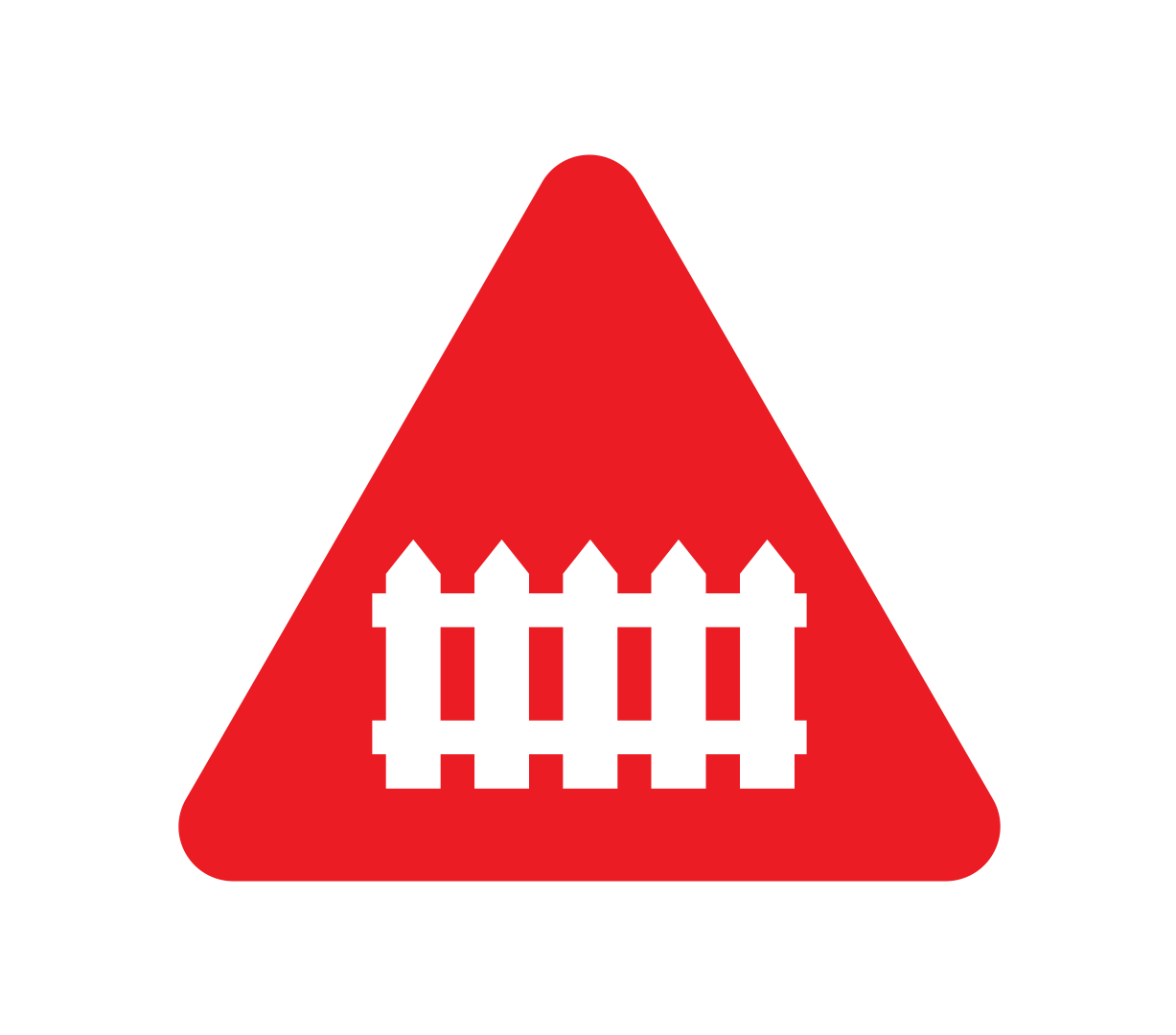
# 

# Level Crossing Predictor



#### Josh Balfour <<jdb45@kent.ac.uk>>

#### 

#### Kieran Jones <<kj90@kent.ac.uk>>

#### 

#### Ryan Wood <<rsw24@kent.ac.uk>>

## Abstract

Level crossing closures are a cause of delay, have a negative impact on the environment due to fumes of stopped transport, and prevent vital emergency services from getting to their destination quickly. The problem is particularly noticeable in Canterbury, which has level crossings on main transport routes.

Can level crossing closures be predicted? Rail timetables plus real time train feeds might make journey planning through level crossings more reliable. Redirecting via longer, but quicker routes or delaying journeys would then be possible.

The project was to develop such a level crossing predictor, integrate with suitable mapping software on the web and/or mobile apps. Field testing for accuracy would be a key part of the final stages of the project.

The project was implemented in PHP and Node JS on the back end, using Angular JS on the front end and a MySQL Database, all managed in a Docker container.

## Introduction

The problem this project solves is the lack of data surrounding the timings of level crossings. There are many solutions on the market for tracking the trains themselves, however there are none for level crossings. Level crossings are a big problem in Canterbury, bringing the city's roads to an almost standstill whenever they are down. Because of this issue, this product has major commercial viability, as the data that it produces has the potential to save the emergency services, bus drivers, and taxi drivers the large amount of money currently spent on employees sitting in avoidable traffic jams caused by level crossings being down. Aside from offering the data to businesses, there is also the potential to integrate with existing route planning services that could factor the data in to optimise their routeing algorithms further, and make the data available to end users.

In this report we shall cover the methodology behind the project's management in detail, the pivots that were taken in both the technology and project approach, detail behind our development tactics, and the real-world testing aspect.

## Background

This project built off of many communities existing efforts, mainly from the rail, technology, and cartography communities. As part of our research we found that many communities had an intersection with the technology community. A prime example of this was the National Rail Open Data (NROD) community, who created a collaborative wiki in order to document the various data feeds and systems that National Rail advertise as public but that are poorly documented. This was a major asset to us, and was referred to often over the course of the project. It's best asset was the amount of tips people had written about that saved us many hours of investigation, as we were able to learn from other's mistakes going forward.

The technology community was immensely helpful in this project, all of the products we used to realise this project were community developed and supported products. Particularly the Docker, Laravel, and Node JS communities. This was particularly apparent when it came to provisioning a local copy of the database for each group member, as we were able to synchronise our environments and version control configuration changes using Docker, and the plethora of libraries available for Node JS and Laravel meant that we were able to focus on building the project itself, as opposed to the underlying libraries.

## Aims

### Step One - Plan

The project's requirements were first broken down to into 6 key points and numbered, in order of priority:

* R1 - Be able to show current level crossing times
* R2 - Be able to display level crossing times in a nice way
* R3 - Predict level crossing times to a 90% degree of accuracy within 2 minutes
* R4 - Have an open API to allow other developers to add value
* R5 - Be able to predict level crossing times to 95% degree of accuracy within 30 seconds
* R6 - Show more data about the level crossing that could be of potential interest (pictures, names, general information such as: accessibility, train frequency, and more)

Due to the size of the project, it was deemed appropriate to class the project as MVP, with a preceding Discovery phase and an extension "Future" phase.

We then broke each requirement down to a functional level, and listed out the functions that the the system would have to fullfill in order to accomplish each respective requirement, numbered in the format Requirement#.Function# for example R1 was broken down to 5 functional requirements:

* R1.F1 - Download and Store Level Crossing Locations
* R1.F2 - Download and Store Train Times
* R1.F3 - Download and Store Rail Station Locations (links R1.F1 and R1.F2)
* R1.F4 - Download and Store Railway Route Locations (R1.F1, R1.F2, and R1.F3)
* R1.F5 - Write algorithm to show if level crossing is up or down based on the above data

We then started a Spike, within which each function was discussed in depth and broken down to a task level, making notes from our own knowledge and researching where there were gaps, bullet points of what needed to be done in order to accomplish the task. This then allowed us to put man hour sizings against each task.

For Example R1.F1 was broken down into 3 tasks:

* R1.F1.T1 Set up a Database - 4 MH
  + DockerFile from mysql
* R1.F1.T2 Find and download the data - 4 MH
  + Needs to be able to be done regularly, so must write script
    - Scrape <http://www.networkrail.co.uk/transparency/level-crossings/>
    - Find download link
    - Download contents
    - Parse XLS file
    - Grab second sheet
    - Import each row into database table
* R1.F1.T3 Write view to sanitise/convert/extract the data we need - 1 MH
  + ID
  + location

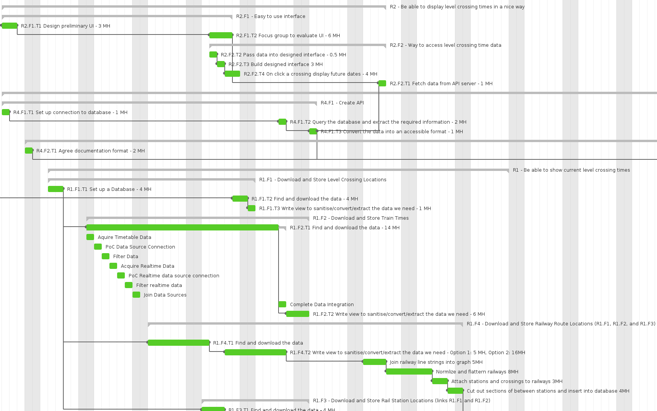
With these Man Hour estimates in hand we were then able to make estimates by aggregating the estimates and including 10% contingency, as to how long each requirement was going to take to fulfill.

Aside from providing us with time estimates this process was incredibly valuable as it also highlighted the project's risks, which is why some tasks have ranges against them instead of an absolute hour value, and it also gave us time to think through the approach we would take, lowering the number of assumptions we made.

Taking these time estimates we then established how we would share the load of the tasks. To do this we needed to input the tasks into a collaborative project management tool, within which you could assign tasks to an individual, input dependancies, and generate gantt charts and workload reports.

After evaluating a number of different project management tools we ultimately chose Asana in combination with a collection other third party services, as it was both free and able to fulfill our needs.

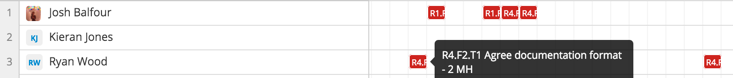
Using the dynamic gantt chart creator Instagantt, we were able to pull our project data from Asana and render it in gantt chart form, like so:



NB: The bars were previously blue before the tasks were marked as completed.

During the discussions surrounding the Spike we had already established where the group member's strengths were and who already had a solid idea about how they would accomplish each task, so assigning responsibilities to group members was reasonably easy.

After allocating a team member to each task, we used the Pending Workload view in Instagantt to set target completion dates, preventing clashes and verifying that the workload was distributed evenly across team members.



NB: I marked tasks as incomplete so that they would show in the Pending Workload view to illustrate our usage of the product, hence they appear as red.

With the project planning process completed, we were able to deliver well thought through, concrete delivery date estimates to our project supervisor for when each requirement in the project would be met.

### Step Two – Execute

#### A Targeted Approach

Within the project plan a Discovery phase was allocated in order to establish the base of the project.

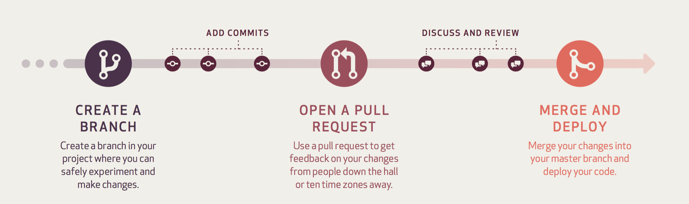
This consisted of the following: A Version Control System, a self-contained runtime environment for development and deployment, and a definition of practices surrounding documentation, testing, communication, and development.

The chosen VCS was GIT, as it met our requirements and would also allow us to use industry standard GitHub's free private hosted offering, which we were all very familiar with.

In order to ensure that software which our project would depend on was in sync between our development machines, and ultimately our production machine, we chose to use Docker to build a low-overhead virtual machine, herein refered to as a container, which would have all our dependancies installed - our database, our application framework, and our web server. This container was defined by a script, known as a Dockerfile, which was under version control in our VCS (GitHub).

Using Docker solved the issue of this project's environment conflicting with other projects on the developer's machines. This also meant that when it came to test and deploy the application, we were able to do it in under a minute with no problems, as the environment was exactly the same as the development environment.

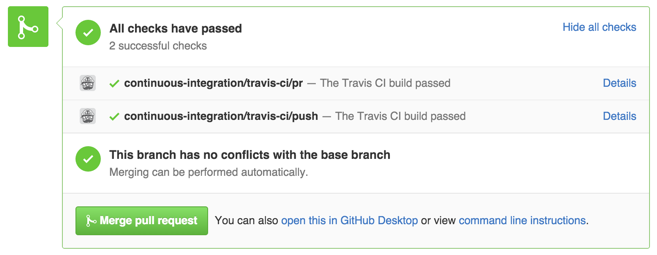
Using GitHub as our VCS allowed us to use GitHub's development workflow which they dubbed "GitHub Flow".



We implemented GitHub Flow by establishing in our development practice that when developing a feature, bugfix, or otherwise changing the contents of the repository, that the developer must first make a new branch from the master branch, perform their task, then open a Pull Request to the master branch. A notification is then sent to the rest of the team, who review the request, and when there is unanimous approval the code from the feature branch is merged into master, and the feature branch is deleted.

To minimise the opinionated aspect of our code review, the process was established objectively as part of our development practices. One of our criteria for approval was that the code could be automatically merged into the master branch, with no conflicts. This was determined by GitHub and could be satisfied by the developer who made the feature request merging the Master branch into their feature branch, and resolving any conflicts in their feature branch before making the pull request. Another of our criteria was that the code passes all tests. In order to minimise the work required to verify this criteria we used a continuous integration service called Travis to continually monitor our code repository and run our automated tests. When a pull request was made against the master branch Travis would test both the feature branch, and the automatic merge from the feature branch into master branch.

The resulting box is then shown to reviewers of the pull request:



This approach minimised developer effort, whilst maximising stability of the product.

Discussions on pull requests were done using a team messaging service called Slack, which allowed us to communicate whilst on our laptops and phones. Our continuous integration service Travis, and our VCS GitHub integrated with Slack, which allowed developers to be notified when other team members were committing to the repository. This prompted increased collaboration and encouraged more knowledge sharing and allowed team members working remotely to feel more involved in the project.

#### Documentation

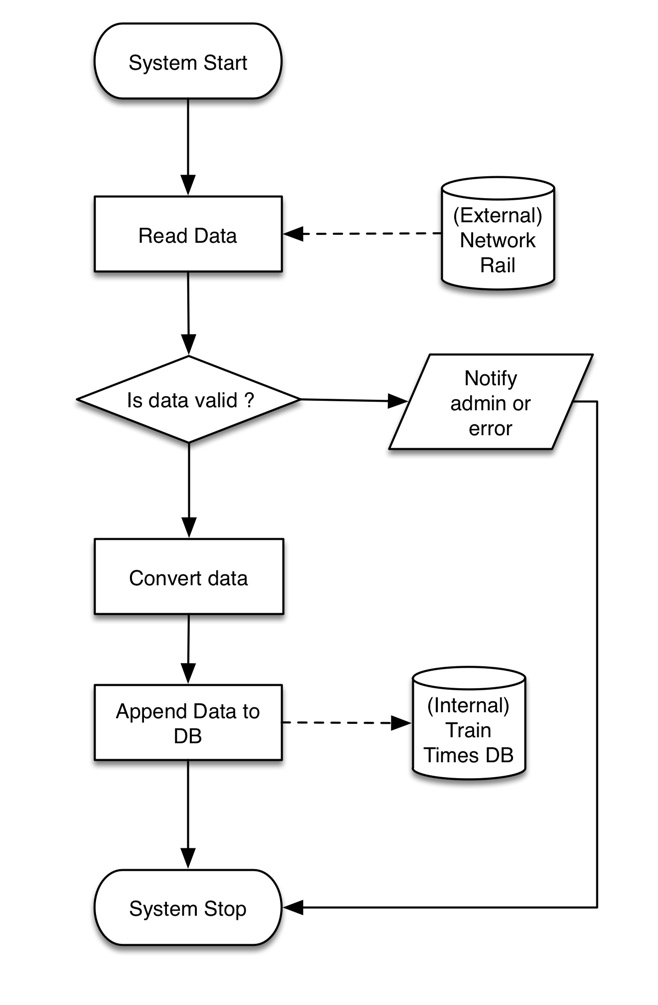
Documentation was taken very seriously in this project. It's level of importance was a combination of the group members all working remotely from one another, and integrating between the different components being vital to not putting ourselves in technical debt. For example if the database was in a different format than the algorithm was expecting then this would create rework further down the line.

It was decided that the documentation should be treated as a first-class citizen alongside the source code, as they sat hand-in-hand in our development procedure, as with every piece of code written a comprehensive accompanying documentation item needed to be written. With code files sharing commits with documentation files, it was easier to keep track of the progression of features' functionality and documentation levels.

Each document, as well as being under version control, is version numbered at the top of the document, this is to make it even easier for the reader to understand which version of the document they are reading.

#### Importer Procedure

As part of our documentation we formalised what we referred to as our "Importer Procedure". The Importer Procedure is a generalised flow of how all of the respective importer scripts will work, formalising the process, which was important as the different importers will be written by different group members.

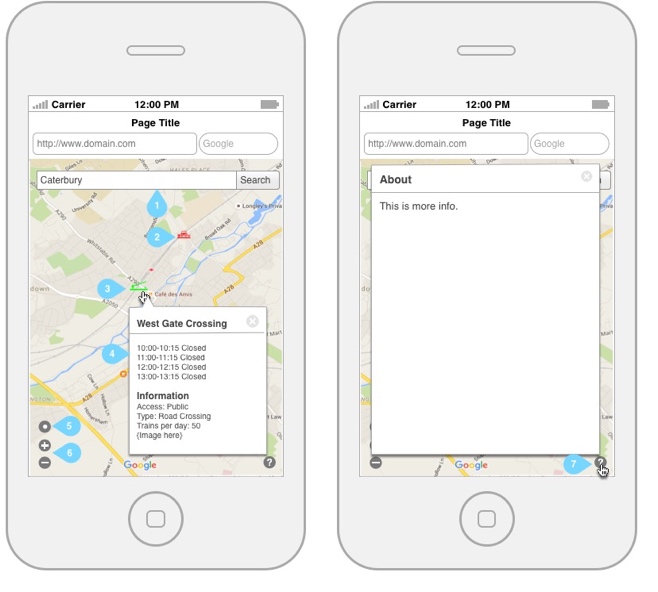


#### Interface testing

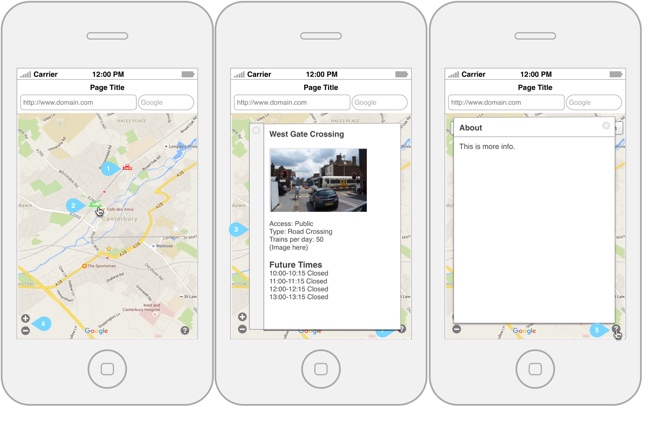
After building a mock-up of the user interface it was tested on a small group of people for usability, along with private testing with different data sets. As a result of the testing 3 changes were made: The Search bar was removed - Due to the limited area of coverage, no one ever attempted to use this, more information bubble was made a model - This provided more room to display data, as the popup provided little space, this was particularly an issue on smaller devices, and the "Go to my current location" button was removed as Google maps does not allow this with their API.

As a result of this the UI became, overall, slightly simplier which is due tousing a map based view which the user can also navigate manually.

##### Before the interface testing



##### After the interface testing



#### Algorithm

The full algorithm deep-dive is available in the project's documentation, but here I will give a brief overview.

##### Station tracks

The first step will be to preprocess all of the train tracks and the stations so that we can produce a simple map which contains a table with a line.

Technical Decision

All of the following process will be passed down to Node.JS because PHP will not provide the performance needed to crunch the numbers in a timely manner.

##### Flatten

All of the rails will be flattened and joined together to produce a continuous graph of connected nodes. This will group lines which run next to each other together and lines that join on junctions that aren't present in the source data file.

Attach stations and crossings

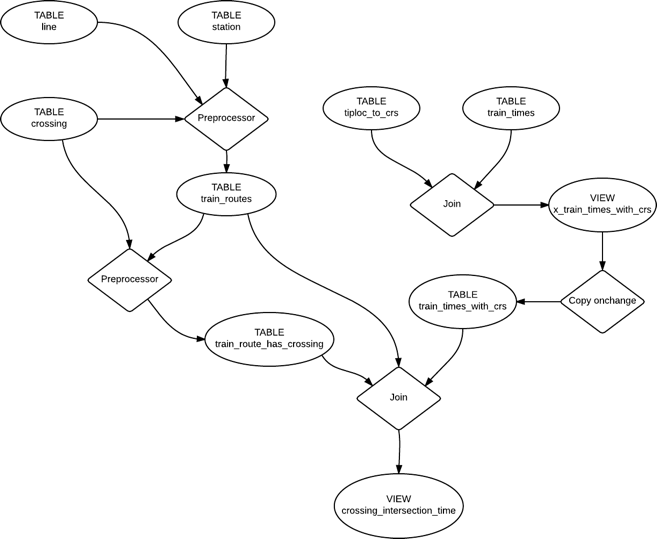
All of crossings and stations will be attached to their nearest node on the line. This will be based on distance but can be based on the shape and size of the station if using general area proves ineffective.

Mapping the result

Starting at each train station and walking along the nodes, if a node has connections then we send the walker down each respective set of nodes. When we reach a node that has a station attached we add the path, the station, and the from and to destinations into the database. Then we continue along into will covert 6 stations. This is a compromise between doing all station to all stations and having a large amount (250GB) of data, and connnecting each station to the nearest neighbour, which National Rail may have not provided.

##### Ouput

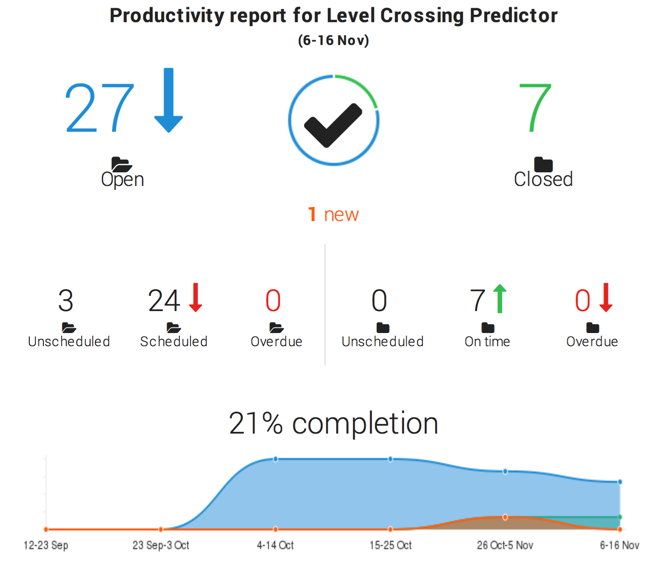
The output of this algorithm will be a table of the track that goes from station to stations.



#### Weekly Standups

Each week, as the project progressed and tasks were marked completed in Asana, the gantt chart automatically updated, visualising the project's progression. This allowed us to report to our project supervisor in graphical form the progress of the project.

Along with the updated gantt chart, a weekly report was produced using the third party service WeekDone which pulled the project's progress data from Asana and generated a progress report, an excerpt of an example of which is below:



#### Pivot Points

Over the course of the project there were several pivot points, which, due to the agile approach we took, we were able to easily accomodate for.

##### Project Pivots

Data sourcing was one of our key challenges on this project, we pivoted many times before we settled on our final set of data sources.

In our initial research we found a website that mapped out the entire UK rail network on a Google Map, and emailed the author to obtain permission from him to use his data. Unfortunately he declined, so we kept searching.

We next looked into using crowd sourced mapping project Open Street Map's data. This however produced an extra set of challenges - mostly surrounding compute resources. This was because although the compressed map itself was only 4 GB, it was 40 GB uncompressed. The uncompressed file then needed to be imported into a Postgres database which had a geospatial extension called PostGIS installed. After evaluating this option it was deemed unfeasible due to the time it would take to extract the railway tracks from the database and the gamble taken on the quality of the resulting data, as it was purely a crowdsourced dataset.

After some in-depth research into the subject we found a government open spatial data initiative called INSPIRE. Backed by European Directive 2007/2/EC, the initiative established an infrastructure upon which government departments can publish geospatial datasets. Within the datasets published we found a listed, but not documented, API endpoint which was backed by Geoserver, an open source geospatial data server. Despite the lack of documentation, we were still able to make use of the API as one of the team mebers, Kieran, had knowledge of how Geoserver behaved so was able to write a reusable script to extract and transform the data we needed. Using this service we were able to source the railway track routes and the railway station locations.

##### Technology Pivots

Over the course of the project we made several pivots in the technology stack that we utilised to deliver this project.

Application Framework

We initially started off using Lumen as the project's overarching framework, as it would allow us to deliver a lightweight solution, but still provided an industry standard project structure for us to use. However, as we started fleshing out our technical implementation plan we realised that a large part of the functionality we were planning to implement was available in Lumen's more feature rich cousin: Laravel. As Lumen and Laravel were made by the same community, we were able to transition over painlessly, within only a half hour. Although this cost us half an hour of development time, it saved us in the long-term many hours. With hindsight it's clear to see that this was the right decision.

Relational Database Management System

After starting off with a Postgres Database, which was a requirement of the Open Street Map datasource, we transitioned to MySQL after switching out our datasource, as the team's skillset was more suited to MySQL, and 3rd party library support was more prevalent.

PHP JIT Compiler

Whilst at a Facebook Hackathon part way through the project Josh discussed the technology stack used in the project with a developer at Facebook, who commented that if we switched from using the stock PHP interpreter to run our code to the HHVM (Hip-Hop Virtual Machine) JIT compiler, we would see a significant performance boost. This comment was taken with a significant grain of salt, as HHVM is a technology that was developed internally at Facebook, and their use case was significantly different to ours, hence performance improvements may be subjective.

After performing an impact analysis of evaluating the performance difference between the two approaches it was determined that the development time that this would take would be minimal, although this is a core part of our stack it's installation was as easy as changing a line in the Docker container's configuration from FROM php to FROM hhvm. This made swapping the technology easy, however performance evaluation still remained. This was handled by our extensive test suite, which was automatically ran against every commit made to the repository by Travis. This both verified that the technology change was non-destructive, and, because the test suite reporter also noted down run-times, allowed us to compare the performance of HHVM against stock PHP.

The results of the analysis were that HHVM used less memory, and out-stripped stock PHP in terms of raw speed, thus HHVM was judged to be more performant than stock PHP, so we switched to using it.

##### Algorithm Language and Framework

The underlying algorithm for the project was originaly written in PHP, and when tested against a subset of the data it was performant, however when it was ran against the entire dataset, PHP proved to not scale. This meant that going forward, in order for the algorithm to run it needed to be rewritten using a different toolset. The choice was made to switch to using JavaScript and Node.JS as this was a language and framework the group had the matching skillset for.

As the approach the algorithm took to the source dataset was already established and proven, the rewrite from PHP to JavaScript was a short process, and the resulting code was quick to run against the entire dataset, and also produced correct results.

#### Challenges

One of the data sources made publicly available by Network Rail is a real time feed of raw data from rail signals, which gave us what train passed what signal at what time. However what was missing from it was the location of the signals. We found an FOIA (freedom of information act) request that was filed for this data, however Network Rail actively refused it.

It was decided as this would potentially give us a higher level of accuracy, that Josh would make use of his active connections with senior management at TFL (Transport for London), who work closely with Network Rail, and explain our situation to them, in the hope of getting the data and their permission to use it. They reached back, confirming that the dataset did exist, but unfortunately were not able to allow us use it.

Our biggest challenge was collecting data for our predictive algorithm to compare against, which was a very time consuming process, and logistically complex as it involved visiting a wide variety level crossings, some of which were far from Canterbury, in order to get a workable reference data set.

Our key pain point was the quality of the the data that was provided to us, as it meant that there was a significant amount of data cleansing that was required in order for the data to be workable.

### Step Three - Deliver

Our completed project was able to predict when a level crossing is going to be up to within 27 seconds of accuracy. It consisted of an easy to use map-based, mobile, tablet, and desktop friendly user interface. The data is also accessible via a well-documented, standard adhering, open JSON RESTful API, allowing developers to build on our efforts. Our finished product's technology stack consisted of a MySQL database, with the algorithm written in Node.JS, the API and data acquisition scripts in PHP, and the front end in Angular JS, with a Google Map view, backed by the Bootstrap CSS library and HTML5. Every day at 3 AM the system reloads the data published by Network Rail, which takes on average 4 minutes, and throughout the day Network Rail feeds in real time timetable corrections if a train is delayed or cancelled.

## Conclusions

Overall we are very pleased with the work that was completed, our requirements were all met, our smooth development flow with automated continuous integration and code quality testing is an approach we plan to replicate across other projects - personal and commercial. We are also pleased that we were able to better our estimates for the web app component, as it was written only once and produced no re-work.

Our main success point was the accuracy of our predictions, this is what we are most proud of. Other keys successes include the data acquisition being completely automatable, which meant that we are able to script the entire spin-up process from start to finish, which has kept our repository very light as we don't have to put large data files into it, also that the prediction aspect of the project was quick to implement once the data was in place in the correct format - as per the design.

While we were debugging we added realtime moving trains onto the same map as the level crossings, which a group another year attempted to do as their final year project, however our implementation exceeded theirs as the trains on our map went along the railway lines, instead of a straight line.

If we were to run this project again, in hindsight we would have written it entirely in JavaScript, now that we know the limitations of PHP, which caused us a man day's worth of rework. We would also split our Docker containers, which we didn't do in this round due to lack of experience with the technology - we didn't know how best to use it - and there are now many tools surrounding Docker which were not proven when we built this project. This would have simplied setup and minimised the time the process took to run. We would also have used the free edition of Oracle's commercial database (Oracle XE) to handle our data processing as it's underlying engine is more efficient and contains many more geospatial optimisation features and would allow us to run far more complex procedural data manipulation code on the database itself rather than using a driver to interact with it from Node JS or PHP code. As we use Docker, it would have been easy to provision a container of it, then destroy it when the data import was complete.

To extend this project we would work on integrating the system with third party solutions, such as TomTom and the routing software used by bus companies and emergeancy services. Although we have an API which third parties could already use it would take some development work on their part in order to integrate with us. Given more time we would develop those integrations ourselves in consultation with them, as this would make our work more accessible, and therefore useful, for the end users. We would also gather more reference data from more level crossings, as this was a major logistical challenge for us but our results are solid we would like to prove our work more.

## Acknowledgements

We would like to acknowledge Peter Rodgers, our project supervisor, the various authors of the Unofficial National Rail Open Data (NROD) wiki, the maintainers of the various data sources we based the project on: Network Rail's spreadsheet, the INSPIRE geoserver containing the railway track and station locations, and those responsible for maintaining Network Rail's FTP server containing the train timetable and real-time updates. We would also like to thank the Docker, Laravel, and Node JS communities for their plethora of modules and libraries that made building this project infinitely easier than it would have been without them.

## Bibliography

## Appendices

* Diagrams
* Test data