

Making websites great again!

(Yes, that's a Trump pun)

The image shows the letters 'DR' in a large, bold, white, sans-serif font. The letters are centered within a dark gray rectangular background. The 'D' and 'R' are connected at the top but have a gap between them.

Creating an application that provides live data to a website from an external API

Who: Dan Christensen

What: Dissertation

Where: Danmarks Radio, TU Web Applikationer

When: 1/10/2016 - 8/11/2016

School: Erhvervsakademi Sjælland, Datamatiker

Supervisor: Anders Børjesson

Table of Contents

Table of Contents	2
Introduction	5
Abstract	5
Overall introduction	5
The structure of the report explained	5
Chapter 1: Project start	6
The initial assignment	6
The people working on the project	6
Chosen tools and frameworks	6
Planned architecture	6
Project Management and Work Assignments	7
My Learning Goals	7
Chapter 2: Problem definition	7
The initial assignment recap	7
Problem analysis	7
Problem definition	8
Chapter 3: Data gathering (Problem Analysis)	8
Introduction	8
Problem definition	8
US Election Research	8
Method	8
Plan	8
Problem Solving	9
Evaluation and result	10
API Research	11
Method	11
Plan	11
Problem Solving	11
Query Explorer	11
Test Election Data	14
Evaluation and result	14
Overall evaluation and result	14
Chapter 4: The Backend	15
Introduction	15
Problem definition	15
Method	15
Plan	15
Problem Solving	16
The API Scraper	16
The Report Scraper	18
The Scraper Wrapper	20
The Test Runner	21

Evaluation and result	22
Chapter 5: Live data	22
Introduction	22
What is Firebase?	22
Problem definition	22
Method	22
Plan	23
Problem Solving	23
Firebase	23
MongoDB	23
Evaluation and result	24
Chapter 6: The Frontend	24
Introduction	24
Problem definition	24
Method	25
Plan	25
Problem Solving	25
The Election Bar	25
Handlebar / HTML	25
Less	27
Javascript	27
The Election Map and State List	32
Making it modular	32
Evaluation and result	32
Chapter 7: Late changes	33
Introduction	33
Problem definition	33
Method	33
Plan	34
Problem Solving	34
3rd Party Candidates	34
Adjusting the Architecture and providing data to TV	34
Firebase	35
Evaluation and result	35
Chapter 8: Project Management	35
Rituals	35
Programming	36
Challenges	36
Chapter 9: Election night	36
Backend	36
Website	37
TV news	37
The night in general	37
Conclusion	38

Overall project evaluation	38
The problem definition	38
Data Gathering	38
Architecture and Frameworks	38
The Backend	38
The Frontend	39
Databases	39
Late Changes	39
Project Management	39
Lessons learned	40
Outlook	40
Final Thoughts	40
Appendix	40
Documents:	40
Files:	41

Introduction

Abstract

During my internship at Danmarks Radio, Team Connect (that I was assigned to) was given the task of delivering live data to the US Presidential and General elections. Our product would be part of a larger whole (DR's website and election event site), and we would only be responsible for the specific elements that dealt with live data. This report covers the whole process, from beginning research, through the agile development with many changing requirements, and all the way up until launch on election night.

Overall introduction

We were given access to an API from Associated Press, and shown some initial drafts¹ of what the end product should look like, design-wise - but otherwise we would have to figure out the rest ourselves. There were a lot of unknown factors when we started the assignment, meaning we had to start with a period of research, both into the API itself, but also into the election in general, to figure out how everything behaved, and how we should handle that, before we could get around to the actual coding.

I chose to use this assignment as the topic for my dissertation for several reasons. First, it was an assignment where I was part of it from the very beginning that it was given to Team Connect, and second, it was a big enough, and interesting enough assignment that I felt I could learn a lot by doing it.

To enable me to see this assignment through to the end, I managed to extend my internship with DR for an extra month, up to and including the week of the election itself. All in all, we spent 9 weeks working on the project.

The structure of the report explained

I have chosen to write this report a bit differently. The assignment itself was a highly iterative process with a lot of discovery and learning along the way. This meant that the parameters of our assignment kept changing as well - especially considering that our first order of business was to figure out which actual problems to solve.

The most important aspect of this though, is the problem definition. Our problem definition at the beginning of the project did not at all reflect our problem definition at the end. Just like the project itself, it evolved through several iterations, and I feel that it's important to show this in the report.

Therefore, the structure of the report will attempt to reflect this, at least in part. I have divided the report into chapters that are *roughly* in chronological order, and grouped by topic / problem set. I say *roughly* because in truth, we were working on chapters 4-6 at the same time, and had to dip back into data gathering / analysis every so often as well.

From Chapter 3 and onwards, each chapter will be a "mini report", containing the problem definition as it looked at that point in time, along with the methods, plan, problem-solving and result of what we worked on. I will start each chapter with an "updated" problem definition, showing the status as it was at the start of that part of the work.

Finally, in the conclusion, I will look back at the project as a whole, and the problem definition as a whole as well, and evaluate based on that.

NB: It's important to note here that this was a large project. While I had a hand in working on most things, to varying degrees, I am going to focus on only the things where I was the primary (or only) person working on them. To cover everything would likely take up more pages than we are allowed in this dissertation. For this assignment, I was the primary on the following things: ***Election Research, API Research, API Scrapers and wrapper, Recording test elections and analyzing data and the Election Bar.***

¹ See Appendix

Chapter 1: Project start

The initial assignment

For my internship in DR, I was placed in a front-end development team called Team Connect. Due to a lack of assignments for that team, I was switched to a different team at the start of week 5. However, at the end of week 5, Team Connect's Product Owner came to me and told me that they had just gotten two new assignments, and that they'd love for me to be a part of one of them. So I switched back and started on the assignment that would be the base for this dissertation.

A news editor from DR's web news department had negotiated a deal with Associated Press for access to their API, which would deliver live data for the US presidential and general election. Furthermore, the editor had already been in contact with the design department at DR, and they had a bunch of non-final designs for how they wanted this live data to be displayed on the website.

Team Connect's assignment was to create everything in between these two things, and make sure everything worked and was able to run live during the election. We were supplied the API, and a vision of the finished product, and had to make sure those two connected.

The people working on the project

Working on this project would be Peter Freltoft, a developer who had been with Team Connect for many years, and who had a great deal of experience working on danish elections, and then myself, as an intern. During the last 1½ weeks, we would be joined by Adam Grausen, another long-time developer at Team Connect, but for the majority of the time, it would just be myself and Peter working on it.

Chosen tools and frameworks

Since Peter was the person responsible for the project at Team Connect, he chose the tools that we would use, based on what he thought would get the job done, and what he was comfortable with:

- NodeJS for the backend
- ExpressJS for the frontend
- Google Firebase as our live data database
- MongoDB as our archive database

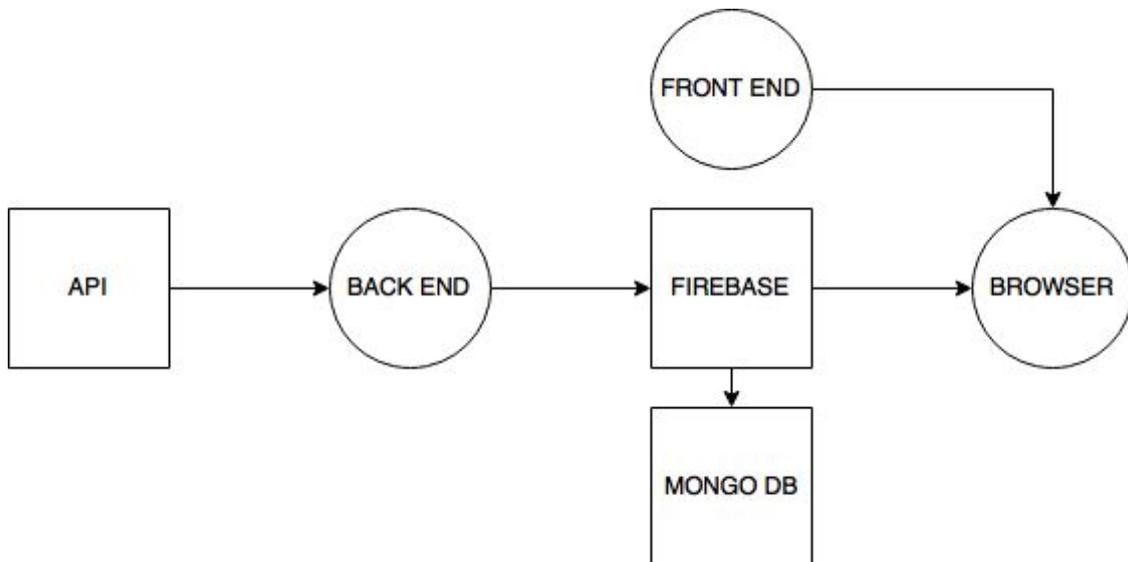
The above tools would handle the architecture part of the application, but we would also need to use several other technologies. Some of them are part of the production pipeline at DR, and some of them are part of the design pipeline for the website - and we of course also had to use those, to make sure everything would fit together:

- HandlebarsJS
- LESS
- Github
- Heroku
- GruntJS

Planned architecture

Peter already had a basic sketch for a system architecture in mind when we started the project, and we used this as our "base assumption". The backend application would query the API for data, process this data into something we could use, and feed it to the Firebase database. The frontend application would then have "hooks" in the Firebase database, which would look for changes, and fetch data whenever changes occurred, thus updating the user's view on the website.

We figured that we would need to also have a mongoDB database to use after the election was over, when the "live" aspect of the application was no longer necessary, but we hadn't entirely decided on how it would be hooked up to the general architecture at this point.



Project Management and Work Assignments

The Web Department at Danmarks Radio all use a variant of Scrum, commonly referred to as ScrumBut². This is not really a good term, but it is accurate. The teams try and stick as closely to Scrum as possible, but other parts of DR do not use Scrum, so often there will be a lot of factors preventing the team from fully utilizing the methodology.

That being said, as a Scrum team, we would have daily standup meetings, and we would have “shared code ownership” in the sense that while one person might be the “primary” working on something, the other members of the team would get a run-through of the code together with the primary, so that they understood what was going on, and so that they could work on it or adjust it if they needed to. I will go into more detail about our usage of Scrum and project management in general in chapter 8.

My Learning Goals

Since this was a purely practical real world assignment, I didn’t set out to do it with any particular learning goals. It was a good chance for me to experience a real project from start to finish, using all the skills I had learned during my education. I would need to do research, analysis and coding, all while using a development framework (Scrum). It would also be my first real experience working with NodeJS, which I was excited to learn.

Chapter 2: Problem definition

Our initial assumptions about the project

The initial assignment recap

So, to sum up, we were given the following set of parameters for our assignment.

1. Get live data from Associated Press API
2. ...
3. Supply live data from the election to DR’s website, using the design guidelines given to us

We had #1, and we needed to get to #3, but everything in between was basically up for discussion, and for investigation. There were a lot of questions that would need to be answered before we could even start coding the application that would get us to #3

Problem analysis

The major, glaring problem with this assignment was that we simply didn’t know enough. We didn’t know what and how much data the API would provide, we didn’t know how much of this data was relevant to the end product, we didn’t know

² <https://www.scrum.org/scrumbut>

how the data would behave in a live scenario, and so on. We realized that we didn't even know the details of how the presidential and general elections worked in the US - and this was something that we'd need to learn before we could create our solution.

What we *did* know however, were the technologies that we wanted to use to create the solution, and we created github repositories for the front end and the back end. Github has a built-in wiki that comes with each repository, so we decided to use the back end wiki to organize our information, issues and clarification questions.

Problem definition

With so many unknowns, our initial problem definition was therefore very open-ended - but it was a start. We realized that in order to meet the requirements of the project, we would need to do some heavy research first. Even though we did not formally make a problem definition at this point (because this was an actual assignment rather than one created for the purpose of my dissertation), our initial problem definition was as follows:

With our chosen frameworks and technologies, how can we provide live data from an outside API when there are many unknowns?

This was our base problem that we had to solve - and it was one that would bring to light many issues and sub-issues as the project came along. This very open-ended question is therefore the basis for my dissertation, and the reason why I'm using an unusual structure for the report.

Chapter 3: Data gathering (Problem Analysis)

Relations between real data and the API. Data accuracy.

Introduction

With the initial meetings and setup work done, it was time to tackle the many unknowns of the project. This required a large amount of research, into both the API, and the actual election.

The first order of business was to investigate the US election in more detail. We needed to figure out the "math" parts of the election - how votes are counted, how winners are determined and so on, and we needed to keep an eye out for any special circumstances or tricky situations that might arise, because we would need to plan for that in our application as well.

After that, we then needed to investigate the API itself. Figure out what kind of data output we were getting, both in terms of format, but also in terms of how they tracked things - and how *they* counted votes, determined winners and so on.

Problem definition

With our chosen frameworks and technologies, how can we provide live data from an outside API when there are many unknowns?

- *How does the US election work in details, and how does this affect our assignment?*
- *What does the data from the API look like, and how does it behave in a live scenario?*

US Election Research

Method

This part of the assignment would require internet research, and a fair amount of it. This would mean finding fairly reliable sources, and then hopefully other sources to back them up. It wasn't necessary to get super high accuracy here, as we were going for more of a general overview of what to look out for rather than detailed historical research.

Plan

1. Find information sources on the internet.
2. Write notes about anything that we need to keep in mind for the application.
3. Write notes about things that we need clarification on from our customer.
4. Gather notes in a centralized location that would be accessible at any time.

Problem Solving

The first step for researching the election was the wiki page for the [2016 election on Wikipedia](#)³. Wikipedia on its own is not a credible resource of course, but properly maintained wiki pages usually also have a list of sources at the bottom for verification and further research, if needed. Since we were just going for an overview of the elections, we decided that it was a “good enough” source for our needs.

Apart from information about the 2016 election, the thing to note here is that they actually have a historical overview of all the elections in the US, going back to 1798. I decided to take advantage of this, and map out the elections from the previous presidential election to this one - ie. from 2012 to 2016. The end result looked like this:

Up for election	Total Seats	Seat Type
2016 Elections		
1	1	Presidential
435	435	Voting-Member House Seats
6	6	Non-Voting-Member House Seats
34	100	Senate Seats
12	50	State Governorships
2	7	Territorial Governorships
2015 Elections		
3	50	State Governorships
2014 Elections		
435	435	Voting-Member House Seats
36	100	Senate Seats
36	50	State Governorships
2	7	Territorial Governorships
2013 Elections		
2	100	Senate Seats (non-standard)
6	435	House Seats (non-standard)
2	50	State Governorships
2012 Elections		
1	1	Presidential
435	435	Voting-Member House Seats
6	6	Non-Voting Member House Seats
33	100	Senate Seats
11	50	State Governorships

You may be wondering why this is relevant to our assignment. The overview actually shows a lot about the things we can expect from the API data, since Associated Press theoretically tracks everything in the election. From this overview, we can conclude the following:

- House members get entirely replaced every 2 years - During the General Election, and the Midterms.
- There are house members that aren't allowed to vote - These are from the “Territories” which do not count as full states, and therefore do not get a vote.

³ https://en.wikipedia.org/wiki/United_States_elections,_2016

- One third of the senate gets replaced every 2 years - Each senate seat is a 6-year appointment, which is why they only swap a third at a time.
- There is no national pattern to when governorships are up for election. Each state has its own rules.
- House and Senate seats can be up for election due to various non-standard occurrences. John Kerry got promoted to secretary of state and resigned as a senator, and one senator from New Jersey died before completing his term. There were similar occurrences in the House as well.

So after the initial overview was done, we could see that there were additional issues that we'd have to look into. Would the special cases affect how we set up our application? How much of the data could we safely ignore? And we were still missing the details of the presidential election with the Electoral College. Finally, there was the issue of Ballot Measures (not shown in the table), that might be of interest to the danish consumer as well - especially with things such as Marijuana legislation, which there has been discussion about in Denmark as well.

I dug further into the topics that our initial overview had shown we needed to look at, and found a few more interesting things:

- There are always many more candidates than just the two that are usually shown. The 3rd party candidates however, usually have such low polling numbers that no-one considers it worth reporting on them.
- Presidential votes are not a direct democracy, but rather a representative democracy. Each state has a number of Electoral Candidates, determined by population size (and possibly other factors). The voters in the state then vote for the candidate that they want, and the winning candidate gets **all** the electoral votes from that state. However, the president is not actually chosen at this time. This happens later, when the Electoral Candidates vote for the president that their state has told them to vote for. It's an odd leftover system from when the candidates had to physically travel to Washington DC to represent their state, and vote on their behalf.
- The winner-takes-all approach to Electoral Candidates is what is used in most states. However, there are currently 2 exceptions: Maine and Nebraska. These two states have split votes, where the Electoral Candidates can be split according to vote percentage.
- Any of the states currently using the winner-takes-all approach are allowed to change to split vote right up until the very last minute before the election.

The information gathered through this research was typed up into notes and put on the Github Wiki that we set up earlier for this purpose.

Evaluation and result

We now had a pretty good overview of how the US General Elections worked. There were a lot more intricacies and details than we had first expected. However, doing this research meant that we were able to spot a lot of issues ahead of time, so we could plan for them in our application. Additionally, we also had a series of follow-up questions that we needed the customer to answer:

- How much data do we need to provide for the website - Which elections do we track?
- What do we do about 3rd party candidates?
- What do we do about split votes?
- Is it relevant to show data regarding the "Territories" - ie. the non-voting members of the House, and the governors of these territories.

Some of these questions could be answered by our own research into the API that we were provided. Now that we had an idea of what kind of data is reported in elections, we had a better idea of how to read the data that the API would provide us. All in all, I spent maybe one full working day on doing this research and taking notes. It wasn't a lot, but it would prove to come in very handy later on.

API Research

Method

As part of the agreement that had been made with Associated Press, we were given several pieces of documentation and tools: An **introductory letter with links**⁴, a **developer guide**⁵ and a link to a **query explorer web page**⁶ where you could try out different queries against their 2012 election data.

While we could read the manual and query existing data, we faced a problem in regards to seeing how the API acted when live data was being fed through it. Fortunately, Associated Press would run a number of "Test Elections" leading up to the election. Each Monday, Wednesday and Friday at 19:00 Danish time, they would run a 2-hour test election where they would feed live data through the API.

We needed to know what the data output looked like, what we could do with our queries and finally, we needed to come up with a solution for the test elections, so that we could figure out the live part of the data.

Plan

1. Read through the introductory letter
2. Read through the API developer guide
3. Read through any related emails that had been forwarded to us
4. Come up with a solution for the test elections.

Problem Solving

I started by reading through the introductory letter. It was a basic "welcome customer" thing, with a few details regarding the API and test elections. By the time we got our hands on the documentation, they had already started the test elections - Team Connect wasn't brought on-board until late-ish in the process.

Next was the Developer's Guide. The guide is meant as a reference and contains information about each entry in their data files, and how to access them. However, reading through it from end to end is very dry reading, and without proper context it can get a bit confusing - So even though I read it all, I only got a superficial understanding from my first read-through. However, I would go back to it several times later to look up specific things, and for this it was very helpful.

The various emails were fairly helpful as well, as they detailed recent changes, or things to look out for. However, one thing that was very strange to us was the fact that they kept changing stuff, and that their test elections were as much for their own benefit as it was for ours. This is odd, because Associated Press has been tracking these elections for many years, and they already had a complete dataset from the 2012 elections - So you would think that they would have things under control by now.

Query Explorer

Finally, it was time to test the Query Explorer web page. Due to confidentiality, I'm not allowed to link directly to the page (as it requires DR's unique API key), but I can show some screenshots to give you an idea of how it works.

⁴ See Appendix

⁵ See Appendix

⁶ The query explorer web page won't work without the API key provided to us, and that is unfortunately confidential.

Elections API Query Explorer

Documentation & Guides

- Developer Guide
- Release Notes
- Calendar 2016
- Elections Online FTP Guide

Welcome

This page lets you explore available elections data by Date, State, and Test/Live setting. You can do this in any order. For example, choose a state and the Election Date menu will show only applicable dates. Alternatively, choose an election date and the statePostal menu will only show applicable states.

Explore the parameters and try queries using 'Issue Request'. Use 'Copy to Clipboard' to export your query.

Election Reports

See and monitor the latest reports

- Link to all **Live** reports
- Link to all **Test** reports

See Available Data & Construct API Queries

API Key CONFIDENTIAL

Parameters

Election Date statePostal test

level officeID winner raceType

format

RaceId, SeatNum, SeatName, UnContested, Party, National

CandidateInfo, SetZeroCounts, OmitResults

Notes

These menus contain currently available election data. Select in any order to see available results. statePostal supports a list, for example: statePostal=nd,sd,ok
This explorer will automatically set 'test=true' for test election dates.

Values in these menus are static; there may not be matching election data. officeID and raceType support lists, for example: officeID=a,b

Get your API responses in either XML or JSON format.

Parameters you can manually add to further filter your query.

Parameters you can manually add to manipulate the response.

Your Query

 Copy To Clipboard  Issue Request

 Select a valid election date

Response

 For demonstration purposes, queries issued from this explorer will return a maximum of one race.

This is the initial screen that you see when you access the web page. There are a number of dropdown menus that you can use to specify your query, to a good amount of detail. However, not *all* the available query options are available through this interface. I have black-boxed the references to the API key here, but you should be able to see how it works anyway.

Elections API Query Explorer [↻](#)

Documentation & Guides

- Developer Guide
- Release Notes
- Calendar 2016
- Elections Online FTP Guide

Welcome

This page lets you explore available elections data by Date, State, and Test/Live setting. You can do this in any order. For example, choose a state and the Election Date menu will show only applicable dates. Alternatively, choose an election date and the statePostal menu will only show applicable states.

Explore the parameters and try queries using 'Issue Request'. Use 'Copy to Clipboard' to export your query.

Election Reports

See and monitor the latest reports

- Link to all **Live** reports
- Link to all **Test** reports

See Available Data & Construct API Queries
API Key

Parameters [↻](#)

Date=2012-11-06 statePostal test

level officeID=P winner raceType

format=xml

RaceId, SeatNum, SeatName, UnContested, Party, National

CandidateInfo, SetZeroCounts, OmitResults

Notes

These menus contain currently available election data. Select in any order to see available results. statePostal supports a list; for example: statePostal=nd,sd,ok

This explorer will automatically set 'test=true' for test election dates.

Values in these menus are static; there may not be matching election data. officeID and raceType support lists; for example: officeID=a,b

Get your API responses in either XML or JSON format.

Parameters you can manually add to further filter your query.

Parameters you can manually add to manipulate the response.

Your Query [↻ Copy To Clipboard](#) [➤ Issue Request](#)

/elections/2012-11-06?format=xml&apikey=_____&officeID=P

Response [➤ Follow Next Link](#)

⚠ For demonstration purposes, queries issued from this explorer will return a maximum of one race.

ℹ Follow the **Next Link**, located at the end of each response, to **receive only updated results**. Keep following Next Links for continual updates.

```
<Vote ElectionDate="2012-11-06" Timestamp="2017-01-02T11:10:28.862Z">
  <Race Test="0" ID="0" Type="General" TypeID="6" OfficeID="P" OfficeName="President" National="1">
    <ReportingUnit StatePostal="US" Name="United States" Level="national" ElectTotal="538" LastUpdated="2016-07-26T17:41:11.483Z">
      <Precincts Reporting="173375" Total="174047" ReportingPct="99.61" />
      <Candidate ID="1918" Party="Dem" Incumbent="1" First="Barack" Last="Obama" PolID="1918" BallotOrder="2" PolNum="1918" VoteCount="62615486" ElectWon="332" Winner="X" />
      <Candidate ID="893" Party="GOP" First="Mitt" Last="Romney" PolID="893" BallotOrder="1" PolNum="893" VoteCount="59142804" ElectWon="206" />
      <Candidate ID="31708" Party="Lib" First="Gary" Last="Johnson" PolID="31708" BallotOrder="3" PolNum="31708" VoteCount="1211982" ElectWon="0" />
      <Candidate ID="895" Party="Grn" First="Jill" Last="Stein" PolID="895" BallotOrder="4" PolNum="895" VoteCount="431719" ElectWon="0" />
      <Candidate ID="713" Party="Ind" First="Virgil" Last="Goode" PolID="713" BallotOrder="5" PolNum="713" VoteCount="118272" ElectWon="0" />
      <Candidate ID="61907" Party="PPP" First="Roseanne" Last="Barr" PolID="61907" BallotOrder="7" PolNum="61907" VoteCount="56349" ElectWon="0" />
      <Candidate ID="43773" Party="JP" First="Rocky" Last="Anderson" Abbrv="Andersn" PolID="43773" BallotOrder="10" PolNum="43773" VoteCount="38747" ElectWon="0" />
      <Candidate ID="62535" Party="AIP" First="Thomas" Last="Hoeftling" PolID="62535" BallotOrder="8" PolNum="62535" VoteCount="33509" ElectWon="0" />
      <Candidate ID="61513" Party="Ind" First="Randall" Last="Terry" PolID="61513" BallotOrder="21" PolNum="61513" VoteCount="12986" ElectWon="0" />
      <Candidate ID="56224" Party="Una" First="Richard" Last="Duncan" PolID="56224" BallotOrder="28" PolNum="56224" VoteCount="12148" ElectWon="0" />
      <Candidate ID="62562" Party="PSL" First="Peta" Last="Lindsay" PolID="62562" BallotOrder="6" PolNum="62562" VoteCount="7588" ElectWon="0" />
      <Candidate ID="100013" Party="NPD" Last="None of these candidates" Abbrv="None" PolID="100013" BallotOrder="27" PolNum="100013" VoteCount="5753" ElectWon="0" />
      <Candidate ID="59208" Party="RP" First="Chuck" Last="Baldwin" PolID="59208" BallotOrder="20" PolNum="59208" VoteCount="4737" ElectWon="0" />
      <Candidate ID="43982" Party="CST" First="Will" Last="Christensen" Abbrv="Chrstnsn" PolID="43982" BallotOrder="29" PolNum="43982" VoteCount="4283" ElectWon="0" />
      <Candidate ID="59660" Party="Obj" First="Tom" Last="Stevens" PolID="59660" BallotOrder="15" PolNum="59660" VoteCount="4066" ElectWon="0" />
      <Candidate ID="61912" Party="SPU" First="Stewart" Last="Alexander" Abbrv="Alexandr" PolID="61912" BallotOrder="13" PolNum="61912" VoteCount="3946" ElectWon="0" />
      <Candidate ID="3486" Party="SRP" First="James" Last="Harris" PolID="3486" BallotOrder="16" PolNum="3486" VoteCount="3849" ElectWon="0" />
      <Candidate ID="62588" Party="GRP" First="Jim" Last="Carlson" PolID="62588" BallotOrder="23" PolNum="62588" VoteCount="3172" ElectWon="0" />
      <Candidate ID="62536" Party="ATP" First="Merlin" Last="Miller" PolID="62536" BallotOrder="14" PolNum="62536" VoteCount="2833" ElectWon="0" />
      <Candidate ID="62537" Party="WTP" First="Sheila" Last="Tittle" PolID="62537" BallotOrder="11" PolNum="62537" VoteCount="2504" ElectWon="0" />
    </ReportingUnit>
  </Race>
</Vote>
```

This is the same web page after a query. You can choose the response as either XML or json - In this case, it's XML. At the bottom of the response (not shown here), is a "Next Link". This link consists of the initial query plus a timestamp marker, and when you query with that one, the API only responds with the values that have been changed / updated since the last query, plus a new Next Link. The Query Explorer also has a button for fishing out the "Next Link" and calling the next query. If I press that button, I get the following response:

Response [➤ Follow Next Link](#)

⚠ For demonstration purposes, queries issued from this explorer will return a maximum of one race.

ℹ Follow the **Next Link**, located at the end of each response, to **receive only updated results**. Keep following Next Links for continual updates.

```
<Vote ElectionDate="2012-11-06" Timestamp="2017-01-02T22:09:46.845Z">
  <link rel="next" href="https://api.ap.org/v2/elections/2012-11-06?officeID=P&minDate=2016-07-26T17:30:41:30:11.483Z" xmlns="http://www.w3.org/2005/Atom" />
</Vote>
```

As you can see, the response is empty, save for the Next Link - this is as it should be, since I'm querying the 2012 election data, and there are no live updates to it. However, this also highlights the problem that we were now facing:

While we could see the structure of the response data, and we could get it in different formats (XML and json), there was still no way to tell how the data would act in a live scenario. We could wait until the test elections and sit and manually update the query explorer, but that would be entirely too much manual labor, plus the test elections would be running

after normal business hours, and the prospect of spending 3 evenings a week doing extra work was not really a good one.

We discussed how we wanted to handle this, and we decided that we needed to be able to “record” these test elections, so that we could run them whenever we wanted to (during normal work hours), and use those datasets as a basis for testing the live portions of our application.

We needed to create our own tools for this. The tools that we created will be detailed in the next chapter, but for now I’m going to skip ahead in time to when we had the tools made and were able to record and replay the test elections.

Test Election Data

During development, we would go on to record 14 test elections. In addition to just “replaying” the elections for testing purposes, we also spent a good amount of time manually analyzing this data. Our initial assumptions about the live data was that Associated Press had been doing this for at least the past 4 years if not longer, so that they would have everything already locked in, in terms of structure and amount of data. However, there were several things that changed from test to test, and there were a number of issues that came to light. At first, we thought that these issues originated from the API itself, but it turned out that some of them were just part of the election itself.

While we were just focusing on Donald Trump and Hillary Clinton, there were actually a total of 31 presidential candidates in this election. Normally, only the Democratic and Republican nominees actually matter in an election, but the 2016 election was far from normal. The two candidates were some of the least popular major candidates in the history of the US elections, and there was the chance that at least one very popular candidate (Bernie Sanders) might run as a 3rd party candidate. This was something that we might need to address - but we would need to get a response from our customer first.

Another big issue was how the presidential election results were called on a state-by-state basis. Associated Press uses both prognosis data and actual data for the API, yet we couldn’t find anything to differentiate between the two. In fact, the only way we could see a winner of a particular election (presidential, senate, house, etc.) was if a `winner=x` flag had been set or not. The normal procedure for determining a winner was therefore:

1. Votes counted.
2. Winner declared at x% counted votes.
3. Winner flag assigned.
4. `electWon` number (or similar) assigned.

However, in some cases, the winner was declared at 0% counted votes. In some cases, the states stopped reporting after the winner flag had been assigned, but before they had updated the `electWon` entry. Other times, the votes would be counted, but a winner wouldn’t be declared for a particular state. In fact, as of the time of writing this dissertation, New Hampshire and Michigan haven’t been officially called, even though 100% of the votes have been counted.

The issue of split electoral votes for the presidential election (for Maine and Nebraska) wasn’t reflected in our test data either. Even though both of these states could theoretically split the Electoral Candidates between two or more nominees, all the test elections awarded all the candidates to one or the other nominee - So we had no idea how Associated Press would actually handle this, especially because the `winner=x` flag seemed to be the only real indicator of how a state was won.

Evaluation and result

Our research into the API itself answered a lot of the questions that we had, but it also brought to light a number of new issues that we had to consider, and get some answers for. We realized that we couldn’t count on *just* the winner flag for our results, because in some cases it never got assigned. There was the unanswered question of split votes. It was still unclear whether we needed to account for 3rd party candidates or not, and there was no way to differentiate between prognosis data and actual data.

Overall evaluation and result

Like most aspects of this project, the research phase went through several iterations. The majority of our research was done up front, but we had to go back to it several times to clarify issues that we discovered along the way. Peter and I didn’t have much decision power in regards to determining what we did and didn’t want to show, and how to handle the special cases that we had discovered, so there were lots of emails being sent back and forth between us and the

customer (the news department), as well as several clarification meetings. In the end though, several decisions were made:

- 3rd party candidates were deemed irrelevant, and were to be ignored.
- With the data being somewhat unreliable, it was important that we didn't add to any issues that might arise. For this reason, we needed to modify the raw API data as little as possible. No rounding of numbers unless absolutely necessary. No manually adding things together if it could be avoided, and so on.
- Since the "reports" part of the API already had things added together, we decided that we would use these for the House and Senate elections. These were deemed to be less important than the Presidential election, so it wasn't necessary to get super live data. The once every 3 minutes update would suffice for this. We didn't need to know the details of these elections, just the total numbers.
- Split electoral votes didn't show up during our research phase (more on this in the "Late Changes" chapter)
- Only actual results mattered. We would use as little prognosis data as possible. This would also mean that we would have no "predictions" on our map, but rather just show it as blank / undecided until a winner had been determined.

All in all, the research that we did was invaluable to the project. Even though the customer (the news department) was technically our "Domain Expert", we still needed to be very well informed about the election and the inner workings of it, in order to properly complete the assignment that we had been given. There was also the issue of communication speed. While we were using Scrum and being very agile in general, communication between us and the customer could sometimes be quite slow, so the more we could answer ourselves, the better.

What all the work we did in the research phase boils down to is basically, **"How can we reduce the number of unknown factors to as little as possible?"**.

Chapter 4: The Backend

The API scraper and test election runner

Introduction

With the initial research completed, and us having a decent overview of how everything worked, it was time to start actually coding the application. Peter had already set up an empty backend project in NodeJS and on Github, so it was just a matter of getting started.

We now knew how the data from the API looked, and we knew how to access it, so it was time to adjust our problem definition a bit. We had discovered a new set of challenges during our research phase, and we would have to address those when we created the backend.

Problem definition

With our chosen frameworks and technologies, how can we provide live data from an outside API when there are many unknowns?

- *How can we reduce the number of unknowns to as little as possible?*
- *How do we fetch the data from the API?*
- *How do we feed the data into our own database?*
- *How can we test the live aspect of our application?*

Method

For the tasks ahead of us, it was a matter of just coding everything in Javascript / NodeJS, and then testing and verifying it. Testing would be done mainly through logging to the console window, to see that we got the data we expected. Pretty much every aspect of the code was subject to logging at one point or another, but we have of course removed most of these from the final product.

Plan

- Create an automated API scraper that can run on a delay / timer.
- Find a way to save data from the API.

- Create a way to pass data into our own database(s).
- Figure out a way to “Replay” the test elections.

Problem Solving

NodeJS works by using “modules” that have different functionality. There are a lot of modules that are officially part of NodeJS, but you can also easily create your own. Using it reminded me a lot of using C# and .Net where you would just grab the libraries you needed, and get leaner code that way. It was also very helpful for separation of concerns, and making sure that each module only contained code relating to one specific set of functionality.

You will notice that we used and created quite a lot of modules for our backend. It can get a little bit confusing if you're not the one who's been working with them, but following the “require” constants at the beginning of each file should let even someone completely new to NodeJS follow the trail of breadcrumbs. We didn't have a set architecture / organization of these modules when we started out, because we didn't *quite* know how many modules we were going to need, but we still managed to keep everything fairly organized inside the folder structure of the project. Partial modules would go into the `modules` folder, while “end modules” - ie. the modules that you would actually run through the console - would go into the root folder of the project.

The API Scraper

What we decided to do was to create an API scraper that we could use to get the queries that we wanted, and then do whatever we needed to do with the data received. Peter and each created our own scraper, and for the first trial run we would run both of them. We ended up just using the one I created, so that's the one I'm going to focus on here. So, the first thing to do is to grab the modules that I would need for the scraper.

```
const rp = require('request-promise');
const fs = require('fs');
const config = require('./../config');
```

[Request-promise](https://www.npmjs.com/package/request-promise)⁷ is a module that lets you chain events in a try-catch block. We use this both for the chaining, and for the built-in error handling. [Fs](https://nodejs.org/api/fs.html)⁸ is NodeJS' file system module, that enables us to save to and read from files - something that we will need for our “Replay” tool. Finally, “config” is actually a link to a folder of our own, containing configuration files that we can re-use across our different tools.

Next thing I needed to do is set up the options for fetching our data.

```
var requestOptions = {
  url: config.AP.PATH + config.AP.ELECTION + "?" + config.REQUEST_FORMAT + config.MODE + config.AP.APIKEY,
  method: 'GET',
  gzip: true,
  json: true,
  headers: {
    'Accept': 'application/json; charset=utf-8',
    'Accept-Encoding': 'gzip'
  },
};
```

We had the relevant API variables stored in our config file. `PATH` refers to the basic url to fetch data from. `ELECTION` is the date of the election that we want to retrieve, `REQUEST_FORMAT` is whether we want it in json or XML (we chose json for this), `MODE` is a flag that tells the API whether we want test or live data, and the `APIKEY` is just what it sounds like.

Now I had the basic setup for the scraper, and it was time to code in the actual functionality. I would need two different functions - one for taking a “full dump”, and one for handling the “next link” updates.

```
var getElectionFull = function() {
```

⁷ <https://www.npmjs.com/package/request-promise>

⁸ <https://nodejs.org/api/fs.html>

```
requestOptions.url = config.AP.PATH + config.AP.ELECTION + "?" + config.REQUEST_FORMAT + config.MODE +
config.AP.APIKEY + "&officeID=P";
// Get requestOptions
rp(requestOptions)
  .then(function(json) {
    // Set url to next request
    requestOptions.url = json.nextrequest + config.AP.APIKEY;
    console.log('Full dump done.\nNext request: ' + requestOptions.url);
    // Write full dump
    fs.writeFile("./scraper-dump/" + Date.now() + "_full.json", JSON.stringify(json, null,
4),
    function() {
      console.log('Full dump file saved\n');
    });
  }).catch(function(err) {
    console.error(err);
  })
}
```

You might notice that I set the `requestOptions` url manually here - this was something that was added in later, because we decided to change our approach slightly, and get our House and Senate data in a different way. To start with though, we just used our default `requestOptions`, which queried ALL the data for that election, rather than just the presidential election data.

The functionality here should be fairly easy to read. I create a function called `getElectionFull`. Request-promise then attempts to query the API using the `requestOptions` variable.

If successful, we receive a json object containing our data. We then change our `requestOptions` url variable to be equal to the "nextrequest" entry in the json file - this is the "next link" that I talked about earlier - and then we log it, so that we can get verification that this has happened. Finally, we use the `fs` module to write the json object to a file inside our `scraper-dump` folder, and we log that as well. If not successful, the application throws an error instead, and then we'll need to figure out what went wrong.

So this takes care of the initial data dump, and it sets our `requestOptions` to query for the next update. We then need to also create the function for this:

```
var getElectionUpdate = function() {
  rp(requestOptions)
    .then(function(json) {
      // Set url to next request
      requestOptions.url = json.nextrequest + config.AP.APIKEY;
      console.log('Update dump done.\nNext request: ' + requestOptions.url);
      // Write update dump
      fs.writeFile("./scraper-dump/" + Date.now() + "_update.json", JSON.stringify(json, null,
4), function() {
        console.log('Update file saved\n');
      });
    }).catch(function(err) {
      console.error(err);
    })
};
```

The functionality here is almost identical to the full dump one. First we query the API using request-promise. Then we adjust our `requestOptions` to fish out the "nextlink" from the received file, and then we write the received json object to a file. The differences here is that we start with a "nextlink" query, and we save the file with a slightly different filename.

Finally, because the scraper will also be a node module, we need to set up the exports, so that we can import the functions in other files.

```
var full = getElectionFull;
var update = getElectionUpdate;

module.exports = {full, update};
```

Here, I set up two simple variables to equal the functions I just made, and then flag them as exports. This tells NodeJS that when we “require” them into another file, that those are the two things that can be called from that other file. You can use these exports to hide internal logic in a module and only allow the other file to access exactly what you want it to access.

As I mentioned previously, we originally used the election scraper detailed above for everything. However, later down the line, once we had gotten clarification about how much House and Senate data we needed to provide, we decided to go a different route with this data. So the election scraper would only be used for the Presidential election, where a high degree of detail was needed.

The Report Scraper

For the House and Senate data, I created a “Report Scraper”. Along with the very detailed election data that Associated Press provided, they also provided “reports” - These reports were basically summaries that would be sent out every 3 minutes during the election.

One reason we chose these reports instead of the more detailed live data was, among other things, that Associated Press did the math for us. Instead of us having to add up things in our own application (and possibly getting math or rounding errors in the process), we could “rely” on Associated Press to do it for us. Also, we didn’t need district-by-district information for this data, but rather just a summarized overview, so it worked out well for us. Additionally, it meant that we could point the finger at Associated Press in the case of any counting errors.

Another reason for choosing the report was the fact that we had a query budget. We were allowed to make 10 queries per minute. This sounds like a lot, but if you’re making separate queries for presidential, house and senate data and getting updates for these, then that budget is quickly spent.

The way the reports worked was that you query the API for a “report overview”. This costs 1 query out of your budget. The report overview in turn contains links to the 6 different reports generated by Associated Press, and these links do *not* count against the budget. This means that we were able to get our House and Senate data at practically no cost to our query budget (1 query every 3 minutes), and have Associated Press do the math part for us. It also meant that there were an extra couple of steps to the scraper part though, so I will detail that below.⁹

```
const rp = require('request-promise');
const fs = require('fs');
const config = require('../config');
var ReportPath = "https://api.ap.org/v2/reports";
```

I started with the basic configuration of the module, where I pull in the other Node modules I need. In addition, I added the path to the reports part of the API.

```
var requestOptions = {
  url: ReportPath + "?apikey=" + config.AP.KEY + "&electiondate=" + config.AP.ELECTION + "&" + config.MODE
+ "&" + config.REQUEST_FORMAT,
  method: 'GET',
  gzip: true,
  json: true,
  headers: {
    'Accept': 'application/json; charset=utf-8',
    'Accept-Encoding': 'gzip'
  },
};
```

⁹ See appendix for an example Report Overview file

```
var requestReport = {
  url: '',
  method: 'GET',
  gzip: true,
  json: true,
  headers: {
    'Accept': 'application/json; charset=utf-8',
    'Accept-Encoding': 'gzip'
  },
}
```

Next step was to set up the query options for fetching the data we needed. Here, I needed two different sets of options. One to get the initial report overview, and a separate one for fetching individual reports.

```
var getReports = function() {
  rp(requestOptions)
  .then(function(json) {
    requestOptions.url = json.nextrequest + "&apikey=" + config.AP.KEY;
    console.log("\nReports page fetched.\n\nNext request: " + requestOptions.url + "\n");
    if (json.reports[0] != null) {
      Object.keys(json.reports).forEach(function(key) {
        requestReport.url = json.reports[key].contentLink + "?" + config.AP.APIKEY +
        "&format=json";

        var ReportName = json.reports[key].categories[1].term;
        if (ReportName === "s" || ReportName === "h") {
          rp(requestReport)
            .then(function(json) {
              fs.writeFile("./scraper-dump/" + "report_" + ReportName +
              "_" + Date.now() + ".json", JSON.stringify(json, null, 4), function() {});
              console.log("Saved: report_" + ReportName);
            }).catch(function(err) {
              console.error(err);
            });
        }
      });
    } else {
      console.log("\nEmpty reports array!");
    }
  })
};
```

Next, I dealt with the actual reports. It uses the same basic build at the election scraper, but with a couple of twists. Since the report overview returns a json object that includes an array of links (to the reports), I had to dig into this array and pull out the reports that I wanted.

The function goes through each item in the reports array (usually 6 items, although this changed from test to test), then looks at the second entry in the categories array inside the reports array, and checks the value of "term". If this value is either "s" (for Senate) or "h" for House, then it requests that report and saves it with an appropriate name. It ignores the other reports, as they are irrelevant to our application.

Normally, there will always be something in the report overview to look at, but several times during the test elections, we would get an empty object back for one reason or another. There wasn't much we could do about this, but I included a log text to warn us that this had occurred, so that we would at least know.

```
module.exports = getReports;
```

Finally, I remember to export my function, so that I can include it in other modules. Since I'm only exporting a single function, I don't need to set it up as a variable first.

The Scraper Wrapper

Now I had to tie the two scraper modules together. I could fetch the data we needed, but I still hadn't dealt with the fact that the test elections would be running after working hours - So I needed to create some sort of timer function for it. For this, I needed a wrapper module that could run the scrapers at different intervals - The presidential election data needed to be updated several times a minute, and the report data once every 3 minutes.

I looked at the [setInterval\(\)](#)¹⁰ function built into JavaScript, and that seemed to be mostly what I was after. However, it had a problem in that while I could set the interval to run, it would run forever until I manually stopped it. For this, I would need to create a custom interval function. I figured that there was no reason to re-invent the wheel if I didn't have to, so I looked around on the internet for answers, and I found a [function](#)¹¹ that perfectly suited my needs. It didn't require any adjustment, just implementation into the wrapper module I was working on.

```
const getElection = require('./modules/scraper-election');
const getReports = require('./modules/scraper-reports')

function setIntervalX(callback, delay, repetitions) {
  var x = 0;
  var intervalID = setInterval(function() {
    callback();
    if (++x === repetitions) {
      clearInterval(intervalID);
    }
  }, delay);
};
```

As usual, I start by requiring in the modules that I need for this to work - in this case, my two scrapers. After that, I define the interval function that I found. I set an internal variable to use as a counter for the number of repetitions in the interval. When `setIntervalX` starts, it will wait for `{delay}` number of milliseconds, and then fire off whatever function is called, and update the counter. Once the counter reaches the desired amount of repetitions, it then triggers `clearInterval()` and stops the loop.

Now, there was an additional challenge here: We wanted our timer to be able to start after a certain delay, but we also wanted our functions to actually start when when we want to - and not after an initial delay! In addition to that, we needed different intervals for our two scrapers. Everything runs asynchronously, so each instance of the scraper had to have its own separate timer as well. This meant that I had to get a little creative, and use multiple nested instances of `setIntervalX`.

```
setIntervalX(function() {

  // FULL DUMP EVERY 30 MINUTES

  setIntervalX(function() {
    getElection.full();
  }, 1800000, 4); // Interval for full dumps. 30 Minutes

  // ONE FULL DUMP, THEN UPDATES EVERY 30 SECONDS

  setIntervalX(function() {
    getElection.full();

    setIntervalX(function() {
```

¹⁰ http://www.w3schools.com/js/js_timing.asp

¹¹ <http://stackoverflow.com/questions/2956966/javascript-telling-setinterval-to-only-fire-x-amount-of-times/2956980#2956980>

```
        getElection.update();
    }, 30000, 240) // Interval for updates. 30 Seconds
}, 1, 1); // first full dump. Milliseconds and number of intervals
// REPORTS EVERY 3 MINUTES
setIntervalX(function() {
    getReports();
}, 180000, 40) // Interval for reports. Milliseconds and number of intervals
}, 1, 1); // Wrapper interval. Use to set initial delay
```

Here's where good code commenting came in really handy. I could delegate the timer settings to external variables and used those, but that would just be unnecessary code bloat, especially when the wrapper was as small as it was.

The internals of the wrapper works like this: First, it fires off a query to get a full report with no delay. From this first full dump, I fish out the "next link", and use this to call my update function every 30 seconds. In addition to that, I also request a full dump every 30 minutes - These were used simply for control, so that we could have something to compare data to along the way. This was especially important during our testing of the live aspect. We needed to insure that the things that got sent to the database actually matched up with what we were getting from Associated Press' API.

Finally, and also independent from the election data, we call the reports scraper every 3 minutes. We adjusted the number of repetitions for each instance of `setIntervalX` to correspond to 2 hours total (the length of a test election), and finally wrapped everything in yet another instance of our interval function, so that we could set an initial delay timer. This meant that once I got home from work, I could set up a computer to run the scraper, set the initial delay, and start it - and then not worry about it until after the test election, when I would then upload the files I had saved.

The scraper was an "end module", so I didn't need to include an export for this file.

The Test Runner

Now we had working scrapers, so we were able to get the data we wanted from the API, and we were able to save that data locally, and ordered chronologically. The next piece of functionality to make would then be a module that would let us "replay" the test elections. For this, we created three small modules.

Peter was the main programmer responsible for these modules, so I'm not going to go into too much detail on them, but just give a basic explanation of how they work.

The first, called `run_getAllFileNames.js` would look into a folder that we specify (in this case `scraper_dump`), and then "walk" down the list of files, using a node module called.. [walk](https://www.npmjs.com/package/walk)¹². It would then generate a `.json` file that was basically just an ordered list of the filenames in the folder containing the data dumps from a given test election.

The second module, called `passElection.js` was responsible for passing election data into our Firebase database. This module only contains the passing functionality. Towards the end of the project, we also added the ability to pass the data into our MongoDB as well. I will touch on our use of MongoDB later on in this report.

The third module, called `run_preRunElection.js` was the one that tied everything together. This module would look at the list generated by our `run_getAllFileNames.js`, and then apply `passElection.js` to each entry in that list. The really important part here was that we were able to set the timings of how fast we wanted to pass the data. This meant that we could do a test election in a couple of minutes instead of the two hours that a test election normally took.

¹² <https://www.npmjs.com/package/walk>

Evaluation and result

We had achieved what we set out to do here. We were able to get the data that we needed from the API. We could save the data to files for later use, and we could do “replays” of the test elections, and do this on our own time, rather than during when the test elections would actually be run. Finally, we were able to push the data to our database(s).

The API scraper was some of the first NodeJS code that I had ever worked on, and it’s something that I’m still fairly proud of. It does what it sets out to do, and it does so in a manner that’s fairly efficient. Not only that, but since we started out by each making our own version of the scraper, and we ended up using mine, it must mean that it’s good for the job it was designed for. Additionally, it turns out the scraper was well designed enough that we were able to use it for the final product as well, with only a few minor changes. It also had a very simple, but very effective way of handling any issues that might arise during the *actual* election.

The Test Runner tool was also invaluable. The most important part of the assignment was that the whole thing had to be able to run live - and it absolutely *had* to work. There would be no second chances, as once the election was over, we would just be fetching static data from our database. Being able to replay the test elections, and see how our front end application reacted to the changing data was great, not only for testing the application, but also for showing our customer how everything would end up looking in a live scenario.

Chapter 5: Live data

Firebase and MongoDB

Introduction

Of course, just getting the data from the API and saving it locally wouldn’t do much for us. We needed a solution that would also be able to handle the live aspect of the application, with theoretically many updates per minute. Peter had already chosen to work with Firebase - but this was his first time doing so, so we had a bit of discovery to go along with this as well. And, to be on the safe side, he wanted to make sure that we had a backup in case anything went wrong - so we needed to be able to fall back to a mongoDB solution. This would also be used once the election was over, and we only needed to display static data. At least, this was the assumption that we worked on to start with.

What is Firebase?

[Firebase](https://firebase.google.com/)¹³ is a platform for both mobile and web that has many tools and features to help build applications. It started out as a real-time cloud-based distributed database though, and that is what we were going to use it for. Firebase is easy to use, very good at handling live data (it’s supposedly fast enough to use for real-time gaming), and has a lot of documentation and tutorials available for it.

Problem definition

With our chosen frameworks and technologies, how can we provide live data from an outside API when there are many unknowns?

- *How can we reduce the number of unknowns to as little as possible?*
- *How do we fetch data from the API?*
- *How do we feed the data into our own database?*
- *How do we set up a database that can handle the live aspect well, and without bottlenecks?*
- *How do we handle our data once the election is over?*
- *How can we test the live aspect of our application?*

Method

Setting up a Firebase account is a very simple matter. It’s mostly just a matter of following a tutorial like the [one they provide](#)¹⁴, and then adjusting it to our needs. Data in Firebase is stored as Json, so as long as we get data in Json format from the API, there is very little that needs to be done to pass it into our Firebase database.

¹³ <https://firebase.google.com/>

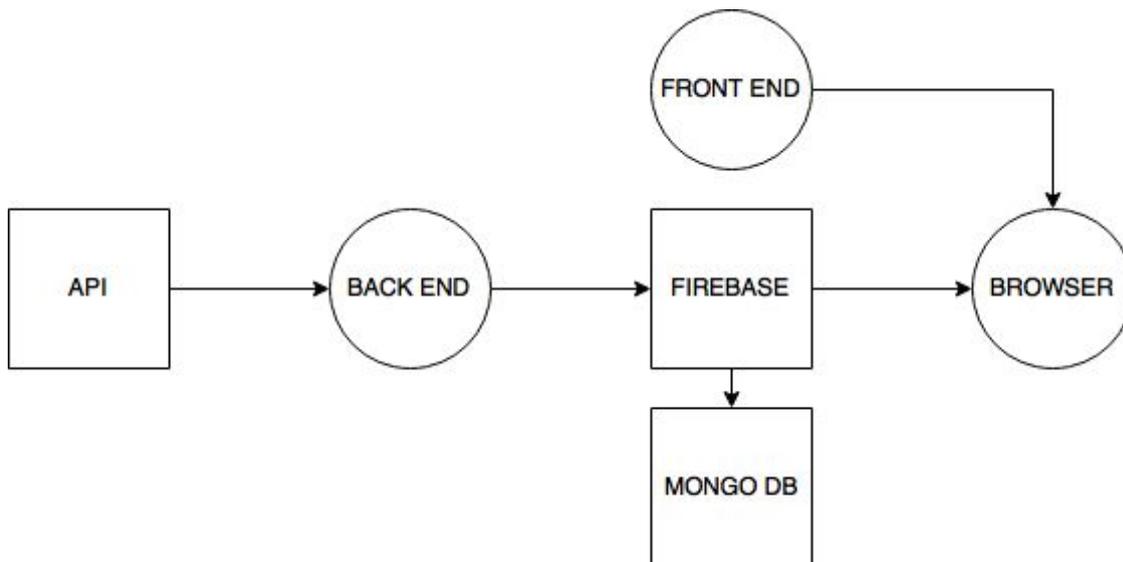
¹⁴ <https://codelabs.developers.google.com/codelabs/firebase-web/#0>

Since we considered the mongoDB as a backup, we didn't implement this until much later. I didn't personally have any hands-on with the mongoDB part of the assignment though - Adam, who joined the team in the last 1½ weeks was the one who handled that - So I will not be going into detail with that, but just providing a very quick overview.

Plan

1. Set up a Google Firebase account, and hook it to our application.
2. Test the solution as much as possible.
3. Set up a mongoDB on our own servers to use as a backup.
4. Figure out how to switch between live and static states.

Problem Solving



For most of our work, we were still under the assumption that we would be using MongoDB as a backup once the live aspect of the application was over - So our planned architecture was still what we were using. We would later have to change that¹⁵, but for now we kept to our original plan.

Firestore

Since we were only using the database aspect, all we had to do was to make an “empty” account, and then feed it data from our API. Putting data into the database was handled by the `passElection.js`¹⁶ module mentioned in the previous chapter.

We tried to keep the structure very similar to the data that we received from the API, but we did have to make some changes along the way, for a couple of reasons: First, we had to make sure that names etc. would make sense in our front-end application, and second, we had to make sure that we used the same naming conventions for everything. This was especially important for the bar that I created, since it would be multi-function and provide views for both the Presidential, House and Senate elections, depending on what kind of data you fed into it.

Implementing Firestore into the front-end application wasn't difficult either. You basically have to add “hooks” into your module that look at specific areas of your database, and as soon as anything changes in the database, it is then automatically synchronized to the browser. I will go into more detail with this in chapter 6.

MongoDB

Our MongoDB solution wasn't added until very late in the development process. As shown by the architecture drawing, it was meant as a fallback, and a post-live solution that would only take occasional snapshots of the Firestore database, and store it. Like I mentioned earlier, I didn't have any direct involvement with the MongoDB part, so I can't go into any detail with it.

¹⁵ See Chapter 7: Late Changes

¹⁶ See Appendix

It was the same basic idea as with Firebase, except for the fact that we wouldn't be pushing live updates to the front-end with our MongoDB. As a feature of the application though, it wasn't developed until much later¹⁷

Evaluation and result

Waiting so long to implement MongoDB was a bit of a gamble for us - we were so focused on the live aspect which, to be fair, was also the most important part, that we kept pushing back the implementation of MongoDB. Had we not done so, we would have been a lot less stressed towards the end of the development cycle, but it would probably have required that we had gotten Adam assigned to our team a good deal earlier than he was.

We assumed that we would need to switch to the MongoDB solution (which was hosted on DR's own servers rather than Google's cloud servers for Firebase), which is why we kept it as part of our plans that it would be implemented - and it's a good thing we did. While it turns out we didn't need MongoDB as a non-live backup after all, we ended up having another very important use for it.

Implementing Firebase was quite easy. We didn't have to really worry about having a database schema or anything like that, and we were able to (and did) change our structure several times along the way without much effort. Together with the tools we created for the backend, it was very easy for us to "replay" a test election any time we wanted to, and this was immensely helpful in making sure the front end worked how it was meant to.

We still hadn't quite figured out how to handle our data once the election was over though, and we didn't figure this out until much later.

Chapter 6: The Frontend

Generic election bar (and live map)

Introduction

For the frontend part of our application, we also used Javascript, and a number of related frameworks, such as ExpressJS, Handlebars & Less. The idea was to keep everything in the same language, more or less. I personally worked on the bar that would be featured both on the election page, but also on the front page of dr.dk, as well as several related news pages. Peter worked on the interactive map, and the accompanying list of states. Since I hardly did any work on the map itself, I'm going to focus on the work that I did with the bar, and how I got that working.

Before I go any further, I will try and explain the basic organization of dr.dk's website in terms of development. The website itself runs in a highly modified version of Drupal 7. Drupal takes care of the "frame" around the rest of the site - ie. the basic menus, shared assets and so on, while more specialized tools are used to create articles and so on. Some sections of the overall website, especially "theme" sites, such as for the olympics, are created as almost separate sites that are then embedded into the larger framework.

What this means in terms of our application is that we just needed to focus on the actual bar and the actual map. The rest election website was created by a different team, and the articles were written/created and managed by a third team (the news department). Because we didn't have control over anything but the elements we were working on, we had to make sure that they were super modular, and that they would work in the environments they would be deployed to.

Problem definition

With our chosen frameworks and technologies, how can we provide live data from an outside API when there are many unknowns?

- *How can we reduce the number of unknowns to as little as possible?*
- *How do we fetch data from the API?*
- *How do we feed the data into our own database?*
- *How do we set up a database that can handle the live aspect well, and without bottlenecks?*
- *How do we handle our data once the election is over?*
- *How can we test the live aspect of our application?*

¹⁷ See Chapter 7: Late Changes

- *How do we display live data from our API?*
- *How can we make sure that our elements will work with the rest of the website?*

Method

Everything in the front-end would be coded in Javascript, with several frameworks added in. We used Less for styling, Handlebars for building the pages, and Grunt to compile this into actual HTML and CSS. The reason for using these tools was simply that they are part of the regular production pipeline at Team Connect, so they were familiar tools to Peter and Adam (but not so much me, since this was my first NodeJS/Express application that I'd worked on).

Plan

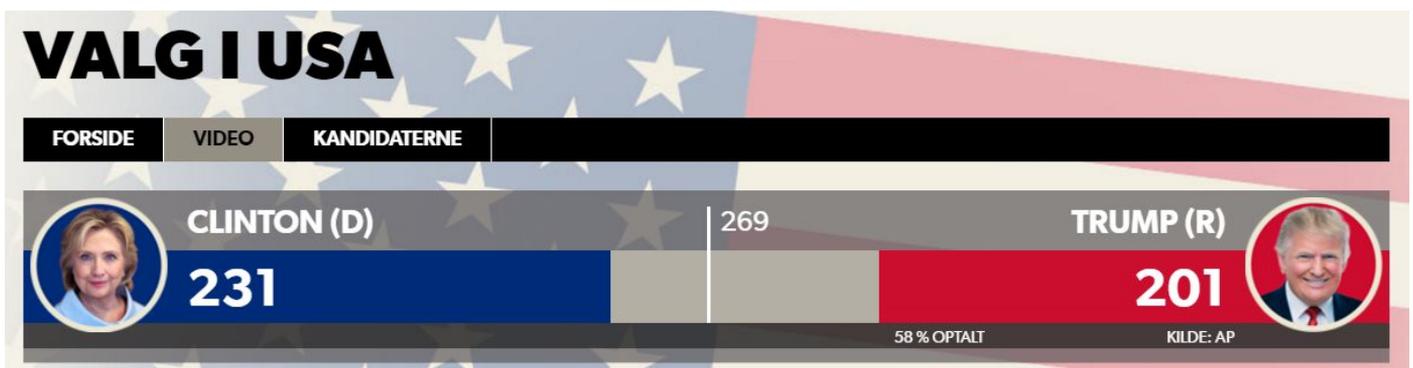
- Create an election bar that is multifunction, and can show all 3 types of election that we're covering.
- Create an interactive svg map of the United States that can show the results of the presidential election.
- Create a state list that can show vote distribution for the presidential election in percentages, as well as show the winner of a state.
- Connect everything to Firebase and make sure we have live functionality.
- Make sure both bar, map and list are modular and can be inserted anywhere on DR's website.

Problem Solving

For the problem solving part of this chapter, I'm going to focus almost entirely on the part that I worked on, the election bar. I will also mention the map, but the map was one of the things I had the least to do with during the project, so I will just give a very quick overview.

The Election Bar

This is the only part of the project that I worked on that you can actually see in action. So far, my involvement had been in creating backend tools that the public would never see, and in doing research to enable us to actually do this assignment.



The bar consists of 3 parts: The `module-bar.hbs` file (that gets compiled into an html file using Grunt), the `election-bar.js` file that holds the functionality, and the `bar.less` file that handles styling.

Handlebar / HTML

The HTML code for the bar is extremely simple. It merely consists of some wrapped div's with classes attached to them. The code itself looks like this:

```
<link rel="stylesheet" type="text/css" href="{{config.baseUrl}}/election-assets/css/bar.css">
  <div class="wrapper {{electSwitch config.params.type type}}">
    <div class="bar-wrapper-title">
      <div class="bar-text-cand-dem"></div>
      <div class="bar-text-cand-oth"></div>
      <div class="bar-text-cand-gop"></div>
      <div class="bar-divider"></div>
      <div class="bar-center-text"></div>
    </div>
  <div class="bar-wrapper-bar">
```

```
<div class="bar-result-dem"></div>
<div class="bar-result-oth"></div>
<div class="bar-result-gop"></div>
<div class="bar-result-dem-text"></div>
<div class="bar-result-oth-text"></div>
<div class="bar-result-gop-text"></div>
<img class="badge-dem">
<img class="badge-gop">
</div>
<div class="bar-wrapper-footer">
  <div class="bar-counted"></div>
  <img class="bar-oth-box"/>
  <div class="bar-oth"></div>
  <img class="bar-insuff-box"/>
  <div class="bar-insuff"></div>
</div>
</div>
<script src="{{config.baseUrl}}/election-assets/js/base-bar.js"></script>
```

There are really only a few things to note here. You will see some text wrapped in `{{ }}` - These are HandlebarsJS expressions¹⁸ used to refer to properties or functions from other files. Of special interest here is the second line of code:

```
<div class="wrapper {{electSwitch config.params.type type}}">
```

`electSwitch` refers to a function in a `helper.js`¹⁹ file that we created for our views. Peter did most of the work on the helper file, but the function I'm referring to in the handlebars code is this one (which I also worked on):

```
// Viser forskellige barrer, alt efter hvilken switch man bruger (pres, house, sen)
exports.electSwitch = function (url_parm, module_parm) {
  var name = "pres";
  if (module_parm !== undefined) {
    name = module_parm;
  } else if(url_parm !== undefined){
    name = url_parm;
  }
  return "container-"+name;
}
```

When you want to embed the election bar on a page, you call it with Handlebars as follows:

```
{{>module-bar type='pres'}}
```

This tells the application to call the bar, and give it the type of `pres`. The application will read the html for the bar, which will trigger the `electSwitch` helper, and the function will then return a value of `"container-"+name`. Everything inside the Handlebars expression will be replaced with the value that is returned, so what the rendered HTML for the second line ends up looking like is:

```
<div class="wrapper container-pres">
```

It might seem like a lot of steps to go through, but it was necessary in order to ensure that the correct bar is shown when you call it. The `electSwitch` helper was actually a fairly late addition to the code that we only discovered we needed after we were able to test our application in a fully live environment. Other than this tricky little part though, the rest of the HTML part of the bar is very simple. Simply divs and classes.

¹⁸ <http://handlebarsjs.com/expressions.html>

¹⁹ See Appendix

The script called at the bottom of the file - `base-bar.js`, is the compiled version of `election-bar.js`, and is almost identical, save for a few added requirements that are handled by the Grunt taskrunner.

Less

The `bar.less`²⁰ file is also very straightforward. It is simply a set of CSS properties for the different classes. Less is part of the production pipeline at DR, but this file could have just as easily been made without it. After using it though, I would say that using Less makes it much easier to keep your css organized, as it supports nesting of properties. I'm not going to spend too much time on this file, as it should be self-explanatory. The important thing to take away here is that it contains css properties for all the classes that I assign to the various divs in the Handlebars file.

Javascript

The `election-bar.js`²¹ file is the heart of everything. It's a fairly large file because it has to be multi-function, so there are a lot of switches and variables to define. We can roughly divide it into 4 parts. First, we have the "setup" part of the script:

```
"use strict";

// Adding Vanilla prototype features
require('./vendor/vanilla.js');

// Application configuration
var config = require('./.././././config');

// Firebase require and initialize
var FBapp = require('./firebase')

var ElectionBar = function() {

    // General election
    var officeType; // Election type. (p)residential, (h)ouse, (s)enate
    if (document.querySelector('.container-pres') !== null) {
        this.mainContainer = document.querySelector('.container-pres');
        officeType = 'p';
    }

    if (document.querySelector('.container-sen') !== null) {
        this.mainContainer = document.querySelector('.container-sen');
        officeType = 's';
    }
    if (document.querySelector('.container-house') !== null) {
        this.mainContainer = document.querySelector('.container-house');
        officeType = 'h';
    }

    var officeTotal; // Total number of seats/electives available. 538 for (P)residential, 435 for (H)ouse,
    100 for (S)enate.
    var demWon; // Number of units won by democrats
    var gopWon; // Number of units won by republicans
    var othWon; // Number of units won by others
    var candDem; // Candidate name.
    var candGop; // Candidate name.
    var candOth; // Candidate name.

    // President
    var pctRep; // Precincts reporting in
```

²⁰ See Appendix

²¹ See Appendix

```
var candShortDem; // Last name.
var candShortGop; // Last name.

// Senate and House
var demHold; // Holdovers from last election(s). Relevant for senate, but house has this data as well.
var gopHold; // Holdovers from last election(s). Relevant for senate, but house has this data as well.
var othHold; // Holdovers from last election(s). Relevant for senate, but house has this data as well.
var demLead; // Expected to win, but not verified.
var gopLead; // Expected to win, but not verified.
var othLead; // Expected to win, but not verified.
var demInsuff; // Insufficient votes to determine.
var gopInsuff; // Insufficient votes to determine.
var othInsuff; // Insufficient votes to determine.

// General utility
var initialDataLoaded = false; // flag to tell "child_changed" code to run if true
var fireRef = FBapp.database().ref(config.firebase.dbPath + '/' + officeType); // Reference for Firebase
//var fireRef1 = FBapp.database().ref(config.firebase.dbPath + '/' + officeType + '/parties'); //
Reference for Firebase
//var fireRef2 = FBapp.database().ref(config.firebase.dbPath + '/' + officeType + '/national'); //
Reference for Firebase
var data; // Container for data received from Firebase
var dataPath; // Variable path to account for "child changed" path changes
```

First, I require in the modules that are needed for the bar. In this case, it's actually just the firebase module, but for consistency, Peter wanted me to include the `vanilla.js` module as well, even if I didn't need it. Since `vanilla.js` isn't needed for the module, I haven't included it in the appendix, but what it contains are some prototype modules that basically can do some of the same stuff that [AngularJS](https://angularjs.org/)²² can do, but much leaner. The reason Peter wanted me to use this was so that we could cut out Angular from the application entirely.

Everything is then wrapped in a function called `ElectionBar`, which is what gets exported at the end. Since I'm defining my function as `var ElectionBar = function()`, it doesn't run automatically when the browser reads through the javascript file. Had I defined it as `function ElectionBar()`, it would have run automatically. By doing it this way, I ensure that it only gets run once when a new instance of the bar is called - ie. when it is imported through a "require".

The first and most important variable, `officeType` is what we use to differentiate between the 3 different elections that this bar is supposed to be able to show. That class name that we spent a lot of energy to set in the Handlebars part of the application? Here's where it is used. The `document.querySelector` will look for the presence of the three classes and set the `officeType` variable accordingly.

Next, we define a ton of variables that are used in the actual bar. The variables are all commented in the code, so there is no need to describe those further. After that, I set up the Firebase functionality.

```
fireRef.once('value', function(dataSnapshot) { // Calls this once, then sets a flag to signify this
    data = dataSnapshot.val();
    switch (officeType) {
        case "p":
            dataPath = data.national;
            break;
        default:
            dataPath = data.parties;
            break;
    };
    initialDataLoaded = true;
    checkElection();
    renderVotes();
});
```

²² <https://angularjs.org/>

The `.once()` function tells Firebase to send the browser a full snapshot of all the data that it has for the selected `officeType`. A switch inside the function changes the data path according to the election, then sets a flag variable for later use, and then runs the two main functions inside `my ElectionBar()`.

Now that we have our initial full snapshot of data loaded into the browser, we need to keep looking at firebase for any changes that might occur. This is the all-important live aspect of the application. Whenever a change occurs inside the Firebase database, a notification is pushed out, and the next function catches that:

```
fireRef.on('child_changed', function(childSnapshot, prevChildKey) { // Listens for changes and updates data accordingly
    if (initialDataLoaded) {
        data = childSnapshot.val();
        dataPath = data;
        checkElection();
        renderVotes();
    };
});
```

The `.on()` function is the thing that keeps an eye on Firebase. Whenever it is triggered, it first checks to see if the `initialDataLoaded` flag is true. This is the flag we set when we got our initial snapshot. If it is not, then it won't run anything inside the `.on()` function. However, if true, then it will update the data in the browser's memory, and run the two main functions again to update the view of the data.

Surprisingly enough, this is all that is needed for a live implementation of Firebase. Load the data once, then look for updates and act accordingly. The `.on()` function is very low performance overhead too, and you can easily have dozens of `.on()` hooks looking at different parts of a database with virtually no performance impact.

After having implemented the Firebase functionality, it's time for the two functions that actually make the bar work.

```
var checkElection = function() { // Checks the type of election and assigns data from the correct path
    switch (officeType) {
        case "p":
            if (dataPath.stateName === "United States") {
                officeTotal = 538;
                demWon = dataPath.Dem.electWon || 0;
                gopWon = dataPath.GOP.electWon || 0;
                othWon = dataPath.Other.electWon || 0;
                pctRep = dataPath.precinctsReportingPct || 0;
                candDem = dataPath.Dem.name;
                candGop = dataPath.GOP.name;
                candShortDem = candDem.split(/[, ]+/.pop()); // Split candidate name and pick
last.
                candShortGop = candGop.split(/[, ]+/.pop()); // Split candidate name and pick
last.

                break;
            };
            break;
        default:
            if (dataPath.officeName === "U.S. Senate" || "U.S. House") {
                demWon = parseInt(dataPath.Dem.Won) || 0;
                gopWon = parseInt(dataPath.GOP.Won) || 0;
                othWon = parseInt(dataPath.Other.Won) || 0;
                demLead = parseInt(dataPath.Dem.Leading) || 0;
                gopLead = parseInt(dataPath.GOP.Leading) || 0;
                othLead = parseInt(dataPath.Other.Leading) || 0;
                demHold = parseInt(dataPath.Dem.Holdovers) || 0;
                gopHold = parseInt(dataPath.GOP.Holdovers) || 0;
                othHold = parseInt(dataPath.Other.Holdovers) || 0;
            }
    }
};
```

```

        demInsuff = parseInt(dataPath.Dem.InsufficientVote) || 0;
        gopInsuff = parseInt(dataPath.GOP.InsufficientVote) || 0;
        othInsuff = parseInt(dataPath.Other.InsufficientVote) || 0;
        //officeTotal = demWon + gopWon + othWon + demLead + gopLead + othLead + demHold
+ gopHold + othHold + demInsuff + gopInsuff + othInsuff;
        if (officeType === 'h') {
            officeTotal = 435;
        } else if (officeType === 's') {
            officeTotal = 100;
        }
        break;
    };
    break;
}
}
}

```

The checkElection function is a very simple switch that sets the values of a bunch of the variables that I defined earlier. It is what matches the data received from Firebase with the variables in the Election Bar. Again, I use officeType here as my main indicator for what type of data it's supposed to use.

As you might be able to see from commented code, I had originally planned to dynamically calculate the officeTotal value. It worked, for the most part, but due to occasional inconsistencies in the reports (ie. calculation errors from Associated Press), the value could change during updates - something that it shouldn't do! So I decided to use a less elegant, but safer approach of just hardcoding the numbers into my function.

Finally, we have the function that would make the data actually appear on the website:

```

var renderVotes = function() { // Renders the votes on the bar

    this.mainContainer.querySelector(".bar-center-text").innerHTML = Math.ceil(officeTotal / 2).toString();
    switch (officeType) {
        case "p":
            this.mainContainer.querySelector(".bar-center-text").innerHTML = Math.ceil((officeTotal /
                2) + 1).toString();
            this.mainContainer.querySelector(".bar-text-cand-dem").innerHTML =
                candShortDem.toUpperCase() + ' (D)';
            this.mainContainer.querySelector(".bar-text-cand-gop").innerHTML =
                candShortGop.toUpperCase() + ' (R)';
            this.mainContainer.querySelector(".bar-result-dem").style.width = (demWon * 100 /
                officeTotal).toString() + '%';
            this.mainContainer.querySelector(".bar-result-oth").style.width = (othWon * 100 /
                officeTotal).toString() + '%';
            this.mainContainer.querySelector(".bar-result-gop").style.width = (gopWon * 100 /
                officeTotal).toString() + '%';
            this.mainContainer.querySelector(".badge-gop").src = config.baseUrl +
                '/election-assets/img/badge_drumpf.png';
            this.mainContainer.querySelector(".badge-dem").src = config.baseUrl +
                '/election-assets/img/badge_clindawg.png';
            this.mainContainer.querySelector(".bar-result-dem-text").innerHTML = demWon.toString();
            this.mainContainer.querySelector(".bar-result-gop-text").innerHTML = gopWon.toString();
            this.mainContainer.querySelector(".bar-insuff").innerHTML = "KILDE: AP";
            if (pctRep !== undefined) {
                this.mainContainer.querySelector(".bar-counted").innerHTML = pctRep.toString() +
                    '% OPTALT';
            }
            if (othWon !== 0) {
                this.mainContainer.querySelector(".bar-oth").innerHTML = "ANDRE KANDIDATER";
                this.mainContainer.querySelector(".bar-oth").style = "right: 20%";
                this.mainContainer.querySelector(".bar-oth-box").src = config.baseUrl +

```

```

                '/election-assets/img/box_oth.png';
                this.mainContainer.querySelector(".bar-oth-box").style = "right: 20%";
                this.mainContainer.querySelector(".bar-counted").style = "right: 30%";
            };
            break;
        default:
            if (officeType == "h") {
                this.mainContainer.querySelector(".bar-text-cand-dem").innerHTML = 'HUSET';
                this.mainContainer.querySelector(".bar-result-dem-text").innerHTML =
                    demWon.toString();
                this.mainContainer.querySelector(".bar-result-gop-text").innerHTML =
                    gopWon.toString();
            } else if (officeType == "s") {
                this.mainContainer.querySelector(".bar-text-cand-dem").innerHTML = 'SENATET';
                this.mainContainer.querySelector(".bar-result-dem-text").innerHTML = (demHold +
                    demWon).toString();
                this.mainContainer.querySelector(".bar-result-gop-text").innerHTML = (gopHold +
                    gopWon).toString();
            }
            this.mainContainer.querySelector(".bar-center-text").innerHTML = Math.ceil(officeTotal /
                2).toString();
            this.mainContainer.querySelector(".bar-result-dem").style.width = ((demWon + demHold) *
100 / officeTotal).toString() + '%';
            this.mainContainer.querySelector(".bar-result-oth").style.width = ((othWon + othHold) *
100 / officeTotal).toString() + '%';
            this.mainContainer.querySelector(".bar-result-gop").style.width = ((gopWon + gopHold) *
100 / officeTotal).toString() + '%';
            this.mainContainer.querySelector(".bar-oth").innerHTML = "ANDRE";
            this.mainContainer.querySelector(".bar-insuff").innerHTML = "IKKE AFGJORT";
            this.mainContainer.querySelector(".bar-oth-box").src = config.baseUrl +
                '/election-assets/img/box_oth.png';
            this.mainContainer.querySelector(".bar-insuff-box").src = config.baseUrl +
                '/election-assets/img/box_insuff.png';
            this.mainContainer.querySelector(".badge-dem").src = config.baseUrl +
                '/election-assets/img/badge_dem.png';
            this.mainContainer.querySelector(".badge-gop").src = config.baseUrl +
                '/election-assets/img/badge_rep.png';
            break;
        }
    }.bind(this);

```

Again, I start off with a switch that looks at the `officeType`, and then sets a bunch of values accordingly. These values then define how the bar looks. Text is changed to the proper names (candidates or parties), pictures are set correctly (candidates or parties), the bar legend at the bottom is set correctly as well.

The *actual* bar is controlled with the `.style.width` property. I calculate the percentage value of house/senate seats or electoral candidates won by both sides, and turn that integer result into a string with a “%” added at the end. This new string value then replaces the original value in the CSS file. Inside the CSS file, I have a transition property set, and it is this that animates when the bar changes values. If I didn’t have a transition, it would simply “jump” between the two values instead. It would still work, but not look quite as elegant.

By using the `.bind(this)` function along with main function, I make sure that the values are only changed for that specific instance of the bar. This is important for pages where more than one bar is shown.

Finally, if you have a keen eye, you might have noticed a little `if` statement hiding in the Presidential bar. According to what I’ve written so far, this is not supposed to be there! We were supposed to ignore any 3rd party candidates, as they were deemed irrelevant by the news team back when we made our initial clarification inquiries in the research phase. There is a reason for the presence of this `if` statement that I will get into later, but for now I will just say that this `if` statement enables us to show a 3rd party candidate on the presidential election bar if we need it.

In the House and Senate however, it is common to have a couple of “Other” (either independent or belonging to one of the minor parties) seats, so the functionality for that is part of the normal rendering of the bar. All in all, the ElectionBar function looks like it’s huge, weighing in at 181 lines of code. In reality though, it runs much less code than that, because of all the switches involved.

The Election Map and State List

Setting up the map and state list was the same general idea. Of course, the HTML for the map was hugely more complex than the relatively simple bar that I had made. Peter spent a lot of time just getting the map to behave correctly, and then hooking up Firebase to properly update results from the Presidential Election down to the state level.

However, once he got the map sorted, he could reuse most of that code to also create a list of the states. Like I mentioned though, Peter was the primary programmer on the map, and I barely touched it (save from helping a bit with the state list and map legend), so there’s not much reason for me to go into further detail here.

Making it modular

During the development process of the front end, we were careful to make sure that our modules would work on their own so that they could be included anywhere. Testing went through several phases. First, we just checked the module completely on its own. Then we created a dummy page that included DR’s site-wide assets so we could get proper styling for our text and so on. Finally, we got a proper test environment up and running.

For a long time, we were working under the assumption that our modules would be [ESI, or Edge Side Includes](#)²³, which is a Drupal term. Actually getting things to include as ESI is something that is currently beyond my skillset and it was something that Peter was working on, and how we implemented our modules on the election page.

However, because the bar would be going up on the front page of dr.dk, I couldn’t *just* have it as an ESI module. I also needed to create a version that could run in an `<iframe>`²⁴ as well. This was a major reason for us implementing `.bind(this)` in my javascript file, to ensure that whatever happened would only target that specific instance of the bar, and not accidentally hit something else.

Evaluation and result

In the end, we managed to get everything up and running like it was supposed to. Both the bar and map would display live data and update in real-time, everything could be included where it was supposed to be, and everything was working as expected.

For my own code, I feel that the bar could have been done a little more elegantly. I kept tweaking and refactoring my code, and in the end, there was only one minor issue that I didn’t manage to “fix”. It wasn’t actually broken, but just a little bit inefficient. When the bar was displaying the presidential election, the data that it received from Firebase included all the State-level data as well - something I didn’t need to display at all. The issue here was one of incorrect pathing to Firebase.

I probably could have fixed it given enough time, but when we actually discovered what it was doing, it was quite late in the development process, and we decided that it was “good enough”, so that I could spend time working on other things instead. While it did increase the amount of data that would need to be transferred back and forth, the final object that would be passed around was only 19kb, so we decided against doing anything else.

Chapter 7: Late changes

Providing live data for TV. US Politics being crazy.

Introduction

As with all agile projects, the requirements tend to change. Our assignment was no different, of course. Add to this the fact that news teams tend to work right up until a deadline, and you have a recipe for many sudden changes. On top of all this, the actual election turned out to be completely crazy as well.

²³ <https://www.drupal.org/project/esi>

²⁴ http://www.w3schools.com/tags/tag_iframe.asp

Very late in the process, there were a number of issues that popped up and that we had to deal with. The biggest one was that we all of a sudden also had to provide data to the TV news. While they had their own “front end” implementation, they needed to get the API data from our backend. This was something that we completely hadn’t planned on.

A potentially critical issue revealed itself in connection with the fact that we had to provide data for TV as well: It turns out that any standard Firebase app gets a limit of 10.000 concurrent users. If you want more than that, you have to negotiate with Google to get them to raise the limit - this was something that the department managers had been trying to do for a while, but without any success. Now, if the limit was just applied to the bar on the website, it would be inconvenient, but not a disaster. It would simply mean that some people would have to wait a little bit longer to get their data if all the spots were filled. However, since we had to also draw on our Firebase to provide live data to TV, we needed to figure out what to do if *they* all of a sudden got locked out.

Another obstacle that popped up was from the election itself. Both Trump and Clinton were SO unpopular that 3rd party candidates suddenly became relevant again. In Utah, Evan McMullin all of a sudden was predicted as having a 25% chance of winning the presidential nomination! Even though we were only talking about 6 possible Electoral Votes, it could still be enough to deadlock the election. If Trump and Clinton got roughly the same number of votes, the ones for McMillan (if he won) could keep either major candidate from being declared president. This means that all of a sudden, a 3rd party candidate was relevant again. Something that hadn’t happened since 1956.

There were several other smaller issues, but I’m going to focus on these 3 big ones for this part.

Problem definition

With our chosen frameworks and technologies, how can we provide live data from an outside API when there are many unknowns?

- *How can we reduce the number of unknowns to as little as possible?*
- *How do we fetch data from the API?*
- *How do we feed the data into our own database?*
- *How do we set up a database that can handle the live aspect well, and without bottlenecks?*
- *How do we handle our data once the election is over?*
- *How can we test the live aspect of our application?*
- *How do we display live data from our API?*
- *How can we make sure that our elements will work with the rest of the website?*
- *How can we provide live data to TV?*
- *What do we do about the concurrent user limit on Firebase?*
- *How do we display 3rd party candidates in the Presidential Election?*

Method

At this point, we were scrambling to get everything done. The 3rd party candidate thing we could code our way out of fairly easily. Raising the limit on Firebase was out of our hands - The management kept trying of course, but we had to think up a different solution, just in case they didn’t succeed.

For data for TV, we needed something entirely new. One reason for this was that unlike the API, backend and Firebase which all worked with JSON objects, the data for TV would have to be sent as pure text strings with very specific formatting requirements. They would also need complete data snapshots sent every time, as their front end solution couldn’t handle just getting updates. For this reason, we decided to revisit our planned MongoDB solution - however, with the threat of the 10k concurrent user limit hanging over us, we needed to also rethink the architecture of our application a little bit.

Plan

- Implement the option for a 3rd party candidate in the election bar, map and state list.
- Adjust architecture so we can bypass Firebase
- Properly implement our MongoDB database, and set it up to feed data to TV
- Cross our fingers and hope the managers manage to negotiate a higher limit on Firebase

Problem Solving

3rd Party Candidates

This one turned out to be an easy fix. Previously when looking at election data, both at the state level and national level, we had only been looking at Democratic or Republican candidates. All others were ignored and not added into Firebase. What we then did was to add some logic that would look at all the other candidates on the ballot, figure out which one had the highest number of votes, and then also add this candidate into our Firebase database.

This third candidate would be categorized as "Other", with the actual party name (if any) as a value inside his dataset. Now that we had the top 3rd party candidate in the database, we could include them when looking for the `winner=x` flag to see who won a state.

For the bar I worked on, this resulted in that little `if` statement that I mentioned in the Frontend chapter. If the `othWon` value is higher than zero, that means a third party candidate has won electoral votes in a state. Then, and only then, would we show these votes as green on the election bar. It wasn't necessary to display a name or anything like that, just the color, so that people could see that something was amiss. A third party candidate winning a state would be sensational enough that it would be all over the news anyway, so there would be context to find there.

Adjusting the Architecture and providing data to TV

The main problem we were facing with our architecture was that MongoDB was supposed to get data from Firebase. If it was just a matter of backing up Firebase data, this wouldn't be a big issue, even with the concurrent user limit possibly cutting off access, because it was not an essential system, and it wouldn't be a disaster if MongoDB was updated a bit slower. However, since we now needed MongoDB to feed data to TV, we had to rethink this.

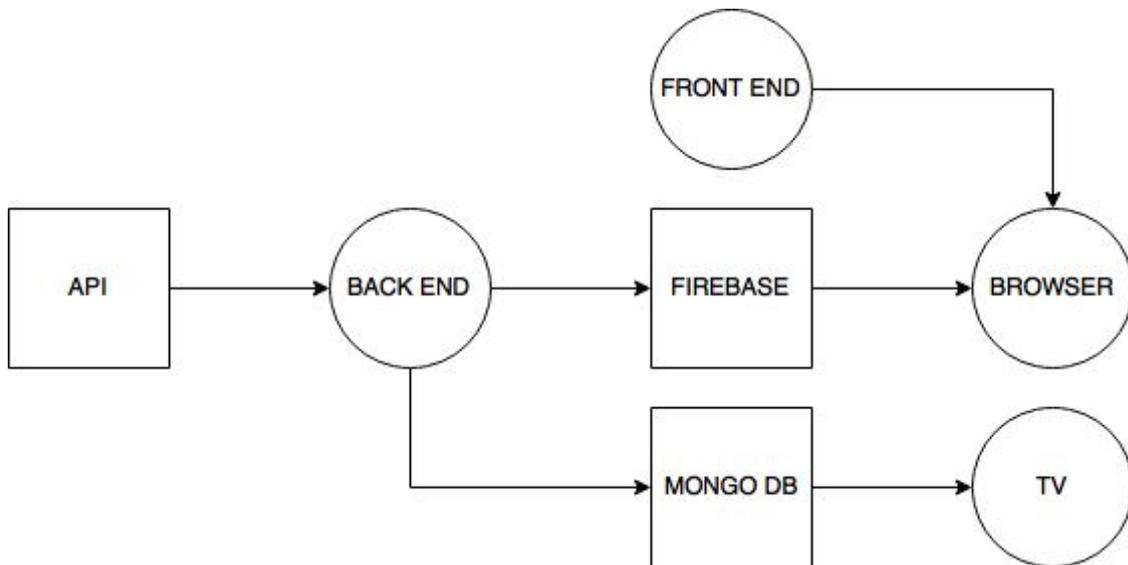
So Monday evening, the day before the election, we were having a crisis meeting. Everyone else had already gone home, and it was just myself, Peter, Adam and one of the department managers who were left. They were talking about possibly having to work all night on an emergency solution that they would have to write from scratch.

Thinking back to our API scraper, I remembered that we were nowhere near using our query budget for the API. The scraper used only 2-3 queries per minute, out of the 6 we were allowed. This meant that we actually had a bit of leeway for this, and that we also had a very simple solution:

Push data to MongoDB directly from the API.

We had the budget for it, and while Associated Press were not really happy with people querying for full data dumps all the time, we figured that we could do a query every 30 seconds or so for a full dump and still be fine. Unfortunately, I couldn't stick around for the actual implementation of this idea, but it was what they ended up going with, and by reusing code from the scraper I made, they set up an emergency solution very quickly.

The final architecture therefore ended up looking like this:



Firestore

A mere 3 hours before the election, we got some good news: Our department manager had made a final plea to Google to raise the limit, and they had agreed to raise it from 10.000 concurrent users to 100.000. We had hoped for just a temporary raise for the duration of the election night, but it was a permanent raise.

This not only meant that we would be able to handle traffic during the election, but it also meant that we wouldn't have to switch to using MongoDB quickly after the election. It could be implemented much later, if it was needed at all.

This also meant that the emergency TV bypass wasn't necessary after all. At least not for what we thought we needed it for. Like I mentioned earlier, TV news especially tend to work up until the very last second that they can. The election website went live at 16:00 on the 8th of November, and therefore Firestore needed to be empty until real data started flowing in. However, the TV guys needed to test their front end and graphics as late as 21:00 that day. Normally, this would have been impossible, but because Peter and Adam had created the emergency fix, it meant that they could bypass Firestore for the TV testing, and therefore the website would be unaffected during this - So the work wasn't wasted after all.

Evaluation and result

Even though the requirements for the assignment kept changing pretty much up until the last minute, we had managed to get everything done in time. Ideally, we would have had "code freeze" (no new features being added) at least a week before the election, but sometimes it doesn't work out that way, and as was the case here, there were a bunch of things that needed to be added at the last minute. Fortunately, we were working very agile throughout the whole process, and managed to tackle the changes as they came up.

Chapter 8: Project Management

Development Methodology and project management

As I mentioned earlier in the report, the de facto software development framework at DR's web department is Scrum... or rather ScrumBut. The rest of the company doesn't use Scrum, so there are several challenges in regards to working agile when everyone around you does not.

Rituals

Every day we would have a daily standup meeting, where we talked about our progress, and updated the Kanban Board. For the most part, this was a great way of keeping up to date with how the project was progressing, but since we were just two people working on the assignment for most of it, it wasn't always necessary, as we would communicate a lot during the day.

Programming

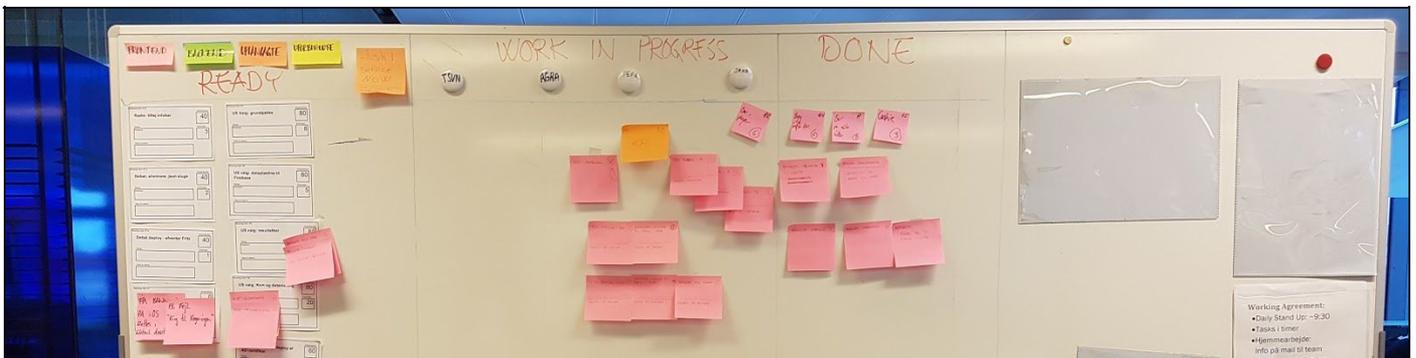
While Peter and I both had aspects of the assignment where we were the “Primary” coder, there were also a lot of situations where we would pair program. This was especially useful when we were working on things we weren’t 100% sure how to implement, or when there were many factors to take into consideration.

Both of the projects were on Github, and it was during this time that I really got some hands-on experience with working actively with repositories. We would create feature branches for whatever we were working on, merging and splitting as we needed, and so on. When used actively, Github is a very powerful tool, even for small teams such as ours.

Challenges

While internally, Scrum was working very well for us, there were many challenges along the way. We had to coordinate with several other departments (News, TV News and Design), and the rest of DR doesn’t lend itself well to the informal nature of Scrum, unfortunately. Whenever we wanted to discuss anything in detail with another department, we had to set up a meeting. It was rare that we could get a meeting on the same day as the issue came up, so we often found ourselves getting delayed, simply because we had to wait for a meeting to be arranged. This was especially frustrating when we had simple clarification questions to ask, and often resulted in tasks stalling for a while.

At one point, we had 10 open tasks in “Doing” on our Kanban Board. Out of those, 9 of them had stalled, waiting for input from either Design or the News department. Every time we stalled and had to wait for input, we started something else so we could keep going.



10 items in “Doing”. 9 of them waiting for outside feedback

This was murder on our Velocity though - not being able to close any of the tasks looked very bad on our productivity charts, which were the metrics used to show management that we were actually getting work done. In the end we managed to close them all of course, but we had several weeks where we were unable to move anything into the “Done” category.

Still, it was interesting to see Scrum used in the real world, and it’s unrealistic to expect that the entire company uses it when we’re talking about a company the size of DR. Compromises have to be made, and I think we stuck as close to the framework as possible, given the circumstances.

Chapter 9: Election night

Brief chapter on how it went during the election

The project culminated with the actual election on the night of the 8th of November. Everything was ready to go, except for the fact that the TV guys still needed to test their stuff. The website went live at 16:00, but data wouldn’t start flowing in until around midnight.

Backend

When I was recording the test elections, I would typically set the scraper to run on my laptop while I went and did something else, either on my desktop computer, or I went out. Since everything was automated, it should be able to just run, and I could check up on it afterwards. However, a couple of times, the recording got interrupted, because Windows

10 decided to update itself and restart my computer. I did manage to turn this off (although Microsoft sure doesn't make it easy!), but it brought to light that this might be an issue during the election night itself.

So we got a hold of an older macbook, made sure that it was up to date, and that all automatic updates were turned off, and we ran the backend from this. We had adjusted my scraper to a "live" version, where it would trigger the passElection script without having to read it from files first. Everything here went as planned, more or less.

The one thing we hadn't gotten a chance to thoroughly test was the connection to MongoDB, and a couple of times, we got communication errors, as updates to MongoDB didn't manage to finish in time for a new update to come in. Most of the time it was fine, but there were a couple of instances with network errors. However, due to how I had designed the scraper, it was a super easy fix: Just stop the scraper and restart it. Since it always got a full API snapshot as the first thing, we didn't have to worry about the data not being up to date.

I think we restarted the scraper maybe 3 times during the entire night. Otherwise it worked flawlessly.

Website

The website behaved exactly as we expected. We had done a lot of testing during the development process, and knew how to expect. The big question of course was how much traffic we would get, considering that the election was during the middle of the night here in Denmark.

It was a VERY fortunate thing that we had managed to get the concurrent users limit on Firebase increased though, as we peaked at around 66.000 concurrent users! Almost seven times as many as the initial limit. Everything worked like it should though, and apart from a minor Drupal hiccup early in the morning (which was completely unrelated to us), it was smooth sailing. The election website is still up and running, although it is of course no longer live. You can see it here:

<http://www.dr.dk/nyheder/udland/valg-i-usa-2016/resultater>

TV news

Even though the TV guys tested their stuff VERY late, the part that we were responsible for also worked as expected. The minor MongoDB hiccups were corrected so fast that it wasn't noticeable during the broadcast

The night in general

Throughout the night, there were quite a lot of emails flying back and forth. Sometimes the News guys would get fidgety when another website had seemingly more "up to date" information than us, but we would point at the fact that they used different data providers, and that a lot of them were using prognosis data. The news guys had themselves specified that they ONLY wanted hard result data, not prognosis data. Also, because we had taken care not to calculate any results ourselves, we could simply point to Associated Press (who also had their own live implementation on their website) and go: "We have the same data as them, and we're getting it at the same time. The problem isn't with us." It's a good thing we did too, because at one point, this showed up in our table:

Iowa	6	42.1 %	51.7 % ✓	99.94 %
Kansas	6	36.1 %	57.2 % ✓	98.83 %
Kentucky	8	32.6 %	62.5 % ✓	100.05 %
Louisiana	8	38.4 %	58 % ✓	100 %

It was great that everything worked out like we had planned, but it unfortunately also meant that it was a very long night. There were no sudden emergencies that we had to fix, no last-minute coding to be done, so all we really had to do was to monitor everything. Peter and Adam, who had both done election nights like this before, said that it was the smoothest election they had run so far - but also the most boring, because there was nothing to do.

Conclusion

Overall project evaluation

After the election itself was over, and we had all gotten some much-needed sleep, we held an evaluation meeting and retrospective. Had we done what we set out to do? What were issues that we could improve on in the future? What was next?

The problem definition

With our chosen frameworks and technologies, how can we provide live data from an outside API when there are many unknowns?

- *How can we reduce the number of unknowns to as little as possible?*
- *How do we fetch data from the API?*
- *How do we feed the data into our own database?*
- *How do we set up a database that can handle the live aspect well, and without bottlenecks?*
- *How do we handle our data once the election is over?*
- *How can we test the live aspect of our application?*
- *How do we display live data from our API?*
- *How can we make sure that our elements will work with the rest of the website?*
- *How can we provide live data to TV?*
- *What do we do about the concurrent user limit on Firebase?*
- *How do we display 3rd party candidates in the Presidential Election?*

Looking at the problem definition as it looked like at the end of the project, we had managed to resolve all the issues that we discovered along the way, except for a single one. We had the option to switch our data feed from Firebase to MongoDB, but there was no rush in doing so. We had even managed to take on extra work that was outside the original scope of the assignment (the TV data feed) and provide a satisfactory solution for that. It had been a lot of hard work, but it had paid off.

Data Gathering

The data gathering process went as well as it could have, I think. The research I did at the very beginning of our project brought a lot of issues to light, such as the 3rd party candidates that we were told to ignore until the election showed just how crazy it was. By the end of the research phase, both Peter and myself were pretty knowledgeable about the election, and it gave us a very good starting base for the project.

The fact that we knew from the beginning that the data we got from the API was unreliable meant that we made a very good decision in delegating most “math” jobs to them. It meant that we could place the responsibility elsewhere and just focus on receiving and displaying whatever data they sent our way. As a part of this, I also made sure that we made it very clear on the website itself that we were just relaying data, and not making any calls of our own on that matter.

Architecture and Frameworks

Our application architecture in general was very loose. Peter had already decided to use NodeJS (with MVC) and the frameworks associated with that, so I wasn't part of that decision. Could we have made the application in something else, like .NET? Sure, we could have, but NodeJS was a good fit for this project, especially because Firebase has easy integration with it.

Personally, I would have preferred to have our application be even more standalone than it was - but DR has rules and an established production pipeline, and we had to follow that, so we had to use shared assets, etc. whether we wanted to or not. Some things seemed unnecessarily complex to me, which is why I would have gone for a simpler solution, but I understand us having to follow company procedure.

The Backend

When Peter first mentioned us having to build a bunch of tools for ourselves, I was a little bit leery. I hadn't really done anything like that before, and it was a bit daunting for me. However, the actual coding involved turned out to not be that bad, and it was a very good decision to do so.

I'm quite proud of the fact that while we both created scrapers to start with, it was mine that we ended up using. I'm also very happy that I was able to build it so well that we could use it for the actual application with minimal modifications needed.

The Frontend

Building the bar was a bit closer to coding like I was used to, but even here I saw some challenges. I could most likely have made the bar with a lot less code, had I been allowed to use AngularJS for instance. However, Peter wanted us to stay as close to standard JavaScript as possible, the reasoning being that no one knows what frameworks will be popular and in use in just a couple of years.

They already had problems with maintaining applications built with frameworks that have since fallen out of favor, so I can understand this push towards going as minimal as possible. For the frontend, I do wish that I had been a little bit more experienced, or had had more time, so that I could have made a more elegant solution than the one I ended up with - but still, it worked like it was supposed to, and performed well, so I guess I shouldn't complain too much.

Databases

Again, I wasn't really a part of the decision-making process in choosing which solution to use here. In fact, I had not heard of Firebase before working at DR. Firebase itself is a *really* cool [Platform-as-a-Service \(PaaS\)](#)²⁵, and it was super easy to learn and implement. However, we *very nearly* ran into big issues with it because of the concurrent user limit. I still think using it was the right choice, but the limit negotiations should have been in place *much* sooner than they were.

In regards to MongoDB, I think we waited too long to implement it. Had we had Adam join the dev team a week or two earlier, I think the days leading up to the election would have looked very different, and we would have been much more on top of things.

Late Changes

These were probably the most stressful to handle, and something I could have done without. I am happy that we were able to be as agile as we were with our development, or it could have ended up quite badly. Some of them were unavoidable, like the 3rd party candidate becoming relevant again, but the Firebase and MongoDB issues could have been avoided, had we been able to start on them earlier. I'm glad we managed to get everything done in time though.

Project Management

This is where I had the most issues. It's also where everyone else had the most issues. Having to stop work on something because you're waiting for answers or material from other teams *sucks*. Not being part of the early stages of the project *sucks*. Had we been brought on board at the same time as the design team, we would have been much better informed, and we would have been able to influence the process a lot more. As it was, we were just given a design and API access and told to make it work.

We spent a lot of time discussing this during our retrospective, and while some of the things were out of our control, Team Connect's product owner agreed that we (the team in general) should try pushing for more and earlier involvement, with the justification that it would lead to better products.

Another point of learning in this area was about communication. Waiting around for answers is frustrating and slows down work. During the big stall where we had 10 assignments in "doing", we were, among other things, waiting for Design to provide us with actual graphics for the application instead of just a paper printout from a powerpoint presentation. Turns out that particular assignment had been "lost" somewhere, and there was no one actually assigned to make those graphics available for us.

A possible solution that we discussed was to create a slack channel or something similar, and make sure that *everyone* who was involved in a project, across departments and teams, were in that channel. That way, when you needed an answer to a question, you could ask it to the channel and hopefully get a response from someone who could make a decision, rather than first having to figure out who it was and then track them down for a meeting.

²⁵ https://en.wikipedia.org/wiki/Platform_as_a_service

Lessons learned

So what did I learn personally from this? The short answer is “A lot!”, but I will try and go into more detail. First of all, I got a chance to learn and work with NodeJS, which I ended up really liking. In fact, I liked it so much that I want to keep using it and learning it. I already have a couple of projects in mind where I can use it.

Second, I learned a lot about working in a big place like DR. Sure, our team was only 2 people, but there were 3 other teams working with us on different parts, and we were all dependent on each other. It showed me just how important good communication and good lines of communication is. Before this, I had only worked in small teams as part of my education, and making decisions and such was a lot easier. While it was frustrating to not really have much deciding power in this project, it was also a good lesson. You have to adjust to your environment, and there are still steps you can take to make the process as smooth as possible.

Third, I got to experience a software development framework in the wild. While we also used Scrum during my internship, this was the first time I’ve seen it applied to a big project that I got to be a part of. Sticking close to the core of Scrum was difficult, because of all the outside factors, but from what I’ve heard and read, pretty much no big company runs pure Scrum. Besides, the whole point of agile development is that you use what works for you, and discard the rest.

Speaking of agile, this was also the first time that I’ve *really* had to work properly agile. We’ve done it for school projects sure, but the main things we had to adjust for were changes due to lack of knowledge, time or technology. It’s the first time I’ve had to deal with changing requirements in an agile setting (although I’ve had to in a waterfall setting before, back when I was freelancing as a web designer), and it was quite interesting to see just how much our assignment changed from beginning to end.

Outlook

As for our application, there is no real outlook for it. The website itself will be “parked” on dr.dk’s website for a couple of years (probably until the next US election), but when the next election comes around, it will be an entirely new project. While some parts of our code might be able to be reused, there’s no telling how much of it.

Maybe DR will get a different API provider next time. Maybe there will be a new and better suited framework available to create the application in, and so on. 4 years is a long time in the IT industry, and what is great today might be old hat tomorrow.

Final Thoughts

This was a very exciting project to be a part of. The fact that I was able to participate from beginning (when the team got the assignment) and to the end has given me great experience with the development process as a whole, and with the different challenges that you can face when working in a big company with a lot of people.

I learned a lot, both from the good parts of the project, and especially from the bad parts, and the mistakes we made along the way. I have realized how important communication is on a project like this, and if I get a chance to be a part of something like this again, I’ll make good communication a priority.

All in all though, I am very satisfied with how it went, and I get to say that something I made was on the front page of the most visited danish website.

Appendix

Unfortunately, I am unable to provide the whole application in my appendix, as some parts of it are confidential. I was allowed to provide the files I worked on though, and a few other items, so I have attached them along with the report itself.

Documents:

AP Introductory letter.docx
AP API Developer Guide.pdf
Initial Design Sketch.pptx

Files:

bar.less
election-bar.js
module-bar.hbs
passElection.js
report_overview.json
run_getAllFileNames.js
run_preRunElection.js
run_scraper.js
scraper-election.js
scraper-reports.js