Development and Deployment of Web Applications as Installable Desktop Applications Using Electron Framework

FH Campus 02

Maximilian Martin Wolf

30 October 2021

Contents

1 Introduction

1.1 What Is Electron?

In December 2012 software engineer Cheng Zhao joined GitHub's team, having previously worked for Intel developing node-webkit, with the task of porting the Atom editor from using Chromium Embedded Framework to node-webkit. Node-webkit being a Node.js module developed by Roger Wang which combined the browser engine used by Chromium - WebKit - with Node.js, making Node.js modules accessible from JavaScript code running inside a web page. (Jensen, 2017)

Porting to node-webkit proved difficult, so GitHub abandoned that approach, and it was decided that a new native shell for Atom would be created. Said shell was dubbed *Atom Shell* and after development was finished and the Atom editor was open sourced by GitHub, Atom Shell soon followed suit and was renamed to *Electron*. Initially developed as a way to deliver an editor, numerous widely known applications like Slack, Discord and Visual Studio have started using Electron to develop and deliver their desktop applications.(Electron Framework, 2021b) But what exactly is Electron?

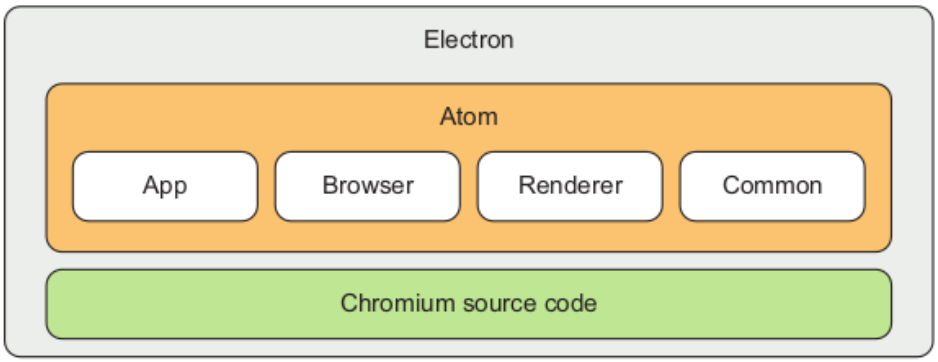
Electron is a framework which allows for the development of cross-platform desktop applications using only popular web programming languages like HTML, CSS and JavaScript. While the advantages will be discussed in detail in the next chapter *Why Use Electron?*, the appeal to developers is obvious: Maintaining one codebase while being able to deliver the app to all desktop operating systems.

Now, as described previously the Electron framework serves the same purpose as node-webkit (later renamed to NW.js), but their approaches do differ in certain ways: (Jensen, 2017)

Without going into detail on how NW.js works, Electron and NW.js share some architectural similarities. However, there are some differences in how Electron combines Node.js with Chromium.

Architecturally, Electron places an emphasis on strict separation from Chromium source code, ass seen in figure 1.1. This looser integration allows for easier updates to the Chromium part of the source code, whereas with NW.js Chromium is patched to allow for Node.js and Chromium to use a shared Javascript state. (Jensen, 2017)

Figure 1: A simplified representation of Electron's architecture. (Jensen, 2017)



On the other hand, this means that Electron has separate JavaScript contexts: A *main* process which starts running with the app window and a *renderer* process for each individual window. Any sharing of state between these contexts, or simply put between the front- and back-end, has to pass through the *icpMain* and *icpRenderer* modules. This means that each JavaScript context is kept separate but data can be explicitly shared, allowing for greater control over what state exists in which app window. (Jensen, 2017)

Electron itself (the part without Chromium) is made up of for different components: App, Browser, Renderer and Common. **App** contains code written in C++ and Objective-C++ responsible for loading Node.js and Chromium's content module. The **browser** folder contains code which handles interactions with the front-end. This is to say functionality such as loading the JavaScript engine, interacting with the UI and binding operating system specific modules. As for **renderer**, this component handles the different renderer processes. Because Chromium works by running each tab as an individual process, as to not crash the entire browser should one tab become unresponsive, each application window in Electron runs as its own process. **Common** contains code which is used by both the main and renderer processes for running the application. Among other things this folder also contains the integration of Node.js' event loop with Chromium's event loop. (Jensen, 2017)

1.2 Why Use Electron?

Now the next obvious question is why a framework such as Electron is needed at all. After all, it is "just" a way to have desktop applications developed using HTML, CSS and JavaScript. So why not just develop native desktop applications or traditional web applications depending on the use case? To answer this one has to examine the bigger picture:

Over the past decade it seems as though software pricing has moved from perpetual licenses towards subscription-based models. If one examines the data regarding end-user spending on cloud applications it is clear that the *Software as a Service* (SaaS) model has grown considerably in revenue and is projected to do so in the future: The worldwide end user spending for Software as a Service has increased from 31.4 billion US Dollars in 2015 to 120 billion US Dollars in 2020. It is projected this growth will progress with spending reaching 171.9 billion dollars in 2022. (Gartner, 2021)

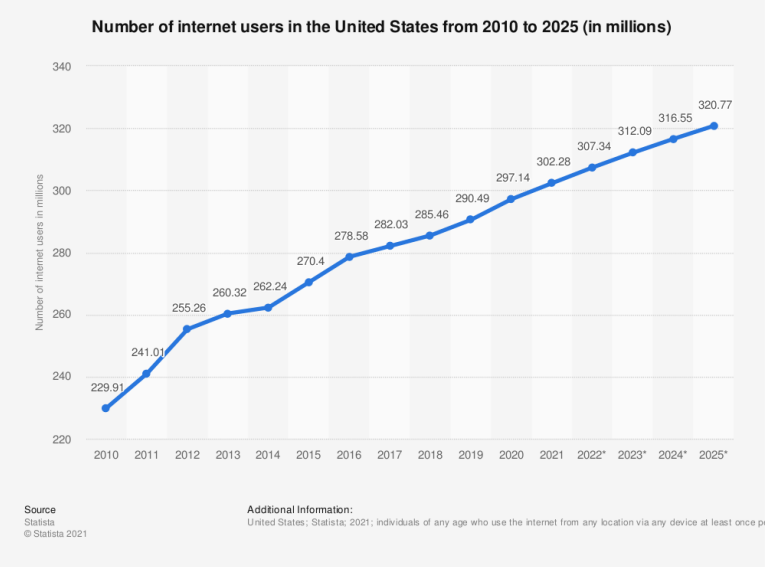
Furthermore, Gartner (2021) forecasts that by 2026, cloud spending will exceed 45% of all enterprise IT spending, up from 17% in 2021. This impressive growth can be attributed to two groups of reasons. Reasons either technical and/or financial in nature. One financial benefit of SaaS is economies of scale: By hosting the application centrally and by extension aggregating users together, providers can benefit financially from leveraging economies of scale. At the simple end, this means benefiting from volume pricing on hardware such as data centers, servers, space and so on. Taking this idea further, SaaS providers can also cut costs by sharing hardware across their customers. It is not cost-effective to use one machine for each customer, instead resources should be shared and dynamically allocated on-demand to each customer's needs. Similarly, as user count increases, the cost of adding on single user decreases. These and other reasons are a big financial motivator for providers of software to switch to the SaaS model. (Jacobs, 2005)

However, technological reasons play a large role as well. According to Jacobs (2005) the ever-increasing maturity of the Web is a major contributor for the rise in popularity of SaaS.

Browsers are significantly more powerful than ever. The *browser wars* of the mid-to-late nineties started with Microsoft and Netscape outdoing each other with new features, faster and overall better browsers leading to significant leaps in browser technology and therefore accelerating the advancements of web browsers. (Jensen, 2017; Mozilla Foundation, 2021)

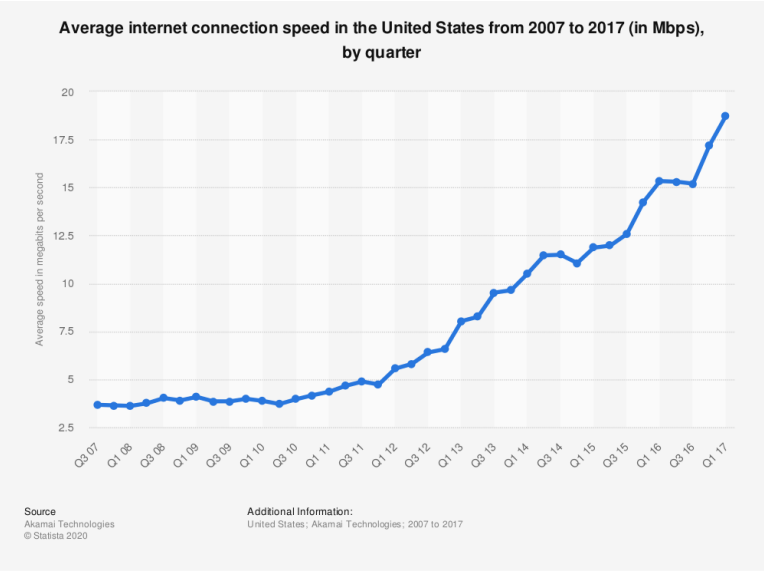
Furthermore, internet access is more widespread than ever. In the United States, the number of internet users rose from 229,91 million in 2010 to 302,28 million in 2021, which constitutes a 31% increase.

Figure 2: Number of internet users in the US from 2010 to 2025. (Statista, 2021)



And not only have the number of internet users risen over the past eleven years, the average connection speed increased as well over the same time period:

Figure 3: Average internet connection speed in the US 2007-2017. (Akamai Technologies, 2017)



As seen above, from Q3 2007 to Q1 2017, average internet speeds across the US rose by 410%. This allowed for much more elaborate websites where larger amounts of data have to be downloaded. Moreover, the number of robust frameworks for web development (be it front-end or back-end), make creating a complex web application easier than ever before. However, while this explains why SaaS is often the billing model of choice, it doesn't fully explain why specifically web applications have risen in popularity. (Statista, 2021)

After all, SaaS can also be delivered as a Desktop Application, as seen with the Adobe Creative Suite for example. (Adobe Inc., 2021)

To answer this, one should examine how desktop applications and web applications differ in more detail.

1.2.1 Desktop Applications

Desktop applications used to be the standard way of delivering software to the end user. Users had to go to a store, buy a CD-ROM, check the system requirements and install the software on their machine. This does of course come with a number of benefits over web applications.

One advantage is that desktop applications aren't reliant on an internet connection. Web applications obviously fail here, as they are accessed over the internet. Furthermore, this reliance on an internet connection leads to more issues when the application is very feature-rich and/or has to support large files. An image editing software as a web application for example can run into limitations when being used with high-resolution images. Similarly, desktop applications start instantly without having to download resources over the internet. In such cases, desktop applications have an edge over comparable web applications. (Jensen, 2017)

There are of course also benefits to using desktop applications from a developer's perspective. As a developer, one does not have to worry about users accessing their web applications over different web browsers as the choice of browser is at the user's discretion. This means not having to consider how different browsers interpret CSS, for example and always being sure about how the UI is being displayed and how the application's code is interpreted. Another benefit is the fact of tighter integration with the user's OS. Browser security limits the use of hardware and can lead to challenges for certain use cases. (Jensen, 2017)

Moreover, having an installable desktop application means not having to continuously support all the necessary infrastructure. There simply is no need to run servers, databases and such when the application is locally installed on each user's machine. (Jensen, 2017)

However, these benefits of desktop applications come with a significant drawback. Developing desktop applications requires developers to be proficient at languages like C++, Objective-C or C#. For a portion of developers, this can be a significant barrier to entry because it means learning an all-new language and in some cases even frameworks as well.

1.2.2 Web Applications

In contrast to desktop applications, the relatively low barrier for entry in web development thanks to the ease of learning the basics of HTML, CSS and JavaScript makes it much easier for developers to create complex web applications. With the amount of open source frameworks developers of web apps have a large selection of different solutions to fit their specific use case. Also, package managers like npm offer a large selection of readily available,

well established packages for developers to use and enhance their projects with.

Another big advantage of web applications is that they are platform-independent. A web application can be reached on any reasonably modern device which runs a web browser. There is no need to create a separate version for all the operating system one wants to support and websites can also easily be accessed on mobile devices.

As described in the previous chapter 1.2.1, web applications need continuously running infrastructure such as web servers and databases. While this constitutes a disadvantage, it also comes with a big benefit for developers, as they can strictly control which version a client uses. Furthermore, the access to real-world data in said databases makes reproducing bugs much simpler. (Jacobs, 2005)

1.2.3 Electron as a Solution

However, web applications do come with disadvantages. As described in 1.2.1 web apps have their shortcomings such as browser security preventing access to a user's file system or having no access as soon as the internet connection fails.

This is where frameworks such as Electron manage to strike the near-perfect balance between desktop application and web app. For instance the drawback of not having access to a user's PC's file system does not apply to applications developed using Electron, as the npm module *osenv* can for example retrieve the user's home folder among other environment settings. (Schlueter, 2012)

Additionally, the disadvantage of having to consider different browsers (and versions thereof) are a non-issue with electron because Electron uses Chromium as outlined in 1.1. Furthermore, internet access is not a requirement with Electron which means applications can have some offline functionality as opposed to web apps.

These are some features and advantages of Electron, though not an exhaustive list. (Electron Framework, 2021b)

Ultimately, it is at the developer's discretion which form of software to use. Native desktop applications, web applications or applications developed using frameworks such as Electron all have advantages and disadvantages, and it is important to consider which solution fits an application's and/or user's needs best.

In order to facilitate such informed decisions this thesis attempts to answer the question of how web applications can be developed and deployed as desktop applications using Electron. To answer this question a proof-of-concept application will be developed, which will be explained in more detail in the following chapter. The aforementioned considerations and points will be examined in development of said application and finally an evaluation will be made on how effective of an alternative to web applications Electron is and whether the biggest shortfalls of web applications can be eliminated or mitigated by using Electron.

2 Method

Using the Design Science approach (Vaishnavi & Kuechler, 2004) an artifact will be developed using Electron. Said artifact is a proof-of-concept application which will be used to gauge whether Electron would be an effective solution in this specific and other different use cases.

The application to be developed will be developed similarly to an already existing application. Said application is a time keeping app developed by and used by Comm-Unity GmbH, a company specialising in e-government software. This web app is used by all employees to record the time worked based on factors such as customers, projects and other internal, domain-specific aspects.

The existing application is a web-only application developed using Vaadin (Vaadin is a server-side Java-only framework for web development)(Vaadin Ltd, 2021). By developing a similar application using Electron one can directly judge where the advantages and disadvantages of Electron lie.

2.1 Artifact Outline

As mentioned before the intention of the application is to facilitate efficient time keeping for employees of Comm-Unity EDV GmbH. As with any

business-oriented time keeping program it is central to be able to book time worked on specific data points not only to be able to bill customers correctly but also to create a clearer picture of which project and/or customer require what amount of attention by employees. In this specific proof-of-concept we will limit said data points to customers and projects.

These can be easily extended customised depending on domain-specific requirements and needs but as far as this example goes, a generalised approach is sufficient and also required as to extrapolate results on other use cases.

The application will be structured into four different views. The main view shows a list of entries which each represent a data point. Said data point has a start and end time a project and customer are assigned to each entry. As the number of customers and projects increases over time it becomes increasingly difficult for employees to quickly find the correct values to attribute their entries to. To make this easier to use users can create templates which limit the possible data points one can choose.

The second view is the so called template view which shows a list of the aforementioned templates. Users can create, update and delete templates which each have a unique ID, a name and a list of projects and customers.

The third view is an overview of customers, which can be created, updated and removed. Each customer is comprised of a name and an address.

The fourth view is similar to the customers view but represents an overview of projects, which can also be created, updated and removed. Each project contains a name and whether said project is active or not.

The application can be split into three distinct parts from a technological/architectural point of view:

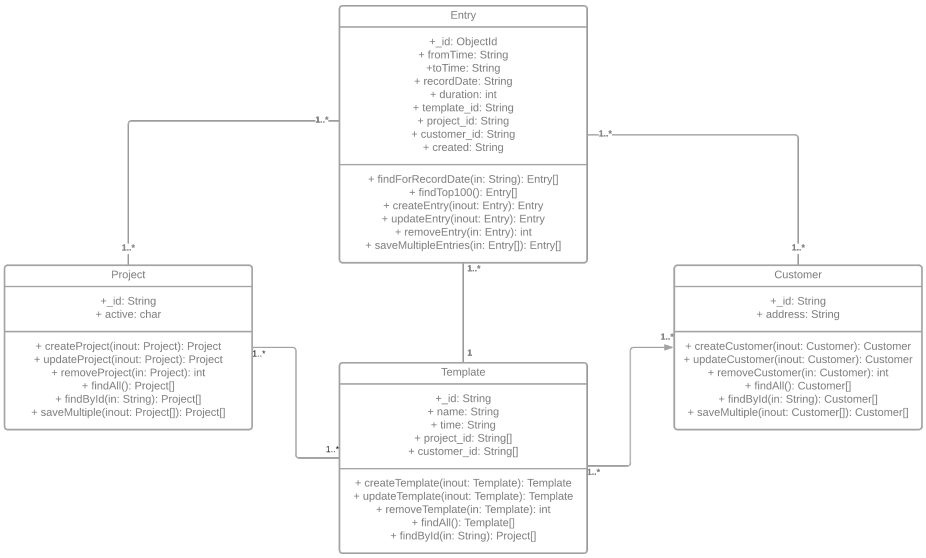
- The back-end API used to fetch and save data.
- The front-end part using Angular(Google, Inc., 2021a).
- The Electron part used to deploy the application and to offer offline functionality.

The back-end uses a MongoDB (MongoDB, Inc., 2021) database to persist data. The decision to choose MongoDB was taken because of the use case: A time keeping application needs to be highly flexible. Entities such as projects and customers and the general company structure and therefore evolving requirements can change over time, possibly necessitating a different database structure. Due to MongoDB being a No-SQL database based on a JSON-like document structure, future changes regarding documents (which are analogous to tables in relational databases) require no changes to the database itself. Such flexibility would greatly increase the future proofing of such an application.

The back-end logic is developed in Python and uses the Flask framework (Pallets Project, 2010) for handling requests from the front-end. As mentioned previously for each entity (project, customer, template, and entry) there is business logic to support create, update, and remove operations.

The front-end uses Angular (Google, Inc., 2021a) and Angular Material (Google, Inc., 2021b). Angular was chosen as it is a very popular framework for single page applications and furthermore, it makes it easy to develop re-usable UI components. As for the components library, Angular Material was used because of its ease of use and well-known design, meaning users can easily adjust to the new user interface.

Figure 4: A class diagram representing all entities, relationships and operations.



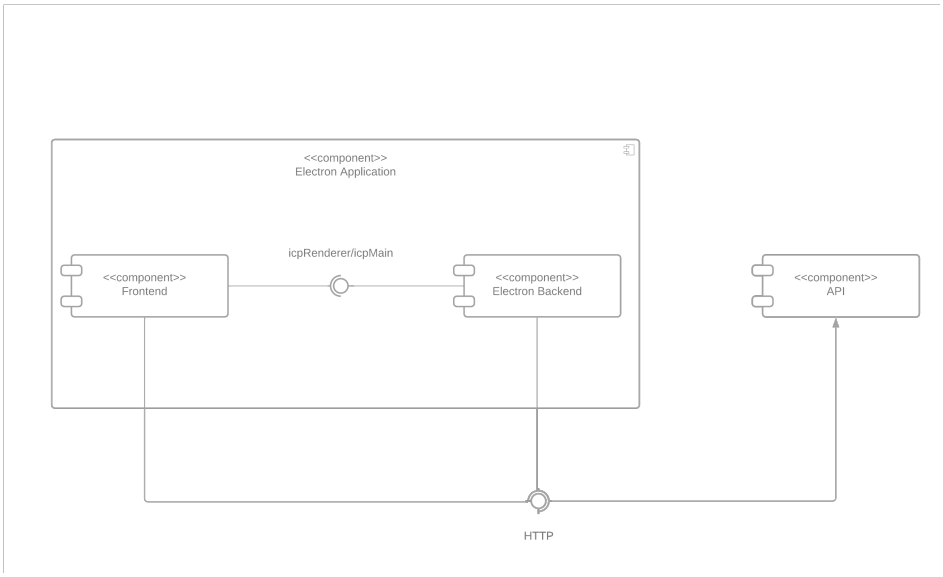
As seen in the figure above, all entities have similar operations and relationships. Each can be created, updated, and removed. Furthermore, each offers an operation to fetch an entity based on its ID, as those are retrieved on-demand and not fetched eagerly. Additionally, the Entry entity contains an operation to fetch the 100 latest entries (ordered by their recordDate attribute) in order to return data for a local backup on each user's machine. As the number of entries will grow once this is in use it is neither practical nor sensible to fetch all records from the database just to make offline usage possible. Another unique operation of the Entry entity is fetching Entries based on their recordDate attribute. This is necessary as users view entries on a per-day basis.

Entry, Project and Customer entities each contain an operation which allows for the creation of multiple of their respective entities. This is to facilitate the offline functionality where users can create entities while not connected to the back-end API. Said entities are saved locally and once connection is restored, they are posted to the back-end and persisted.

An instance of a template always references at least one customer and project, meaning that each entry which always references exactly one tem-

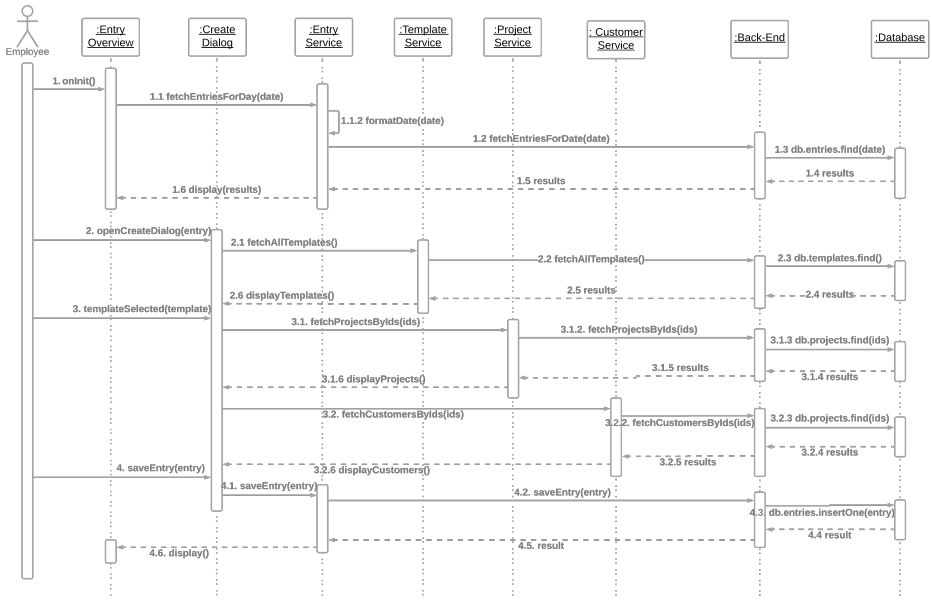
plate always references at least one project and customer. One template can of course be referenced by multiple entries and one template can reference multiple project and vice-versa. On the database level, a template holds an array of project and customer IDs as references. The IDs of customer and project entities are a string representation of their given name as those have to be unique.

Figure 5: A schematic overview of the application components, simplified.



The above figure shows a simplified overview of the project's components. For the sake of illustration some implementation details have been omitted, such as the details of the database implementation in the back-end API. In essence there are two ways the front-end can communicate with a data source: Either over HTTP requests with directly the back-end or through the icpMain and icpRenderer processes with the Electron-provided back-end. The details of this implementation will be discussed in a later chapter. To further illustrate the interaction between the application's components, see the following sequence diagram which illustrates the workflow of creating an entry.

Figure 6: A sequence diagram of the Create Entry use case.



After the user has started the application they will find themselves on the entry overview which lists all entries for the selected day. This causes a request to the back-end to fetch all entries for the specified day. Once the user opens the create dialog all templates are fetched and displayed to the user to choose from. When the user chooses a template the projects and customers defined in the selected template are fetched from the back-end. After the user has entered all other necessary data they can save the newly created entry which sends the object to the back-end and persists it in the database. This is an illustration to show what the workflow behind user creation of an entry looks like and how entries, templates, projects and customers all work in accordance.

2.2 Developing with Electron

As previously described, the front-end component of this application is developed using Angular and Angular Material. As these aspects are independent of Electron, the details of said implementation will be omitted, except the parts where a direct communication between the front-end and angular takes place.

During the course of this chapter a closer look will be taken on how an Electron application is to be developed. That is to say, what the general structure should be, how it is set up and bootstrapped and finally how one can use Electron's functionality to cater to the needs specific to this application.

2.2.1 Creating an Electron Application

As with any framework, installation comes first. Electron can be simply installed using npm, the node package manager: (npm, Inc., 2021)

```
1     npm install -g electron
```

This installs Electron as a global dependency. The next step is to create a project directory and a package.json file for the Electron configuration, among other things. The entry point for any Electron application is a JavaScript file. In this specific case it is named app.js:

```
1 {
2     "name": "pze",
3     "version": "1.0.0",
4     "main": "app.js"
5 }
```

Note that the name property on line two is an abbreviation of the German term for this application: Projektzeiterfassung, meaning project time keeping and abbreviated to PZE.

As mentioned the starting point for any Electron application is a JavaScript file. This file (in this case app.js) is responsible for loading all required parts of the Electron application and any windows that it should display. See the following example for creating a browser window in electron:

```
1  const { app, BrowserWindow, ipcMain } = require("electron");
2  let mainWindow;
3  function createWindow() {
4    mainWindow = new BrowserWindow({
5      width: 800,
6      height: 600,
7    });
8    mainWindow.loadURL(
9      url.format({
10       pathname: path.join(__dirname, `/dist/pze/index.html`),
11       protocol: "file:",
12       slashes: true,
13     })
14   );
```

```
15   mainWindow.on("closed", function () {
16     mainWindow = null;
17   });
18 }
19
20 app.on("ready", createWindow);
```

After importing the necessary parts of Electron one needs to create a window. As the "ready" event is fired once the app has finished loading one can listen to said event and then call the createWindow() function to create a window and more importantly load a file to display. On line three a window is created with the specified dimensions, 800 by 600 pixels in this case. Electron's approach to window sizing means that a developer can specify different window sizes for each window directly in code rather than having to specify such properties in a configuration file, as is the case with NW.js. (Jensen, 2017) Furthermore, parameters for the initial positioning of the window can also be passed to the BrowserWindow constructor.

The next step is to load a URL. This URL needs to point to an HTML file containing the content to display. In this case the output file of the Angular part of this application is loaded. Finally, after the window has been closed the variable will be set to null.

However, not every platform behaves the same when an application is closed by clicking the X symbol in the top right or left corners. As opposed to Windows and Linux, MacOS does not quit an application once a window is closed. To check for the platform so the correct behaviour can be implemented one can use the process.platform (OpenJS Foundation, 2021) property provided by node.js:

```
1 app.on("window-all-closed", function () {
2     if (process.platform !== "darwin") app.quit();
3 });
```

As shown the code listens for the "window-all-closed" event and if the user's operating system is anything but MacOS the app quits. These steps form the basic scaffolding of an Electron project and can now be run with the electron command.

2.2.2 Electron and Angular

As described previously the front-end in this application is a single page application developed with Angular. This chapter describes the process of

integrating Angular with Electron. After taking the steps to create and scaffold the Angular application, some configuration work needs to be done. In the angular.json file the output directory needs to be set to the previously specified directory where Electron looks for the index.html file:

```
"options": {
    "outputPath": "dist/pze",
},
```

Another modification is required for the start scripts in the package.json file where the angular application has to be built and then the electron application:

```
"scripts": {
        "start": "ng build --base-href ./ && electron .",
},
```

With this npm start can be executed and first ng build will be called, building the angular application after which the Electron start script will be called and the Electron application will be started.

Now the app is up and running. An Electron instance will start with the Angular application running inside it. To speed up the development process, the Angular front-end should be developed separately. Not only does this lead to looser integration of Angular and Electron but when saving edits made to Angular developers need to restart the Electron build process which takes considerably longer than just reloading the Angular app. This means during development - and as long as the Electron part is not needed - developers should start the application with ng serve as changes will be adopted almost instantly whereas with Electron one needs to run the entire start script again.

For communicating between the front- and back-end Electron provides the ipcRenderer and ipcMain modules. In the front-end part events are emitted with ipcRenderer. Electron offers various different ways of sending and receiving events. The methods to listen to events on the ipcRenderer are as follows: (Electron Framework, 2021b)

- ipcRenderer.on(channel, listener):
  This method listens to a channel and when a message on the corresponding channel arrives, the listener is called.
- ipcRenderer.once(channel, listener):
  Acts similarly to ipcRenderer.on, but the listener is removed after the invocation

With ipcRenderer.removeListener(channel, listener) or ipcRenderer.removeAllListeners(channel) listeners can be removed. Note that with ipcRenderer.removeAllListeners(channel) the channel parameter is optional and when omitted all listeners get removed. The ipcMain Event Emitter has the same methods as ipcRenderer to listen to events, with the only addition being ipcMain.handle(channel, listener).

Similarly to receiving events, Electron offers multiple methods to send events from the ipcRenderer Event Emitter. Those methods include: (Electron Framework, 2021b)

- ipcRenderer.send(channel, ...args):
  This methods sends an asynchronous message to ipcMain via the specified channel.
- ipcRenderer.invoke(channel, ...args):
  This methods also sends an asynchronous message. It does however expect a result and returns a Promise<any> to resolve the response. Note that on ipcMain handle() should be used to intercept these events.
- ipcRenderer.sendSync(channel, ...args):
  Same as ipcRenderer.send(channel, ...args) with the difference of expecting a synchronous result.
- ipcRenderer.sendTo(webContentsId, channel, ...args):
  A method for sending an event directly to a specified window using the webContentsId parameter.
- ipcRenderer.sendToHost(channel, ...args):
  Behaves the same as ipcRenderer.send(channel, ...args), with the difference being that the event is sent to the <webview> element rather than the main process.

Having listed the options provided by Electron for inter-process communication the next step is to integrate the ipcRenderer into Angular. For this, there are multiple solutions. To simplify development, developers can use an NPM module called ngx-electron developed by Thorsten Hans. (Hans, 2019) Ngx-electron makes calling Electron's API from Angular simpler for developers by exposing all methods available within Electron's render process. This means the above mentioned methods would all be accessible through this module. To include it in the Angular application one needs to import the module:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

import { NgxElectronModule } from 'ngx-electron';
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MaterialModule,
    HttpClientModule,
    NgxElectronModule,
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

After then importing the ElectronService class from the ngx-electron module in the necessary components one can send or listen to events by accessing the API:

```
this.electronService.ipcRenderer.send('getProjects');
```

For this solution to work there is a caveat however: If one tries to run this example with the latest stable Electron release (at the time of writing 17.1.0), the following error will be encountered:

```
Error: node_modules/ngx-electron/lib/electron.service.d.ts
    :17:31 - error TS2694:
Namespace 'Electron.CrossProcessExports' has no exported member
     'Remote'.

17     readonly remote: Electron.Remote;
                                 ~~~~~~
```

This error occurs because in version 12 of Electron, the remote module has been deprecated and in version 14 removed from Electron itself and moved to another package, @electron/remote. (Electron Framework, 2021a) This of course leads to an error because ngx-electron cannot locate the required module. Because the latest release of ngx-electron (version 2.1.1) happened in October of 2019, one can assume that this issue (which is still marked as open on GitHub as of March 2022) will not be fixed in the foreseeable future. (Ch3shireDev, 2021) What this means for developers is that if they wish to use ngx-electron, the latest usable stable release of Electron is 13.6.9. While said release was last updated (at the time of writing) on February 2nd 2022,

being forced to use a deprecated feature and being locked into a specific version of any framework can pose a worry to many developers.

As an alternative, developers can skip the use of ngx-electron and directly import the required electron module.

```
const electron = (<any>window).require('electron');
```

Sending an event through the ipcRenderer would be very similar to the code sample above which is using ngx-electron:

```
// Using ElectronService from ngx-electron
this.electronService.ipcRenderer.send('getProjects');

// Directly accessing ipcRenderer
electron.ipcRenderer.send('getProjects');
```

The advantage would be greater future-proofing because the use of a newer Electron stable release would be possible.

2.2.3 Development Workflow with Electron

2.2.4 Finished Project

3 Analysis

3.1 Results

3.2 Vaadin as an Alternative to Electron

References

Adobe Inc. (2021). Plans and pricing for creative cloud apps and more. Retrieved December 14, 2021, from https://www.adobe.com/creativecloud/plans.html

Akamai Technologies. (2017, May 1). Average internet connection speed in the united states from 2007 to 2017 (in mbps), by quarter. Retrieved December 14, 2021, from https://bit.ly/31PSvFV

Ch3shireDev. (2021). Namespace 'electron.crossprocessexports' has no exported member 'remote'. number 71. Retrieved March 13, 2022, from https://github.com/ThorstenHans/ngx-electron/issues/71

Electron Framework. (2021a). Electron 14.0.0. Retrieved March 13, 2022, from https://www.electronjs.org/blog/electron-14-0#removed-remote-module

Electron Framework. (2021b). Osenv. Retrieved December 14, 2021, from https://www.electronjs.org/docs/latest

Gartner. (2021, August 1). Global public cloud application services (saas) market size 2015-2022. Retrieved December 14, 2021, from https://bit.ly/3lZb8xV

Google, Inc. (2021a). Angular. Retrieved March 7, 2022, from https://angular.io/docs

Google, Inc. (2021b). Angular material. Retrieved March 7, 2022, from https://material.angular.io/

Hans, T. (2019). Ngx-electron. Retrieved March 13, 2022, from https://github.com/ThorstenHans/ngx-electron

Jacobs, D. (2005). Enterprise software as service. *Enterprise Distributed Computing*.

Jensen, P. B. (2017, May 1). *Cross-platform desktop applications using node, electron, and nw.js*. Manning Publications. https://bit.ly/3EPy0r1

MongoDB, Inc. (2021). Mongodb. Retrieved March 7, 2022, from https://docs.mongodb.com/

Mozilla Foundation. (2021). The history of web browsers. Retrieved December 14, 2021, from https://www.mozilla.org/en-US/firefox/browsers/browser-history/

npm, Inc. (2021). Npmjs. Retrieved March 7, 2022, from https://docs.npmjs.com/about-npm

OpenJS Foundation. (2021). Node.js. Retrieved March 13, 2022, from https://nodejs.org/api/documentation.html

Pallets Project. (2010). Flask. Retrieved March 7, 2022, from https://flask.palletsprojects.com/en/2.0.x/

Schlueter, I. (2012). Osenv. Retrieved December 14, 2021, from https://www.npmjs.com/package/osenv

Statista. (2021, January 1). Number of internet users in the united states from 2010 to 2025. Retrieved December 14, 2021, from https://bit.ly/3yq6LkE

Vaadin Ltd. (2021). Flow. Retrieved March 7, 2022, from https://vaadin.com/docs/latest/flow/overview

Vaishnavi, V., & Kuechler, B. (2004). Design science research in information systems. *Association for Information Systems*.

List of Figures