfaces are not necessary. Nonetheless it is considered „best-practice“ to use these interfaces.

Template Syntax - Bindings

A template in the Angular context is equivalent to a MVVM View and is written in HTML. All HTML tags are allowed, though some tags like `<html>` or `<body>` are meaningless in this context. The official set of tags is extended with tags that refer to components as described in subsubsection 2.1 Components on the previous page.

Like many MVVM frameworks, Angular supports bindings. These are used to display data obtained from the ViewModel or, in case of input controls, send data back to the ViewModel. Interpolation is used to display data. The syntax used by Angular is {{`myObject.myProperty`}} as shown in Listing 2.2 on the following page, Line 3. Angular will evalute the text between the curly brackets as an expression and then display the result. Usually, the context for the expression is the component's instance.

```
1  <mat-expansion-panel *ngIf="CurrentItem.type === 'submenu'">
2    <mat-expansion-panel-header>
3      <mat-panel-title>{{CurrentItem.label}}</mat-panel-title>
4    </mat-expansion-panel-header>
5    <app-nav-item *ngFor="let subItem of CurrentItem.submenu; let i=index"
         [CurrentItem]="subItem" (ItemClicked)="ItemClicked.emit()">
6    </app-nav-item>
7  </mat-expansion-panel>
8
9  <mat-list-item *ngIf="CurrentItem.type === 'normal' &&
       CurrentItem.visible" (click)="itemClickHandler()">
10   <p> {{CurrentItem.label}} </p>
11 </mat-list-item>
12
13 <div [ngSwitch]="CurrentItem.type">
14   <p *ngSwitchCase="'submenu'"> Item has children </p>
15   <p *ngSwitchCase="'normal'"> Item doesn't have children </p>
16   <p *ngSwitchCase="'separator'"> This is a separator </p>
17 </div>
```

Listing 2.2: Example of an angular template.

In case data from the ViewModel should be passed on to another component, which is referenced via its HTML tag, the square bracket notation is used (Listing 2.2 on the current page, Line 5). It is also possible to use the `bind-` prefix e. g. `bind-CurrentItem="subItem"`. In order to work, both methods require that the receiving component needs a property in its ViewModel which is declared with the `@Input()` decorator, see Listing 2.1 on the preceding page, Line 11.

For events, the standard JS round bracket notation (Listing 2.2 on the current page, Line 5) or the `on-` prefix e. g. `on-ItemClicked="ItemClicked.emit()"` are used. Custom components can declare their own events using the `@Output()` decorator on a property (Listing 2.1 on the preceding page, Line 13). The property usually is of type `EventEmitter`. If RxJS observables are used, `Observable<T>` or `Subject<T>` are also possible [Les+18].

Forms often use elements like a „Combobox“ or a „Dropdown“ menu. The `SelectedItem` property of these elements is supposed to be „mirrored“ to a property in the component. This gives the developer easy access to read komma? **??** and also change the value from within the ViewModel. In this case a two-way binding can be used. Angular uses the „Banana-in-a-box“ notation which are round brackets inside square brackets e. g. `[(SelectedItem)]="userSelection"`. The two-way binding can also be achieved using the `bindon-` prefix e. g. `bindon-SelectedItem="userSelection"`. [Goo18h]

Template Syntax - Directives

Angular exposes different directives which add certain functionality to the HTML. The `*ngIf` directive determines whether the containing tag, as well as its children, should or should not be part of the Document Object Model (DOM). This was used in the Jukebox project for showing `NavItems` as an expander for items with children, or regular for single items, as shown in Listing 2.2 on the previous page, Line 1.

If an item contains children each of these child items has to be displayed. For this the `*ngFor` directive is used. Its syntax (Listing 2.2 on the preceding page, Line 5) is similar to the `foreach` syntaxes of many popular languages. The index tracking is optional, but useful in some cases. Also, it is possible to use the trackBy feature. This is similar to the `equals()` C# method. It allows the `*ngFor` directive to decide on the equality of two items not by their instance references, but by a custom-defined method. For this to work, the developer is required to implement a method with the following signature:

```
public myTrackBy(index: number, object: object): number
```

The third built-in directive is the switch-case directive which lets the developer show content based on a property. Listing 2.2 on the previous page, Line 13 to 17 is an example of how the `NavItem` template could be debugged.

Template Syntax - Pipes

Angular also offers support for pipes which are very similar to Windows Presentation Foundation (WPF) converters, with the exception that an Angular pipe only works in one direction. This is useful for formatting the „raw“ data e. g. `{{title | uppercase | lowercase}}`. As shown in the example, pipes can be chained. It is also possible to add parameters to a pipe, one example is the `date` pipe, where the name for a date format can be passed as an argument e. g. `{{ myDate | date:'longDate'}}`. Angular offers many built-in pipes, but it is also possible to create a custom one. To do so, the developer creates a new class which implements the `PipeTransform` interface and uses the `@Pipe()` decorator as shown in Listing 2.3 on this page. [Goo18h]

```
1 import { Pipe, PipeTransform } from '@angular/core';
2
3 @Pipe({ name: 'booleanYesNo' })
4 export class BooleanYesNoPipe implements PipeTransform {
5     public transform(value: boolean): string {
6         return value ? 'Yes' : 'No';
7     }
8 }
```

Listing 2.3: A custom pipe to convert a boolean to 'yes' or 'no'.

Services

An Angular service is a class which is used to do a specific, well-defined task. It usually performs operations on data or communicates with outside services for the purpose of obtaining or pushing data to an API.

```
1  import {Injectable} from '@angular/core';
2  import {ElectronService} from "ngx-electron";
3  import {AngularMenuItem} from "./models/angular-menu-item";
4
5  @Injectable()
6  export class MenuItemService {
7    private _electronService: ElectronService;
8
9    constructor(electronService: ElectronService) {
10     this._electronService = electronService;
11   }
12
13   public createElectronMenuItem(menuItem: AngularMenuItem)
14     : Electron.MenuItem
15   { [...] }
16
17 }
```

Listing 2.4: A service for managing menu items.

Angular services are written in TS. They are a class with the `@Injectable()` decorator (Listing 2.4 on the current page, Line 9) which makes this class visible to the Dependency Injection (DI) components of the Angular framework. With exception of the decorator, a service is simply a class which can implement interfaces and declare methods like any other class.

Services are used to encapsulate functionality which helps promote reusability and modularity. To determine whether a method should be implemented in a service or a component, the question is: Is the method related to how something is displayed? If the answer to this question is yes, then the method should be implemented in a component, if the answer is no, it should be implemented in a service class.

Modules

When talking about modules in an Angular application, it is important to distinguish between Angular modules, also called NgModules, and JS (ES2015) modules. Those two are different concepts and can be used side by side. While JS modules are used to manage collections of JS objects, NgModules are more like containers for related chunks of code. An NgModule contains components, services, and/or other code files which all belong to one domain, a workflow, or are performing similar tasks. From an architectural perspective they bear great resemblance to a code library like a Dynamically Linked Library (DLL). Like

DLLs, they can import other NgModules and choose which classes, services, and/or components should be exported and which should only be visibile to classes inside the NgModule in question.

```
1  import {BrowserModule} from '@angular/platform-browser';
2  [Imports...]
3
4  @NgModule({
5    declarations: [
6      AppComponent
7    ],
8    imports: [
9      BrowserModule,
10     [...]
11     //App Modules load here
12     MenuModule,
13     [...]
14     //Has to be last !!!
15     AppRoutingModule
16   ],
17   exports: [],
18   providers: [
19     AuthGuard,
20     {
21       provide: HTTP_INTERCEPTORS,
22       useClass: HttpClientErrorInterceptor,
23       multi: true
24     },
25     [Interceptors...]
26   ],
27   schemas: [CUSTOM_ELEMENTS_SCHEMA],
28   bootstrap: [AppComponent]
29 })
30 export class AppModule { }
```

Listing 2.5: The Jukebox AppModule file.

NgModules are TS classes which are decorated with the `NgModule()` annotation (Listing 2.5 on this page, Line 4). Oftentimes the class is completely empty, since the decorator arguments are the important part of the module, essentially the module's metadata. Each application has to have at least one module, the AppModule.

The **declarations** argument (Listing 2.5 on the current page, Line 5) is used to specify all components which are part of this module. If a module wants to import other modules, those modules need to be listed using the **imports** argument (Listing 2.5 on this page, Line 8). If the developer chooses to encapsulate the routing configurations into separate modules, these modules should always be the last ones imported to avoid dependency issues. The **exports** argument (Listing 2.5 on the current page, Line 17) defines which components, services, and/or modules are visible to other modules which import this module. Angular's DI configuration is strongly tied to its NgModules. The **providers** argument references all services, pipes, and guards the developer

wants to be available for injection. It is also possible to use interception, which has to be defined here as well. To be able to use custom HTML tags (see subsubsection 2.1 Components on page 4), the `CUSTOM_ELEMENTS_SCHEMA` must be included under the **schemas** argument. Usually, the **bootstrap** argument is only used in the AppModule. All components which are added here are marked as entry components for the app start. proofread **??**

Routing

Every Angular application is a Single Page Application (SPA), which means that the client downloads the entire web application in the beginning and keeps it in cache to enable faster response times once the app is loaded. This also means that normal navigation through different „subsites“ of a website will not trigger the browser to download new HTML and thus these „subsites“ do not exist in the traditional sense. Even though it is called a **Single** Page Application, displaying a gigantic site with everything in it is unwieldy and not user-friendly. Therefore, Angular offers its own routing.

```
1  <mat-toolbar color="primary">
2    <button mat-button id="btn_menu" (click)="sidenav.toggle()"
         *ngIf="!isElectronApp && mobileQuery.matches">
3      <mat-icon>menu</mat-icon>
4    </button>
5    <span id="title">Jukebox</span>
6  </mat-toolbar>
7  <mat-sidenav-container>
8    <mat-sidenav #sidenav id="sidenavContainer">
9      <mat-nav-list id="sidenavList">
10       <app-nav-item *ngFor="let item of _navItems" [CurrentItem]="item"
             (ItemClicked)="navItemClicked()"></app-nav-item>
11     </mat-nav-list>
12   </mat-sidenav>
13   <mat-sidenav-content>
14     <router-outlet></router-outlet>
15   </mat-sidenav-content>
16 </mat-sidenav-container>
```

Listing 2.6: The Jukebox navigation bar and router-outlet.

Routing is done through the `<router-outlet />` tag in HTML and the `RouterModule`. The `<router-outlet />` tag tells Angular that the developer wants to display different „subsites“ at this location in the DOM. Usually an application has some kind of menu or navigation bar which is always visible and then the rest of content is displayed through the router outlet, as shown in Listing 2.6 on the current page, Line 14.

Angular's routing module also changes the URL in the browser's address bar. This allows for deep links into the site, but also requires any web server which

serves this SPA to return the same HTML for all URLs without a redirect to the index.html route, otherwise it would break Angular's routing.

```
1  import {NgModule} from '@angular/core';
2  [Imports ...]
3
4
5  @NgModule({
6    imports: [
7      RouterModule.forChild([{
8        path: 'auth',
9        children: [
10         {path: 'changePassword', component: ChangePasswordComponent},
11         {path: 'login', component: LoginComponent},
12         {path: 'register', component: RegisterComponent},
13         {path: 'resetPassword', component: ResetPasswordComponent},
14         {path: 'logout', component: LogoutComponent, canActivate: [AuthGuard]}
15       ]
16     }])
17   ],
18   exports: [
19     RouterModule
20   ]
21 })
22 export class SecurityRoutingModule {
23 }
```

Listing 2.7: The Jukebox security route definitions.

To define the application routes and map their components, the `RouterModule.forRoot` or `RouterModule.forChild` methods are used (Listing 2.7 on this page, Line 7). The `RouterModule.forRoot` method may only be called once and is usually located in the app-routing.module.ts file. Following the Angular code style guidelines, every module should have its own routing module. This is possible through the `RouterModule.forChild` method which may be called as often as needed.

The **path** argument must be set for each route. In case of a simple route, which maps to a **component**, the component in question must be specified (Listing 2.7 on the current page, Line 10). A route can also be used as an organizing element for the application's URLs and so may not have a component, but multiple routes (Listing 2.7 on this page, Line 9). This is usually done once per module and allows the developer to create well-structured URLs. There are many more arguments which define details such as path matching strategies or specific router-outlets.

One very important feature of the Angular routing system is that of guards. A guard is a method which is called by the `RouterModule` if a certain condition is met. Guards are specified in the route configuration, as shown in Listing 2.7 on the current page, Line 14. The conditions are `canActivate`, `canActivateChildren`, `canDeactive`, `canDeactivateChildren`, and `canLoad`.

Guards are always invoked after a navigation is queried, but before it is completed. The return value is of type `Observable<boolean>`, `Promise<boolean>`, or `boolean` and determines whether the navigation may proceed. In the example of the `AuthGuard` (Listing 2.7 on the preceding page, Line 14), the `AuthenticationService.isLoggedIn()` method is used to determine whether the user was logged in and can therefore now be logged out.

Angular also allows the use of asynchronous routing, also called lazy loading. This is useful if certain parts of the application are rarely used, or only used by a small number of users. In this case, the application is separated into several chunks and only the needed code parts are served during the initial loading of the application. Once the user accesses a route to a module which is not yet loaded, Angular will trigger the download of the missing module and display it as soon as the download has finished. This will improve initial loading times, but some routes may be slow, as they have to be loaded on demand. It is also possible to configure the `RoutingModule` in a certain way which prioritizes the important parts of the application. In this case, the initial download might contain the index page as well as commonly-used modules. Once they are loaded and displayed, the `RouterModule` will download the remaining modules in the background. This will improve initial loading times without the drawback of bottlenecks later on. It should be noted that even if a module is configured to load asynchronously, it will be loaded with the initial chunk in case it is referenced in some other component. This is due to the fact that the other component depends on this module.

Dependency Injection and Interception

The Angular framework relies heavily on DI and, as of version 4.3 [Che17] also offers interception. As described in subsubsection 2.1 Modules on page 8, the configuration of Angular's DI capabilities is achieved through the metadata of NgModules. Every component and provider listed in the module can be requested. Additionally, it is possible to configure providers in the `@Injectable()` and `@Component()` decorators. In doing so, the configuration will override any configuration specified **??** at the NgModule level. To make the injection work, Angular uses its own `Injector` class. Instances of this class are created by the framework and the developer doesn't have to touch them in most cases. Though, if required by a certain action, it is possible to create and manipulate instances of this class. Every service is a singleton inside its scope, which is the `Injector` instance. Angular automatically nests `Injector` instances. Every component has its own `Injector` instance which mirrors the component tree structure (see Figure 2.1 on the following page). A request bubbles up from the `Injector` instance of the component it was requested in. The first `Injector` which can satisfy the request is used to resolve the dependency.

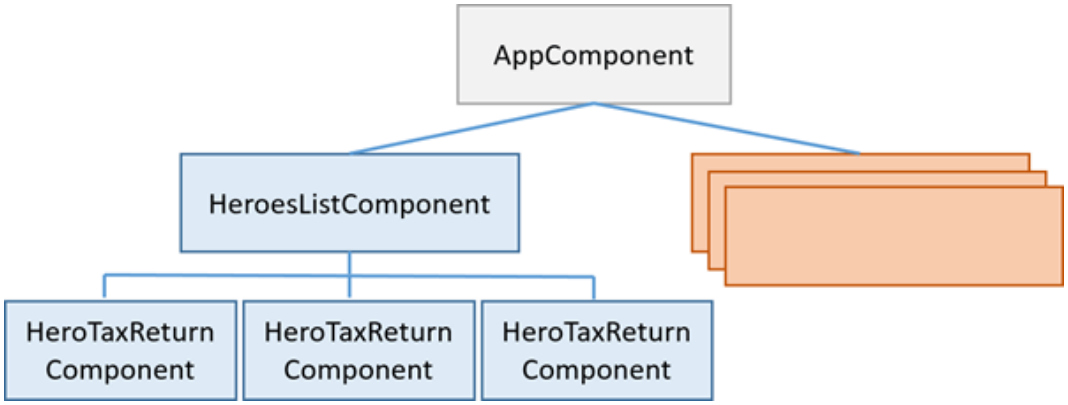

Figure 2.1: Component hierarchy in the Angualr framework. [Goo18g]

As of version 4.3 Angular offers the possibility of interception. Angular's interception is limited to only the `HttpClient` class, contrary to other DI frameworks where any class can be intercepted. To specify one or multiple interceptors, the developer has to configure a provider as shown in Listing 2.5 on page 9, Line 21.

```
1  import {Injectable} from '@angular/core';
2  [Imports ...]
3
4  @Injectable()
5  export class AuthenticationInterceptor implements HttpInterceptor {
6    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
7      // Get the auth header from the service.
8      let header = req.headers.set('Content-Type', 'application/json');
9      if (AuthenticationService.isAuthenticated()) {
10       header = header.set('Authorization', `Bearer
             ${AuthenticationService.loginToken.accessToken}`);
11     }
12     // Clone the request to add the new header.
13     // Pass on the cloned request instead of the original request.
14     return next.handle(req.clone({headers: header}));
15   }
16 }
```

Listing 2.8: The Jukebox interceptor to automatically include a JWT Token.

The interceptor is a class which implements the `HttpInterceptor` interface and thus the

`intercept((req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any»`

method. In the Jukebox project, it was used to automatically include a JSON Web Token (JWT) as well as the „Content-Type“ header (Listing 2.8 on this page). With this technique it is also possible to catch any request which returns a 401 HTTP status code, get a new JWT access token and then retry the request without any service noticing that the initial request failed.

2.2 Angular Material

Angular Material is a framework, also developed by Google, which provides pre-built controls. These include buttons, trees, datepickers and many more. All of these controls are styled with the Google Material style guidelines in mind. To be able to use this framework, the developer has to install it via NPM or a similar package management tool. [Goo18b]

Components

Each control is an Angular component. This means that controls can be used in the HTML by including the corresponding tag, as shown in Listing 2.6 on page 10, Line 1. Any HTML tag which starts with the `mat-` prefix references an Angular Material control/component.

This prefix was changed from `md-` to the current `mat-`. At the same time, Angular Material changed how the components are organized into modules. Previously, there was a single module called `MaterialModule` which became deprecated and was replaced by dedicated modules for each single control, e. g. `MatButtonModule`. The Jukebox project currently uses 17 different controls, which results in having to import 17 different modules. To keep the app.moudle.ts file tidy, a custom module was created. This so-called `MaterialMetaModule` imports and exports all 17 „submodules“, so that the `AppModule` only has to import this one meta module.

Many controls like buttons or the toolbar are only referenced in the HTML because they don't contain much logic. These components are designed to communicate through their events and properties. This can be achieved through bindings in the HTML code (see subsubsection 2.1 Template Syntax - Bindings on page 5). Especially for dialogs, a developer wants to have more control over how and when the dialog is displayed. Dialogs are also invoked through the TS, for instance inside some method or event handler. Typically, the dialog contains some sort of question and the answer determines how the application should proceed. Since a dialog is a modal element which is always positioned as an overlay, it is unclear where the element should be placed in the DOM. Angular Material solves this by injecting the `MatDialog` into the component's TS file (Listing 2.9 on the next page, Line 14). This means that the placement in the DOM is done by the framework and the developer doesn't have to worry about it.

A filepicker dialog was implemented for the Jukebox project. This dialog serves the purpose of selecting a folder which the server should use to index *.mp3 files. Once the user clicks the „addPath“ button, a dialog is created. This is done via the `MatDialog.open()` method, as shown in Listing 2.9 on the following page, Line 23. The `MatDialog.open()` method requires the type of

an Angular component as its first argument. This determines the component to be displayed as the dialog's content. In addition, a configuration object may be passed to the method. The configuration object is of type `MatDialogConfig` and is used to configure the dialog's appearance. One of its properties is the `data` property which allows the developer to pass an arbitrary object into the dialog. This feature was used to pass a custom configuration object of type `FilePickerConfig` (Listing 2.9 on the current page, Line 24).

```
1  import {Component, OnInit} from '@angular/core';
2  [Imports...]
3
4  @Component({
5    selector: 'app-settings',
6    templateUrl: './settings.component.html',
7    styleUrls: ['./settings.component.css']
8  })
9  export class SettingsComponent implements OnInit {
10   private _isIndexing: boolean;
11   private _settingsService: SettingsService;
12   private _dialog: MatDialog;
13
14   constructor(settingsService: SettingsService, dialog: MatDialog) {
15     this._settingsService = settingsService;
16     this._dialog = dialog;
17   }
18 [...]
19   public addPath() {
20     let filePickerConfig = new FilePickerConfig();
21     filePickerConfig.selectDirectory = true;
22
23     this._dialog.open(FilePickerDialogComponent, {
24       data: filePickerConfig
25     }).afterClosed()
26       .mergeMap(() => {
27         if (filePickerConfig.selection.length < 1)
28           return Observable.empty<void>();
29
30         this._musicPaths.push(filePickerConfig.selection[0]);
31         return this._settingsService.setMusicPaths(this._musicPaths);
32       })
33       .subscribe();
34   }
35 [...]
36 }
```

Listing 2.9: Sample dialog in the `SettingsComponent`.

The `MatDialog.open()` method returns a reference to the newly created dialog which is of type `MatDialogRef<T>`. This class exposes certain methods to configure event handlers. Most notably, the `afterClosed()` and `afterOpen()` methods. These methods return `Observable<R | undefined>` and `Observable<void>`, respectively. The `afterClosed()` method was used to retrieve the selected path from the `FilePickerConfig` object and initiate an HTTP call to save this path.

```
1  import {Component, Inject, OnInit} from '@angular/core';
2  [Imports...]
3
4  @Component({
5    selector: 'app-file-picker-dialog',
6    templateUrl: './file-picker-dialog.component.html',
7    styleUrls: ['./file-picker-dialog.component.css']
8  })
9  export class FilePickerDialogComponent implements OnInit {
10   private readonly _config: FilePickerConfig;
11   private _filePickerService: FilePickerService;
12   private _dialogRef: MatDialogRef<FilePickerDialogComponent>;
13
14   constructor(
15     dialogRef: MatDialogRef<FilePickerDialogComponent>,
16     @Inject(MAT_DIALOG_DATA) config: FilePickerConfig,
17     filePickerService: FilePickerService) {
18       this._dialogRef = dialogRef;
19       this._config = config;
20       this._filePickerService = filePickerService;
21   }
22
23   public done() {
24     this._config.selection.push(this._selectedPath.directoryFullPath);
25     this._dialogRef.close();
26   }
27
28   public cancle() {
29     this._dialogRef.close();
30   }
31
32   [...]
33 }
```

Listing 2.10: Excerpt from the `FilePickerDialogComponent`.

Once the `MatDialog.open()` method is called, the required component will be created. During that process all necessary objects will be injected into this component. At this point, the developer can request dialog-specific objects next to the service classes which the component needs to function. A reference to the displaying dialog is necessary in almost any case. For this purpose, the developer asks for an object of type `MatDialogRef<T>` where `T` is the type of the displayed component (Listing 2.10 on this page, Line 15). With this reference, it is possible to close the dialog from within the TS code, as shown in line 25. If a configuration object was passed, its data can be accessed via the `@Inject()` decorator (Listing 2.10 on the current page, Line 16). For this to work the `MAT_DIALOG_DATA` argument must be passed into the decorator. [Goo18c]

Icons

Using intuitive icons is a big part of the visual design of any application. There are many ways how icons can be encoded and used, e. g. they can be images

or vector graphics. Angular Material uses another, also popular method called „Font Icons“. The same concept is used by FontAwesome [Fon18] and many others. A font usually displays some sort of letter but technically it is just a way of representing a scalable image. Icons are mostly used in buttons or menus which are all designed to work with some sort of font. Also, fonts are an old concept which means support for them is fairly good.

There are many ways to use the Material icon set. The Jukebox project mainly uses the `<mat-icon />` tag (see Listing 2.6 on page 10, Line 3), but it is also possible to associate CSS classes with specific icons or use the `<svg />` tag. It should be noted though, that the latter two examples are usually needed if the provided icons aren't sufficient or if e. g. another icon font is used in addition. The font of the Material icon set had to be included via `@font-face` CSS rule in oder to use the icon set. **??**

During the development of the Jukebox project, a bug was discovered which resulted in no icons being displayed. This only occurred in the Internet Explorer and Edge and was fixed by changing how icons are selected. Angular Material offers the possibility to use an icon's name to reference it inside the HTML tag e. g. `<mat-icon>menu</mat-icon>`. Another method is to use the unicode explicitly. This method has the drawback that it isn't obvious anymore what an icon represents or looks like by just looking at the HTML code. Some users in the community have created command line tools, which integrate into the Angular CLI or work as standalone scripts, which replace all names with the corresponding unicodes. Since this thesis is focused on compatibility between Electron and Angular, none of these tools were evaluated. **??** [Goo18d]

2.3 Websockets

The HTTP protocol was designed for web pages and is based on client-initiated communication. There are techniques like long polling which simulate a two-way communication channel but even there, the client needs to initialize the communication. This becomes a problem in web applications which rely on events or push notifications rather than on polling. In the Jukebox project, one requirement was that a music player could be remote-controlled. Websockets were used to push events like „Next“ or „Play/Pause“ to the player.

Websockets can be used by calling the `WebSocket.send()` and `WebSocket.close()` methods as well as listening to the various events. Another approach is to wrap the WebSocket into a RxJS `Subject`, as shown in Listing 2.11 on the next page [Les+18] [Moz18a]. To do so, the developer creates an `Observable` which binds to the events of the `WebSocket` object (lines 4 to 8). Then an `Observer` is created (line 11). In this example, the `Observer` is an object which has a `next()` method that is empty. This is due to the fact that in the Jukebox project notifications are only received through the websocket. This `next()` method

is normally used to call the `WebSocket.send()` method to send data back to the server. A RxJS `Subject` is created from the `Observer` and `Observable` objects. This `Subject` can then be subscribed to in order to receive incoming messages. To send messages the `Subject.next()` method may be used. send was used for the player control - clarify that **??**

```
private openNotificationSocket() {
  this._notificationSocket = new
      WebSocket(`${environment.websocketBaseUrl}/api/notification/ws`);

  let observable = Observable.create((obs: Observer<MessageEvent>) => {
    this._notificationSocket.onmessage = obs.next.bind(obs);
    this._notificationSocket.onerror = obs.error.bind(obs);
    this._notificationSocket.onclose = obs.complete.bind(obs);
    return this._notificationSocket.close.bind(this._notificationSocket);
  });

  let observer = { next: () => { } };

  this._subject = Subject.create(observer, observable);

  this._subject.subscribe((messageEvent: MessageEvent) => {
      this._otherSubject.next(JSON.parse(messageEvent.data))
    }
    , err => NotificationService.handleSocketError(err)
    , () => this.handleSocketCompleted());
}
```

Listing 2.11: Sample websocket usage.

3 Electron

Electron (formerly known as Atom Shell [Saw15]) is an open-source framework developed and maintained by GitHub [Git18d]. It is based on the Chromium V8 browser engine [Goo18f] and the Node.js JavaScript runtime [Nod18a]. The Electron framework was initially developed for the Atom text editor [Git18b] in 2013 and both were open-sourced in the beginning of 2014 [Git18a].

Electron enables a developer to build cross-platform applications which are written in HTML, CSS, and JS. Its API offers desktop integration with APIs such as the OSX and Windows notifications, the Windows taskbar, the Ubuntu launcher and many more. With this framework, it is possible to maintain a single code-base while serving all major desktop operating systems, as well as hosting a website on a traditional web server.

3.1 Electron's Architecture

Every Electron application has a main file which is the root or starting point for the application. This file is specified in the package.json configuration file with the key „main“. By convention, this file is called „main.js“, but any name is possible.

This main file is mostly used to set event handlers on the app object. These include handlers for the „ready“, „window-all-closed“, and „activate“ events (see Listing 3.1 on this page). Also, the event handler for the ready event usually contains code to create a `BrowserWindow` object and load some HTML. In the Jukebox project, the main.js file is used for the entire application startup sequence as well as for some Electron debugging methods.

```
1 app.on('ready', () => {
2   Menu.setApplicationMenu(menu);
3   setupWindowsNotifications();
4   startApi();
5   createWindow();
6   console.log(isSecondInstance);
7 });
```

Listing 3.1: The Jukebox app „ready“ event handler.

Electron uses the multi-process architecture of its underlying Chromium V8 engine. This means that the main.js file is called from the main process but every „web page“ is running in a separate window and process, which is called a renderer process [Git18e]. Figure 3.1 on the current page illustrates the multi-process architecture.

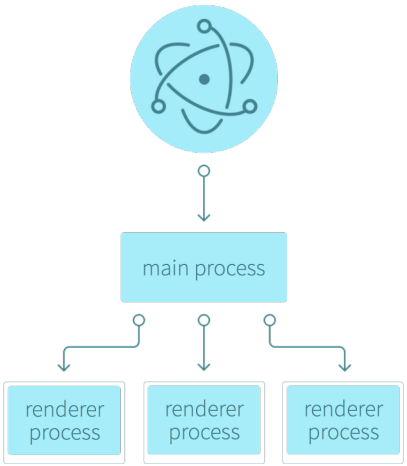

Figure 3.1: Overview on the architecture of Electron. [Pos15]

There is exactly one main process at all times. It is responsible for managing all of the renderer processes. A renderer process only knows its web page and is isolated from the rest of the application. It is thightly coupled to the `BrowserWindow` object. If the window is destroyed, the process is also terminated.

Inter-process Communication

As all renderer processes are isolated from the main process as well as each other, there needs to be a way to communicate data between those processes. This is done via the Electron Inter-process Communication (IPC) API which is split into the `ipcMain` and `ipcRenderer` objects, which are both instances of `EventEmitter`.

```
const {ipcMain} = require('electron')
ipcMain.on('asynchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.sender.send('asynchronous-reply', 'pong')
})

ipcMain.on('synchronous-message', (event, arg) => {
  console.log(arg) // prints "ping"
  event.returnValue = 'pong'
})
```

Listing 3.2: Handling messages with the ipcMain module. [Git18f]

The `ipcMain` is only accessible in the main process. It is used to handle asynchronous as well as synchronous messages sent from a renderer process (Listing 3.2 on the preceding page). To handle messages the `ipcMain.on()` method is used. Its first argument is a string which is referred to as „channel“ and used as a key. The same channel has to be used for the sender and the receiver. The second argument is a delegate which takes an event object as well as many arguments. This delegate will be executed every time a renderer process sends a message using the specified channel.

The `ipcMain` object also provides several other methods, e. g. a method to execute a listener only once, another method to remove a single or all listeners. As shown in Listing 3.2 on the previous page, the main process can reply via the `event.returnValue` property which results in a synchronous message. It could also send a new message to the renderer which would be an example of an asynchronous communication. The `ipcMain` object does not contain a send method like the `ipcRenderer`, since it would not be clear where to send the message. Instead, the `webContents` object of a given `BrowserWindow` provides this method. This is the same object held by the `event.sender` property. [Git18f]

The `ipcRenderer` object is only available inside a renderer process and is similar to the `ipcMain` object. The only notable differences are the `send` methods. It is possible to send asynchronous as well as synchronous messages. The `ipcRenderer.send()` method always sends the message to the main process. If the message needs to be sent to a different renderer process, the `ipcRenderer.sendTo()` method may be used. [Git18g]

IPC was used in the Jukebox project to quit the application from the taskbar item. This was neccessary since the `app` object is only available within the main process, but the system tray is set up inside an Angular service class which runs in a renderer process. Also, IPC may be used if results of a dialog have to be communicated back to the main application.

The App Object

Electron's `app` object provides access to the application's lifecycle events, as well as properties which hold information about the application. This includes methods to manipulate the Windows jumplist in order to add items to the taskbar context menu. It is possible to import certificates, get application metrics, access the OSX dock, or get the application's root folder path. In the Jukebox project, this was used for the notifications and system tray functionalities and will be described in more detail in the corresponding sections.

Node

Node.js is a JS runtime environment which offers libraries for a variety of typical desktop application tasks. The Jukebox project uses mostly the „fs“ and „os“ libraries.

The „fs“ library grants access to the filesystem [Nod18b]. It is used to check that ?? the Windows start menu folder for this application exists and if not, create it (Listing 3.8 on page 29, Line 7). Without the ASP.NET Core server this library would be used to access the music files.

The „os“ library provides access to operating system specific information and methods [Nod18c]. It is used to determine the current operating system and architecture to be able to register the correct notification libraries.

3.2 Accessing Electron APIs from within Angular

One of the challenges in the Jukebox project was accessing the Electron APIs from within Angular. This access was needed for setting the menus and their corresponding `click()` handlers, as well as showing notifications or determining the current environment.

require('electron')

The first approach was to try and `import` or `require()` the Electron package. This, however, failed due to configuration errors. Angular can be configured via system.js or webpack, but the developer rarely gets to see or manipulate these files, since they are generated by the Angular CLI during the build only. It is possible to „eject“ the webpack configuration file and change it to be able to `require()` the Electron package. However, this is rather complicated and creates problems with other packages, since the Angular CLI does not automatically adapt the configuration file anymore. [Soh17]

ngx-electron

The alternative to `require('electron')` is to use the ngx-electron package [Han18]. This package provides an Angular service, the `ElectronService`, which is exposed as an interface and contains properties for all of the Electron APIs.

This package checks the user agent property to determine whether the application is running inside Electron. If it is, the Electron package is `required()`. If not, all properties of the `ElectronService` return `undefined`, except for the `isElectronApp` property. This property returns a `boolean` and makes it

easy to check whether the application is running inside Electron or as a web page in some browser.

3.3 Menus

A big difference between web pages and traditional desktop applications is the placement of menus. Web applications often integrate menus into the web page, sometimes as navigation bars, sometimes at prominent locations in the layout. For desktop applications the placement of the menu is rather clear. For Windows applications it is a menu bar right under the top border the application window and for Ubuntu (Unity) and OSX it is also a menu bar, but located at the top of your screen.

Application Menu

For the Jukebox project, a custom `MenuItem` class was created: the `AngularMenuItem`. This is due to the fact that a menu is needed for both the web version as well as the desktop version. With this approach, the custom class can be used for the entire project and only a small part, namely the `MenuItemService`, depends on the Electron APIs. Since Electron is doing some initialization, it is not possible to use a custom `MenuItem` object, even if the custom object has the same signature. Therefore, a converter method was implemented, as shown in Listing 3.3 on the next page. This method creates an instance of `Electron.MenuItem` for each `AngularMenuItem` recursively.

maybe say something more about this **??**

```
public createElectronMenuItem(menuItem: AngularMenuItem): Electron.MenuItem {
  let subItems = [];

  menuItem.submenu.forEach(value => {
    subItems.push(this.createElectronMenuItem(value));
  });

  let item = new this._electronService.remote.MenuItem({
    label: menuItem.label,
    click: () => menuItem.click(),
    submenu: subItems.length > 0 ? subItems : null,
    type: menuItem.type,
    enabled: menuItem.enabled,
    visible: menuItem.visible,
    id: menuItem.id,
    position: menuItem.position,
    accelerator: menuItem.accelerator
  });

  let visibleChangedSub = menuItem.visibleChanged.subscribe(() => {
    let oldMenuItem = this._electronService.remote.Menu
                        .getApplicationMenu().getMenuItemById(menuItem.id);
    if (oldMenuItem != null)
      oldMenuItem.visible = menuItem.visible;
  });

  let enabledChanged = menuItem.enabledChanged.subscribe(() => {
    let oldMenuItem = this._electronService.remote.Menu
                        .getApplicationMenu().getMenuItemById(menuItem.id);

    if (oldMenuItem != null)
      oldMenuItem.enabled = menuItem.enabled;
  });

  menuItem.subscriptions.push(visibleChangedSub, enabledChanged);
  return item;
}
```

Listing 3.3: Excerpt of the Jukebox `MenuItemService` class.

System Tray

Another style of a menu is the system tray and its context menu. Especially for applications which are supposed to run in the background and only show a notification when an event occurs, the ability to minimize the application to the system tray is very important. Electron supports this feature with the `Electron.remote.Tray` API.

The `Electron.remote.Tray` object takes a `Electron.remote.NativeImage` or a `string` as constructor arguments. Both variants were tried, but the `string` variant did not show the icon, even though the system tray was there and clickable. Once the system tray object is created, a menu may be set.

```
win.on('close', (event) => {
  if (!isQuitting) {
    event.preventDefault();
    win.hide();
  }
  else {
    win = null;
    app.quit();
  }
}
```

Listing 3.4: The modified `BrowserWindow.close()` event.

In the Jukebox project, the system tray is used to minimize the application without quitting it when all windows are closed. To quit the application, a „Quit“ menu item was added to the system tray. For this purpose, the `BrowserWindow.close()` event had to be modified, as shown in Listing 3.4 on this page. The „Quit“ menu item's `click()` handler then uses IPC to tell the main process to quit the application via a call to `app.quit()`.

```
let trayMenuTemplate = [
  {
    label: 'Jukebox APP',
    id: 'app',
    click: () => this._win.show(),
    position: 'first'
  },
  {
    label: 'Quit Jukebox',
    click: () => {
      this._electronService.ipcRenderer.send('quitApplication');
    },
    position: 'last',
    id: 'quit'
  }

];
```

Listing 3.5: Example of a menu template.

In the Jukebox project, the system tray menu is implemented using templates, as shown in Listing 3.5 on the current page. This means an array of anonymous objects is created which all follow a certain signature. The properties of these objects are then read by the Electron framework and an `Electron.MenuItem` is created using this blueprint. The advantages are that the developer only has to specify the properties needed for this menu item and the framework can ignore additional properties. Therefore, it is possible to use existing custom objects in this array, without the need for implementing interfaces.

Challenges while Implementing Menus

One of the challenges encountered was routing from a menu item's `click()` handler. Many menu items are used to navigate through the application e. g. to get to the settings page. In Angular, routing is usually done via the `Router.navigateByUrl()` method. If the `click()` handler of a menu item was set to just this line of code, navigation would occur, but the old page was still displayed and the new page was added underneath.

Angular uses zone.js to enable its `<router-outlet />` feature to exchange parts of the web page at runtime and manage different execution contexts [Goo18e]. Since the `click()` event handler is called from the Electron framework, it is called from a different context, which causes this faulty behavior.

To fix this situation, a reference to the `NgZone` object is needed, which can be obtained through injection. The `click()` handler then has to be modified to run the navigation code in the corresponding zone, as shown in Listing 3.6 on this page.

```
1 this._zone.run(() => this._router.navigateByUrl('/auth/register'))
```

Listing 3.6: Fixed `click()` handler.

Another issue is that of top level menu items not behaving correctly. This is a known and ongoing issue of the Electron framework. It results in top level items ignoring the change of values of the `enabled` and `visibile` properties. Currently, there is no fix for this bug. [Cao17]

3.4 Notifications

Electron uses the HTML 5 notification API to send basic notifications to the user. This API works on all operating systems. proofread **??**

The globally available `Notification` object is used to manage (it doesnt do it by itself) **??** the permission to display notifications for each web page. This is done through the static `Notification.permission` property which returns one of the three following values: „granted“, „denied“, or „default“. The „default“ value tells the developer that the user's choice is unknown at this moment. In this state, the browser behaves as if the user denied permission, which results in showing no notifications at all. Using the static `Notification.requestPermission()` method, the user is asked whether he wants to see notifications or not.

Once permission is granted, a notification can be displayed, as shown in Listing 3.7 on the next page. The HTML 5 notification API offers many more properties on a notification object, but these are not implemented in all browsers

and therefore should not be relied on. The `appId` property is not part of the HTML 5 notification API, but is needed for the Windows operating system. If the property is omitted Windows will not show any notification at all proof-read **??**. The property can be used in non-Windows environments without risk because the constructor uses templates and the property will simply be ignored komma? **??** when it is not needed. [Moz18b]

```
1 private static displayHTMLNotification(notificationOptions:
      UserNotificationOptions) {
2   new Notification(notificationOptions.title, {
3     body: notificationOptions.body,
4     icon: notificationOptions.icon,
5     appId: environment.appId
6   });
7 }
```

Listing 3.7: Displaying a basic HTML notification.

Simple Windows Notifications

Under Windows, the „electron-windows-notifications“ [Rie18b] and „electron-windows-interactive-notifications“ [Rie18a] packages can be used to create visually sophisticated, interactive notifications.

The „electron-windows-notifications“ package offers two kinds of notifications. The first type is called `TileNotification`. This notification updates the primary or secondary tiles of the application. In this case the application must be run inside the Universal Windows Platform (UWP) model. The second type is called `ToastNotification`. This is the „standard“ notification which is usually displayed on the bottom right corner of the screen and can later on be found in the Windows Action Center.

A prerequisite for showing any kind of notification is that the application has set up an `appId`. Configuring this `appId` is a multi-step procedure. First, a start menu folder has to be created for the application. Then, a shortcut to the application has to be placed in this folder. During testing, it was discovered that it does not matter where this shortcut points to in order to show the notification. If, however, the shortcut points to the wrong file, the click action on the notification might not work as expected. Next, the „System.AppUserModel.ID“ data store property of the shortcut must be set to the `appId` [Bis17]. For debugging purposes it is possible to tweak the windows registry so it shows the „System.AppUserModel.ID“ in the file explorer details view [Kal13]. During the startup sequence of the application, the `app.setAppUserModelId()` method has to be called. Finally **??**, the `appId` property must be set for every notification object, be it the `ToastNotification`, `TileNotification`, or the HTML 5 `Notification`. The `appId` is an arbitrary

`string` which should be unique. A Globally Unique Identifier (GUID) was used for the Jukebox project.

The notification package uses the „nodert-win10-au“ native Node module to call native Windows APIs. The `ToastNotification` and `TileNotification` objects are used similarly to their corresponding C++ classes. Most notably, the notifications are constructed using an Extensible Markup Language (XML) template. This template allows the placement of banner images, cropped images, and many more visual elements in the notification. **??**

Although many features of the Windows 10 notification API work, not all do and some only do in certain circumstances. The Windows Dev Center (WDC) page, e. g. lists a progress bar feature which does not work at all. Also, there are many ways to display images e. g. as app logo override, as hero image, or as inline image. Microsoft offers three different protocols to reference an image: „http://“, „ms-appx:///“, and „ms-appdata:///“. The „http://“ protocol did not show an image, while the „ms-appx:///“ and „ms-appdata:///“ worked, but only if the application was run inside a UWP model. [Mic17f]

Interactive Windows Notifications

Windows notifications also support buttons, comboboxes, and inline replies. These interactive elements are not supported by default and the additional „electron-windows-interactive-notifications“ package is needed.

This package works by registering a Component Object Model (COM) server. Whenever a user interacts with a notification, the COM server is called with the information on the action chosen by the user and **??** any related data. The COM server then encodes this data into a string and calls the application via a protocol link.

At installation time the package requires two parameters. The first parameter to be set is called „TOAST_ACTIVATOR_CLSID“. This needs to be a unique string and it is strongly recommended to use a GUID. According to the package's documentation, this can be achieved through setting the CLSID in the package.json configuration file or setting the „TOAST_ACTIVATOR_CLSID“ environment variable. Tests showed that the package.json method does not work [Mor18]. Second, the application protocol has to be set. This is possible via package.json and environment variables and here, the package.json does not work either. Both of these values have to be set at installation time, since a DLL will be compiled during the installation of the package. In the DLL these two values are saved as constants.

During the startup sequence of the Electron app, both the COM server and the „TOAST_ACTIVATOR_CLSID“ have to be registered. This can be achieved with the `registerActivator()` and `registerAppForNotificationSupport()` methods, respectively, as shown in Listing 3.8 on the following page. To register

the application as default protocol handler, the `app.setAsDefaultProtocol-Client()` method may be used. This method requires the protocol name as well as a path to the executable as arguments.

```
function setupWindowsNotifications() {
  if (os.platform() !== 'win32')
    return;

  const {registerAppForNotificationSupport, registerActivator} =
      require('electron-windows-interactive-notifications');

  if (!fs.existsSync(shortcutFolder))
    fs.mkdirSync(shortcutFolder);
  registerAppForNotificationSupport(shortcut, appId);
  registerActivator();
}
```

Listing 3.8: Registration of the electron-windows-interactive-notifications package.

Whenever the operating system calls a protocol link, the application registered for that protocol is started. This means a new instance of the application is executed. In most cases this is not the desired behavior. In order to communicate the information back to the first instance a communication channel is needed.

```
const isSecondInstance = app.makeSingleInstance((argv) => {
  if (win == null)
    return;
  win.webContents.send('protocolActivation', argv[1]);
  console.log(argv[1]);
});

if (isSecondInstance) {
  setTimeout(() => {
    isQuitting = true;
    app.quit();
  }, 1000)
} else {
  // Create window on electron intialization
  app.on('ready', () => {
    Menu.setApplicationMenu(menu);
    setupWindowsNotifications();
    startApi();
    createWindow();
    console.log(isSecondInstance);
  });
}
```

Listing 3.9: Jukebox startup code to handle multiple instances.

Electron makes this possible with the `app.makeSingleInstance()` method. This method returns a `boolean` which is `false`, if the current instance is the

only one running, and is `true`, if not. Also, this method takes a delegate which is executed only if another instance of the application is present. This delegate will be executed within the context of the main instance and is used to communicate any information transmitted via the protocol link (Listing 3.9 on the previous page).

Mac

For macOS the „node-mac-notifier“ package may be used. This package allows the developer to specify the `canReplay` property which is of type `boolean`. When set to true the user will be presented with a text input field which may be used for a reply. Apart from this reply feature the „node-mac-notifier“ package offers no other features.

As of Electron v2 the documentation states that more functionality of the HTML5 API is implemented, but only for macOS. Now the `actions` property on the `Notification` object may be used. An action may have a label and must be a button. When more than one action is specified for a notification, only the first one is shown. To see all available actions the user will have to hover the mouse over the first action. If the `hasReply` property is set to true no actions will be shown. Furthermore, the app needs to be signed and have its `NSUserNotificationAlertStyle` set to `alert` in the `Info.plist`. This could not be tested due to the very recent update of the Electron framework.

Linux

There are no special packages available for Linux. Notifications will be shown using the „libnotify“ library which can show notifications on any desktop environment that follows the Desktop Notification Specifications [HHM18] [Git18i].

4 ASP.NET Core

ASP.NET Core is a cross-platform, open-source framework for building cloud- and web-based applications. It is the latest of Microsoft's web frameworks. Although the name is very similar to the ASP.NET framework, ASP.NET Core is not its 5th version, but a new framework rewritten from the ground up. It combines the previously independent ASP.NET MVC and ASP.NET Web API frameworks into one single programming model. It uses the new .NET Core framework which is platform-independent, but projects may also target the Windows-only .NET framework. [Mic18d]

NuGet

The ASP.NET Core framework relies on the NuGet Package Manager for its modularity. The NuGet Package Manager is very similar to Node's NPM. The packages to be used in a project can either be configured using the command line, or directly editing the *.csproj file, or using the Graphical User Interface (GUI) tools most .NET Core Integrated Development Environments (IDE)s provide. This makes it possible to download the necessary parts of the ASP.NET Core framework only. Microsoft also allows third parties to publish libraries on its NuGet platform. This was also done in the Jukebox project for some of the middlewares.

OWIN

Open Web Interface for .NET (OWIN) is another big aspect of ASP.NET Core's modularity. OWIN is not a specific implementation, but rather a specification which helps developers to create a modular architecture and was used to create the ASP.NET Core framework. Katana is the codename Microsoft gave its OWIN implementation and should not be confused with OWIN itself.

The most important concept for a developer is that of a middleware. In the OWIN specification, a middleware is defined as an exchangeable programming block with the following signature: `Func<IDictionary<string, object>, Task>`. This means a middleware is a delegate which is given an `IDictionary` object containing key-value-pairs and which has to return a `Task` object. By using a `Task`, Microsoft enables programmers to take advantage of the Task Parallel Library (TPL) and its asynchronous programming model [Mic17g].

Middlewares are usually provided by the ASP.NET Core framework, or they can be created by the developer for specific tasks. In order to create a middleware the developer has two options. He may simply create a delegate with the correct signature. Since this is similar to functional programming, Microsoft also provided a method to use classes. When a class is used, the ASP.NET Core framework will check whether this class contains a method which matches the signature of a middleware. Using this method, the developer may take advantage of DI for any dependencies the custom middleware requires. [Att15]

Another concept defined by the OWIN standard is the middleware pipeline. Every HTTP request will traverse this pipeline and be processed along the way. A request is often referred to as context and is contained in the `IDictionary` object of the middleware delegate. Depending on the circumstances, a middleware might alter the request and call the next middleware, or complete the request. Each request starts at the first middleware. Yet not every request reaches the last middleware, since it might be processed before it gets to the end. Once a request is processed, it travels back to the beginning of the pipeline. This is illustrated in Figure 4.1 on this page.

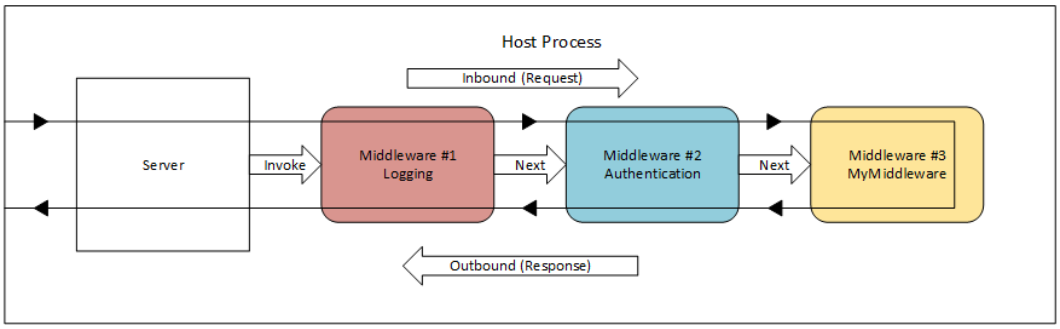

Figure 4.1: Example of an OWIN middleware pipeline. [Att15]

The `Program` class

Like any other C# project, an ASP.NET Core project is rooted in the `Program` class and its `Main()` method. As of .NET Core 2.x, the code required to start a `WebHost` has been simplified, as shown in Listing 4.1 on the next page [Mic18a]. In the Jukebox project, the Uniform Resource Locator (URL) is explicitly defined. It is still possible to configure options like the Internet Information Services (IIS) integration, a custom content root path, and many more through the `WebHostBuilder` class. Any configuration through the `WebHostBuilder` will overwrite the configuration specified **??** via the launchSettings.json file.

The underlying web server of ASP.NET Core is called Kestrel. It is a minimalistic web server which only provides three features. Those are HTTPS, Websockets, and Unix sockets for high performance behind Nginx. Kestrel

should never be used as a standalone solution, but rather in combination with another web server like IIS, Nginx or Apache [Mic18e].

```
public static IWebHost BuildWebHost(string[] args)
{
    return WebHost.CreateDefaultBuilder(args)
                  .UseUrls("http://0.0.0.0:5000")
                  .UseStartup<Startup>()
                  .Build();
}
```

Listing 4.1: The `Main()` method of the Jukebox project.

The `Startup` Class

The `Startup` class is used to configure an ASP.NET Core application. It is separated into two methods: the `ConfigureServices()` and the `Configure()` method. This class is called `Startup` by convention, but may be named differently. If a another name is chosen, the developer has to specify it via the `WebHostBuilder.UseStartup<T>()` method as shown in Listing 4.1 on this page. The `Startup` class does not implement any interfaces. The ASP.NET Core framework detects the required methods via reflection. This is necessary since these methods have multiple applicable signatures, which are handled differently by the framework.

DI is an integral part of the ASP.NET Core framework and is configured in the `ConfigureServices()` method. This method has one parameter of type `IServiceCollection` which is an interface similar to the Autofac `ContainerBuilder` class. In order to make classes and framework parts available, such as the Model-View-Controller (MVC) functionality of ASP.NET Core, they have to be registered using this interface.

The `IServiceCollection` interface provides many features which are sufficient for most projects, but it is not possible to configure interception using this interface. If the developer needs or wants to use interception, he has to change the signature of the `ConfigureServices()` method to return an `IServiceProvider` interface. This interface is Microsoft's abstraction of a `Container` class which is used to resolve dependencies. The Jukebox project uses the Autofac DI framework was used, because it supports all of ASP.NET Core's DI interfaces [Aut18b]. It should be noted that it is also possible to use other DI frameworks, but custom adapters may have to be implemented [Mic18c].

The Jukebox project uses a dual approach. ASP.NET Core components are registered via the `IServiceCollection` interface because extension methods are provided by the framework. All other classes are registered using the `ContainerBuilder` class to enable interception. Then all registrations from

the `IServiceCollection` interface are transferred to the `ContainerBuilder` using the `Populate()` method. Finally, a `IServiceProvider` is created using the `AutofacServiceProvider` class. This is illustrated in Listing 4.2 on the current page. warum abgeschnitten ? **??**

```
public static IServiceProvider ConfigureJukebox(this IServiceCollection services,
                                                 IConfiguration config)
{
  services.ConfigureServices(config);

  var builder = new ContainerBuilder();
  builder.ConfigureContainerBuilder(config);

  builder.Populate(services);
  builder.ConfigureControllers();
  return new AutofacServiceProvider(builder.Build());
}
```

Listing 4.2: Configuring the `IServiceCollection` in combination with an Autofac `ContainerBuilder`.

Once all components are registered, the ASP.NET Core framework calls the `Configure()` method. This method is used to build the OWIN middleware pipeline. Its first argument is of type `IApplicationBuilder` and is required. The second and third arguments are of type `IHostingEnvironment` and `ILoggerFactory`, respectively, and are optional.

The `IApplicationBuilder` interface is used to set up the pipeline, as shown in Listing 4.3 on the following page. ASP.NET Core provides extension methods for all of the framework's middlewares. There are multiple options to register a custom middleware. The `IApplicationBuilder.Run()` method can be used to specify a delegate which will run and terminate the pipeline at this point. In order to chain middlewares the `IApplicationBuilder.Use()` method can be used. This method also takes a middleware delegate as parameter. The `IApplicationBuilder.UseMiddleware<T>()` method is used for middlewares which are written as a class. [Mic18b]

It is important to bear in mind that the middleware which is registered first, will always be called first by the framework for each request that is processed. This means that the authentication middleware should always be registered before the MVC middleware, otherwise requests might not be authenticated properly. In the Jukebox project, the `ExceptionMiddleware` is registered first since it is supposed to catch any exception thrown in the pipeline later on. Next, the `Swagger` and `SpaMiddleware` are added because they do not require authentication. These are followed by the `Authentication`, MVC, and `WebSocket` middlewares. Finally, the `StaticFiles` middleware is added.

```
public virtual void Configure(IApplicationBuilder app,
                              IHostingEnvironment env,
                              ILoggerFactory loggerFactory)
{
    app.ApplicationServices.GetService<DataContext>().Database.Migrate();
    app.UseExceptionMiddleware();
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
    app.UseSwaggerUi3(typeof(Startup).GetTypeInfo().Assembly);
    app.UseSpaMiddleware();
    app.UseAuthentication();
    var websocketOptions = app.ApplicationServices
                              .GetService<IOptions<WebsocketOptions>>()
                              .Value;
    app.UseWebSockets(new WebSocketOptions
                    {
                        KeepAliveInterval = websocketOptions.KeepAliveInterval,
                        ReceiveBufferSize = websocketOptions.BufferSize
                    });
    app.UseMvc();
    app.UseStaticFiles();
}
```

Listing 4.3: Building the OWIN middleware pipeline.

4.1 ASP.NET MVC Core

Middlewares are a great concept, but developing an entire application on the basis of just middlewares is very tedious. To make the development of web applications easier ASP.NET Core offers the pre-built MVC middleware. This middleware allows the developer to write his code in separate, independent classes which know nothing about OWIN or the concept of a middleware. As the name suggests, using the MVC middleware encourages the developer to implement his classes following the Model-View-Controller pattern. The MVC package has to be registered with the DI container, as shown in Listing 4.4 on page 37.

MVC is a popular pattern in software development and consists of three main components: Model, View, and Controller (Figure 4.2 on the following page). The model represents the state of the application and any business logic or operations needed. All visual elements and any logic to manipulate these elements are considered the view. In the ASP.NET Core context the Razor view engine is used to embed .NET code in HTML markup. This allows for easy manipulation of the view. The controller is responsible for handling user interactions. It decides which view to render, and which methods to call in order to manipulate the model. In case of the Jukebox project, the view part was not used since all view-related actions are located inside the Angular code. [Mic18g]

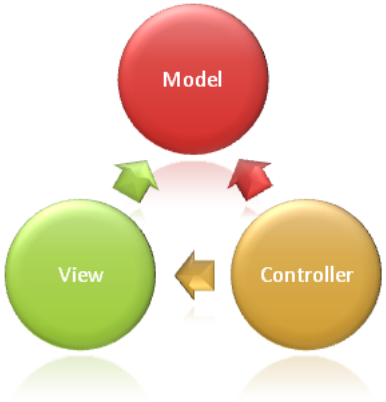

Figure 4.2: Main components of the MVC pattern. [Mic18g]

Jukebox MVC configuration

The `AddMvc()` extension method offers the possibility to configure the MVC middleware. In the Jukebox project, authorization handling and controller registration are configured explicitly.

To prevent unauthorized access, the MVC package offers many ways to protect certain routes, also called actions. A developer may accept the default security settings or choose between filters and attributes to protect certain routes. It is important to note that the MVC package does not authenticate or authorize any request at all. Another middleware is used specifically for this purpose. The JWT Bearer middleware is the one used in the Jukebox project (see subsubsection 4.2 JWT Bearer Token Authentication on page 39). MVC only checks the `HttpContext` to know whether and how a request is authorized, and then acts accordingly. By default, every route is unprotected. However, certain actions should only be executable if the request is authorized. There are two common solutions to this problem. First, a blacklist approach, where every route is accessible by default and the protected ones are declared explicitly. In this case, the developer uses the `[Authorize]` attribute on any controller method which needs to be protected. The second option is to use a whitelist approach, where every route is protected by default and the unprotected routes are specified explicitly. As shown in Listing 4.4 on the following page, a filter is created using the `AuthorizationPolicyBuilder` class and is applied to all routes. Then, all routes which shall be accessible without authorization are marked with the `[AllowAnonymous]` attribute. The whitelist approach is used in the Jukebox project.

There are two ways of registering controllers for the MVC package. By default, the MVC package registers all classes with the DI container that are derived from the abstract `Controller` base class and are located in the „Controllers“ folder. If more functionality, such as interception, is needed, the

`AddControllersAsServices()` configuration option can be used, see Listing 4.4 on the current page, Line 8. This option forces the developer to register all controller classes explicitly. For the controllers registered in the Jukebox project, an interceptor is configured. This interceptor checks for null references passed to controller methods. If such a null reference is detected, the interceptor will throw a specific exception which results in a 400 HTTP response (see subsubsection 4.3 ExceptionMiddleware on page 43).

```
1  private static IServiceCollection ConfigureMvc(this IServiceCollection services){
2      //Creating Global Authorization Filter
3      var globalAuthFilter = new AuthorizationPolicyBuilder()
4                             .RequireAuthenticatedUser()
5                             .Build();
6
7      services.AddMvc(options => { options.Filters.Add(new
           AuthorizeFilter(globalAuthFilter)); })
8              .AddControllersAsServices();
9      return services;
10 }
```

Listing 4.4: Registering the MVC package.

Mapping URLs

Mapping a URL to a controller action works either by convention or by explicit configuration. If nothing is specified, the convention is to map a URL to the controller's name followed by the method's name. For example the „/Home/Authors“ URL is mapped to the `HomeController` and its `Authors()` method. Custom conventions may be specified when the MVC middleware is added to the middleware pipeline. As shown in Listing 4.5 on the current page, a default route is created, which maps to the `HomeController` and its `Index()` method.

```
1 app.UseMvc(routes =>
2 {
3    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
4 });
```

Listing 4.5: A custom route mapping convention. [Mic17d]

Attributes are another way of explicitly configuring mappings. The `[Route]` attribute is used for controllers only. This attribute was used in the Jukebox project, since all controllers are supposed to be part of the „api/“ route. The `[Route]` attribute also offers macros. With macros it is possible to use the same attribute for all controllers: `[Route("api/[controller]/[action]")]`. The `[controller]` macro tells the framework to use the name of the controller which is decorated by the `[Route]` attribute. Similarly, the `[action]` macro references the name of the method or its custom-specified route.

For methods, also called actions, there are several attributes which are derived from the abstract `HttpMethodAttribute` class. There is a matching attribute for each HTTP verb, e. g. `[HttpPost]`, `[HttpGet]`, and so forth. Their main purpose is to specify the HTTP verb, in addition they include a template option to configure the MVC mapping. This is neccessary to decide which method should be called on the controller if two URLs only differ in their HTTP verbs. A custom route is specified as a parameter in the HTTP verb attribute, e. g. `[HttpPost("my/custom/route")]`. Also, many Representational State Transfer (REST) APIs use integers in the URL to reference a business object by its ID. This is supported through the curly bracket notation. A possible scenario is: some method should be executed on the `Player` object with ID 11. This could be mapped to a URL like „/api/player/11/doSomething“ and would result in an attribute such as `[HttpPost("{playerId}/doSomething")]`. [Mic17c]

Transfering Data

Especially in case of a web API, the transfer of data is very important. The most common options to transfer data via a HTTP request are: sending the data in the HTTP body, embedding the data into the URL using a query string, or inserting IDs into the URL as described in subsubsection 4.1 Mapping URLs on the preceding page. For small bits of information like a `string` or `int`, the query string method is recommended. The `[FromQuery]` attribute tells ASP.NET Core that a method argument should be mapped to a query string argument, see Listing 4.6 on the current page, Line 4. In this case the annotated argument's name must match the key in the query string.

For the transfer of objects or large amounts of data the HTTP body should be used. In this case the `[FromBody]` attribute indicates that data from the HTTP body shall be parsed, as shown in Listing 4.6 on this page, Line 2. Usually, the data is transferred as a JavaScript Object Notation (JSON) string, but it is also possible to use XML. ASP.NET Core automatically tries to parse the JSON string into an object of the specified type, in this example an object of type `PlayerCommand`.

```
1 [HttpPost("{playerId}/executeCommand")]
2 public virtual Task ExecutePlayerCommand([FromBody] PlayerCommand cmd
3                                          , int playerId
4                                          , [FromQuery] string playerName)
```

Listing 4.6: Different data transfer methods in ASP.NET Core.

For the Jukebox project, many classes like the `PlayerCommand` class were written. These classes are referred to as a Data Transfer Object (DTO) and are usually `sealed` classes which only contain properties and constructors. Each DTO is annotated with the `[DataContract]` attribute and every property to

be serialized is annotated with the `[DataMember]` attribute. Some properties are used to convert data from one type to another. These are not annotated with the `[DataMember]` attribute and are therefore ignored during serialization. This works well for most data types, the only exception are enums. Enums are serialized as integers by default, because the integer is the underlying type of a default enum in C#. The default behavior can be changed with the `[JsonConverter(typeof(StringEnumConverter))]` attribute to serialize an enum as a string. This improves readability on the front-end part and provides compatibility for the TS implementation of an enum. [Mic18f]

4.2 Security in ASP.NET Core

Securing an application against data leaks and malicious activity is especially important for a web application as it is accessible to a wide range of people. In ASP.NET Core the security aspect consists of two topics: authentication and authorization.

ASP.NET Core Authentication

Authentication is the process in which a client proves that he is the one he claims to be. Traditionally, a client will provide a username and a password. The server will check them against a database and decide whether the client has provided valid credentials or not. This basic form of authentication is possible in ASP.NET Core and is mostly referred to as „cookie authentication“. Its benefit is its simplicity, but the drawback is its inflexibility. With the ASP.NET Core Identity package the developer can choose between using a simple username-password combination or an external login provider such as Facebook, Google, Microsoft Account, Twitter, or others. This approach utilizes the OAuth 2.0 [Par18] or OpenID Connect [Ope18] standards. [Mic17a]

JWT Bearer Token Authentication

For the Jukebox project, the JSON Web Token (JWT) Bearer authentication was chosen [Aut18a]. JWT is an open standard for securely transmitting information in a self-contained, compact JSON object. The information is secured either using the symmetric keyed-Hashing for Message Authentication (HMAC) algorithm [Sta99], or one of the asymmetric algorithms, namely Rivest-Shamir-Adleman (RSA) or Elliptic Curve Digital Signature Algorithm (ECDSA) [Max14]. JWT was chosen because not only does it yield a token the server uses to authenticate a client, but it also transmits authorization information. This is especially important because the front-end can use this

information to hide parts of the application depending on the user's level of authorization.

Configuring the JWT package is a multi-step procedure. First, the `TokenValidationParameters` are set, as shown in Listing 4.7 on the current page. Then, a `JwtTokenOptions` object is configured (Listing 4.8 on the following page) and a method for creating a JWT token is implemented (Listing 4.9 on page 42). Finally, the JWT package is added to the middleware pipeline using the `app.UseAuthentication()` extension method.

For the `TokenValidationParameter`, a `SymmetricSecurityKey` object is created which must be identical to the key object that is used to validate a JWT token later on. Then, an „Issuer" and an „Audience" are set, both are optional. This serves the purpose of identifying the application (issuer) and the client (audience). Finally, the option to validate the lifetime of a token is set to `true` and the clock skew is set to zero. [Bar16]

```
1  private static IServiceCollection ConfigureAuthService(this IServiceCollection
       services){
2      var signingKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(SecretKey));
3      var tokenValidationParams = new TokenValidationParameters
4                                  {
5                                      // The signing key must match!
6                                      ValidateIssuerSigningKey = true,
7                                      IssuerSigningKey = signingKey,
8                                      // Validate the JWT Issuer (iss) claim
9                                      ValidateIssuer = true,
10                                     ValidIssuer = "JB_AUTHORITY",
11                                     // Validate the JWT Audience (aud) claim
12                                     ValidateAudience = true,
13                                     ValidAudience = "JB_AUDIENCE",
14                                     // Validate the token expiry
15                                     ValidateLifetime = true,
16                                     // If you want to allow a certain amount of clock
                                           drift, set that here:
17                                     ClockSkew = TimeSpan.Zero
18                                 };
19
20     services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
21             .AddJwtBearer(options => { options.TokenValidationParameters =
                   tokenValidationParams; });
22
23     services.AddSingleton(tokenValidationParams);
24     return services;
25 }
```

Listing 4.7: Configuration of the JWT package.

The `JwtTokenOptions` object was written specifically for the Jukebox project, it stores information needed to create new JWT tokens. It consists of properties for the „Issuer" and the „Audience". Furthermore, there are properties which define the expiration of the JWT token, as well as the expiration of its refresh token. Finally, the `JwtTokenOptions` object contains the

`SigningCredentials` which consist of the key and an enum to determine the algorithm to be used. As shown in Listing 4.8 on this page, the HMAC algorithm is used in the Jukebox project.

```
builder.RegisterInstance(new JwtTokenOptions
{
   Issuer = "JB_AUTHORITY",
   Audience = "JB_AUDIENCE",
   Expiration = TimeSpan.FromHours(1),
   RefreshTokenExpiration = TimeSpan.FromDays(30),
   SigningCredentials = new SigningCredentials(signingKey,
       SecurityAlgorithms.HmacSha256)
});
```

Listing 4.8: Configuration of the `JwtTokenOptions`.

The last step in setting up the JWT authentication package is to create a method which issues a new token. This method, shown in Listing 4.9 on the next page, is part of the `JwtAuthenticationService` in the Jukebox project. A `ClaimsIdentity` object and a `JwtTokenOptions` object are passed into the method. `ClaimsIdentity` is the C# object to manage a user in a claims-based authorization system, as described in subsubsection 4.2 Claims-Based Authorization on the following page. First, a list of claims is instantiated. For this list, the JWT Subject „Sub“, JWT ID „Jti“, and Issued At „Iat“ claims are created. The JWT subject claim usually contains the username or email address. The JWT ID claim contains a unique identifier, in the Jukebox project a GUID is used. The Issued At claim contains the current timestamp in Universal Time Coordinated (UTC) time. Then all claims stored in the `ClaimsIdentity` are added to the claims list. These are claims defined by the web application and could contain a claim which grants access to the administrative section of the application. Then, a `JwtSecurityToken` object is created. The `JwtTokenOptions` are used to set the expiration, the issuer, and so on. Next, a refresh token is created. For the sake of simplicity, the Jukebox project uses a GUID as a refresh token. Finally, an `AuthToken` DTO object is created which is used to send the information back to the client. So far, the `JwtSecurityToken` is a C# object which still needs to be serialized into a JSON string. The serialization is handled by the `JwtSecurityTokenHandler.WriteToken()` method. Both the `JwtSecurityToken` and the `JwtSecurityTokenHandler` classes are provided by the JWT package.

```
public static AuthToken CreateBasicAuthToken(ClaimsIdentity identity,
    JwtTokenOptions options){
  var now = DateTime.UtcNow;
  var claims = new List<Claim>();
  claims.Add(new Claim(JwtRegisteredClaimNames.Sub, identity.Name));
  claims.Add(new Claim(JwtRegisteredClaimNames.Jti, options.NonceGenerator()));
  claims.Add(new Claim(JwtRegisteredClaimNames.Iat,
      now.ToUnixEpochDate().ToString(), ClaimValueTypes.Integer64));
  claims.AddRange(identity.Claims);

  var token = new JwtSecurityToken(options.Issuer,
                                   options.Audience,
                                   claims.ToArray(),
                                   now,
                                   now.Add(options.Expiration),
                                   options.SigningCredentials);

  var refreshToken = Guid.NewGuid().ToString();
  var refreshTokenExpiration = now.Add(options.RefreshTokenExpiration);

  return new AuthToken
        {
            RefreshToken = refreshToken,
            AccessToken = new JwtSecurityTokenHandler().WriteToken(token),
            AccessToken_ValidUntil = now.Add(options.Expiration),
            RefreshToken_ValidUntil = refreshTokenExpiration
        };
}
```

Listing 4.9: Generating the JWT Token.

Claims-Based Authorization

After a client is authenticated, the JWT token will be read and all claims will be extracted. These claims are used to check whether the client is authorized to execute an action. A claim describes an individual right or action this client possesses. It can also be proof of a piece of information about the client, such as his email address, birth date, or phone number.

Two custom claims are implemented in the Jukebox project. First, the `UsernameClaim` which is used to store the username of a client. Second, the `RoleClaim`. This claim is used to identify a client as „SystemAdmin“, „IndexAdmin“, or „PlayerAdmin“ and grants permission to execute certain actions. Claims will be serialized when they are sent to the client inside the JWT token. To identify a claim after it has been serialized, the claim needs a unique type which is saved as a string. By convention, a URL is used. For example the „http://Jukebox/Claims/Security/Username“ string is assigned to the `UsernameClaim` type in the Jukebox project. [Bus06]

4.3 Custom Middlewares

The Jukebox project uses two custom middlewares. One is the `ExceptionMiddleware` which translates custom exceptions thrown further down the middleware pipeline into HTTP responses. The other is the `SpaMiddleware` which is needed to enable deep links into a SPA.

ExceptionMiddleware

The `ExceptionMiddleware`, shown in Listing 4.10 on the current page, is the first middleware in the OWIN pipeline. This middleware encloses the rest of the pipeline in a try-catch block which catches any `InvalidRestOperationExceptions` that are thrown. The `InvalidRestOperationException` class is an abstract base class for exceptions like the `NoContentException` (HTTP 204), `ForbiddenException` (HTTP 403), and many more. Every exception derived from the `InvalidRestOperationException` is required to implement the abstract `ResponseCode` property which specifies the HTTP status code to be returned. Furthermore, the `InvalidRestOperationException` contains a `CustomErrorCode` property. This property is of type GUID and has to be set via a constructor argument.

Once such an exception is thrown, it will be caught in the catch block of the middleware. In this catch block, the `StatusCode` property of the `HttpContext.Response` object is set to the `ResponseCode` property of the `InvalidRestOperationException`. Then, an `ExceptionDTO` object will be created which contains the error message and error GUID. Although the exception class is already marked as `[Serializable]`, this extra object is used in order to transmit only the error message and GUID - contrary to an exception which is serialized with its stack trace and other debug information not relevant in this scenario. Finally, the `ExceptionDTO` will be serialized to a JSON string and sent to the client.

```
1  public async Task Invoke(HttpContext context)
2  {
3      try
4      {
5          await _next(context);
6      } catch (InvalidRestOperationException invalidRestOperationException)
7      {
8          context.Response.StatusCode = invalidRestOperationException.ResponseCode;
9          await context.Response.WriteAsync(JsonConvert.SerializeObject(new
               ExceptionDTO(invalidRestOperationException)));
10     }
11 }
```

Listing 4.10: The `ExceptionMiddleware`.

SpaMiddleware

ASP.NET Core is intended for use in combination with the Razor view engine to create traditional web applications. A Single Page Application (SPA) works quite differently, because there only exists one HTML file for the entire application instead of one file for every subsite. Additionally, the front-end framework, in this case Angular, oftentimes comes with its own routing component. This means that a user might see a URL like „http://jukebox/account/settings“. This URL is used by the front-end framework to display the settings component, but the back-end knows nothing about such a URL and therefore would return a HTTP 404 error code.

These URLs are called deep links and are used by many web applications. The `SpaMiddleware` is a piece of code which enables the deep link feature for web applications hosted via ASP.NET Core. It has to be added to the middleware pipeline before the MVC middleware and, since no authentication is needed, should also be added before any authentication middleware. As shown in Listing 4.11 on the following page, the middleware checks every request's URL to determine whether it is a deep link or not.

Whether a URL is a deep link or not is determined by a set of rules. Most importantly, the middleware checks whether the request has a file extension or not. This is needed because the index.html file references other files such as the bundle.js file. Furthermore, a configuration object is used. This configuration object stores paths which are not deep links because they are part of another middleware or feature of the web server. The Jukebox project uses the Swagger web API documentation tool which can be accessed through the URL „http://jukebox/swagger“ [Sma18]. Therefore, the Swagger URL is not recognized as a deep link but as a „SpecialRoute“, which causes the middleware to pass the request further down the pipeline.

When a deep link is found the middleware sets the HTTP status code to 200 OK and returns the index.html file. First attempts to solve this problem used a redirect mechanism to serve the index.html. This worked to the extent that the index.html file was served, but failed overall, because the deep link was lost due to the redirect. Since redirecting is not possible, the `SpaMiddleware` needs to know where the index.html file is located. This can be achived by convention or explicit configuration. The middleware checks the ASP.NET Core ContentRoot folder whether it contains a file called „index.html“ without distinguishing between upper and lower case. If no file is found and none is specified through the `PathToIndex` property on the configuration object, an exception is thrown.

```
public async Task Invoke(HttpContext context)
{
    if (!_options.SpecialRoutes.Any(x =>
         context.Request.Path.StartsWithSegments(x))
        && !Path.HasExtension(context.Request.Path.Value))
    {
        context.Response.StatusCode = (int) HttpStatusCode.OK;
        await context.Response.WriteAsync(File.ReadAllText(_options.PathToIndex));
    }
    else
    {
        await _next(context);
    }
}
```

Listing 4.11: The `SpaMiddleware`.

4.4 Entity Framework Core

To manage its users and song index, the Jukebox project uses the Entity Framework Core (EF Core) [Mic16b]. EF Core is an Object Relational Mapper (ORM) framework which enables developers to work with databases using .NET objects and programming paradigms like Fluent API. It also eliminates the need to write Structured Query Language (SQL) code. EF Core features many different database providers to access My Structured Query Language (MySQL), Microsoft Structured Query Language (MSSQL), SQLite, and many other database engines.

SQLite

For the Jukebox project, a datastore was needed, but an entire database server would have been too unwieldy in its configuration and maintenance. Therefore, the SQLite database engine was chosen [SQL18]. SQLite is a self-contained, file-based database engine which does not require any server application to connect to. Compared to a MSSQL or a MySQL database engine, there are some limitations [Mic17e], but these features are not needed in the Jukebox project.

Configuration

When configuring a database with EF Core, the developer may choose between the „Model-First-Approach“ and the „Code-First-Approach“. In the „Model-First-Approach“ an Entity Relationship Diagram (ER Diagram) is created and saved as a *.edmx file. The developer can manipulate this model via GUI tools. The *.edmx file is then used to create a class structure which represents the

database. This approach, however, requires an IDE which supports *.edmx modeling files. Also, it lacks flexibility because the classes created from the model are not meant to be edited, yet the GUI tools oftentimes limit functionality.

The „Code-First-Approach“ is more widely used and also the recommended way of configuring a database with EF Core [Mic16a]. This approach is more tedious, because the developer will have to implement all classes instead of having them generated by the EF Core code generation tool. The advantage is a far greater control over the model. This approach offers the developer three options for the configuration of his database: Annotations, Fluent API, and/or accepting the default conventions.

The easiest way is to accept the default conventions. For example, EF Core will use properties which are called „Id“ as a primary key, create foreign keys for properties of types registered in the `DbContext`, and use the class name as the table name [Ent18]. This method is appropriate for small databases or test dummies.

The second option is to use DataAnnotations (Listing 4.12 on the current page). These are attributes placed on properties which instruct the framework to handle these properties in a special way. Examples for annotations are the `[Key]`, `[ForeignKey]`, and `[NotMapped]` attributes. This method works well for configuring columns and tables.

```
1  public class Song{
2      [Key]
3      public int Id { get; set; }
4      public string FilePath { get; set; }
5      public string Title { get; set; }
6
7      [ForeignKey(nameof(Album))]
8      public int AlbumId { get; set; }
9      public Album Album { get; set; }
10     public DateTime LastTimeIndexed { get; set; }
11
12     [NotMapped]
13     public SongSource SourceType { get; set; } = SongSource.CustomBackend;
14 }
```

Listing 4.12: Configuration of the `Song` class using DataAnnotations.

The third method is to use the Fluent API (Listing 4.13 on the following page). The `DbContext` abstract base class contains a protected virtual method called `OnModelCreating()`. This method has to be overwritten to use the Fluent API. Its only argument is of type `ModelBuilder` and is used to access the `EntityTypeBuilder<T>` configuration objects. These `EntityTypeBuilder<T>` objects are then used to configure properties as primary keys, set the table name, configure the `CascadeOnDeletion` option, and many other settings. Using the Fluent API, anything related to the database can be configured, but

the drawback is its obscurity since a developer cannot know how an entity is configured by simply looking at the entity class.

```
private static void ConfigureSong(EntityTypeBuilder<Song> builder)
{
    builder.HasIndex(x => x.FilePath)
           .IsUnique();

    builder.HasOne(x => x.Album)
           .WithMany(x => x.Songs)
           .HasForeignKey(x => x.AlbumId);
}
```

Listing 4.13: Configuration of the `Song` class using Fluent API.

A mixture of all three approaches has proven to be more practical than sticking to just one. The default conventions are used to recognize properties which are called „Id“ as primary keys. Properties which are not part of the database, but part of the C# class, are annotated with the `[NotMapped]` attribute and everything else is configured using the Fluent API.

Migrations

An application's data model will grow during the development of the application. EF Core supports changes in the data model with its migrations feature. A new migration can be created using the command line tools with the command `dotnet ef migrations add <migrationName>`. Similiarly, the `remove` command can be used to remove a migration. With the `add` command EF Core will create a migration class and then update the model snapshot class. The model snapshot class is used as a base line for comparison when a new migration is added. The created migration class contains code which describes the differences between the old model, as detailed in the snapshot, and the current model, as specified in the class structure. Every migration is derived from the abstract base class `Migration` and has to implement the abstract `Up()` and `Down()` methods. As shown in Listing 4.14 on the next page, the `Up()` method contains instructions on how to apply the migration and the `Down()` method contains instructions on how to revert the migration. [Mic17b]

There are two ways to apply one or more migrations. The first solution is to use the EF Core command line tools to directly migrate the database using the command `dotnet ef database update`. This is usually the preferred method, but may not be feasible depending on the setup of the database. When direct migration is not possible, the EF Core command line tools can also be used to create a SQL script. This is achieved using the command `dotnet ef migrations script`. Both methods offer the option to revert the database by specifying the `From` and `To` migrations explicitly.

One limitation of the SQLite database engine is that a table or column cannot be dropped once it has been created. Therefore, the Jukebox project only contains one migration. This limitation prevents a series of migrations which include drop operations. The practical solution to this problem is to create a new database each time a migration would occur. This is not problematic as long as an application is under development, but as soon as it is used productively, a migration strategy will have to be developed separately.

```
public partial class Initial : Migration {
  protected override void Up(MigrationBuilder migrationBuilder) {
    migrationBuilder.CreateTable(
       name: "Albums",
       columns: table => new
       {
           Id = table.Column<int>(nullable: false)
               .Annotation("Sqlite:Autoincrement", true),
           ArtistId = table.Column<int>(nullable: false),
           Name = table.Column<string>(nullable: true)
       },
       constraints: table =>
       {
           table.PrimaryKey("PK_Albums", x => x.Id);
           table.ForeignKey(
               name: "FK_Albums_Artists_ArtistId",
               column: x => x.ArtistId,
               principalTable: "Artists",
               principalColumn: "Id",
               onDelete: ReferentialAction.Cascade);
       });
       [...]
  }
  protected override void Down(MigrationBuilder migrationBuilder) {
    migrationBuilder.DropTable(
               name: "Albums");
    [...]
  }
```

Listing 4.14: Excerpt of a migration class.

4.5 Architecture

Modularity is a key aspect when developing any application. In ASP.NET Core, modularity can be gained by several methods, one of which are middlewares, as discussed in subsubsection 4 OWIN on page 31. Although using middlewares is a good start, there are many other parts of an application which can be modularized, for example separating database code, business logic, and validation logic from each other.

Project structure

The back-end C# Solution, shown in Figure 4.3 on the next page, contains six different projects. They are classified as database projects, framework projects, acceptance tests, and the ASP.NET Core project.

The database projects contain one project with specific SQLite database classes and another one with in-memory database helper classes. Both projects contain neither domain classes nor the abstract data context because those classes are used by all databases and should therefore be located in a separate project. Using a separate project for database engine specific classes makes it easy to support another engine later on. This is due to the fact that a migration is engine-specific and the EF Core command line tools do not support multiple engines in one project. It is possible to use a single set of migrations for different database engines, but it requires the developer to write the migrations on his own.

The framework solution folder contains the `Jukebox.Common` and `Jukebox.Common.Abstractions` projects. All interfaces for business and validation logic classes are part of the `Jukebox.Common.Abstractions` project. Furthermore, the domain classes and the abstract data context are part of this DLL. All business and validation implementations are contained in the `Jukebox.Common` project. This separation makes it possible to swap implementations without breaking too many dependencies.

Every good application should be tested to a certain degree to ensure that it works correctly. The Jukebox project does not contain many tests, because it is a proof-of-concept application. However, acceptance tests are easy to write and maintain, and thus provide an economic compromise between development time and quality.

Finally, the `JukeboxAPI` project contains the ASP.NET Core specific code, like controller classes and the `Startup` and `Program` classes. This is the only project which references any ASP.NET Core specific DLLs.

Services and Validation

Another step towards a modularized application is the use of a service layer. The service layer handles all the business logic the application needs to function. An example for this is the `IPlayerService`, shown in Listing 4.15 on the following page. This service contains methods for creating a `Player` object, adding a song to a `Player` object, executing a command on a certain player, or getting details about one specific or all players available.

For each domain there is one service class. These services are accessed through constructor injection and are used in controllers as well as other services. Each service and all of the controllers use the TPL and the C# Async and Await

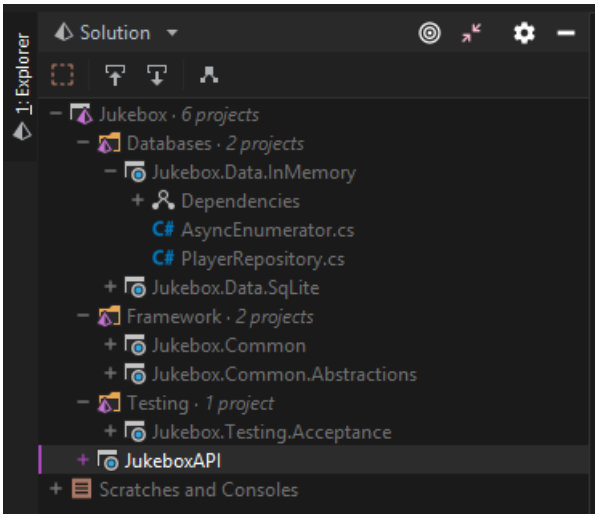

Figure 4.3: C# project structure.

pattern [Mic17g]. This pattern simplifies asynchronous programming provided that the programmer abides by certain rules. Once the developer chooses to use the Async and Await pattern, every method should return a `Task` object which in consequence means that no part of the application should use a synchronous programming model anymore. A `Task` should never be awaited using the `Result` property or `Wait()` method, because these may cause deadlocks. Instead, the `await` keyword should be used.

```
1  namespace Jukebox.Common.Abstractions.Players
2  {
3      public interface IPlayerService
4      {
5          Task<IEnumerable<Player>> GetAllPlayersAsync();
6          Task<Player> GetPlayerByIdAsync(int playerId);
7          Task CreateSocketPlayerAsync(WebSocket socket);
8          Task AddSongToPlayerAsync(int playerId, int songId);
9          Task ExecuteCommandAsync(int playerId, PlayerCommand cmd);
10     }
11 }
```

Listing 4.15: The `IPlayerService` interface.

Validation is an important aspect of any application. The back-end has to verify that the data sent by a client is structurally and semantically correct and that the client is authorized to execute the action. This is done by interception in the Jukebox project. Each service has an associated interceptor as well as an associated validator class. The interceptor decides which aspects of a method have to be validated and the validator class contains the logic for the validation. This separation is necessary because the interceptor is tightly coupled to the service it is intercepting, while the validator should only be concerned with the object it is supposed to validate. The `IPlayerService`

interface contains a method called `AddSongToPlayerAsync()` (Listing 4.15 on the previous page). This method will be intercepted and the interceptor has to validate that both the player and the song with the given ID do exist. For this purpose the interceptor holds a reference to the `PlayerValidator` and the `SongValidator` objects which contain the needed methods for validation, as shown in Listing 4.16 on this page. If the validation methods were to reside within the interceptor, this scenario would not be possible.

```
1 public Task AddSongToPlayerAsync(int playerId, int songId){
2     _playerValidator.ValidatePlayerExists(playerId);
3     _songValidator.ValidateSongExists(songId);
4     return null;
5 }
```

Listing 4.16: Intercepting the `AddSongToPlayerAsync` method.

4.6 ASP.NET Core Websockets

To use websockets in ASP.NET Core, the websocket middleware must be added to the middleware pipeline. Here, the developer can configure settings like the `KeepAliveInterval` and the `ReceiveBufferSize`. Whether a request is a request for a websocket or not, can be detected using the `IsWebSocketRequest` property on the `HttpContext.WebSockets` object. When such a request is detected, the communication can be established using the `AcceptWebSocketAsync()` method on the `HttpContext.WebSocket` object. This method returns a `Task<WebSocket>` of which the `WebSocket` is used for the entire communication. [Mic18h]

After accepting the socket, the Jukebox ASP.NET Core server will authenticate the client and, if the authentication was successful, create a `Player` object. Since the websocket connection will be terminated once the request is finished, the server will use a „keep-alive loop“, as shown in Listing 4.17 on the next page. This loop will wait for any message coming from the player. Once a message arrives, the server checks whether the client wants to terminate the session or has sent data. In case of termination, the websocket will be closed and the `Player` object will be destroyed. If the client has sent data, the server will handle the message in its `HandlePlayerMessage()` method.

```
private async Task HandlePlayerOwnerWebsocket(Player player, WebSocket socket) {
    try
    {
        while (true)
        {
            var buffer = new byte[_websocketOptions.BufferSize];
            var result = await socket.ReceiveAsync(new ArraySegment<byte>(buffer),
                CancellationToken.None);

            if (result.CloseStatus.HasValue)
            {
                await socket.CloseAsync(result.CloseStatus.Value,
                    result.CloseStatusDescription, CancellationToken.None);
                throw new Exception("Trigger catch block");
            }

            HandlePlayerMessage(Encoding.ASCII.GetString(buffer), player.Id);
        }
    } catch
    {
        _playerRepository.RemoveByPlayerId(player.Id);
    }

}
```

Listing 4.17: The „keep-alive loop“ for a websocket.

5 Results and Conclusion

The Jukebox project makes use of most of Electron's prominent features. These include the notification API, the menu API, the system tray API, Electron's IPC components, and a number of smaller features. None of these features is completly unusable given this specific combination of frameworks. Adaptations and third-party libraries were needed to make the communication between Electron's APIs and Angular's services work (section 3.2 Accessing Electron APIs from within Angular on page 22). A solution for connecting an Electron menu item with a click event handler which calls Angular APIs (section 3.3 Menus on page 23) is neither documented in the Electron nor in the Angular manual. Research provided little information on this topic, apparently this use case is not very common. Compared to the challenges posed by the menu API, Electron's IPC API works very well, even the communication back and forth between an Angular service inside a renderer process and Electron's main process is simple to implement and works as described in the documentation. In conclusion, even though not everything works without further adjustment, it is possible to use this combination of frameworks to create a platform-independent application.

Compared to a pure web application, the hybrid Electron approach does require more architectural decisions. It has to be considered whether the application shall be hosted as a web page in addition to being hosted through Electron or not. Electron's APIs abstract most of the low level APIs provided by the operating systems. Nonetheless, there are still APIs like the notification API which require different implementations for each operating system. They can be abstracted, but the developer will have to implement these abstraction layers because they are not provided by the framework. Additionally, if the application is hosted in a web environment, some APIs might not be accessible because they require special hardware or other components not available there. Therefore, these functionalities require a modularized approach in order to easily exclude them in a web environment. One such example from the Jukebox project are menus. When hosted as an Electron application, Electron's menu API is used. The web page version of the application uses Angular menu items. One solution is to always use Angular menu items because they are available in both environments. However, the Jukebox project uses an Angular service to abstract menu items. This service then checks the environment and applies the correct menu items for each environment.

The obvious benefit of this hybrid approach as opposed to a pure web application is its offline usability. The main advantage, howerever, is the set of

numerous and more comprehensive features available. Although some of these features require native node modules, the possibility to implement them at all is well worth the effort involved. Many native node modules are provided through various web portals. For other, more specific use cases, a custom implementation is needed.

A reason for using a pure desktop application might be to avoid performance problems. This is due to the fact that it is difficult to implement multithreading using the Electron framework. JS generally runs in one single thread. Libraries like RxJS might give the impression to be multithreaded because they use terminology like awaitables, and so forth, but this is not the case. They still work with just one thread and use an event loop to simulate multithreading. It is possible to create more than one thread which is usually referred to as a „Web Worker“, but there are no means to synchronize two threads using locks, semaphores, or monitors [Par11]. Electron's documentation also discourages the use of native node modules inside a web worker because most of the native node modules were not designed to work in a multithreaded environment [Git18h]. Although performance is generally speaking an important point, many applications will work well within these limitations. It is also possible to use Electron for most of the application and move only the performance-critical code into a native node module, or, as it was done in the Jukebox project, into an ASP.NET Core C# back-end.

The biggest benefit of using Electron is the single code base for all operating systems. Many traditional desktop application frameworks are limited to a single platform and none have the additional benefit of also producing a web page. (many web developers)

Bibliography

[Att15] John Atten.
ASP.NET: Understanding OWIN, Katana, and the Middleware Pipeline.
4.01.2015.
URL: http://johnatten.com/2015/01/04/asp-net-understanding-owin-katana-and-the-middleware-pipeline/ (visited on Aug. 1, 2018).

[Aut18a] Auth0 Inc.
Introduction to JSON Web Tokens.
2018.
URL: https://jwt.io/introduction/ (visited on Aug. 7, 2018).

[Aut18b] Autofac.
Autofac ASP.NET Core Documentation.
2018.
URL: https://autofaccn.readthedocs.io/en/latest/integration/aspnetcore.html (visited on Aug. 2, 2018).

[Bar16] Nate Barbettini.
ASP.NET Core Token Authentication Guide.
2016.
URL: https://stormpath.com/blog/token-authentication-asp-net-core (visited on Aug. 7, 2018).

[Bis17] Gregor Biswanger.
Notification API do not work with Windows 10 16299.19 (fall creators update).
2017.
URL: https://github.com/electron/electron/issues/10864 (visited on July 26, 2018).

[Bus06] Michele Leroux Bustamante.
Fundamentals of WCF Security: Claims-Based Identity Model.
2006.
URL: https://www.codemag.com/article/0611051 (visited on Aug. 7, 2018).

[Cao17] Mike Cao.
Menu enabled/visible property doesn't affect top-level menu items.
2017.
URL: https://github.com/electron/electron/issues/8703 (visited on July 25, 2018).

[Che17] Ryan Chenkie.
Angular Authentication: Using the Http Client and Http Interceptors.
2017.
URL: https://medium.com/@ryanchenkie_40935/angular-authentication-using-the-http-client-and-http-interceptors-2f9d1540eb8 (visited on Aug. 16, 2018).

[Ent18] EntityFrameworkTutorial.net.
Conventions in Entity Framework Core.
2018.
URL: http://www.entityframeworktutorial.net/efcore/conventions-in-ef-core.aspx (visited on Aug. 9, 2018).

[Fon18] Fonticons Inc.
Font Awesome.
2018.
URL: https://fontawesome.com/ (visited on July 13, 2018).

[Fra18] Fractal Innovations.
gulp.js.
2018.
URL: https://gulpjs.com/ (visited on Aug. 21, 2018).

[Git18a] GitHub.
About Electron.
2018.
URL: https://electronjs.org/docs/tutorial/about (visited on July 23, 2018).

[Git18b] GitHub.
Atom.
2018.
URL: https://atom.io/ (visited on July 23, 2018).

[Git18c] GitHub.
autoUpdater.
2018.
URL: https://electronjs.org/docs/api/auto-updater (visited on Aug. 21, 2018).

[Git18d] GitHub.
Electron.
2018.
URL: https://electronjs.org/ (visited on July 23, 2018).

[Git18e] GitHub.
Electron Application Architecture.
2018.
URL: https://electronjs.org/docs/tutorial/application-architecture (visited on July 24, 2018).

[Git18f] GitHub.
Electron ipcMain.
2018.
URL: https://electronjs.org/docs/api/ipc-main (visited on July 24, 2018).

[Git18g] GitHub.
Electron ipcRenderer.
2018.
URL: https://electronjs.org/docs/api/ipc-renderer (visited on July 24, 2018).

[Git18h] GitHub.
Multithreading.
2018.
URL: https://electronjs.org/docs/tutorial/multithreading (visited on Aug. 20, 2018).

[Git18i] GitHub.
Notifications (Windows, Linux, macOS).
2018.
URL: https://electronjs.org/docs/tutorial/notifications (visited on July 26, 2018).

[Goo18a] Google.
Angular Component Documentation.
2018.
URL: https://angular.io/api/core/Component (visited on June 7, 2018).

[Goo18b] Google.
Angular Material.
2018.
URL: https://material.angular.io (visited on July 12, 2018).

[Goo18c] Google.
Angular Material Dialog.
2018.
URL: https://material.angular.io/components/dialog/overview (visited on July 13, 2018).

[Goo18d] Google.
Angular Material Icons.
2018.
URL: https://v5.material.angular.io/components/icon/overview (visited on July 13, 2018).

[Goo18e] Google.
Angular zone.js.
2018.

URL: https://github.com/angular/zone.js/ (visited on July 25, 2018).

[Goo18f] Google.
Chromium.
2018.
URL: https://www.chromium.org/Home (visited on July 23, 2018).

[Goo18g] Google.
Hierarchical Dependency Injectors.
2018.
URL: https://angular.io/guide/hierarchical-dependency-injection#the-injector-tree (visited on Aug. 16, 2018).

[Goo18h] Google.
Template Syntax.
2018.
URL: https://angular.io/guide/template-syntax (visited on June 21, 2018).

[Han18] Thorsten Hans.
ngx-electron.
2018.
URL: https://github.com/ThorstenHans/ngx-electron (visited on July 25, 2018).

[HHM18] Mike Hearn, Christian Hammond, and William Jon McCann.
Desktop Notifications Specification.
2018.
URL: https://developer.gnome.org/notification-spec/ (visited on July 26, 2018).

[Kal13] Gaurav Kale.
How to show more details for shortcuts in Windows 8.1, Windows 8 and Windows 7.
2013.
URL: https://winaero.com/blog/how-to-show-more-details-for-shortcuts-in-windows-8-1-windows-8-and-windows-7/ (visited on July 26, 2018).

[Les+18] Ben Lesh et al.
RxJS.
2018.
URL: https://rxjs-dev.firebaseapp.com/ (visited on July 18, 2018).

[Max14] Maxim Integrated Products Inc.
The Fundamentals of an ECDSA Authentication System.
2014.
URL: https://www.maximintegrated.com/en/app-notes/index.mvp/id/5767 (visited on Aug. 7, 2018).

[Mic16a] Microsoft.
Creating a Model.
2016.
URL: https://docs.microsoft.com/en-us/ef/core/modeling/index (visited on Aug. 9, 2018).

[Mic16b] Microsoft.
Entity Framework Core.
2016.
URL: https://docs.microsoft.com/en-us/ef/core/ (visited on Aug. 9, 2018).

[Mic17a] Microsoft.
Custom storage providers for ASP.NET Core Identity.
2017.
URL: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity-custom-storage-providers?view=aspnetcore-2.1.

[Mic17b] Microsoft.
Entity Framework CoreMigrations.
2017.
URL: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/ (visited on Aug. 13, 2018).

[Mic17c] Microsoft.
Routing to controller actions in ASP.NET Core.
2017.
URL: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-2.1 (visited on Aug. 7, 2018).

[Mic17d] Microsoft.
Routing to controller actions in ASP.NET Core: Setting up Routing Middleware.
2017.
URL: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-2.1#setting-up-routing-middleware (visited on Aug. 6, 2018).

[Mic17e] Microsoft.
SQLite EF Core Database Provider Limitations.
2017.
URL: https://docs.microsoft.com/en-us/ef/core/providers/sqlite/limitations (visited on Aug. 9, 2018).

[Mic17f] Microsoft.
Toast Content.
2017.

URL: https://docs.microsoft.com/en-us/windows/uwp/design/shell/tiles-and-notifications/adaptive-interactive-toasts (visited on July 26, 2018).

[Mic17g] Microsoft.
Task-based Asynchronous Programming.
30.03.2017.
URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming (visited on Aug. 1, 2018).

[Mic18a] Microsoft.
ASP.NET Core fundamentals.
2018.
URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-2.1&tabs=aspnetcore2x (visited on Aug. 2, 2018).

[Mic18b] Microsoft.
ASP.NET Core Middleware.
2018.
URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-2.1&tabs=aspnetcore2x#creating-a-middleware-pipeline-with-iapplicationbuilder (visited on Aug. 2, 2018).

[Mic18c] Microsoft.
Dependency injection in ASP.NET Core.
2018.
URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.1 (visited on Aug. 2, 2018).

[Mic18d] Microsoft.
Introduction to ASP.NET Core.
2018.
URL: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.1 (visited on Aug. 1, 2018).

[Mic18e] Microsoft.
Introduction to Kestrel web server implementation in ASP.NET Core.
2018.
URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?tabs=aspnetcore2x&view=aspnetcore-2.1 (visited on Aug. 2, 2018).

[Mic18f] Microsoft.
Model Binding in ASP.NET Core.
2018.

URL: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.1 (visited on Aug. 7, 2018).

[Mic18g] Microsoft.
Overview of ASP.NET Core MVC.
2018.
URL: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview (visited on Aug. 6, 2018).

[Mic18h] Microsoft.
WebSockets support in ASP.NET Core.
2018.
URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-2.1 (visited on Aug. 15, 2018).

[Mor18] Jan Morawietz.
package.json config gets ignored.
2018.
URL: https://github.com/felixrieseberg/electron-windows-interactive-notifications/issues/6 (visited on July 26, 2018).

[Moz18a] Mozilla.
JavaScript Websocket API.
2018.
URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket (visited on July 18, 2018).

[Moz18b] Mozilla.
Notification Documentation.
2018.
URL: https://developer.mozilla.org/en-US/docs/Web/API/notification/Notification (visited on July 26, 2018).

[Nod18a] Node.js Foundation.
Node.js.
2018.
URL: https://nodejs.org/en/ (visited on July 23, 2018).

[Nod18b] Node.js Foundation.
Node.js fs Documentation.
2018.
URL: https://nodejs.org/api/fs.html (visited on July 24, 2018).

[Nod18c] Node.js Foundation.
Node.js os Documentation.
2018.
URL: https://nodejs.org/api/os.html (visited on July 24, 2018).

[Nod18d] Node.js Foundation.
Node.js v10.9.0 Documentation: child_process.spawn(command[, args][, options]).
2018.
URL: https://nodejs.org/api/child_process.html#child_process_child_process_spawn_command_args_options (visited on Aug. 16, 2018).

[Ope18] OpenID.
Welcome to OpenID Connect.
2018.
URL: http://openid.net/connect/ (visited on Aug. 7, 2018).

[Par11] Parallel programming / Synchronization using JavaScript Web Workers.
MatthewJ; Scravy.
2011.
URL: https://stackoverflow.com/questions/8333676/parallel-programming-synchronization-using-javascript-web-workers (visited on Aug. 20, 2018).

[Par18] Aaron Parecki.
OAuth 2.0.
2018.
URL: https://oauth.net/2/ (visited on Aug. 7, 2018).

[Pos15] Kristian Poslek.
Building a desktop application with Electron.
2015.
URL: https://medium.com/developers-writing/building-a-desktop-application-with-electron-204203eeb658 (visited on July 24, 2018).

[Rie18a] Felix Rieseberg.
Electron Windows Interactive Notifications.
2018.
URL: https://github.com/felixrieseberg/electron-windows-interactive-notifications (visited on July 26, 2018).

[Rie18b] Felix Rieseberg.
Electron Windows Notifications.
2018.
URL: https://github.com/felixrieseberg/electron-windows-notifications (visited on July 26, 2018).

[Saw15] Kevin Sawicki.
Atom Shell is now Electron.
2015.
URL: https://electronjs.org/blog/electron (visited on July 23, 2018).

[Sma18] SmartBear Software.
Swagger: The Best APIs are Built with Swagger Tools.
2018.
URL: https://swagger.io/ (visited on Aug. 9, 2018).

[Soh17] Sohail.
How to use electron APIs inside angular2 app.
2017.
URL: https://discuss.atom.io/t/how-to-use-electron-apis-inside-angular2-app/41674/12 (visited on July 25, 2018).

[SQL18] SQLite.
SQLite.
2018.
URL: https://www.sqlite.org/index.html (visited on Aug. 9, 2018).

[Sta99] William Stallings.
The HMAC Algorithm.
1999.
URL: http://www.drdobbs.com/security/the-hmac-algorithm/184410908 (visited on Aug. 7, 2018).

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum	Unterschrift