Online Dispute Resolution for Maritime Collisions

Final Report for CS39440 Major Project

Author: Christopher Benjamin Ashton (cba1@aber.ac.uk)
Supervisor: Dr. Georgios Gkoutos (geg18@aber.ac.uk)

30th April 2015
Version: 1.0 (Release)

This report was submitted as partial fulfilment of a BEng degree in Software Engineering (G600)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature .....

Date .....

Consent to share this work

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature .....

Date .....

Ethics Form Application Number

The Ethics Form Application Number for this project is: 936

110059347

Acknowledgements

I'd like to thank my supervisor, Dr. Georgios Gkoutos, for always being available to answer my questions and for giving me direction, particularly in the early stages of the project. I'd also like to thank Dr. Constantina Sampani and Dr. Alexandros Giagkos for introducing me to the concept of online dispute resolution and for their technical expertise throughout the project.

Rosie Bettles and Lani Cossins also deserve an acknowledgement; I'd like to thank them for helping to create some of the images on the project website.

I am grateful to all of the third-party services that were generous enough to provide their services for free or at a substantial student discount. These include GitHub, Travis, CodeClimate, Gemnasium, ShareLatex and others.

Finally, I'd like to thank all of the hard-working contributors to the open-source frameworks and libraries that made this project possible.

Abstract

This paper explores artificial intelligence driven solutions in the field of Online Dispute Resolution (ODR), using maritime collisions as an example implementation. Maritime law being reasonably terse and straightforward, it was hypothesised that it should be possible to apply the rules of maritime law to the business logic of an ODR platform, but in an abstract way such that any module encapsulating any type of law could extend the core system.

Alongside this report is an open-source and extensible ODR platform, prototype maritime collision module and vendor website. This paper justifies the design, implementation and testing strategy for all three components, and discusses where this platform, dubbed *SmartResolution*, would head next beyond the scope of an undergraduate dissertation and what this could mean for the field of ODR in general.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background & Objectives

1.1 Introduction

Online Dispute Resolution for Maritime Collisions was an idea put forward by Dr. Constantina Sampani, a Lecturer in Law at Aberystwyth University, and presented as a Major Project idea through Dr. Alexandros Giagkos, a Research Associate in the Computer Science department at the same university.

The idea was to create an Online Dispute Resolution (ODR) platform that is able to suggest a resolution using artificial intelligence (AI). Maritime law was considered to be a good example to demonstrate this, as it was thought to be quite concise and relatively straightforward to translate into code. It was hoped that, with the foundation work laid out in the maritime law business logic, interpretation of more complex laws could be automated in future.

With this in mind, as well delivering the maritime collision AI, it was important to create an abstract ODR platform that could take any arbitrary module so that it could be utilised in increasingly sophisticated ways in future iterations of the project.

1.2 Background & Analysis

1.2.1 An introduction to ODR

ODR is a specialised type of Alternative Dispute Resolution (ADR), which refers to "any means of settling disputes outside of the courtroom" and can include negotiation, mediation and arbitration. [22] ODR is being increasingly considered as a viable alternative to traditional courses of action; a recent report by the Civil Justice Council suggested that ODR could be used to settle non-criminal cases of less than £25,000 to reduce the expenses generated by taking cases to court. [11]

There are various motivations for ODR in addition to cost-saving. Since neither party is required to travel to a physical courtroom, disputes can be settled more quickly and conveniently. From an e-commerce perspective, ODR presents an opportunity to increase customer satisfaction and help consumers retain their trust in the retailer.

Existing ODR platforms facilitate discussion, allow the attachment and perusal of evidential documents and offer an unbiased platform in which both parties can try to reach an amicable

resolution. However, in most cases, existing platforms do not contain any AI that helps to influence the outcome of a dispute; resolution is strictly a manual process.

To reiterate what was previously discussed, *Online Dispute Resolution for Maritime Collisions* aims to deliver an abstract ODR platform whereby a module containing maritime law business logic can be plugged into the system. The module itself should ask specific, structured questions, interpret the answers by both parties and suggest a resolution according to maritime law.

Although it is the maritime collision AI that has never been attempted in ODR before - thus breaking new ground - the most valuable deliverable in this project will be the core ODR platform itself for being something abstract and extensible enough to support such a module. As such, the greatest development emphasis has been put on the ODR platform, rather than the module.

1.2.2 Existing ODR platforms

The market leader for online dispute resolution is Modria, who claim to have resolved over 400,000,000 disputes to date. It is a cloud-based platform built by the team that created the world's largest online dispute resolution systems at eBay and PayPal. [29]

This company provides proprietary software as a service (SaaS), existing on Modria's own cloud-based servers. Although it is marketed towards high-volume, low-value disputes such as those that arise in e-commerce, Modria claims to be able to resolve disputes of "any type and volume" and its platform has been utilised to resolve divorce cases and property tax assessment appeals. [28]

Modria stands for "modular online dispute resolution implementation assistant" and aims to "be the operating system for online dispute resolution [for] any kind of dispute, no matter how complicated or how simple, how high volume or low volume", [28] through the use of the following "building blocks": problem diagnosis, technology facilitated negotiation, mediation and evaluation[1].

The first block gives the parties involved an idea of the scope of the problem, the kind of resolutions available, how long the resolution might take, and so on. The second block provides a communication platform to allow the two parties to negotiate their terms. The third block introduces a third-party neutral to assist the parties in their negotiation, and the final block attempts to deliver an evaluative outcome. Each block is an iterative measure only introduced should the dispute require it. For example, 90% of eBay's 60,000,000 annual disputes are resolved without the need of a third-party neutral.

1.2.2.1 Examining Modria's Resolutions Console

Subscribers decide rules through their 'Resolutions Console', an example of which is on Modria's website:

```
If (Customer is Low Risk) and (Dispute Amount is less than $10) and (
    Customer Disputes Filed Account Lifetime is 0) then (Authorize Full
     Refund) and (Close Case).
```

[1] Modria have deliberately chosen the term 'evaluation' rather than 'arbitration' because a lot of the evaluative processes they build are "not necessarily enforceable in a court".

This appears to be quite a powerful feature and allows artificial intelligence to be automatically applied to resolving disputes. However, it is difficult to know the scope to which these rules can be modified.

Given the heavy emphasis on e-commerce transactions, the examples on the website refer only to customers, dispute amounts, transaction periods and so on. For resolving something like a maritime collision, the Resolutions Console would require heavy configuration to be able to interpret maritime law into a set of rules:

```
If (First Agent claims it was an Accident) and (Second Agent claims it
    was an Accident) and (First Agent says they were responsible) and (
    Second Agent says the other Agent was responsible) then (First
    Agent must Pay All Damages)
```

The above example shows how Modria might be able to support some interpretation of the law, though this is already becoming quite complicated. Now let's consider other factors we might want to factor into the resolution. We may want to retrieve the 3 most similar historic cases and feed them back into the resolution logic:

```
If (Similar Case 1) and (Similar Case 2) and (Similar Case 3) are all
    in favour of (First Agent) then (Second Agent must Pay All Damages)
```

Again, this looks doable in theory, but how do we retrieve the most similar historic cases? Are we able to feed that into the Resolutions Console? Surely we'd need to write custom code to implement such functionality, and the Modria platform would need to support the execution of arbitrary code. Modria may indeed be made up of "building blocks", but these building blocks seem quite generic to all dispute types. Is there support for a custom building block containing domain-specific business logic?[2]

This and any other examples more complicated than simple e-commerce disputes highlight the void that still exists in the world of ODR. What seems to be missing is an ODR platform that allows developers to create and plug in arbitrary modules of logic. These could be custom built to be ideally suited to resolving a specific kind of dispute, in this case maritime collision disputes.

Such a platform would probably also have to be open-source, since the developer needs to be able to hook into events and/or functions exposed by the underlying platform. Modria is unlikely to ever offer plugin functionality as it would not make sense for it to go open-source. In the words of open-source advocate Eric Raymond, "when the rent from secret bits is higher than the return from open-source, it makes economic sense to be closed-source". [30] It is unlikely that the return from independent peer-review would be more valuable than the subscriptions to Modria's closed-source platform on account of there being no alternative option. If Modria were open-source, subscribers would have the option instead to download and compile the source code onto their own servers, and Modria would lose out financially.

1.2.3 Maritime law

Some time was spent examining different maritime law documents. Maritime law differs depending on the jurisdiction of the territories in which the collision took place, so it would be necessary

[2] As I am not an e-commerce company, a request for a demo and several follow-up emails have gone unanswered, and unfortunately there is no other information available on the website. I would have liked to have known whether or not there are any plans to allow developer expansions through a remote API or embedded module/plugin facility.

for the maritime law AI implementation to take this into account. This might mean asking the parties where the collision took place and then applying the correct maritime law for that jurisdiction, or it might mean creating several different maritime collision AI modules and having the parties manually select the correct one for their dispute.

In my research, I identified two key maritime conventions which could be implemented in the module. The first and most comprehensive of these was the *Convention on the International Regulations for Preventing Collisions at Sea*, split into five large sections and comprising a total of 38 overarching rules. [2] This would be a good candidate for implementing in the module, but would require very thorough testing and perhaps quite an advanced logic mechanism, such as a neural network, due to its size and complexity.

On the other hand, the *Convention for the Unification of Certain Rules of Law with respect to Collisions between Vessels* describes a relatively simple, condensed maritime law totalling 17 short articles, each of which looks fairly straightforward to translate into code. [1] On the advice of Dr. Constantina Sampani (hereafter referred to as 'the customer'), this was used as the basis for the business logic in the module.

A few days were spent reading around the subject of ODR. Some publications were useful for describing existing ODR systems [20] and discussing where ODR systems might be heading; [10] others raised interesting legal questions in terms of AI not being able to encode justice [36] or EU law not catering for the recent rise in legitimate disputes concerning free SaaS. [25] Additionally, some time was spent gathering around 150 historical maritime collision cases from various sources for possible inclusion in the module. [14] [8] [19]

1.3 Objectives

These were the objectives clarified early on in the project, to be tackled incrementally:

1. **To build an Online Dispute Resolution platform**. This is the minimum viable product, and is easily substantial enough to be a Major Project on its own, encompassing front-end development, back-end business logic and database integration.
2. For this Online Dispute Resolution platform to be **tailored to Maritime Collisions**, providing some sort of conclusion given the details of the dispute. As a worst-case scenario, this may mean hard-coding business logic, rules, database schemas, and so on, to fit with maritime collision properties.
3. For this Online Dispute Resolution platform to be **an abstraction, able to take a module of business logic** (perhaps Maritime Collisions, perhaps something else). The aim was to bypass step 2 altogether to arrive at this stage, as it is important to keep the system abstract.
4. As an additional feature, the system should be able to **retrieve the most similar historical cases**, which should be of use to the lawyers involved. This will involve a large degree of setup work, including sourcing the cases and representing them in a consistent data structure.
5. Following on from the ability to retrieve similar cases is the ability to **feed the details of the similar cases *into the current dispute***, thereby influencing the court simulation and making this feature even more accurate and valuable.

Chapter 2

Requirements

2.1 Project Development Methodology

Before delving into requirements analysis, it was important to decide upon a project development methodology, since the approach taken to gather the requirements differs according to the methodology chosen.

After some consideration (discussed in appendix A), it was decided that the approach should be a hybrid one of Waterfall and Agile. The project would begin with a strong set of requirements and there would be some up front design for parts of the project that are unlikely to change, such as the database schema. At the implementation stage, the project would switch to a test-driven approach that utilises the best of the agile processes.

2.2 Roles

Simple ODR platforms targeted towards e-commerce disputes typically involve just a buyer and a seller. Our ODR platform is intended to be applied to more serious cases and thus has four main roles:

- **Law Firm** - registered to the system by an authorised individual (such as managing director). A Law Firm can have many Agents.
- **Agent** - a lawyer, working on behalf of a Law Firm. An Agent must be in one Law Firm.
- **Mediation Centre** - a company specialising in the mediation of disputes. A Mediation Centre can have many Mediators.
- **Mediator** - working on behalf of a Mediation Centre. A Mediator must be in one Mediation Centre.

2.3 Use cases

A good place to begin was to create various use case diagrams representing the roles in the system and the actions they ought to be able to perform. By doing so, it would also be possible to derive common classes and actions by examining similar or duplicating use case scenarios. These use cases were derived from early meetings with the customer and other stakeholders.

2.3.1 Registration

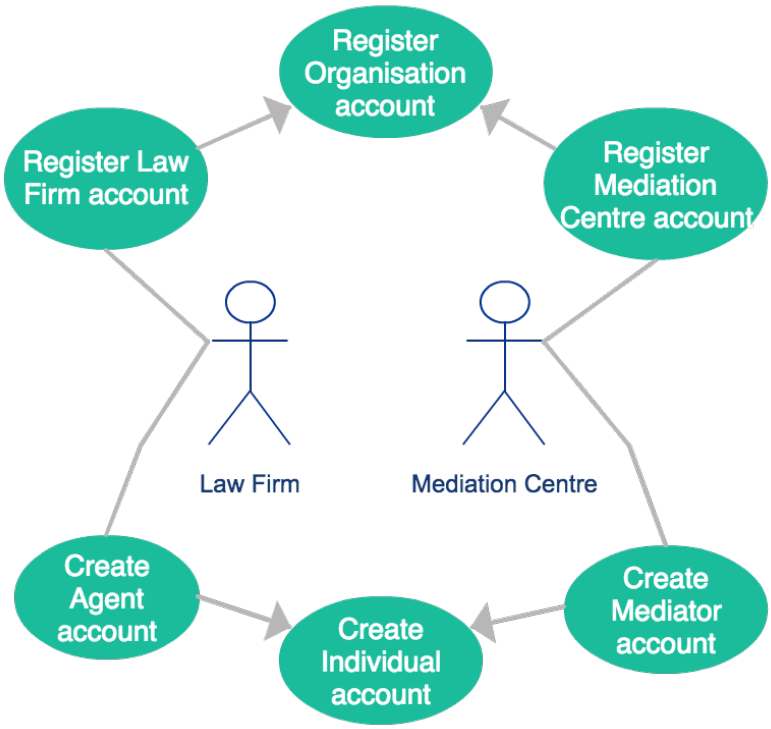

Figure 2.1: Use case diagram showing registration feature

Authorised individuals should be allowed to register accounts representing their company (be it a law firm or mediation centre), and within that organisation account they should be able to register individual accounts. These individual accounts should be agents or mediators depending on the organisation type.

Figure 2.1 shows this in terms of the law firms and mediation centres. A generalised action has been added in both the organisation and individual registration, showing where it might be possible to use a common class or database table to accomplish both goals.

2.3.2 Disputes

Figure 2.2 shows the roles and actions involved in the creation, mediation and closing of a dispute. Arrows denote where one action has a dependency on another.

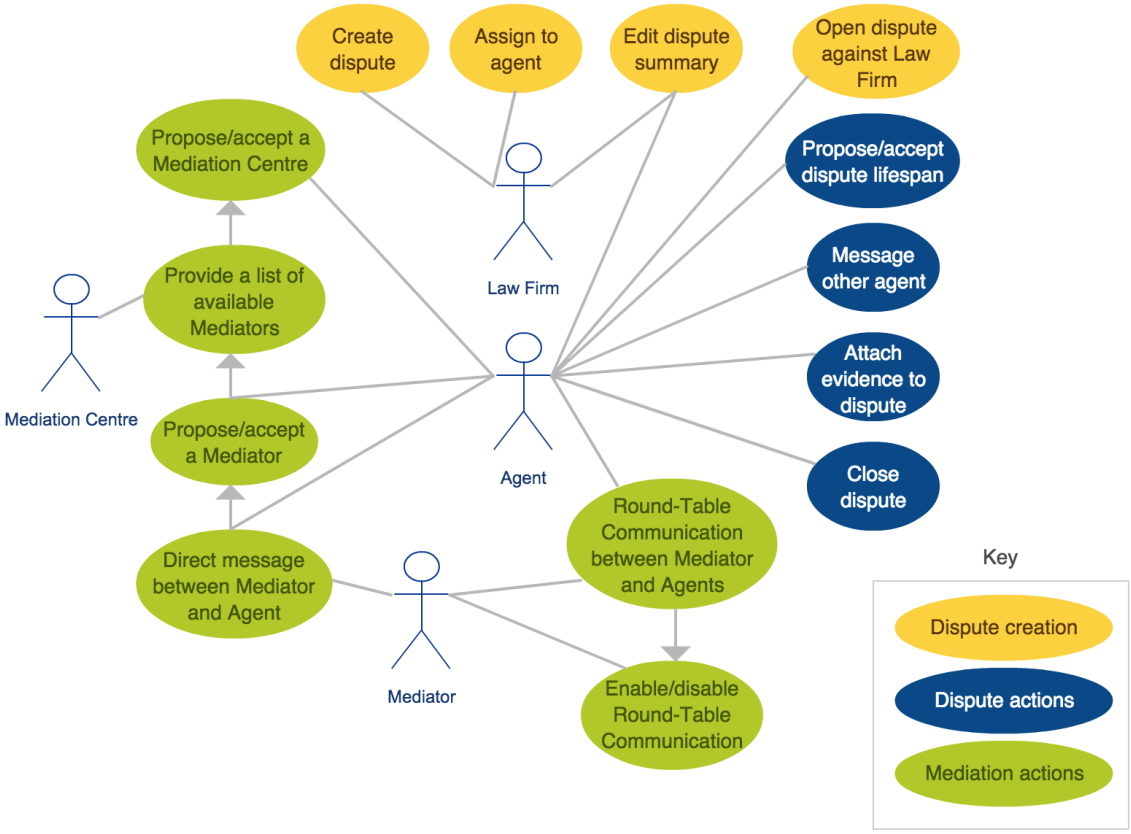

Figure 2.2: Use case diagram showing actions available in a dispute

Only law firms can create new disputes. There is then some back-and-forth assignment between law firms and agents until both sides of a dispute are represented by opposing agents. This is identified as the 'dispute creation' stage.

Inside a dispute, agents should be able to negotiate the dispute lifespan, exchange messages and evidence, and be able to close the dispute. In a best-case scenario, this is all that is required to successfully resolve a dispute. This is known as the 'dispute' stage.

Should it be required, an agent can propose mediation, and there is a defined process of administration between the agents and mediation centre required to get the dispute 'in mediation'. Once in this state, the agents can communicate only through the mediator, unless the mediator feels the dispute is close to resolution and decides to enable round-table communication.

2.3.3 Miscellaneous

Other, lesser elements of functionality are shown in the miscellaneous use case diagram in figure 2.3. For example, agents should be able to peruse a mediator's CV before making a decision as to which mediator to opt for; this suggests the need for a "view profile" facility with custom fields for the CV, which could be as simple as a HTML textarea or as complicated as an integrated PDF uploader and viewer. Given the tight deadline of the project and the scale of the system, it was decided that these miscellaneous features should be kept as simple as possible.

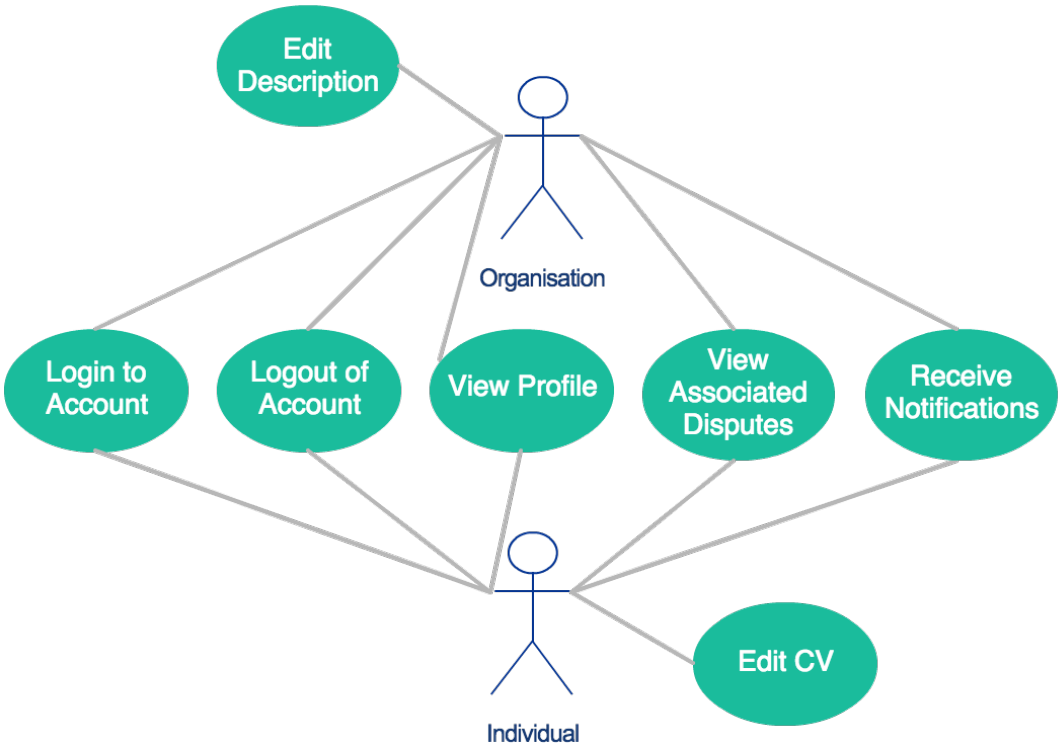

Figure 2.3: Use case diagram demonstrating other miscellaneous requirements

2.4 Dispute process

Although the use case diagrams describe the features required by the system, they do not make it very clear when those features should or should not be available. An early meeting with the customer emphasised that a dispute should follow a very specific workflow.

The activity diagram in figure 2.4 shows the creation of a dispute and the features that become available to the agents when the dispute has been initialised. The activity diagram continues into figure 2.5, which shows what is involved in putting a dispute into mediation. In both diagrams, red boxes indicate the current 'state' of the dispute.

Through the requirements-gathering process it became clear that the end-to-end lifecycle of even a simple dispute is actually quite complicated. I later developed a webpage, "How does SmartResolution work?", to help explain the process of a dispute. The complete workflow can be found in appendix B. If the dispute-creation or mediation process is still unclear, it is recommended that you read and understand the contents of appendix B before continuing.

2.5 Features

Following on from the use case diagrams, it was critical to explicitly define the project requirements in a textual way. In a traditional Waterfall model, a requirements specification is a key deliverable created at the beginning of the process, whereas in an agile model, features are represented as user stories which are then estimated, prioritised and tackled iteratively. My approach

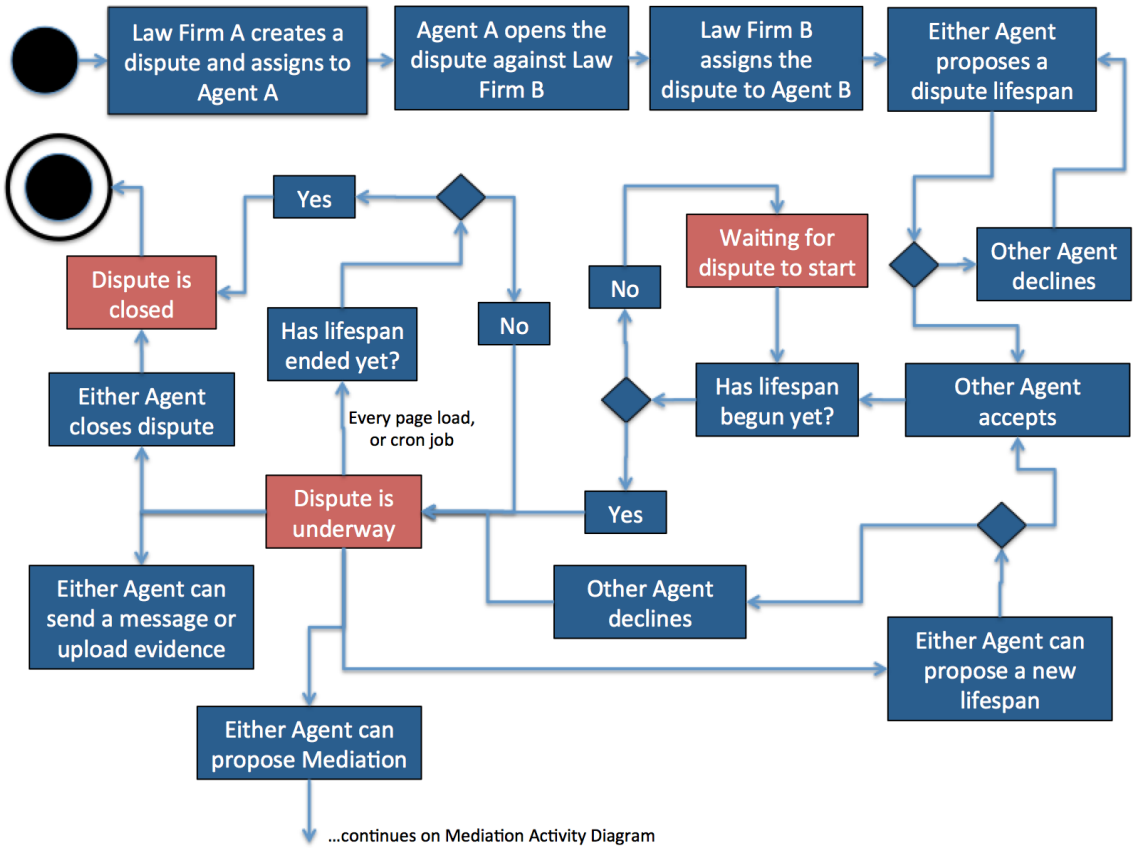

Figure 2.4: Activity diagram showing the workflow required in creating a dispute

to textualising requirements is a compromise between plan-driven and agile approaches. In this project I specified requirements in the form of Cucumber features, which are executable in a BDD way that lends itself perfectly to the agile practices of TDD and CI.

BDD, or business-driven development, allows you to write executable features in a human-readable way so that a business analyst is able to understand the requirements but does not need to know the technical implementation. The features follow a convention known as the Gherkin syntax so that each feature step can be matched to a corresponding step definition represented in code, allowing automated end-to-end testing.

Appendix C contains the full set of Cucumber features which were originally signed off; these features act as the requirements specification. The nature of these being a part of the codebase means that they can evolve over time, which is simultaneously an advantage (for always being in line with the implementation) and a disadvantage (for allowing "requirements creep"). As a result, the Cucumber features at the end of the project are now markedly different to the original set of signed-off features. Though these current features could also have been included in the appendix, they only represent the features at the time of publication and are likely to change should this project be developed further in future. Given that there is little historical value in this, I have decided not to include the current set of Cucumber features in the appendix. For the most up-to-date Cucumber features, please refer to the technical hand-in or to the GitHub repository.

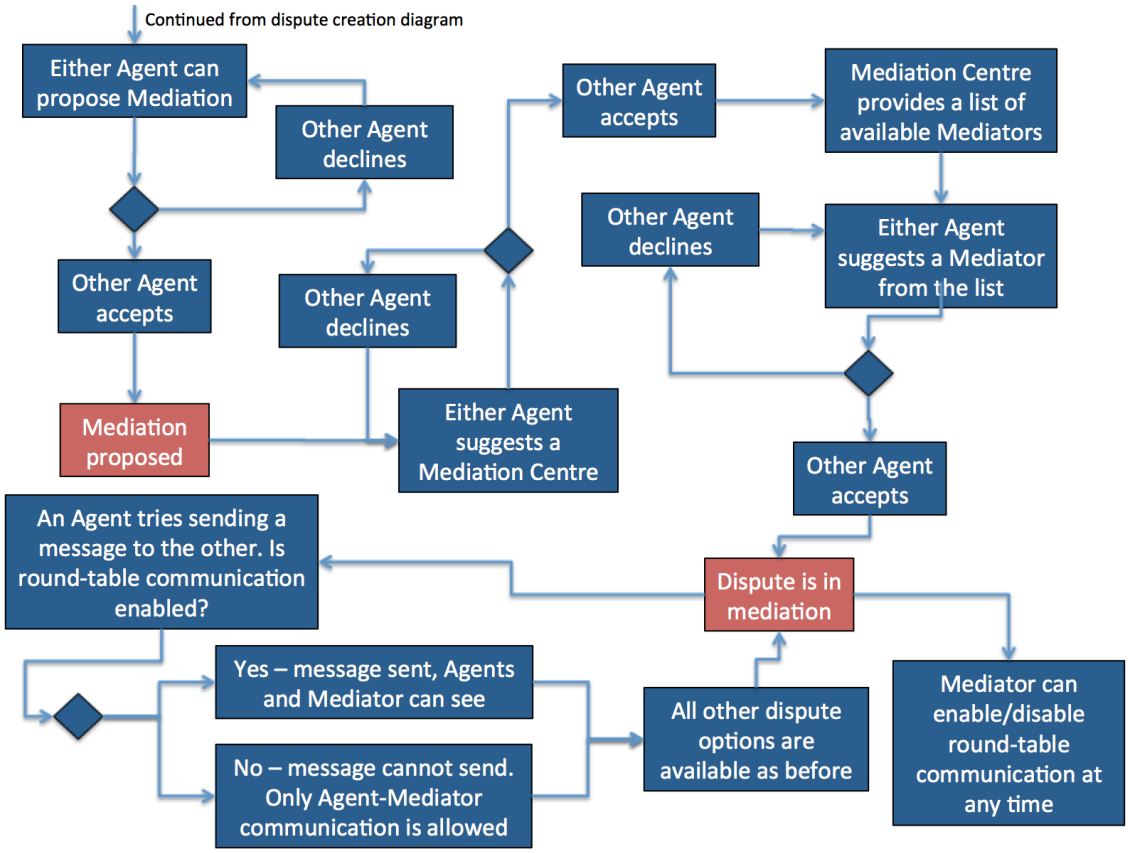

Figure 2.5: Activity diagram showing the workflow involved in getting a dispute into mediation

2.6 Development Methodology versus Project Management

Each of the development methodologies discussed has an associated project management approach. Waterfall projects tend to use Gantt charts to plan progress, whereas agile approaches tend to use sprints to plan individual iterations. As this project would be using a hybrid development methodology, the question was raised as to whether or not it should be using a hybrid project management approach.

Given that the first half of the project would be plan-driven and that significant efforts were made in the early stages of the project to clarify exact requirements, it made sense to adopt the plan-driven approach of creating a Gantt chart which plans out the implementation of the features.

The Gantt chart in figure 2.6 shows two timelines. The time periods in blue represent the original Gantt chart I intended to follow for this project. The time periods in orange represent what actually happened over the course of the project. The reasons for the disparities between the two are discussed in section 6.3.

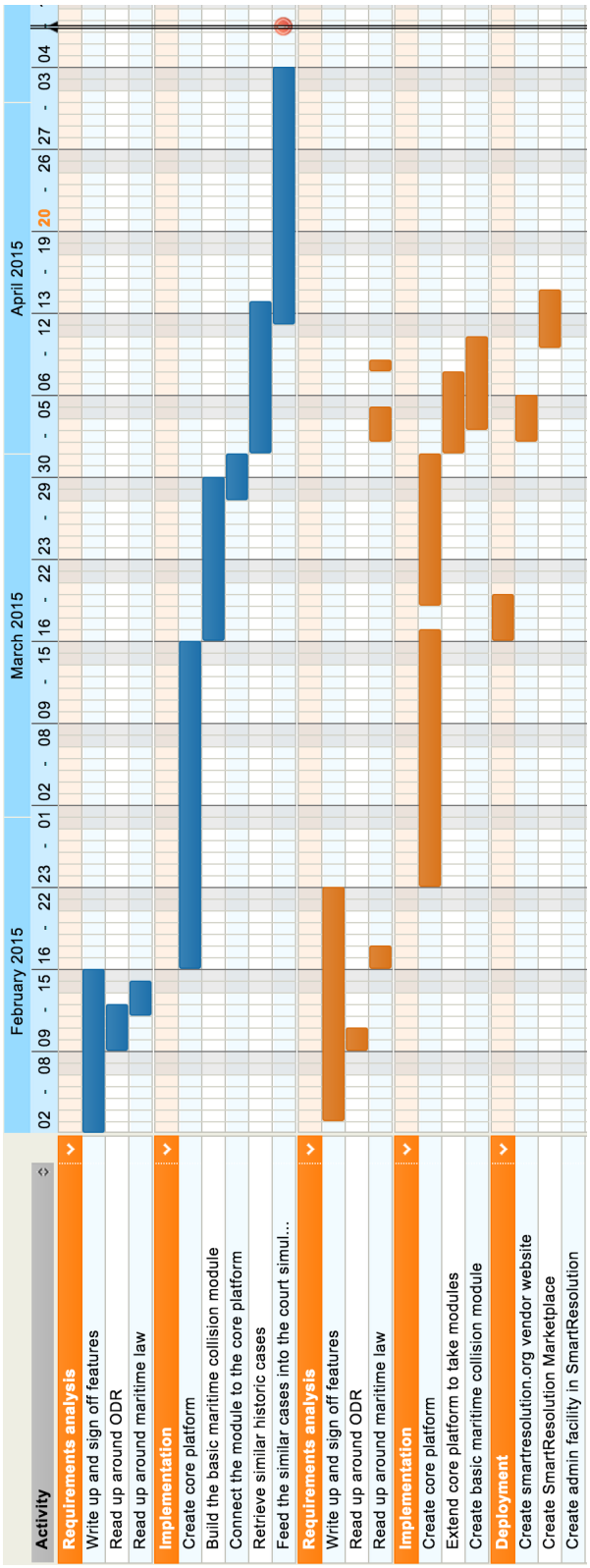

Figure 2.6: Gantt chart showing the original plan, compared with the plan actually followed

2.7 SmartResolution

The report thus far has discussed the project in the form of two main components: the core ODR platform and the maritime collision module. Realistically, a third component was required: a vendor website. This website would host the core software and provide instructions on how to install it onto a server.

If developers were to be excited enough about ODR to develop modules of functionality like the maritime collision module, then this project would require a brand. I felt that the term 'SmartResolution' embodied what the ODR platform was all about: online dispute resolutions done in a smart way, by interpreting disputes using artificial intelligence to automatically suggest resolutions. SmartResolution is the term I'll use to refer to the core platform from this point onwards.

Far from being just an information resource, the SmartResolution website would later provide a facility to download and install SmartResolution modules directly through the SmartResolution installation itself, a little like downloading an app to an Android device directly through the Google Play store. The 'SmartResolution Marketplace' was implemented towards the end of the project.

Chapter 3

Design

To recap, this project used a hybrid approach of a plan-driven methodology followed by an agile implementation. The design stage is where the first methodology began to merge into the other.

A lot of time was invested in clarifying requirements so that the common classes and methods could be established and fed into parts of the initial design. As a result, certain aspects of the design could be designed up front as they were unlikely to change unless there was a change in requirements. Other aspects such as class diagrams would not be suitable for up front design, since the code would be refactored throughout the process and would ultimately fall out of line with the documentation.

3.1 Development choices

3.1.1 Programming language

As a web-based solution, the options were somewhat constrained to server-side languages. PHP, Node and Ruby on Rails all came to mind. It was decided that PHP would be the most appropriate choice.

PHP is currently the most popular server-side programming language worldwide, and, perhaps more importantly, it's the server-side language I had by far the most experience using. [34] My thought process was this: though I could have spent a few weeks learning the intricacies of ASP.NET to deliver my project, it would have been an unwise use of my time given that there was so much to build. I didn't want to find myself desperately trying to debug an obscure error in an unfamiliar language a week before the deadline.

Development experience aside, PHP being the most popular server-side language means that there is extensive support and documentation online: any problems faced during development have likely already been solved through other peoples' experiences. Moreover, if the project were to go open-source, it would be more likely to acquire a sizeable and active developer community than a lesser-known language. Finally, server-support for PHP is very common: by opting for PHP as the language of choice, SmartResolution is more likely to be able to be installed and used by more people.

3.1.2 BDD Framework

There are many options for parsing feature files. Cucumber was originally built for Ruby, but now there are other variants such as CucumberJS (for JavaScript) and Lettuce (for Python). The de facto standard BDD implementation for PHP is Behat.

Personally, I dislike Behat because of its syntax. It requires that you define your step definition as a concatenated function name and write metadata above the function definition:

```
/**
* @Given /^I am in a directory "([^"]*)"$/
*/
public function iAmInADirectory($argument1)
{
throw new PendingException();
}
```

This just isn't as *clean* as Cucumber and Ruby:

```
Given(/^I am in a directory (.+)$/) do |argument|
# pending
end
```

Syntactical disagreements aside, Matthew Daly's blog post about testing PHP web applications with Cucumber discusses the benefits of Ruby and Cucumber compared with Behat. He opts for the former because RubyGems makes it easy to install Cucumber's dependencies and the Cucumber community appears to be very active. [12]

Finally, by choosing a BDD framework whose language is different to that of my project, I'd be forcing decoupling between my application and its regression tests, making it impossible to make certain inappropriate testing choices such as directly including application PHP classes inside the step definitions. For all of the reasons outlined above, I decided to opt for Ruby and Cucumber as my BDD framework.

3.1.3 Reusing open-source software

It was hoped that there would be an existing open-source ODR platform upon which to base the project. One would have been able to make the necessary modifications to support the maritime collision module without having to build the entire platform from scratch, thus allowing more time to be spent on the maritime collision module.

Unfortunately, after fairly extensive research, I was unable to find an existing ODR platform that was not proprietary. Most ODR providers, be they service or platform providers, charge a subscription or one-off fee and would be at a commercial disadvantage if they were to go open-source. Without an existing ODR platform to base the project upon, there was no option but to develop the core platform from scratch.

3.1.4 Project Framework

It makes sense to minimise development effort through the use of a framework, letting the combined efforts of hundreds of contributing developers do the heavy lifting so that we are free to implement the application-specific code. Dozens of PHP frameworks exist, so deciding on one can be difficult.

Large frameworks such as Zend and Symphony are quite constraining, requiring you to stick to their idea of what is the best development approach or else perform some very heavy, non-standard configuration to make it suit your project. Either of these heavyweight frameworks could be perfectly well suited to ODR, but generally speaking I try to stay away from non-transferable frameworks. I like to understand exactly how my application works, rather than delegate that understanding to a third-party library. It is also advantageous to be able to change frameworks painlessly at a later date - something which is not suited to heavyweight frameworks.

Lighter options exist, including Huge, Laravel and Fat-Free Framework. Huge looked promising to begin with but then appeared to be more of a middleweight framework, strongly encouraging certain directory structures. It would be a base that the ODR software would have to build upon, rather than a library that could be plugged into the system, only called as and when needed.

Laravel was the next one considered and looked a little like PHP's answer to Rails, as it generates an entire application structure through a single command. Again, this framework comes with a set of defaults which may not necessarily be suited to your project, locking you into a pre-conceived idea of how an application should be structured.

It would be impossible to thoroughly evaluate every framework to make a truly considered decision, and I'm sure that any one of the above frameworks would have been a perfectly suitable starting point. However, the most promising-looking framework for an agile project was Fat-Free Framework, or F3 as it is commonly known. It had a low learning curve and an unconstraining nature, and its modular build meant that I could cherrypick the elements of functionality needed for the project, rather than go for an all-or-nothing installation. F3 is fundamentally different to its competitors because I could slot F3 into my code, rather than slotting my code into F3. For these reasons, I chose F3 as SmartResolution's framework.

In a recent interview I presented this argument to Gavin Love, Chief Technology Officer at MyBuilder.com, who agreed that modularity seems to be the way that PHP frameworks are heading. This suggests that delegating units of functionality to a lightweight framework might be a future-proof decision. Developing software on a heavyweight framework which subsequently loses support can be very difficult to refactor, as Gavin says he found when upgrading MyBuilder.com from Symphony 2 to Symphony 3.

3.2 Overall architecture

The project can be broken down into three main components.

3.2.1 Core platform

SmartResolution is the ODR platform offering the core online dispute resolution requirements, such as organisation and user registration, dispute creation, messaging, file uploads, and so on. It

offers the minimum facilities necessary to successfully negotiate a dispute online.

As this was the basis of all the other components, it was also the most thoroughly engineered, backed up with integration and unit tests, continuous integration, automated code quality checks and dependency status checks.

3.2.2 Maritime collision module

This module is separate from the core platform and lives in its own repository. It can be manually installed to any SmartResolution installation simply by being copied to the `modules` directory, or automatically installed through the SmartResolution Marketplace facility.

As discussed in the background and initial requirements, it was intended for this to be a feature-rich module that could find similar historic cases, cross-check agents' answers with maritime law, and approximate the likelihood of an agent's success in court.

3.2.3 SmartResolution Marketplace

smartresolution.org is the vendor site for SmartResolution, explaining what SmartResolution is and offering a download link to a production-ready version. Its subdomain, demo.smartresolution.org, has a live demo of the SmartResolution software installed so that users can try out the software before they download. However, one of the main reasons for developing the website was the creation of the 'marketplace' facility, discussed later in this chapter.

3.3 WordPress: a comparison

The commercial viability of the software (analysed in appendix D) drew some parallels with the blogging software WordPress. WordPress can be said to have a three-tier system:

1. **WordPress platform**: blogging software that can be downloaded, installed and hosted on a LAMP server.
2. **WordPress plugin**: a self-contained package that can be installed to a WordPress installation and which augments the core installation with additional functionality.
3. **Plugin Directory**: a searchable area of the WordPress.org website which links to thousands of available plugins. This facility is tightly coupled to the administrative functions in the core software, as administrators are able to browse, download and install plugins from wordpress.org, within the WordPress installation itself.

These three tiers map to the SmartResolution core platform, the maritime collision module and the SmartResolution Marketplace respectively.

Some developers can be quite narrow-minded about WordPress, but it is a platform that powers 23% of the internet. [35] The reasons for this are that it is open-source, highly customisable, and the developer documentation is very good. The brand is reliable: people trust it. I will try to emulate all of this as much as possible in the development of SmartResolution.

3.4 Choice of Licenses

The design section may seem an incongruous place to discuss the issue of licensing, but licensing can be thought of as a high-level form of design. Linus Torvalds, chief architect of the Linux kernel, once said he considered his choice of the GPL license for the Linux kernel "one of the very best design decisions" he made, since it is designed to protect the author's legal and moral rights to their code. [37]

I opted for the GPLv3 license for the core SmartResolution software. This is almost identical to WordPress' choice of a GPLv2 license, which is presumably still at version 2 for legacy or backwards-compatibility reasons.

The benefits of a restrictive license is that nobody has the ability to download, modify and then sell the software without making the source available to all. Multinational corporations who can afford to compensate developers for their time are not able to use SmartResolution as a basis for closed-source software[1]. Any improvements they make to the software must be kept open-source, as it is a derivative work.

On the other hand, the GPL allows for legitimate use-cases for SmartResolution. For instance, anyone can download the software, install it on their own server and start providing ODR services. They can even charge a subscription fee, and would owe nothing to the SmartResolution developers. The choice of GPLv3 for SmartResolution protects me from other people making money selling *software* derived from this project. It doesn't stop people making money providing ODR as a service, and it won't infect their website "like a cancer", to quote Steve Ballmer, ex-CEO of Microsoft. [6]

Though I originally hoped to release the maritime collision module under the MIT license, WordPress and Drupal believe strongly that any plugins or themes for their CMS are classed as derivative works, stating: "The GPL on code applies [also] to code that interacts with that code". [13] Thus, like the parent software, these derivatives should be bound to a GPL-compliant license. Any modules which call PHP functions defined by the core SmartResolution software must therefore also be bound to the GPL license, so I have released the module under the GPLv3 license.

The fact that all distibuted modules are bound to the terms of the GPL shook my confidence in the commercial viability of creating proprietary modules for the system. I found it difficult to understand how WordPress' premium plugins were able to be sold at all, as one of the four freedoms enabled by the GPL is "the freedom to redistribute copies", meaning anybody who purchases a premium WordPress plugin is able to redistribute the plugin for free. [18] Chris Lema suggested a number of explanations for the success of premium plugins in his blog post, mainly alluding to the fact that only plugins which are bought legitimately get the latest updates, bug fixes and support for a specific period of time. [23] The fact that so many plugins and themes *are* able to be sold through WordPress indicates that the same business strategy could be applied to SmartResolution modules.

I saw no advantage in making the SmartResolution website itself open-source, since I'm not encouraging anybody to download and re-use the website itself. I have kept the source-code for smartresolution.org private and all of its contents are under my copyright.

[1] This only applies if the company is *distributing* the software. In-house software can remain closed-source.

3.5 Design: SmartResolution

3.5.1 Choice of Database Management System

Before going down the route of designing a database schema, a decision had to be made as to whether to go for a relational database management system (RDBMS) or an object-oriented database management system (OODBMS). The former is the traditional means of application persistence, whereas the latter is still fairly new and is intended to allow developers to think purely in terms of object-orientation.

RDBMSs have more widespread support than OODBMSs, and I wanted SmartResolution to be as accessible as possible for those wanting to provide their own ODR services. In addition, my choice of PHP framework has built-in database support and is geared towards relational databases.

OODBMSs gives an advantage to those companies that are geared towards multimedia presentation or organizations that utilize computer-aided design. [9] In contrast, SmartResolution is quite text-heavy and is perfectly well suited to a RDBMS implementation.

My next decision was which RDBMS to go for. The market leader is MySQL, but many alternatives exist, such as PostgreSQL, MSSQL, Oracle and SQLite. F3 supports all of these and more. [15] The quickest database to get started with was SQLite, as it meant that I could develop locally without having to worry about creating the database administrator accounts required by other RDBMSs, such as MySQL. SQLite databases exist simply as a file on the system, whose contents can be viewed and edited through a program such as DB Browser for SQLite. The fact that SQLite databases exist as files on the system makes having multiple databases trivial, so my system can easily switch between test and production databases depending on the context in which the application is accessed. For these reasons, I selected SQLite as the RDBMS of choice for the project.

It is important to note that this can be swapped out for another RDBMS relatively easily, since all database interactions in SmartResolution go through one `Database` class. This class calls F3's `SQL` class, and as discussed above, F3 has implementations for all of the main RDBMSs. Indeed, if I had a SmartResolution installation in a production environment which was being used by thousands of people, I would consider changing to a RDBMS such as MySQL. Even the SQLite developers suggest that SQLite is best suited as a stand-in for an enterprise RDBMS: "SQLite is often used as a surrogate for an enterprise RDBMS for demonstration purposes or for testing". [32]

3.5.2 Database schema

Figure 3.1 shows the database schema for the SmartResolution core platform, where `P` symbols refer to primary keys, `F` symbols refer to foreign keys, and `U` symbols refer to unique integrity constraints. Lines generally denote foreign key relationships. Other integrity constraints exist, such as 'organisations.type' only being one of "law_firm" or "mediation_centre". These have been omitted from the diagram.

For a full explanation and justification of the database design, please refer to appendix E.

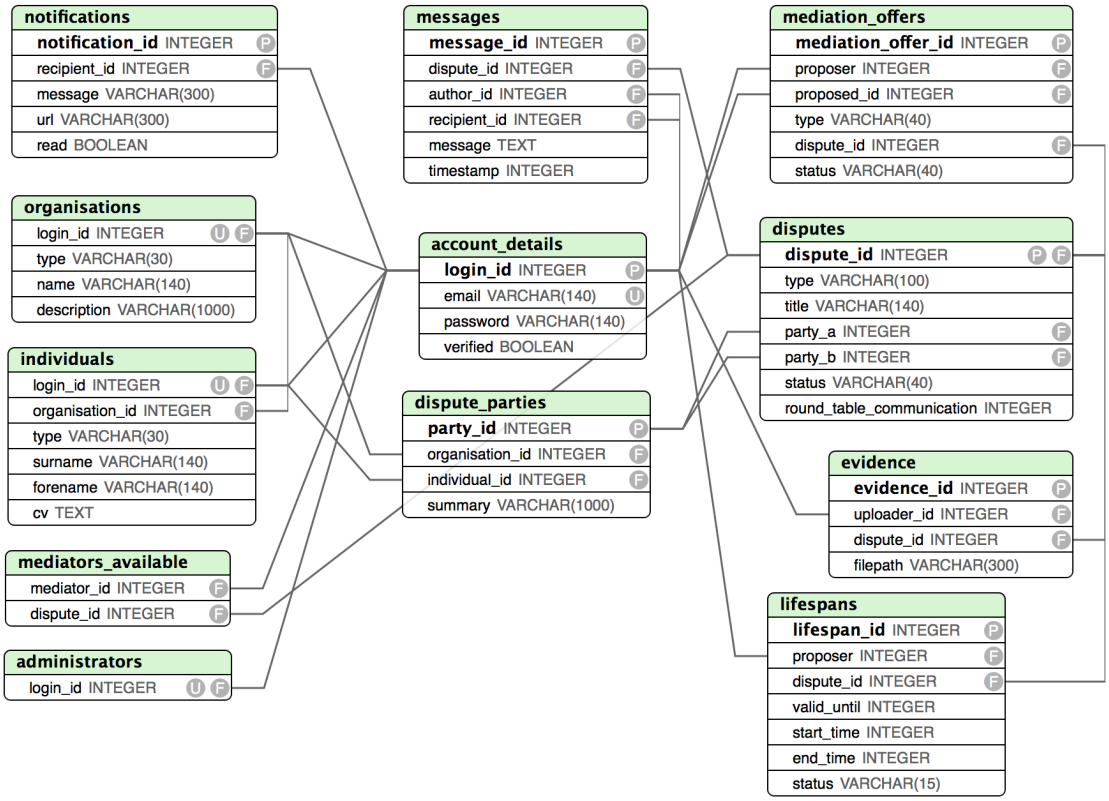

Figure 3.1: Database schema for the SmartResolution core platform

3.5.3 Module support

This section of work is separate from the maritime collision module work, and is concerned with how the core platform should support modules at an abstract level. There should be no maritime-collision-specific code in the core SmartResolution platform, but the platform must support all functionality required by the module.

Taking inspiration from WordPress' Plugin API, the project uses the concept of 'hooks' in a publish-subscribe design pattern. WordPress fires events at various points throughout the normal running of a WordPress installation; these events can be subscribed to and a custom function executed to achieve some arbitrary purpose.

For example, below is WordPress' `add_filter` function:

```
add_filter('img_caption_shortcode', 'my_filter', 10);
```

The first argument is the event to listen for, the second argument is the custom function to execute, and the third argument is an integer denoting the priority of our subscription. Subscriptions with a higher priority are executed before those of a lower-priority. SmartResolution fires hooks which can be subscribed to in the same way as the WordPress function above.

F3's routing API can extend this route-handling concept further:

```
$f3->route('GET /some-route' => 'MyClass->handler');
```

F3's routing allows the developer to assign handling to a public method of a class, rather than a global function, keeping the codebase namespaced and tidy. This is something I've carried over to SmartResolution's parsing of the function name.

These principles formed the basis for the design of the module support:

```
on('event', 'function_to_call', 'priority');
```

'event' is the name of the event to subscribe to, 'function_to_call' is the name of the function to call (and can be a global function, a named class function or an anonymous function), and 'priority' represents the priority with which the function should be called. The priority can be an integer or a string such as 'high', 'medium' or 'low'.

3.5.4 Exposing other methods

One needs to be able to manipulate the rendered output of SmartResolution, e.g. when adding a menu item to the dashboard of a dispute.

This *could* be accomplished by interacting directly with the core platform, as in the example below.

```
global $dashboardActions;
$item =  array(
    'title' => 'Some Action',
    'image' => get_module_url() . '/images/icon.png',
    'href'  => get_dispute_url() . '/custom-route'
);
array_unshift($dashboardActions, $item);
```

This example encourages tight coupling between the module and the underlying platform, locking the core platform into a particular design and risking breaking backwards compatibility should SmartResolution be refactored in the future. If the `$dashboardActions` global variable was removed from the core platform, or the dashboard actions represented with something other than an array, it would break any existing modules relying on that specific implementation.

Again, WordPress provided inspiration for the design. WordPress exposes a number of global functions, e.g. `get_the_id`, which gets the ID of the current post. In this style, SmartResolution exposes a number of global functions. One could now manipulate the rendering of the dashboard like this:

```
dashboard_add_item(array(
    'title' => 'Some Action',
    'image' => get_module_url() . '/images/icon.png',
    'href'  => get_dispute_url() . '/custom-route'
));
```

Not only does this decouple the module/SmartResolution interaction, but this is much cleaner and easier from the module developer's perspective. It is important that there should be as few barriers as possible when it comes to module development, if developers are to get excited about SmartResolution.

3.5.5 Module persistence

Persistence was the most difficult area to tackle, as interacting with the database introduces persistent changes which could be harmful if the developer is not careful. On the one hand, the system has to trust developers. Regardless of the database access methods explicitly exposed by the underlying system, there's nothing stopping developers from accessing the global `Database` class used by the core platform, especially since SmartResolution is open-source and the developer can easily find out what system-level classes are available. At an even lower level, there's nothing stopping a developer from running PHP's `shell_exec` function and executing any command they wish.

The system should trust developers, but it should not trust developers to write perfect code. Though most developers would use an SQL-execution ability only for querying the database for a legitimate reason, they may expose a vulnerability if not thoroughly tested. For example, if they're writing a search engine module which takes a user input, and if they do not sanitise the query, then they're letting the *end user* run arbitrary SQL.

With security in mind, it was decided that the global function definitions should be expanded to define functions supporting specific SQL interactions, e.g. creating tables, selecting rows, updating records, and so on. Early on it was tempting to allow table schema updates on the fly, creating columns as and when they were needed, but this felt dangerous and was tricky to implement. I settled on an up front table design solution by way of a `declare_table` function:

```
declare_table('my_table', array(
    'a_text_field' => 'TEXT NOT NULL',
    'an_int_field' => 'INTEGER DEFAULT 0',
    'initiated'    => 'BOOLEAN'
));
```

The newly created table can then be inserted into and queried using specific, named functions. This does somewhat restrict what the developer can do - for example, there is no method for doing SQL table joins - but more often than not, the developer can still achieve what they need to achieve using pure application code.

In the above example, `my_table` is not the name of the created table. Instead, it is namespaced as `module__[module_name]__my_table`. The module developer doesn't need to know this, and can continue to refer to `my_table` in all of their queries as if that is the name of the table. This means we have the advantage of namespacing our tables (preventing naming conflicts where different modules use common table names) but without the technical overhead of having to remember to prefix table names with that namespace.

Most modules will require some sort of persistence layer to be useful, but developers are not necessarily locked into this database setup. They could use PHP's `file_put_contents` to save to a file, perhaps in conjunction with `json_encode` to save a PHP array as a JSON file. They could even use PHP's `shell_exec` command to create a new SQLite database in their module's directory, giving them complete freedom and the responsibility it comes with. The database functions offered by SmartResolution cut out some of the complexity of doing persistence from scratch, but are by no means the only way of storing data persistently.

3.6 Design: Maritime collision module

This module was developed in tandem with the module support in the core platform. As such, it uses most of the events and hooks exposed by the platform.

As discussed in the requirements section, the basis for the business logic in this module would be the *Convention for the Unification of Certain Rules of Law with respect to Collisions between Vessels*. Everything that is required to apply the Convention can be gathered from a few simple questions, visualised in the flowchart in figure 3.2.

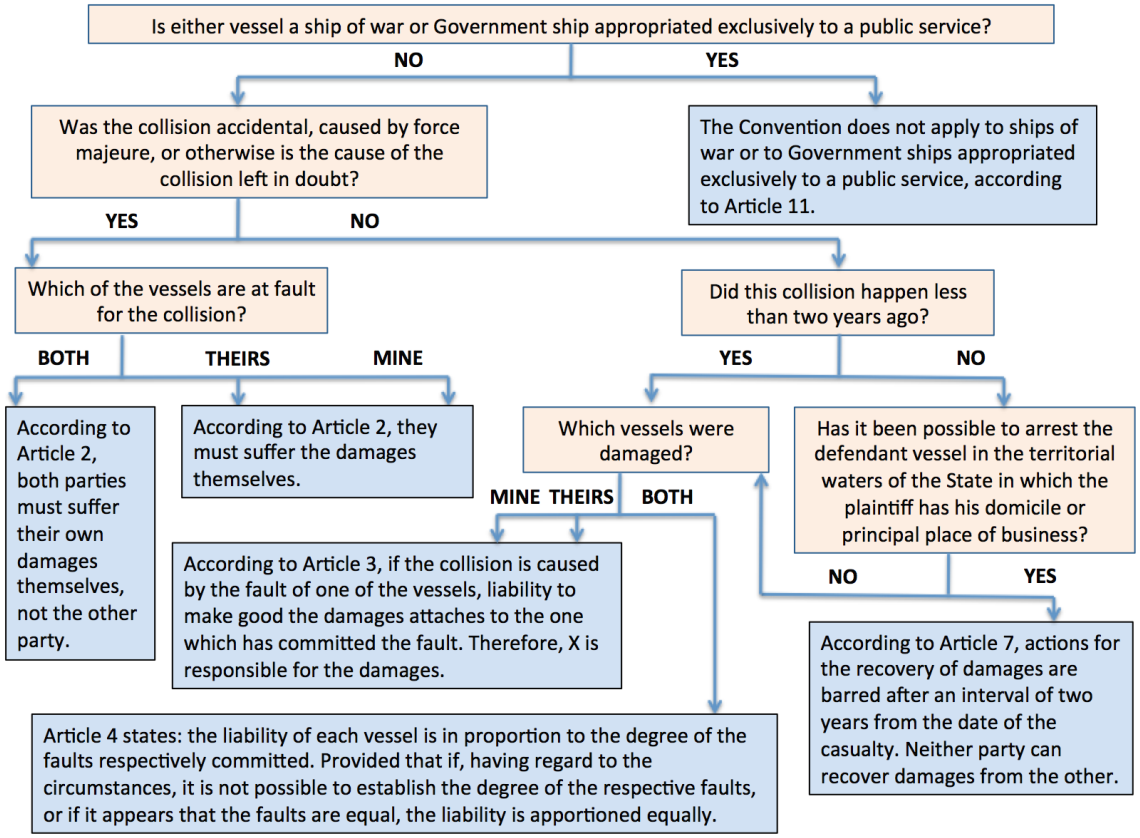

Figure 3.2: Logic for Maritime Law results

Some of the business logic is encapsulated in the JSON representing the questions. For example, some questions should only be displayed if certain prerequisites are satisfied. The questions are represented in the following format:

```
{
    "prerequisites": [
        {
            "question_id":     "article_11",
            "required_answer": "no"
        }
    ],
    "id":   "article_1",
    "text": "Which vessels were damaged?",
    "type": "select",
    "options": [
        {
            "text" : "The vessel of my client",
            "value": "mine"
        },
        {
            "text" : "The vessel of the other party",
            "value": "other"
        },
        {
            "text" : "Both vessels",
            "value": "both"
        }
    ]
}
```

In the above example, the question whose id is `article_1` (corresponding to Article 1 of the Convention) is only displayed if the agent answers the question whose id is `article_11` with the answer "no".

When the agents have answered all of the relevant questions, the following happens:

- The answers are compared to make sure they tally. If one agent says both vessels were damaged and the other agent says only their vessel was damaged, the answers do not correspond and a conclusion cannot be reached. The module cannot be expected to cope with conflicting information. The module would inform the agents of this.
- Provided both answers correlate, the module's `ResultsCalculator` class makes a call to the `deduceSummary` method, which contains the hard-coded maritime law logic. A resolution is then presented to the agents.

Due to the lack of time, the resulting module is a little simplistic. The resolution suggested by the module is deduced through a simple series of if/else statements and probably doesn't tell lawyers anything about maritime law that they don't already know.

However, over-examination of the maritime collision module would be missing the point, which is that the core platform supports any arbitrary module. Modules can be as big and complex as time allows. To go back to the WordPress analogy, on one end of the scale there are many single-file plugins that satisfy a lone developer's personal itch. On the other end, there are enterprise-level plugins that have taken months to engineer and which extend the WordPress platform to be a feature-rich online shop or forum. Given more time, the maritime collision module could be much more heavyweight than it is, encompassing many more aspects of maritime law.

3.7 Design: SmartResolution Marketplace

Early versions of SmartResolution used a PHP array to describe which modules were installed and whether or not they were active. A more user-friendly solution was the creation of an admin dashboard: the ability to sign into an administrator account on your SmartResolution installation, view the installed modules, and activate/deactivate them through a user interface. Following on from this is the ability to view a list of modules, pulled in from smartresolution.org, and download and install them directly through SmartResolution, like WordPress does with plugins.

To accomplish this, there is a JSON feed of featured modules on smartresolution.org[2]. This feed is pulled in and converted to HTML, both directly on the marketplace itself[3] and the SmartResolution admin marketplace dashboard option, emulating what WordPress does with its plugins.

From the admin dashboard on SmartResolution, the modules JSON feed is converted into a HTML page, and server-side logic detects whether or not the module is already installed. If not, a button is rendered which allows the downloading and installation of the module in one click.

In addition to the 'Marketplace' admin option, there is a 'Modules' option which lists all of the installed modules and whether or not they are active. From this screen, the admin can activate, deactivate or delete the module from their SmartResolution installation.

The ability to define which modules to display externally on smartresolution.org gives the SmartResolution maintainers the freedom to change the contents of that JSON, and therefore change which modules are presented to administrators of SmartResolution installations, irrespective of the version of SmartResolution they have installed. This presents a commercial opportunity, allowing smartresolution.org to categorise paid-for modules as 'featured', as hinted by the "Coming soon" modules for Divorce and Breach of Contract.

3.8 User interface

Bootstrap was used to set up much of the initial design defaults in the core SmartResolution platform and also provides a basic UI framework for the layout. Bootstrap has a 12-column layout requiring a specific HTML structure and class-naming convention, but it does mean that the theme has some built-in responsiveness by default.

The CSS overrides provide a clean and attractive look, and helper classes such as `bg-info` and `bg-danger` allowed me to style error messages and other components with minimal effort, meaning I could spend more time on the server-side logic.

Though responsiveness was not a requirement of the core SmartResolution software, I felt it was important to make the SmartResolution *website* responsive. The website fulfils a different need and is likely to be stumbled upon by someone browsing on their phone, whereas lawyers using the SmartResolution *software* are likely to be accessing it on a desktop or laptop at work.

Like the core platform, the website uses Bootstrap for the front-end, but it also uses the SlickNav JavaScript library to create the mobile navigation. Figure 3.3 shows the landing page of the finished website. Figures 3.4 and 3.5 show the SmartResolution website responding to the viewport width, indicating how the website looks on a tablet and a mobile device respectively.

[2]The JSON feed is viewable at `http://smartresolution.org/marketplace/feed`

[3]The human-friendly marketplace is available at `http://smartresolution.org/marketplace`

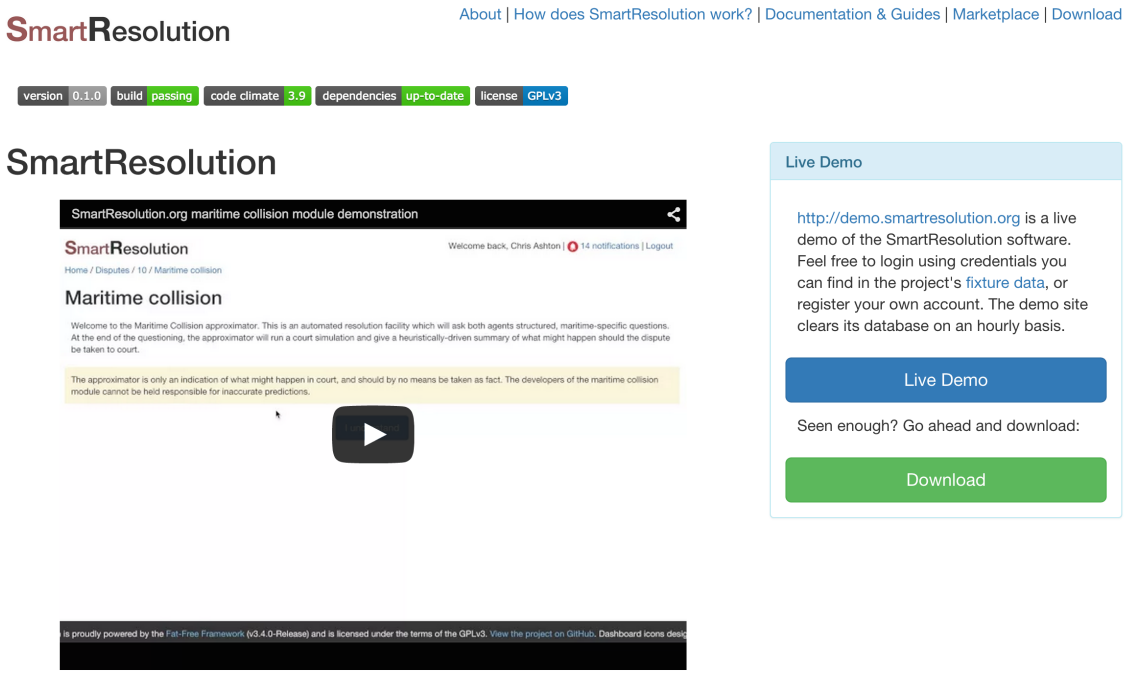

Figure 3.3: Screenshot of SmartResolution vendor website in ‘desktop’ view

Though it would have been nice to have put the same effort into the core software, the size of the software would mean additional (and manual) testing time which could not be afforded on an aspect which was not a core requirement. Moreover, it was hoped that SmartResolution would eventually be able to support swappable themes: development effort would be better expended on that facility than on making sure the default theme is responsive.

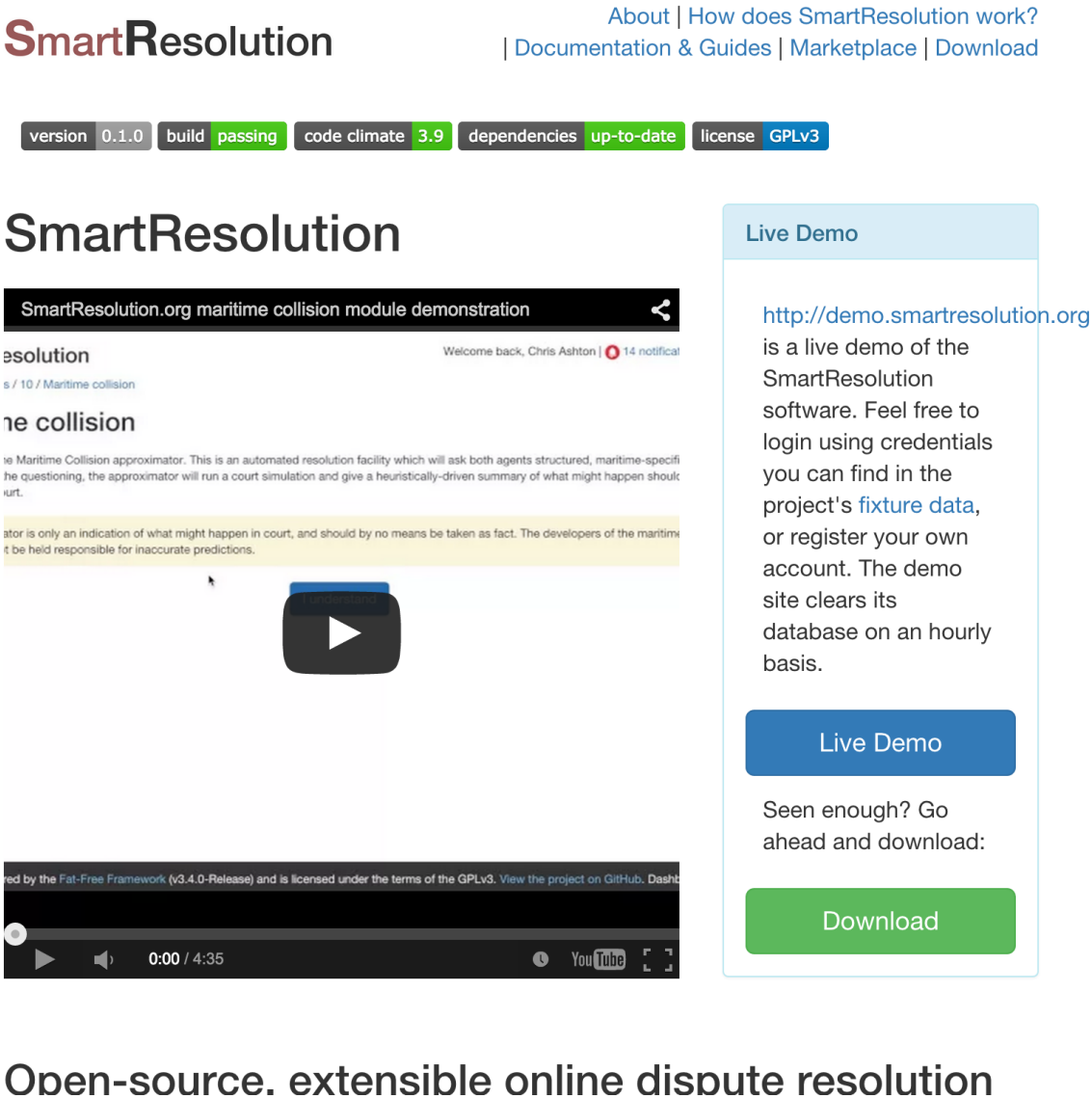

Figure 3.4: Screenshot of SmartResolution vendor website in ‘tablet’ view

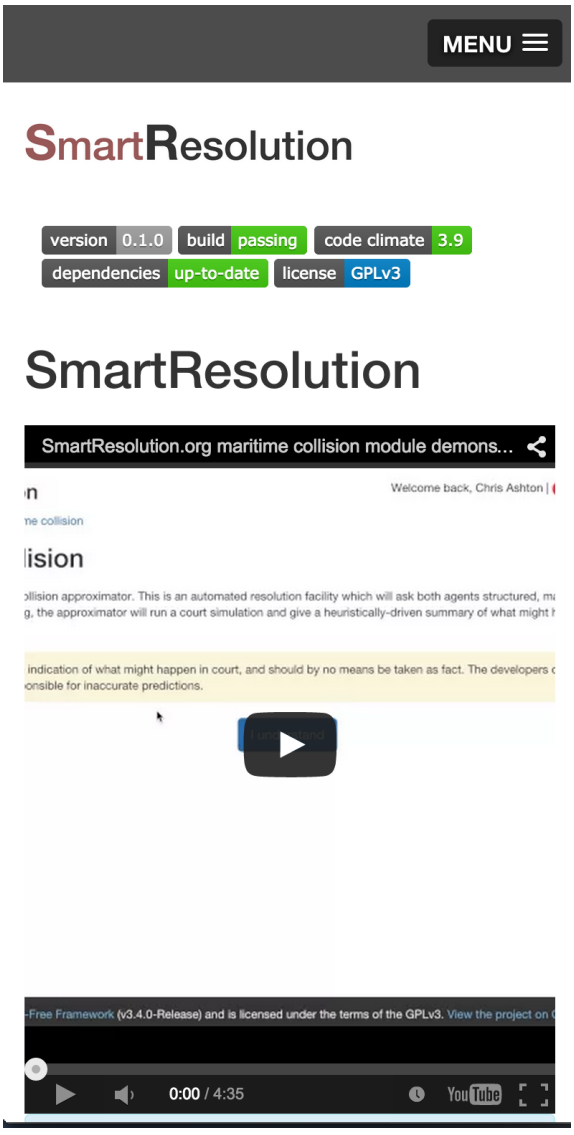

Figure 3.5: Screenshot of SmartResolution vendor website in ‘mobile’ view

Chapter 4

Implementation

4.1 Third-party service configuration

Early on in the project implementation, a few days were spent setting up and utilising third-party services to cut down the costs of maintenance further on in the project's lifecycle. Even in the very early stages when SmartResolution had just one or two unit tests, time was spent configuring a Travis file so that the tests would automatically be run on Travis' CI platform after every pushed commit.

Dependencies such as F3 and PHPUnit were important for the implementation of the project, but they should not exist in the project repository. They also should not add unnecessary complexity to the installation instructions, as developers shouldn't have to manually fulfil each project dependency. For these reasons, I utilised Composer: a dependency manager for PHP which makes it easy for me to specify the project dependencies whilst also making it easy for other developers to install those dependencies. For the same reasons, I utilised RubyGems to specify the Ruby dependencies for the Ruby and Cucumber tests.

Dependency tracking is important, as newer versions of dependencies fix bugs and vulnerabilities discovered in older versions. By not updating SmartResolution's dependencies regularly I would risk crackers being able to exploit unpatched vulnerabilities. This is especially relevant since SmartResolution is open-source and anybody can see which version of which third-party library SmartResolution is using. For these reasons, I signed up to the Gemnasium dependency-monitoring service, which warns me whenever my project's dependencies are out of date.

An important thing to monitor with each commit is code quality, something that CodeClimate is trying to automate. Whenever I push a commit to SmartResolution, CodeClimate queues a scan of the codebase and informs me on a level of 1-4 whether the quality of my codebase has improved or deteriorated, using metrics such as variable name length, code duplication, the number of possible paths through a block of code, and so on. Utilising this third-party service helped fight the temptation to hack a bit of functionality in, instead encouraging good engineering practice and providing suggestions as to how the codebase could be made more maintainable.

4.2 SmartResolution directory structure

As the implementation followed an agile methodology, the design evolved over time and thus, the directory structure could not be determined up front. Most (but not all) of the key directories and folders are outlined below:

```
data/
features/
test/
vendor/
webapp/
    core/
        api/
        controller/
        db/
        model/
        view/
    modules/
        other/
        config.json
    uploads/
    .htaccess
    index.php
    routes.php
```

`data` contains fixture data for tests. This is also where the test and production SQLite3 databases reside.

`features` contains the Cucumber features and Ruby step definitions.

`test` contains all PHP unit tests.

`vendor` is an automatically generated directory, created by Composer, containing all of SmartResolution's PHP dependencies.

`webapp/core` contains the core ODR platform, which uses an MVCR composite design pattern. The `model`, `view` and `controller` directories are self-explanatory and `webapp/routes.php` defines the routing component.

Also inside the core is the `db` directory, which contains middleware classes connecting the model classes to the database, since models should encapsulate the concept of whatever it is they are representing but not be responsible for the relational database to object mapping. This is discussed in detail later on in this chapter.

This folder also contains an `api` directory, which defines all of the global functions available to modules. Having these in their own directory made generating module-specific API documentation easy.

Going back up a level, we have `webapp/modules`, which contains any installed SmartResolution modules, such as the maritime collision module. A `config.json` file (generated in a user-friendly way through the admin dashboard) denotes which modules are installed and whether or not they are active.

Finally, the `.htaccess` server configuration file ensures that all HTTP requests are routed to `index.php`, which is the driver for the application.

4.3 Routing

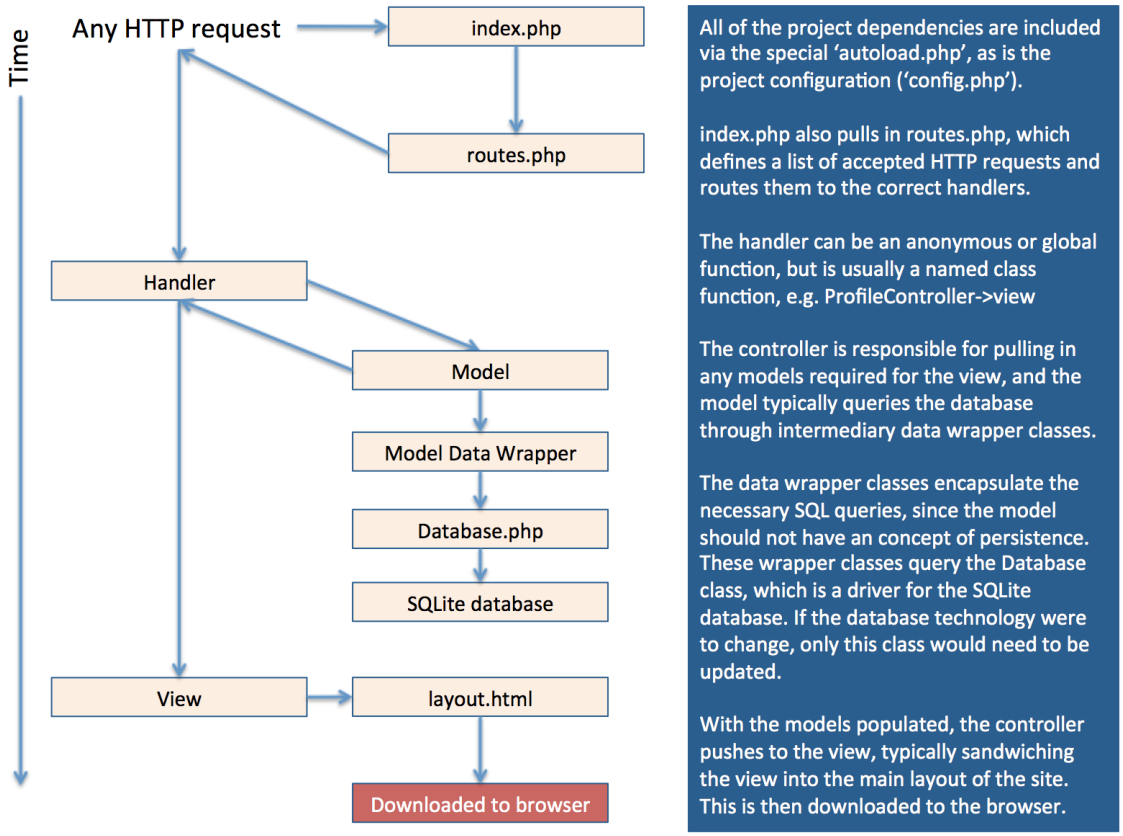

Figure 4.1: How SmartResolution processes HTTP requests

Figure 4.1 shows how SmartResolution routes HTTP requests and renders data-driven pages. As described in the image, the HTTP request is processed by `routes.php` and forwarded to the appropriate controller, which then instantiates the models and renders the view.

4.4 Class diagrams

The system was developed in an agile way, hence these class diagrams are not in the design section but the implementation section. The class diagram in figure 4.2 shows mainly the classes in the `webapp/core/model` directory, as these models encapsulate the business logic of the application. The following subsections describe the class diagram in more detail.

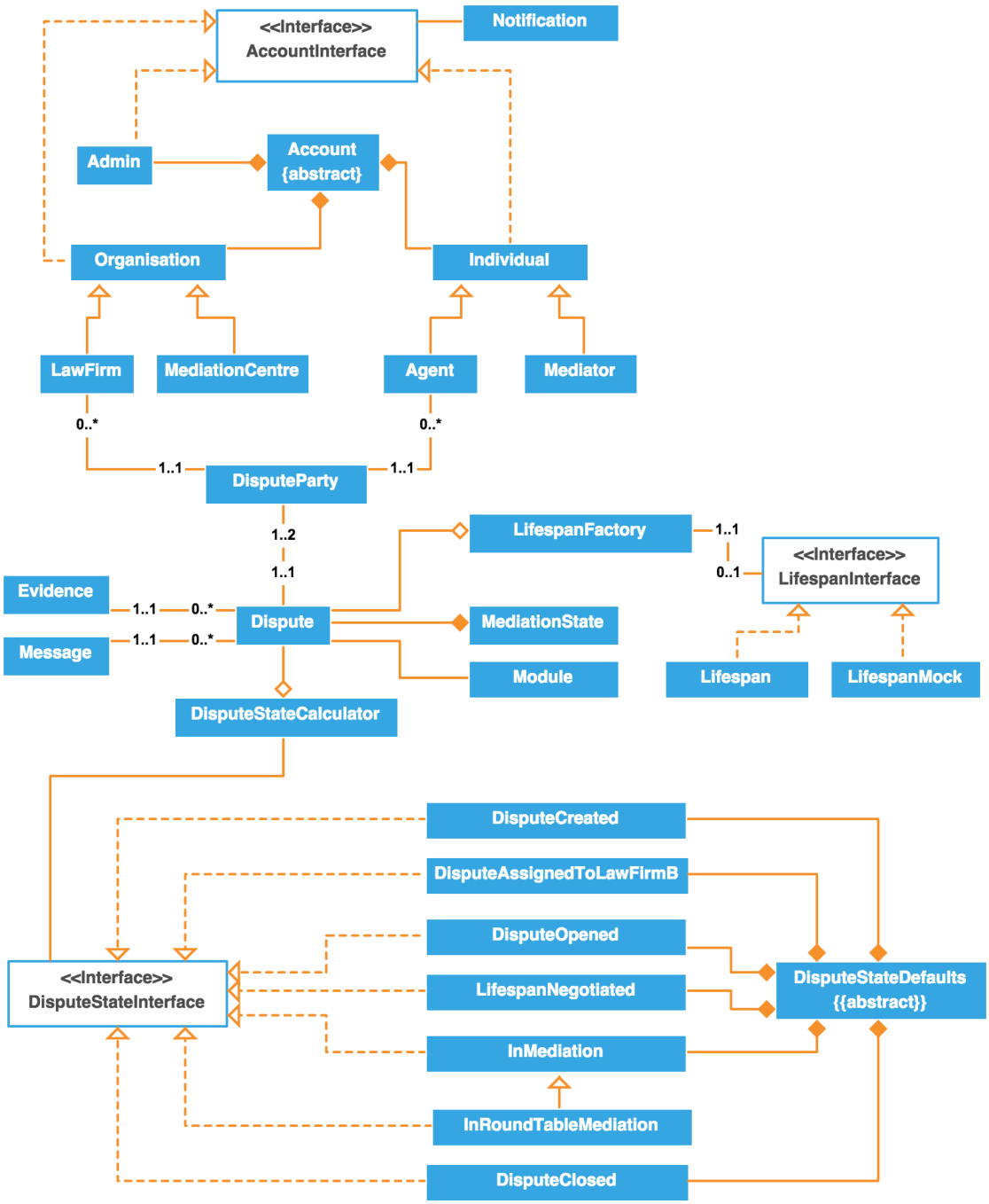

Figure 4.2: Class diagram showing the model classes in SmartResolution

4.4.1 Accounts

The `AccountInterface` defines the core methods which all account types need to implement, whether the account is a law firm, a mediator or even an administrator. These methods include `getEmail()`, `getLoginId()`, `__toString()` and so on.

Many of the methods in the interface will have the same implementation regardless of the

account type. In these cases, the abstract class `Account` defines the common implementations, but it does not implement the `AccountInterface` interface since it cannot implement all of the methods for every account type.

The `Organisation` and `Individual` account types implement the `AccountInterface` interface and extend the `Account` abstract class to further define any missing function definitions specific to their own account type. Finally, the specialised subclasses inherit from `Organisation` or `Individual` and override or call parent function definitions where necessary.

4.4.2 Disputes

Every fully instantiated dispute has two dispute parties, each of which is comprised of a law firm, an agent and a dispute summary. A dispute can be associated with infinite items of evidence (uploaded by the agents) and infinite messages sent back and forth between the agents. Every dispute should also have a lifespan: this is more complicated than it might first appear so is discussed in detail in the next subsection.

Disputes have a *state*, which roughly maps to the red box states indicated in figures 2.4 and 2.5. This is discussed in detail in subsection 4.4.4.

A dispute's type denotes whether or not a module is pulled in to expand the options available to it. By default, all disputes are of type 'Other', which adds nothing to the core functionality of the system. However, with the maritime collision module activated, agents have the option to change the dispute type to 'Maritime Collision', thereby unlocking the AI that the module offers.

Finally, disputes have a *mediation state*: that is, a dispute may or may not be in mediation, or it may be somewhere in between. For example, the agents may have decided upon a mediation centre but not yet chosen a mediator. This is all encapsulated in the `MediationState` class.

4.4.3 Lifespans

Every dispute should have a lifespan. This becomes somewhat complicated, since a lifespan can be proposed by either party, must be agreed by both parties, must only be applied between the start and end points of a lifespan, and can be renegotiated at any time.

With this in mind, every dispute can be said to have two lifespans: the 'current' and 'latest' lifespans. Figure 4.3 highlights the difference between the two.

The 'current' lifespan is the most recent accepted Lifespan that has been attributed to the given dispute. If no Lifespan has been accepted, this retrieves the most recent *offered* Lifespan. If no Lifespan has even been offered, this returns a mock Lifespan object so that all of the lifespan-related method calls still work, saving us from having to complicate our dispute object.

For most purposes, the 'current' lifespan is what is required. This is the lifespan that has generally been agreed by both parties and is used for checking if a dispute is still ongoing before allowing an agent to send a message, for example.

The 'latest' lifespan ignores whether or not a lifespan has been accepted and retrieves the very latest lifespan proposal. This is required in special cases, such as in the rendering of the lifespan on the dispute dashboard, to highlight the fact that a new lifespan has been proposed and needs to be accepted or declined.

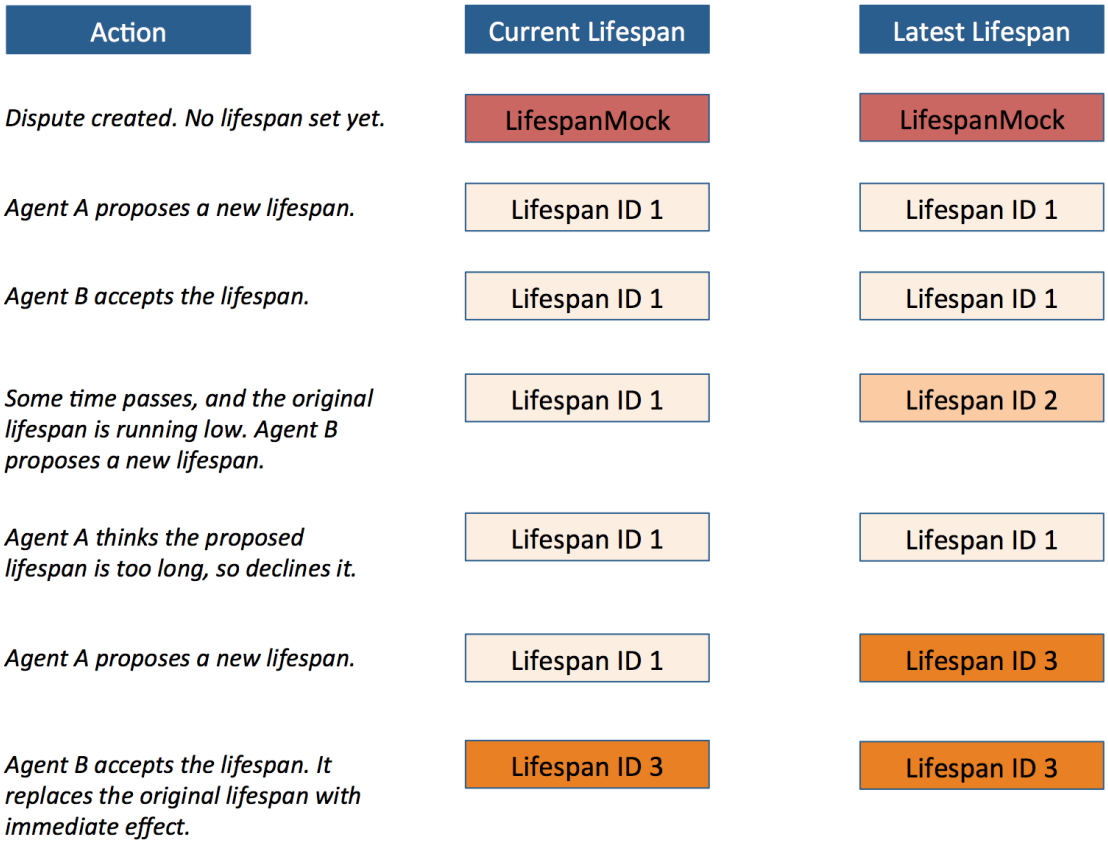

Figure 4.3: Visualisation showing the difference between the 'current' and 'latest' lifespans

4.4.4 Dispute State

Separately from the 'mediation state' is the higher level dispute state. A dispute is in a different state depending on whether or not all agents have been assigned, a lifespan has been negotiated, and so on. The state of the dispute dictates what actions are available to the parties involved.

Early versions of the project began to get quite complicated because classes throughout the system would manually determine a dispute's state when deciding whether or not they could perform some action. This led to complicated queries like the one below:

```
if ($dispute->state() === 'Open' || $dispute->state() === 'InMediation'
    || $dispute->state() === 'NegotiatingLifespan') {
    doSomething();
}
```

Figure 4.4 shows how the state pattern was adopted to overcome problems like this. Classes throughout SmartResolution could now query the dispute's state directly, as the responsibility of whether or not an action could be performed was encapsulated inside the state itself. The above example could thus be rewritten as follows:

```
if ($dispute->state()->canDoSomething()) {
    doSomething();
}
```

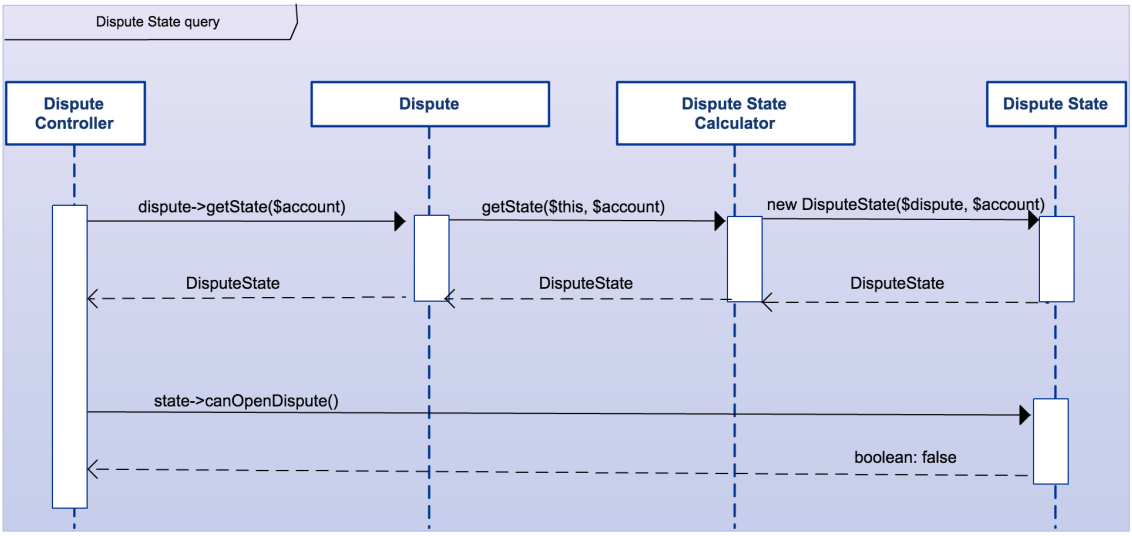

Figure 4.4: Sequence diagram showing the state pattern in SmartResolution

The list of possible states are as follows:

- **DisputeCreated** - this is the very first state of the dispute and represents a dispute that has just been created. At this stage, the first dispute party is complete (it will have a law firm and an agent associated with it), but it has not been opened against another law firm.
- **DisputeAssignedToLawFirmB** - this represents the state of the dispute when it has just been assigned to the other law firm. At this stage, one dispute party is complete, whilst the other only has the law firm. We are still waiting for the law firm to assign an agent.
- **DisputeOpened** - all law firms and agents have been assigned. Now a lifespan must be negotiated.
- **LifespanNegotiated** - the agents have managed to negotiate a lifespan and there is nothing more to do to initiate the dispute. When the start date is surpassed, the dispute is underway and the agents are free to perform all dispute-related actions. When the end date passes, the dispute is automatically closed.
- **DisputeInMediation** - the agents have decided to put the dispute into mediation and have negotiated a mediation centre and a mediator. It is important to note that not all disputes will necessarily reach this stage.
- **DisputeInRoundTableMediation** - the dispute is in mediation, but all parties are free to communicate openly. By default, a dispute in mediation disables direct communication between the two agents. The mediator can enable round-table communication to put the dispute into this state.
- **DisputeClosed** - the dispute is now closed, either because an agent closed it or because the lifespan of the dispute came to an end. It may have been closed successfully (the dispute was resolved) or unsuccessfully (the dispute had to be resolved by other means, e.g. court).

As can be seen in the class diagram in figure 4.2, most of these states inherit from the DisputeStateDefaults class, which defines the default permissions regarding dispute actions. Inheriting from this class means we don't have to specify long lists of true/false values where only one or two items may have changed between states. The exception to this rule is the `InRoundTableMediation` state, which is only very slightly different to the `InMediation` state and thus extends that class rather than the default class.

4.5 Refactoring to pure models

4.5.1 Problem with original design

The early design had all models populated with an ID that corresponded to a row of data in the database. For example, the `Message` constructor expected to be passed an ID corresponding to the database field `messages.message_id`. Inside the constructor, a function call was made to a data wrapper class which contained the business logic for retrieving the class-specific data (such as message content, author ID, etc) from the database. The returned array was then used to populate the model. This behaviour is suggested in the routing visualisation in figure 4.1.

The decision was made to implement the models like this to make life easier for the controllers. Let us examine this with some pseudo-code:

```
// inside a controller
$disputeID = getDisputeIDFromUrl();
$dispute = new Dispute($disputeID);
```

Inside the Dispute model, we had something like this:

```
function __construct($disputeID) {
    $disputeDetails = DisputeDatabaseConnector::getDisputeDetails(
        $disputeID);
    $this->title = $disputeDetails['title'];
    // and so on
}
```

This implementation was advantageous insofar as I could write simple code in the controllers that only had to worry about getting an object's ID. This meant that I could chain method calls together, passing returned IDs into subsequent constructors to quickly and easily get the objects I needed.

The disadvantage of this implementation was that the models were made impure. Though the models were somewhat decoupled from the database - all SQL queries were encapsulated in the intermediary database querying objects - they still had to know about the existence of the database, and they could not exist independently of a persistence layer.

Database querying was not restricted to the constructors; any side-effect functions[1] were also tightly coupled to the database. For example:

[1] These are functions which perform some side-effect when called. The internal state of the object changes as a direct result of calling the function.

```
// inside the Dispute model
function closeSuccessfully) {
    SomeDatabaseConnector::updateField('disputes', 'status', 'resolved'
        , $this->disputeID);
    $this->closed = true;
}
```

Again, this muddled the responsibilities of the model and made it difficult to test and to maintain. The model was tightly coupled to the underlying table representation, and functions had to know how to update individual fields related to the object.

4.5.2 Refactored solution

The decision was made to change models to take an array of data rather than a database row ID, fully decoupling them from the database. The array of data may or may not have come from a database: the models have no concept of persistence.

Our earlier pseudocode examples can now be rewritten as:

```
// inside the Dispute model
function __construct($disputeDetails) {
    $this->title = $disputeDetails['title'];
    // and so on
}

// inside the controller
$disputeID = getDisputeIDFromUrl();
$disputeDetails = DisputeDatabaseConnector::getDisputeDetails(
    $disputeID);
$dispute = new Dispute($disputeDetails);
```

After extracting the database array retrieval out of the model, I extracted the database *updating* out of the model. When moving this behaviour into the controllers, I decided to encapsulate all updates inside one update method that corresponds to the model:

```
// inside the Dispute controller
$dispute->closeSuccessfully();
DBUpdate::instance()->dispute($dispute);
```

Instead of updating an individual field value, I now pass the entire Dispute object to an update method which knows how to map the dispute object to its representation in the database. It calls the necessary methods on the dispute object (such as `$dispute->getStatus()`) to collect all of the information it requires to update the record. This behaviour is demonstrated in figure 4.5.

This large refactoring task perhaps hints at a broader problem: object-relational mapping. It could be argued that at various points in my codebase I have not approached the problem in the object-oriented way I should have, because I've had to manually map the object to be in terms of its underlying relational database representation. Perhaps, in hindsight, an object-relational database would have been more appropriate than an RDBMS, to delegate the understanding of the object-relational mapping to a third-party.

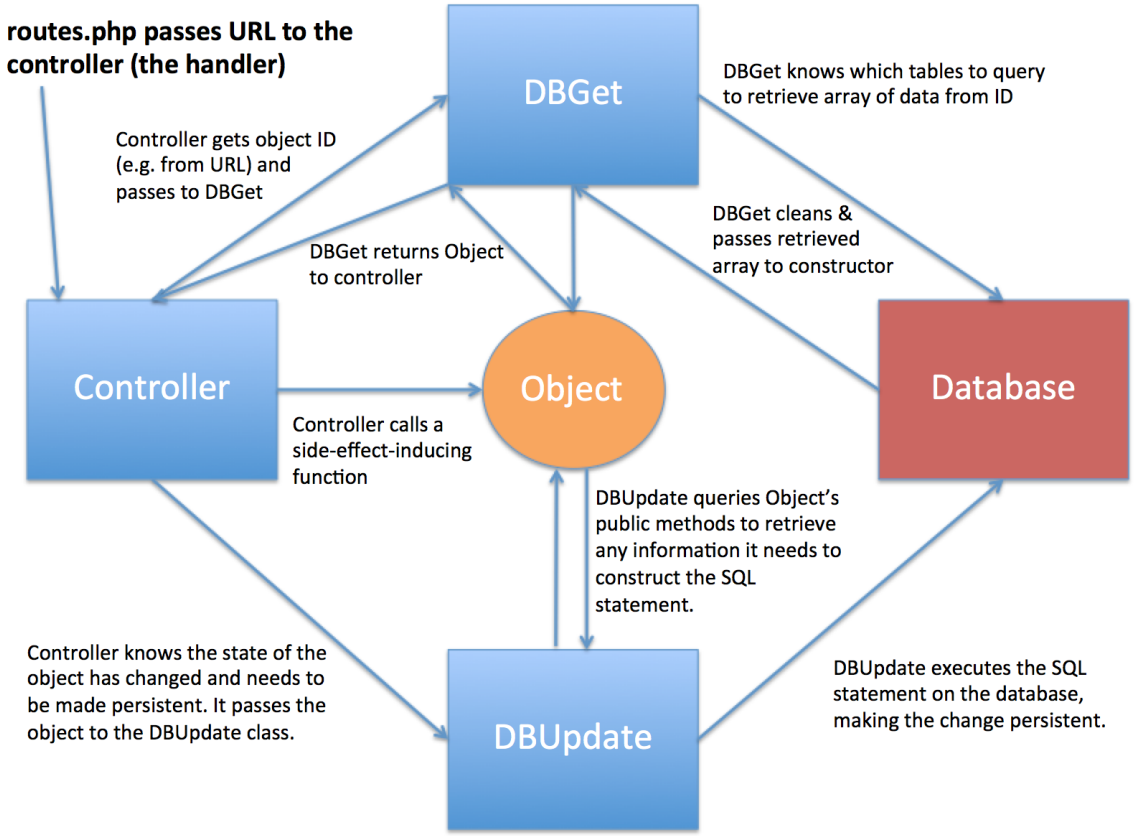

Figure 4.5: Demonstration of how changes to the model are made persistent

4.6 Controllers

Controllers can be split into two main types: handlers and database middleware. The handler in figure 4.1 is deduced by the routing file and is likely to be a controller in `core/controller`. Handlers simply implement the behaviour seen in the use case diagrams in chapter 2 and, as such, a fully comprehensive class diagram was not deemed necessary. However, a demonstration of how these controllers work can be seen in figure 4.6.

Database middleware classes live in `core/db` and handle the mapping between the models and the database. Each class fulfils a distinct database-querying need:

- **Database.php** - links with F3 and provides the database connection.
- **DBCreate.php** - responsible for creating the necessary rows in the database for a given object type.
- **DBGet.php** - retrieves array values from the database, given a row ID.
- **DBUpdate.php** - given an object, it finds the corresponding database entry and updates the values. Used for making object changes persistent.
- **DBQuery.php** - defines miscellaneous queries that would not be efficient or cohesive if encapsulated elsewhere.

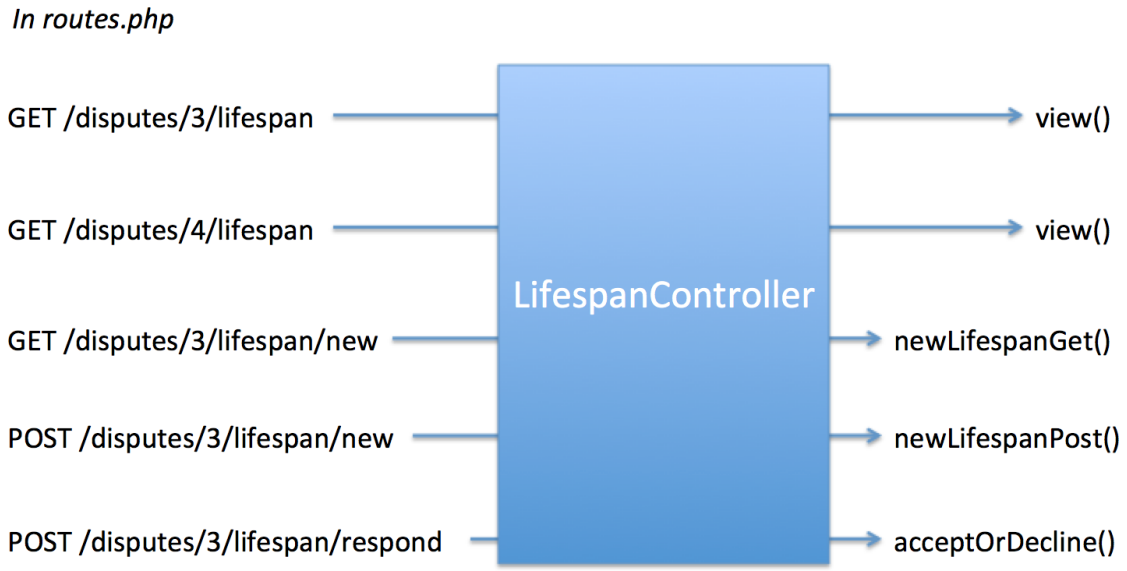

Figure 4.6: Visualisation of HTTP requests being routed to controller method

4.7 Module API implementation

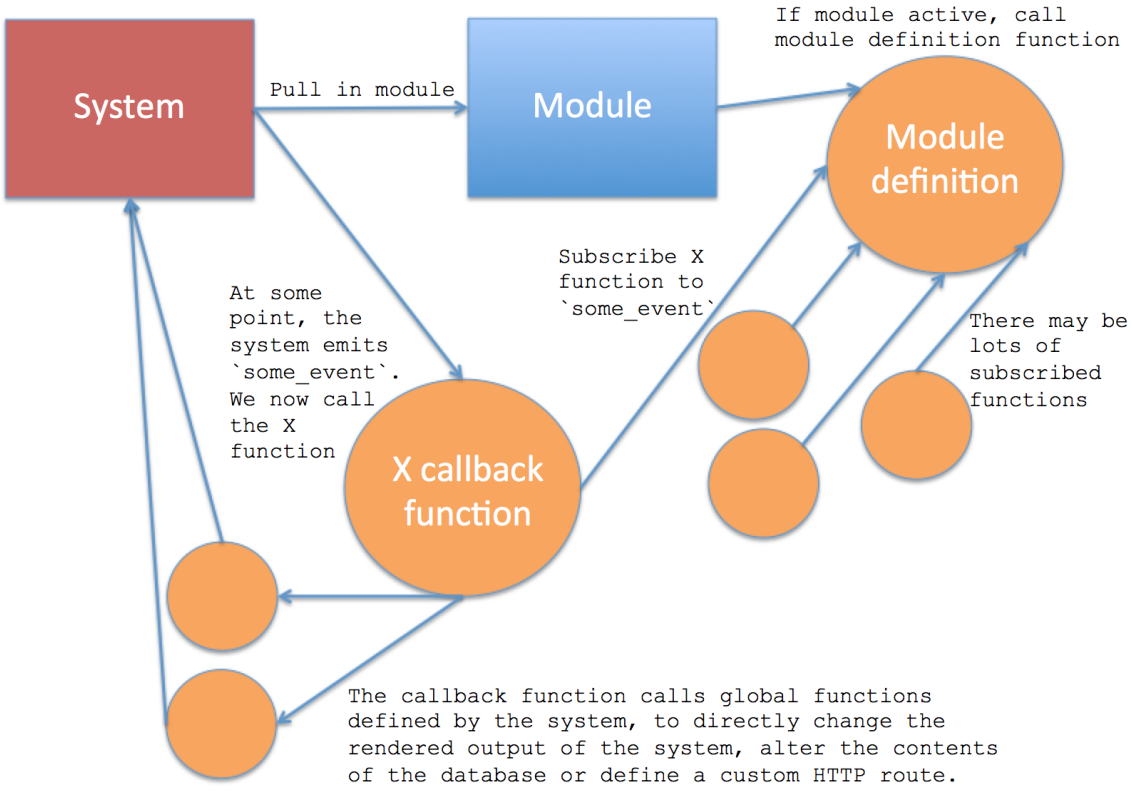

Figure 4.7: Visualisation showing how modules are registered to the system

The design for the module API was discussed in the design section. This section discusses how the module API was actually implemented, as demonstrated in figure 4.7.

The configuration JSON file is edited through the admin dashboard using a user-friendly GUI: administrators can activate or deactive modules at the press of a button. This file maintains a list of all known modules in the installation and whether or not they are active.

On every page load, `config.php` is called. This iterates through the modules described in `config.json` and loads the module's `index.php` file, which contains the module declaration call. All modules are pulled in regardless of their activity state, since the system still needs to know about them for the purposes of the admin dashboard.

Note that the module declaration function call is different to the module definition function. The former is always executed, as it declares the module to the system. It *contains* the module definition as an anonymous function which is only executed if the module is 'active'.

The module declaration function is defined in `api.php` and in turn calls the `registerModule` function in the `ModuleController`. That function instantiates a new `Module` object representing the module and then queries its activity state. If the module is active, its module definition function is called and the module is now hooked into all of the necessary events and global functions.

4.7.1 Module definition function

The module definition function contains calls to the module API. Some of these are top-level, such as defining custom routes: these are executed immediately.

The most interesting API call is the publish-subscribe function `on`, first discussed in subsection 3.5.3. This function calls the `ModuleController->subscribe` function, which adds the callback function to an internal array of subscriptions. The position at which the subscription is added to the array is denoted by the priority of the subscription.

If and when an event is emitted by a class in the SmartResolution core platform, any subscriptions to that event are retrieved. They are then iterated through and the following check is performed (demonstrated here in pseudocode):

```
if the event is dispute-agnostic:
    call the subscribed function

else if the event DOES rely on a dispute:
    if the subscribed function module name matches the type of
       the current dispute:
        call the subscribed function
```

This means that SmartResolution supports both dispute-independent hooks (i.e. "call this function when this event is emitted") and dispute-dependent hooks (i.e. "call this function when this event is emitted, but only if the dispute emitting this event is the same type as the name of this module"). An example of a dispute-dependent hook is described in the next subsection.

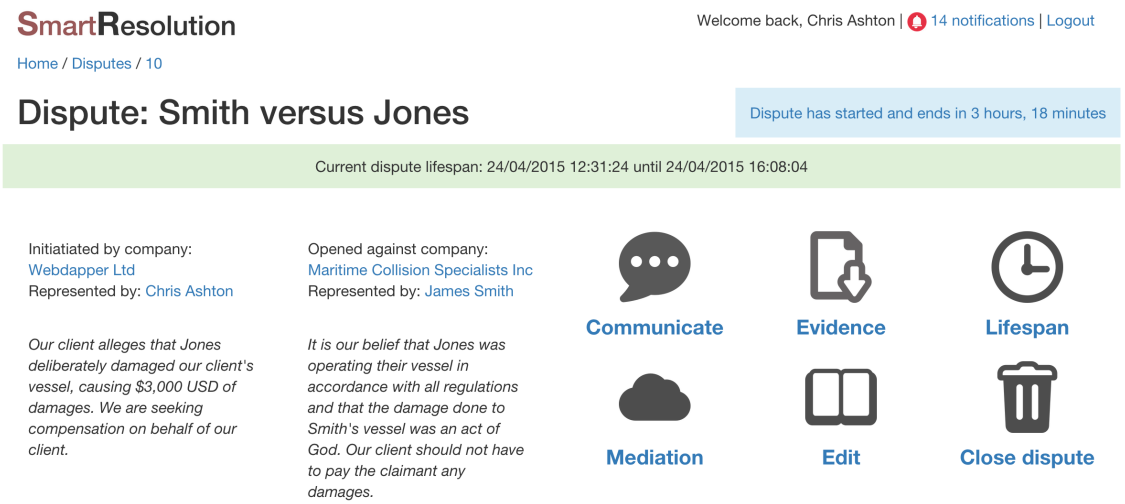

Figure 4.8: Screenshot of a dispute populated with fixture data

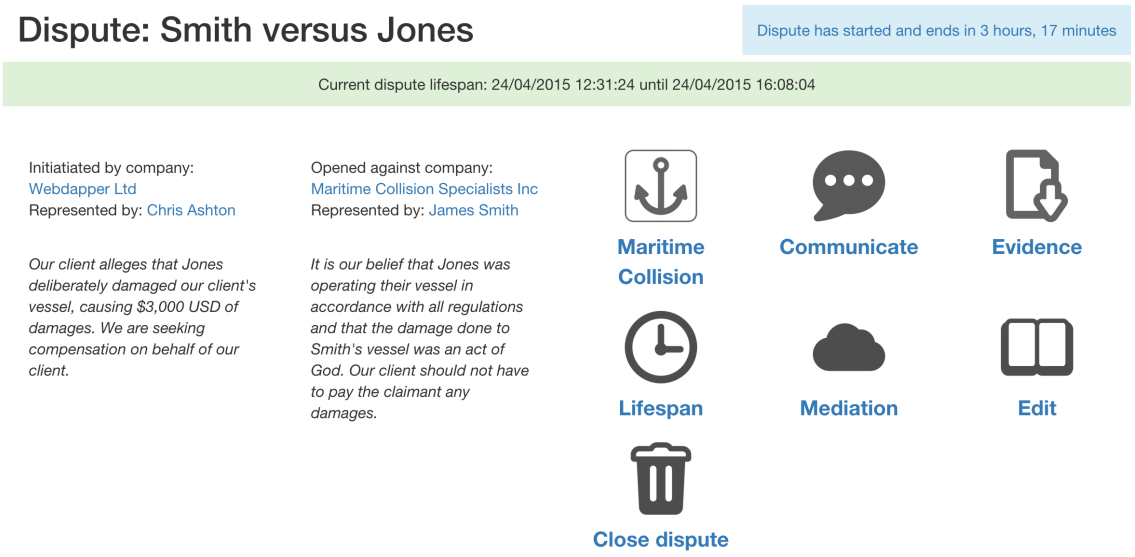

Figure 4.9: Screenshot of the same dispute changed to the ‘maritime collision’ dispute type

4.7.2 Demonstration of module implementation

Figure 4.8 is a screenshot of an example dispute. Contrast this with figure 4.9, which shows the same dispute but with the dispute type changed from ‘Other’ to ‘Maritime Collision’. By doing so, when the dispute-dependent `dispute_dashboard` event was fired, the maritime collision subscribed function was executed. Inside that function, the `dashboard_add_item` function was called, specifying the maritime collision icon, title and link.

The dashboard item links through to the maritime collision landing page which displays domain-specific questions, after both agents have accepted a disclaimer message. One of the questions pages is depicted in the screenshot in figure 4.10.

When all of the questions have been answered, the module’s AI interprets the answers to the

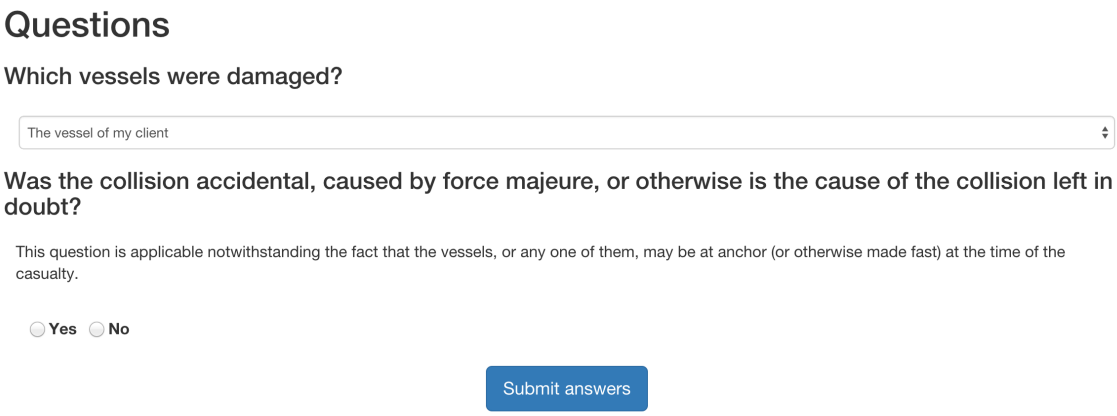

Figure 4.10: Screenshot of one of the questions screens of the maritime collision module

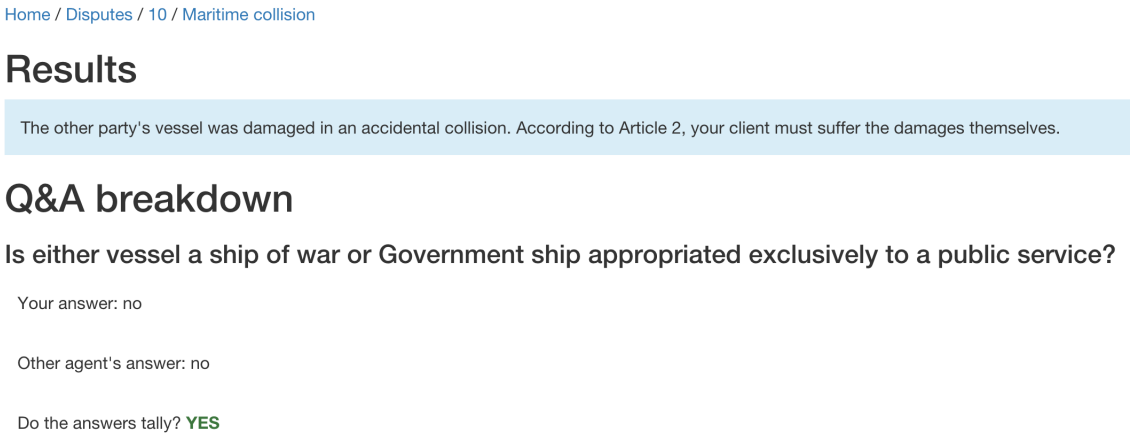

Figure 4.11: Screenshot of a possible results screen of the maritime collision module

questions according to maritime law and presents each agent with a personalised result, shown in figure 4.11. Underneath the overall result is a summary of all of the questions asked and the answers provided by both agents.

4.8 SmartResolution Marketplace

The SmartResolution website was developed locally using many of the same technologies as the core SmartResolution software, including F3, Bootstrap, Composer and so on. For the SmartResolution Marketplace functionality to be implemented, it was clear that the website would need to be deployed somewhere remotely.

4.8.1 Choosing a server

A locally configured, physical server would not be practical for hosting the SmartResolution marketplace, as it would not be able to cope with a spike in network traffic if there were large numbers

of people downloading the software or browsing the documentation. Moreover, it is an unnecessary maintenance and up front expense when so many alternatives exist.

Cloud computing as a web hosting service is becoming increasingly popular, Amazon Web Services (AWS) chief amongst these in terms of exponential growth. [7] AWS is being utilised by companies including the BBC, its popularity down to its speed and scalability, as well as the low cost to market. [33]

AWS has server farms in 11 geographical regions including the US, Ireland, Tokyo and Australia, making it possible to make optimised regional sites or even content delivery networks (CDNs). [4] It can be configured to automatically deploy additional EC2 (Amazon Elastic Compute Cloud) instances when there is a surge in website traffic, to cope with fluctuations in demand.

Other alternatives exist, of course, such as Unlimited Web Hosting, which is a cloud-based web hosting service I like to use for my personal projects. However, SmartResolution requires shell access for the installation process in order to download dependencies through Composer, set up the SQLite database and so on.

Unlike Unlimited Web Hosting, AWS provides shell access, and though other services also provide this facility (DigitalOcean, for example), it made sense to invest time in configuring an infrastructure that would be able to cope with increases in traffic should SmartResolution ever require it.

4.8.2 Configuring the server

At the point where an EC2 instance is started, a dynamic IP address is generated, making the instance available at that given IP. By default, that IP address is not persistent and every 24 hours or so the IP addresses are reallocated and the instance must be reached through a different IP.

AWS offers an 'Elastic IP' facility, which allows you to generate a permanent IP address and allocate it to an EC2 instance. This costs a little extra but is a necessary step to ensure the website is always locatable.

Finally, Amazon's Route 53 service is a scalable DNS and allows you to link a domain/subdomain name to an IP address so that browsers querying that domain name are redirected to the content at the IP address endpoint.

smartresolution.org was purchased through the domain registrar gandi.net and the nameservers for Amazon's Route 53 service were specified as the DNS. On the server itself, the EC2 instance is automatically deployed as a LAMP server running Amazon's own flavour of Linux, Apache, MySQL and PHP. [3]

The decision was made to separate the two concepts of the SmartResolution software and the SmartResolution website, so a live demo of the software should be available on a subdomain rather than the main site index. To accomplish this, a VirtualHost was specified in the Apache configuration to redirect any requests for demo.smartresolution.org to a specific demo folder containing the SmartResolution software, which could be easily updated independent of SmartResolution website updates. All of this is demonstrated in figure 4.12.

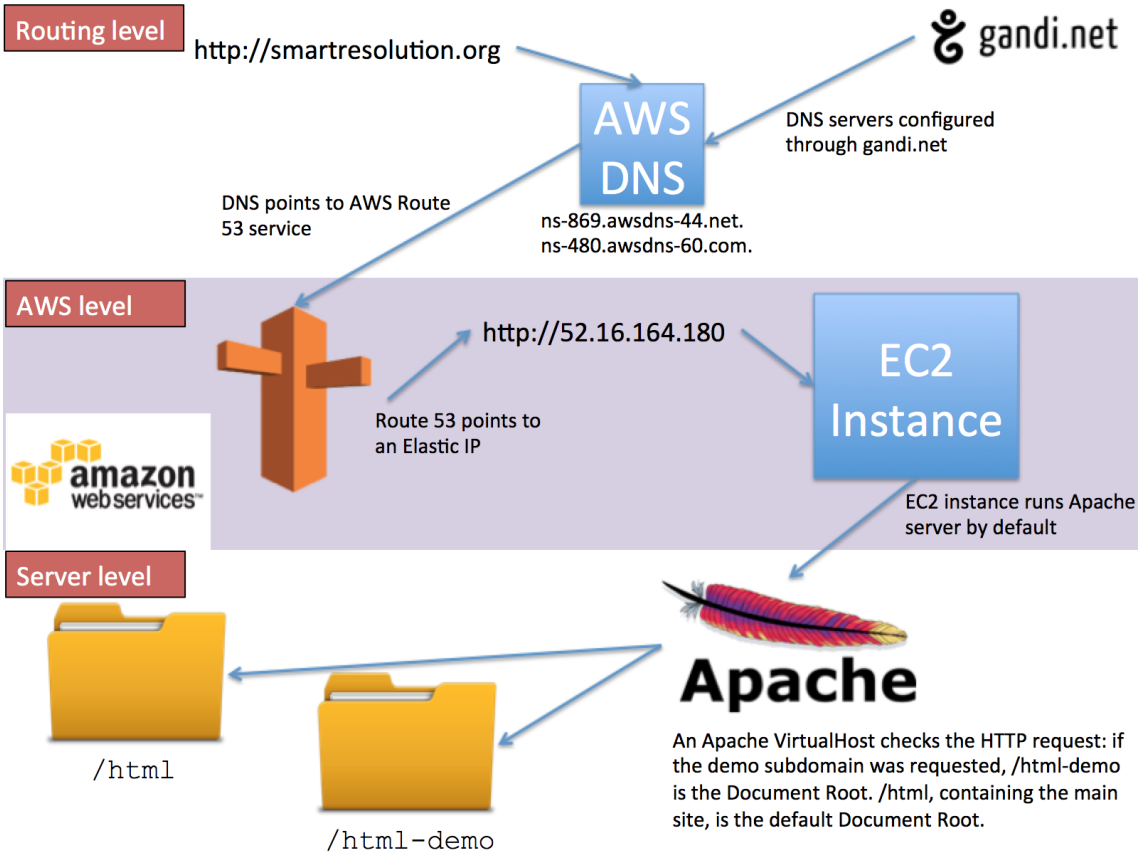

Figure 4.12: The server configuration for smartresolution.org

4.8.3 Continuous deployment

Though continuous integration is important for ensuring that the codebase remains fully functional, continuous deployment is important for ensuring that the version of SmartResolution available for demo and for download on the SmartResolution website is fully up to date. It needed to be simple to keep both elements at the latest version, and everything ought to be as automated as possible.

I wrote a bash script specific to the SmartResolution vendor site which clears the html and html-demo folders outlined in figure 4.12, then re-downloads and installs the website and the software from the website repository and the software repository respectively. The script calls PHPDoc to generate the SmartResolution core API documentation and module documentation, and the script also strips out unnecessary files such as tests and Travis configuration before zipping up the file as a production-ready download.

In an ideal world, this script would be triggered via a Git web hook so that the website, demo and download file are all updated whenever an update is pushed to either the website or software repository. However, triggering this bash script through PHP is a security issue and is made very difficult to accomplish through AWS' default Apache and PHP configuration. After some failed attempts and with time pressing on, it was decided that manually signing into the EC2 instance and running the update script was not too much of an inconvenience.

4.9 Documentation

4.9.1 Comments

I believe in the agile principles of TDD and user stories as a replacement for documentation, and further believe that any documentation that is not intrinsically linked to the code itself will always eventually become misaligned with it.

For example, SmartResolution uses Cucumber features, which is a form of 'executable' documentation. The same can be said of unit tests, though these are less descriptive. Code should be self-documenting, and anything in the code that requires an explanatory comment is a sign that the code itself should be refactored.

All this in mind, I made the difficult decision to write Javadoc-style comments throughout the SmartResolution codebase: this decision is discussed in detail in appendix G. Though I'm concerned that even these kind of comments eventually fall out of place with the code that they refer to, they do at least have the advantage of being able to be parsed to generate HTML documentation, which can be deployed to the project website. This ensures that the process of keeping API documentation up-to-date on the website is made as simple as possible, even if the applicability of the comments themselves cannot be fully verified without looking at the code.

SmartResolution API documentation, aimed at contributors to the SmartResolution platform, is available at:

`http://smartresolution.org/docs/index.html`

Developers who wish to create SmartResolution *modules* can find module-specific API documentation at the following address:

`http://smartresolution.org/module-docs/index.html`

4.9.2 Installation & how-to guides

The same philosophy cannot be applied to installation instructions or how-to guides, as there is currently no technological implementation that allows these to be generated from the code itself. To maximise the usefulness of SmartResolution and the likelihood of its continued existence, I expended quite some effort in creating high-quality, relevant additional documentation.

Instructions for installing SmartResolution on your own server can be found at the following address:

`http://smartresolution.org/installation`

A how-to guide for developing SmartResolution modules can be found at the following address:

`http://smartresolution.org/module-how-to`

Chapter 5

Testing

SmartResolution is an open-source platform being marketed to those wanting to provide ODR services, and as such it is very important that the platform itself is thoroughly tested. Subscribers need to have confidence that the core platform works as it should, and module developers need to have confidence that the underlying system is robust enough to support their module. For these reasons, the core platform implementation was done alongside unit and integration tests throughout.

In an ideal world, the same principles would have been applied to the maritime collision module and to the SmartResolution website and marketplace. However, developing the core platform in a test-driven way turned out to be quite a slow, methodical process - read more in section 6.4- and I felt that doing the same for the lesser two components was a luxury I could not afford with an impeding deadline. Therefore, this section is mostly concerned with testing the core SmartResolution software.

5.1 BDD

The project was developed in a business-driven way, as demonstrated in figure 5.1. The BDD process is as follows.

To begin with, a Cucumber feature is selected and executed. It should fail. Now a unit test is written to describe a partial implementation of the feature. The unit test is run and it should also fail. The next step is to write the simplest code possible to make the unit test pass.

With the test passing, now is the opportunity to take a step back and look at the code in the context of the codebase as a whole. Can anything be refactored? If so, apply the refactoring step, then run the unit tests again to ensure that everything still works as correctly. Repeat this refactoring step until you are satisfied with the design of the code. Continue writing unit tests using this "red, green, refactor" workflow until we have enough of the feature implemented to make a step of the feature pass. Repeat the above steps for each step of the feature until the feature as a whole is completed and passes the integration test.

Occasionally, features were too cumbersome to implement in a BDD manner, as they required spanning multiple architectures (routing, database layers, models, controllers, and so on). Wherever following BDD wasn't appropriate, I added unit and end-to-end tests post-implementation.

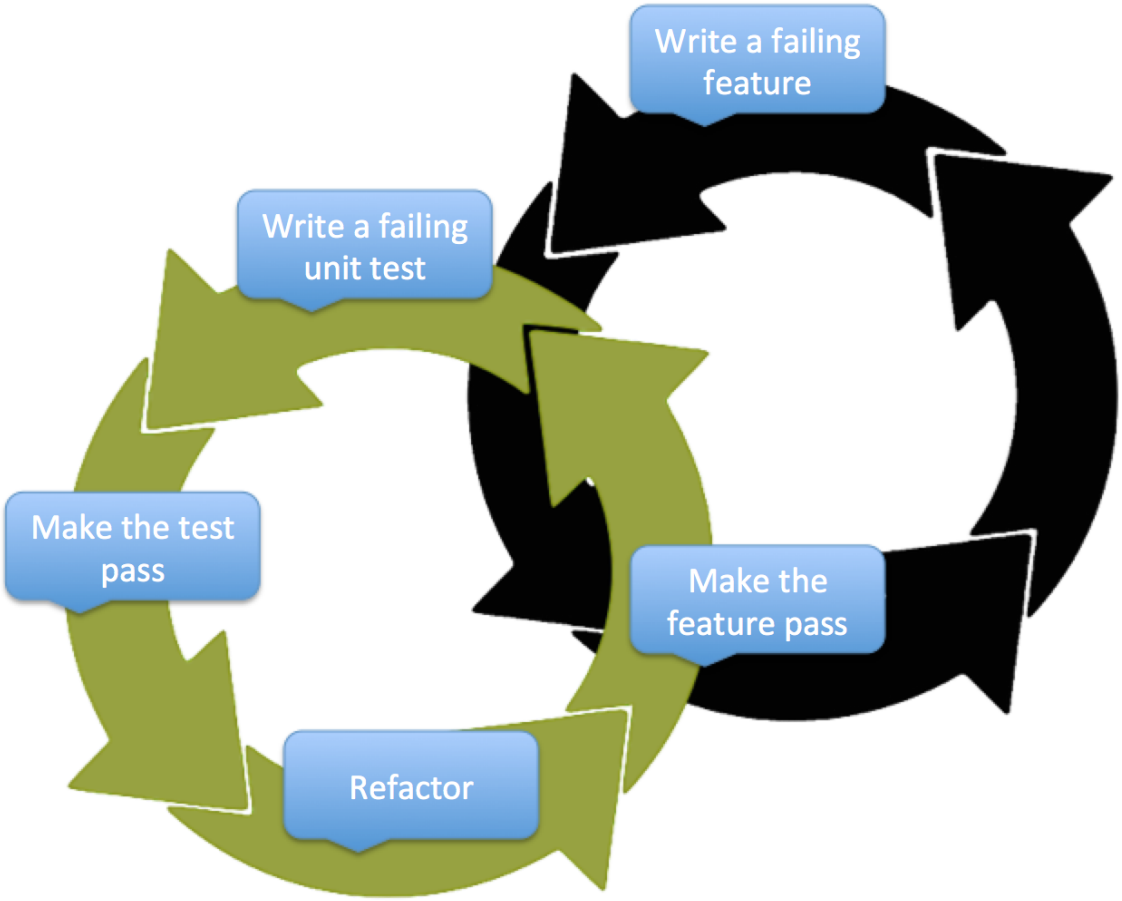

Figure 5.1: The BDD development process: an extension of TDD

5.2 Test database

Cucumber regression tests are end-to-end and use a headless browser to emulate the browser environment, allowing the driver (in our case, Poltergeist) to click buttons and fill in forms. Capybara can then query the state of the HTML and RSpec assertion methods are used to validate that an element of functionality worked as expected.

To accomplish this, the driver accesses the server like a normal browser. However, our tests rely on the database having certain fixture-data and will also be making persistent changes to the database, so it's essential that the tests use a test database, rather than the production database.

Poltergeist allows you to override the HTTP headers sent with each request, so my Cucumber tests modify the User-Agent property to be either `Poltergeist` or `Poltergeist--clear`. Both headers inform the application that it should use the test database, but the latter provides an additional instruction that the database should be cleared and re-populated with test data before processing the request. This is demonstrated in figure 5.2.

There's nothing stopping the end-user changing the headers that they send, so they too could access SmartResolution and interact with the test environment rather than the production environment. Ultimately, this is completely harmless. The test database is cleaned every time the test suite

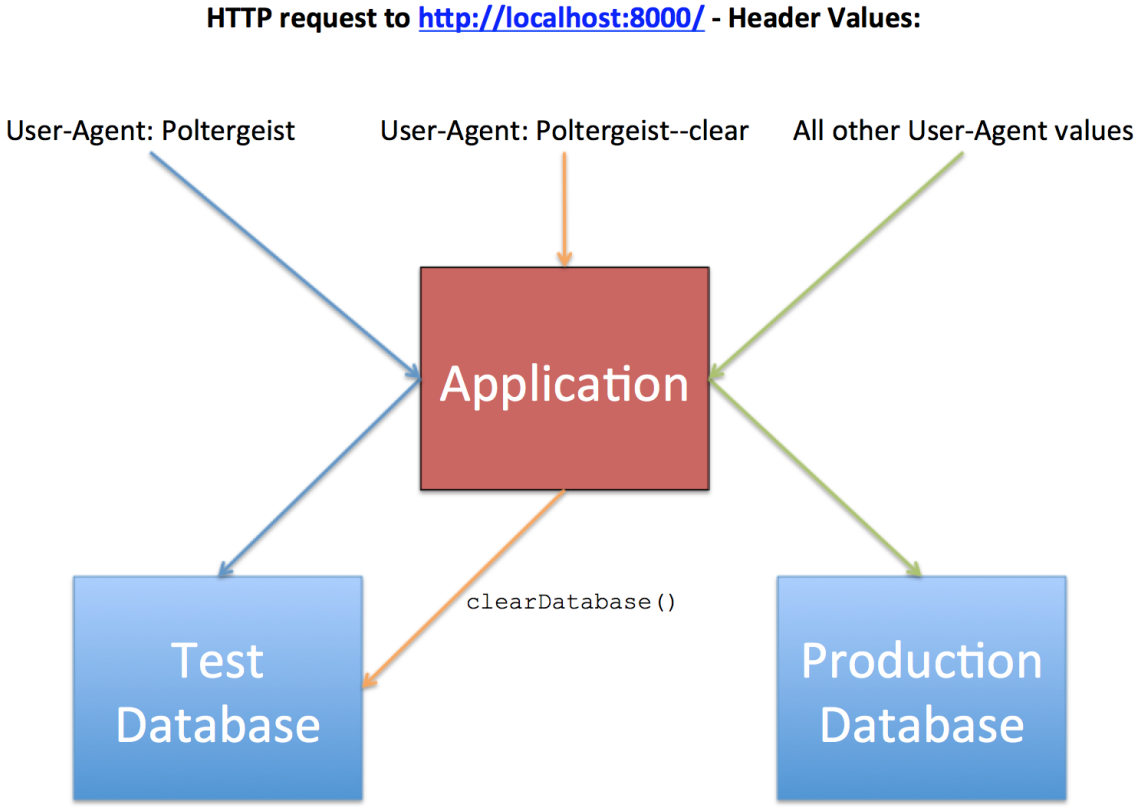

Figure 5.2: Visualisation of how SmartResolution determines which database to use depending on the received HTTP headers

is run, so they would not even be able to sabotage the tests. This is a tried and tested technique, and has been adopted by companies including the BBC. [27]

5.3 Unit Testing

Unit tests verify that the publicly available methods of a class work as expected, both when accessed correctly and incorrectly. Almost every class in the system has an associated unit test. This does not apply to library classes, such as those provided by F3, since the responsibility for testing those classes resides with the third party.

The nature of unit tests should be that they are loaded into memory and are very quick to run. "Depending on your platform, your testing tool should be able to run at least 100 unit tests per second." [31] True unit tests are also entirely self-contained. According to Michael Feathers, a "test is not a unit test if it talks to the database, it communicates across the network, it touches the file system or it can't run at the same time as any of your other unit tests." [16]

Some of the unit tests in SmartResolution are not 'true' unit tests, as they rely on a database connection and the existence of specific fixture data. Additionally, some unit tests might alter, add or remove items from the database.

Towards the end of the project I refactored my tests to cut down on the number of unit tests

that were tightly coupled to the database: this is discussed later on in this section.

5.3.1 Database transaction threads

5.3.1.1 Database transactions and PHP

Often, we want to accomplish several things in one 'transaction', for example, debiting one person's account and crediting the other person's account. If we are unable to debit the first person's account for some reason - for example, if doing so would make their account balance negative - then we don't want the other person's account to be credited. Both queries must happen, or neither must happen.

Transactions are supported in PDO (PHP Data Objects) and in F3's wrapper for PDO, where we are able to define the start and end points of a transaction. The above example could be represented in code as follows, and if either query in the transaction fails, neither would be applied:

```
$db->begin();
$acc1->debit(400);
$acc2->credit(400);
$db->commit();
```

5.3.1.2 Issues with testing database transactions

Between test suites, SmartResolution runs the 'clear database' command to revert the test database to a known constant: an untainted database filled with predefined fixture data. This reverses any changes that may have been made to the database when running the previous test suite.

This is a useful technique in ensuring that test suites don't corrupt one another's results and that each suite works with the same set of data. However, it is slow and takes around a second on a reasonably powerful machine, becoming a problem if the clearing script is called too often.

I made a conscious effort to keep these resets to a minimum, but a problem I faced early on in my unit tests was the following error message:

```
PDOException: There is already an active transaction!
```

The exception refers to a situation in database transactions whereby a transaction was initiated, perhaps some queries were queued for the transaction, and then there was another request to begin a transaction. Somewhere in SmartResolution, a transaction was not being committed or rolled back.

This exception was usually raised after testing a part of my application that should (and does) raise an exception, such as trying to assign a law firm as the agent of a dispute. The expected exception was being raised and that was preventing the change (the assignment of the law firm to the dispute) from being made persistent, but subsequent attempts to begin new transactions complained that a transaction was already in progress.

To fix this issue, I was able to run the 'clear database' command between any unit tests that raised this exception, as removing and re-creating the database would naturally clear any outstanding transactions. This was a bit of a hack, but felt justified as the test and production environments are very different. In the test environment, hundreds of transactions are initiated when running

the unit tests and all tests are hooking into the same database connection. In the production environment this would not happen: if an exception is raised, an error page is triggered and the user is faced with an error message relating to the exception. Once PHP delivers the error page to the user's browser, the database connection is no longer open and thus a new transaction can begin.

For several weeks, I was happy with this solution, but it did mean that my tests were slow to run. You can see in a Travis build around that time that it took almost a minute to run 90 unit tests[1], violating James Shore's principle that unit tests should run almost instantly.

Eventually, I decided to spend some time investigating the issue. I deduced that the transaction was not being committed because an exception was being raised before it could be committed. What I did not realise was that transactions were not being automatically rolled back when an exception was raised - it was something that needed to be handled manually in each exception.

Handling the transaction rollback in each raised exception would mean writing code like this:

```
$db->begin();

// several lines of code and database queries

if (// some condition) {
    $db->rollback();
    throw new Exception('Error!');
}

// more lines of code and database queries

if (// another condition) {
    $db->rollback();
    throw new Exception('Another error!');
}

$db->commit();
```

This seemed laborious, error-prone and messy. Instead, I defined a custom exception function which throws an exception, but also rolls back any existing transactions.

```
    public function throwException($message) {
        try {
            Database::instance()->rollback();
        } catch (Exception $PDOException) {
            // do nothing - we only wanted to roll back the transaction
                if one existed.
            // since one doesn't exist, there's nothing to roll back.
               Let's just continue and
            // throw the Exception we wanted to throw in the first
               place.
        }
        throw new Exception($message);
    }
```

[1] A build on 15th April, before refactor: https://travis-ci.org/ChrisBAshton/smartresolution/builds/58603809

Now that the root of the problem had been fixed, unit tests could be run without clearing the database in between each test. This led to significant improvements in test speed: at this stage, I could now run 112 unit tests in 34 seconds.[2]

I still needed to clear the database between each test *suite*, as the tests would still alter the state of the database. But now, by and large, most of the tests *within* those suites did not need to run on a new database connection.

5.3.2 Refactoring to pure unit tests

Earlier I described how 'pure' unit tests should not query a database. An unfortunate consequence of the implementation of my original code was that database querying was essential to unit test my classes, as my models were populated by an ID representing a row in the database, rather than populated with an array of data which may or may not have come from a database.

Back in section 4.5, I discussed the late move to refactor my codebase to use only pure models, moving the database querying out of the models and into the controllers. The advantage of the refactored solution is that the models became pure: they now had no concept of a persistence layer. We could now test our models by passing arrays of hard-coded data, removing the need to retrieve from and write to a database.

Changing object constructors to take arrays rather than database record IDs meant I could cut down test times significantly. After implementing the refactored solution, my test suite performed 109 unit tests in just 12.7 seconds.[3] The results are even more dramatic if one looks at the number of assertions made rather than just the number of unit tests performed. This information is all summarised in comparison table 5.3.2, where UTPS stands for 'Unit tests per second' and APS stands for 'Assertions per second'. It is worth noting that those are the results on Travis: on my MacBook Pro, the entire unit test suite runs in under six seconds.

Stage in project	Unit tests	Assertions	Time taken (s)	UTPS	APS
Pre-refactor	90	221	57.70	1.58	3.83
Fixed database transactions	112	263	34.24	3.29	7.68
Refactored to use only pure models	109	309	12.72	8.57	24.29

Table 5.1: Table showing the unit test times after various refactoring steps

5.4 Functional Testing

The SmartResolution core software is described as a collection of Cucumber features, associated with step definitions. These features are executable and can automatically verify that the system is working as expected. As a reminder, the full list of tested features is outlined in appendix C, though these later evolved as the project was implemented.

[2] A build on 17th April, after refactor: `https://travis-ci.org/ChrisBAshton/smartresolution/builds/58932345`

[3] A build on 22nd April, after final refactor: `https://travis-ci.org/ChrisBAshton/smartresolution/builds/59532839`

5.4.1 Functional tests structure

Every Cucumber feature has a corresponding step definition, defined in a step definition file of the same name as the feature. Each step definition defines a step in a Cucumber scenario in terms of interacting with the page, through methods provided by the Capybara acceptance test framework. Capybara requires a browser driver such as Poltergeist to request the webpage, and a browser (in our case, PhantomJS, which is bundled with Poltergeist) to render the webpage. The browser can be headless[4] or a window browser. PhantomJS, as the name might suggest, is headless.

In cases where step definitions were beginning to rely on methods defined in other step files, I extracted the methods out as a common helper class. For example, step definitions have access to a 'Session' helper class which provides methods to log in with specific credentials, or log into account types (e.g. `Session.login_as_agent`). Figure 5.3 demonstrates the relationship between a Cucumber feature, its step definition, and the helper classes.

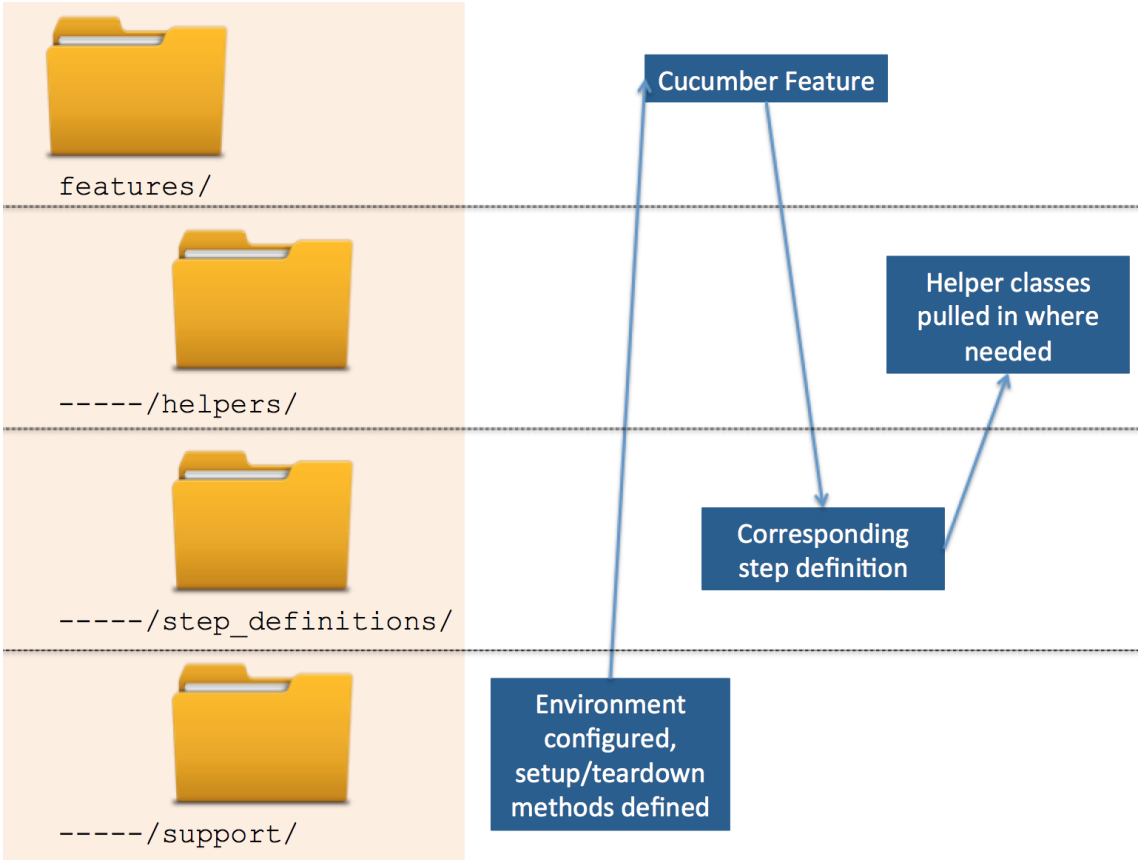

Figure 5.3: Visualisation of which files are involved in defining a Cucumber feature

Finally, there are also numerous steps which are repeated across different features, such as "Given I am logged into an Agent account". In cases such as these, it became difficult to maintain those step definitions when it could have been buried in any of the step definition files whose features depend on it, so these were extracted to a `_common.rb` step definition file. This is now the first place one should look when trying to find and maintain a step definition.

[4]Headless browsers run as a background process, hidden from view. This is in contrast to most 'normal' browsers, which open a window and have a user interface associated with them.

5.4.2 Database-clearance optimisation

As already discussed, the process of clearing the database is slow and involves removing the SQLite database entirely, creating a new SQLite database, executing the table-setup SQL (all performed via shell commands through PHP's `shell_exec` function) and then seeding the database with fixture data. This fixture data is defined in `data/fixtures/fixture_data.yml` and the mapping of YAML data to SQL tables is defined in `data/fixtures/seed.php`.

Towards the end of the project, I refactored my unit tests to cut down the number of 'clear database' instructions, but this was an optimisation I had in mind from the start when it came to my Cucumber features. Unlike unit tests, Cucumber features test the system from the user's perspective and by their very nature require interaction with the database. The database needs clearing and re-populating between some tests to prevent the actions of one test from corrupting the other, ensuring that every test has access to the same, consistent dataset.

I kept the number of 'clear database' instructions to a minimum in my Cucumber tests by annotating specific features with the `@clear` tag to indicate that the database should be cleared before executing that feature. Any features without that tag are less likely to be testing something that makes persistent changes or relies upon specific fixture data. The inclusion of the `@clear` tag tells Poltergeist to send the `Poltergeist--clear` header rather than the `Poltergeist` header, which clears the database as shown in figure 5.2.

5.4.3 A note on feature style

There is a trade-off between having a verbose Cucumber feature and a verbose step-definition. The former bogs the feature down in detail and risks devaluing it, whereas the latter can make maintenance more difficult.

I argued this same internal battle in the *Developing Internet-Based Applications* assignment, discussing the pros and cons of each in some detail. The relevant extract of that report is included here in appendix H.

My conclusion was that, though the two need to be balanced, it isn't detrimental to specify specific fields and error messages in the Cucumber feature itself. The alternative - hiding that information away in the step definition - can make it difficult to maintain state throughout the scenario. Following this conclusion, some of the Cucumber features decided upon in the initial requirements were later reworded to fit in with this philosophy, though their purpose and relevance remained the same.

For example, this was an original 'lifespan negotiation' scenario:

```
Scenario: Accepting a Dispute lifespan offer
  Given the other Agent has sent me a Dispute lifespan offer
  Then I should be able to accept the offer
  And the Dispute should start
```

This is how the scenario looks now that the step definitions have been implemented:

```
Scenario: Accepting a Dispute lifespan offer
  Given the other Agent has sent me a Dispute lifespan offer
  Then I should be able to Accept the offer
  And I should see the message 'Dispute starts in'
```

The original and amended scenarios are very similar, but the latter is a little more tied to the implementation. The 'I should see the message' step definition is reused across many scenarios and means that the step definitions as a whole are simpler and cleaner, even if the Cucumber features themselves are slightly more verbose than before.

5.5 User Testing

Whereas it is the software engineer's responsibility to ensure that the project is built right, it is the customer's responsibility to ensure that the right project is built. Therefore, user testing with the customer is critical.

As has already been alluded to in the choice of project development methodology in appendix A, the customer for this project was extremely busy and was unable to meet more than a couple of times. This made user-testing very difficult, but I tried my best to offer alternative options.

It was good that I spent so long clarifying requirements at the beginning of the project. In the absence of regular communication and feedback, this original set of features was invaluable in ensuring that the system contained all of the required functionality.

Mid-way through the project, I deployed SmartResolution to smartresolution.org and invited all stakeholders to try out a beta version of the project, describing how to log into the system, what functionality had already been implemented, and what functionality had yet to be implemented.

Throughout development I adhered to the agile principle of regular releases, pushing the latest stable version of SmartResolution to the website on a daily basis so that the customer would be able to feel the benefit and experience a more feature-complete demo.

Later on in the project I made it possible to demo the software without physically trying it, by producing a short video demonstrating the software and the maritime collision module. This is embedded on the homepage of smartresolution.org and enables users to see what the system is capable of without having to manually log in and out of demo accounts representing different user roles.

Where feedback was lacking from the customer, I turned to friends, family and the Twitter community to try out the demo and give me constructive comments to work on. These comments were fed back directly into my design, leading to the following improvements:

- Smaller icons. My early designs used dashboard icons around 3 times larger than the final design. I was trying to create a clear and minimalist dashboard but my implementation was something consistently questioned in the feedback I received.
- Larger font size. Bootstrap's default font was too small given the sparseness of the design as a whole. I increased the base font size; this proportionally increased all font sizes across SmartResolution as my CSS defined fonts in terms of `em`.
- Responsive improvements. It was never a requirement to make the SmartResolution software fully responsive: the fact that Bootstrap supports responsiveness by default was just an added bonus. However, some of my design decisions - such as absolutely positioning the lifespan status at the right hand side of the dispute - made for a poor experience on mobile.

> When this was pointed out in the feedback, I added a media query so that the positioning would only apply on devices of a minimum screen width.

Feedback regarding the workflow of a dispute had to be treated differently to UI feedback as the customer was the law expert and the people giving feedback were not law experts. However, their comments were collected and fed back to the law expert, helping to negotiate simpler dispute workflows.

One of the original requirements was for a formal resolution-offering process. An agent would be able to make an offer by filling in a HTML form denoting offer details, damages awarded, and so on. I was able to simplify this in light of the feedback: agents are human beings and can reach that kind of resolution through the communication option alone. They can then close the dispute successfully, with no need to officially form a structured offer that SmartResolution understands. Removing these artificial constraints made for a more user-friendly workflow and reduced the development overheads significantly.

Chapter 6

Evaluation

6.1 Were the requirements met?

As a reminder, the original requirements were identified as the following, to be tackled iteratively:

1. Find or build an Online Dispute Resolution platform.
2. Tailor the platform towards maritime collision disputes.
3. Make the platform abstract, able to take a module of business logic.
4. The maritime collision module should be able to retrieve the most similar historical cases.
5. The details of these historical cases should be fed back into the details of the current dispute, thereby influencing the court simulation.

The first three of these were the core requirements, but I distinctly remember that the fourth and fifth requirements were my own idea. The thought of making use of historical maritime collision cases excited me at the time and felt like a logical progression for the module. Both my supervisor and the customer were happy to agree to make this a part of the requirements specification.

Over the course of the project, I was able to look beyond the specialised area of maritime law. The light bulb moment came when I considered the commercial viability of the project (see appendix D), particularly the comparison with WordPress' commercial model. It was then that I saw the big picture and realised that the success of the platform would depend entirely on its extensibility, its ease of use, its ease of development and its branding.

Though the first three requirements remained, the fourth and fifth requirements took a back seat whilst I developed the platform surrounding SmartResolution. It could be argued that the first three of the original requirements were correctly identified and delivered, whereas the remaining two were surpassed by more pressing and relevant requirements. These were also subsequently delivered.

I wanted to create something that could be physically used in the real world at the end of my project and tried to see the big picture at all times. The big picture is a robust and fully extensible core ODR platform supporting an infinite number of modules that can be developed to fulfil an infinite number of uses.

It was important that the core platform offered enough hooks and an expansive-enough API to allow for the requirements we haven't even considered yet. It was equally important that there should be a straightforward way to install said modules into one's installation of the SmartResolution software. At this stage I created an embedded marketplace and installation wizard, taking inspiration from WordPress' admin panel.

If I had foregone the SmartResolution marketplace component entirely, I could have invested around a fortnight of extra development time in the maritime collision module, and perhaps accomplished something more exciting and groundbreaking in the AI. However, without a centralised means of browsing and installing modules, and indeed a centralised means of downloading the core platform itself, I feared for the future of the project. I like to think that, with the platform built and the documentation plentiful, developers are empowered to develop their own modules for this sector which has recently been gaining in popularity and public awareness.

In essence, the minimum requirements were a working ODR platform and a maritime collision module prototype. Anything beyond that in terms of delivery was my own prerogative, whether that was a SmartResolution Marketplace or a maritime collision module that is influenced by historical cases. In that respect, the core software and the basic maritime collision module satisfy the original needs of the customer.

To summarise, the software deliverables evolved from an extensible ODR platform and a sophisticated maritime collision module, into the following:

- Core ODR Platform
- Simple maritime collision module
- SmartResolution website and comprehensive documentation
- SmartResolution live demo and automated deployment
- SmartResolution 'marketplace', allowing the perusal and downloading of all available modules

6.2 Comparison with Modria

At the beginning of this report, I identified Modria as the market leader for online dispute resolutions. I'd like to take this opportunity now to highlight the differences between the two and the pros and cons of each.

Modria is well suited to low-value e-commerce disputes. It provides a hosted platform which online retailers can become subscribers to, after which they can sign into their hosted area and view the disputes against their organisation. No developer knowledge is required and setup is minimal.

Subscribers are able to set resolution rules to automate the results of disputes as much as possible, thereby cutting costs and freeing up staff for other duties. In that respect, Modria has encoded some artificial intelligence into its system, which the subscriber is able to configure to their own needs and desires.

What Modria cannot offer at this stage is a platform that is capable of concepts beyond e-commerce disputes and terminology. Disputes come in all shapes and sizes, and automated resolution suggestions for all of these can only be possible with a modular architecture supporting domain-specific code.

SmartResolution provides this platform. Unlike Modria, it is not hosted: it is open-source and requires some basic developer knowledge to install and configure, though it does come with detailed installation instructions. That being said, it is perfectly feasible that SmartResolution could offer hosted solutions, perhaps on a commercial site. WordPress does this itself: wordpress.org is the platform site and wordpress.com offers both free and premium hosted blogging solutions.

The important thing is not that SmartResolution is open-source, but that it is extensible. SmartResolution supports arbitrary modules of business logic, allowing developers to define custom dispute types and heuristically-driven resolutions.

The platform comes tightly integrated with the SmartResolution Marketplace, giving developers a public and legitimate means for distributing their modules. A large enough collection of quality modules - and a platform that supports the easy delivery of said modules - can be a catalyst for rapid growth, as demonstrated with Google's Android software, Apple's App Store, or indeed, WordPress' Plugin Directory.

To summarise, I think that simple disputes between two people and almost all disputes related to e-commerce are well suited to Modria's hosted ODR platform. Specialised disputes and any disputes that require representation through lawyers are best suited to the SmartResolution ODR platform.

6.3 Time management

To recap, the Gantt chart in figure 2.6 shows both the intended and the actual progress of the project, in blue and orange respectively. As you can see, though both project dimensions start off roughly the same, they become markedly different from mid-March onwards.

The most striking difference is probably the changing of the self-imposed deadline. Whereas the original plan had me working right up until the beginning of May, I realised later on in the project that the report would be a substantial effort of work and would also require time to print and bind, so aimed to be code-complete by the middle of April. This significantly reduced the number of weeks I'd planned to use for the development of the project.

Another noticeable difference is the amount of time spent on the core platform. I had intended to complete the core platform in just one month, but it ended up taking about 50% longer than expected. I put this down to being overly ambitious with my expectations, not being able to find an open-source base to build upon, and disciplining myself to develop in a strictly TDD fashion. This is discussed in detail in the next section.

The very deliverables of the project also changed as time went on: there was a lower emphasis on the maritime collision module itself and more emphasis on the platform surrounding and supporting the module.

Finally, in the final stages of development, many of the tasks appear to overlap: this reflects the coupling between the various components of the project. You cannot create a maritime collision module without extending the underlying platform, but you don't necessarily know what

is required of the underlying platform until you start developing the maritime collision module. You cannot create a SmartResolution Marketplace without having a finished module to host in that marketplace. You cannot create a maritime collision module without at least some understanding of maritime law. And so on.

The fact that the two Gantt dimensions are so different raises an important question: was a Gantt chart ever going to be compatible with an agile implementation, even if the first half of the project was plan-driven? In hindsight, perhaps it would have helped my planning if I had only used the Gantt chart to plan up until the early design stage, switching to agile sprints thereafter.

6.4 Speed of progression

The design and development of the ODR platform was a major project in itself. As a result, I only had a few short weeks to concentrate my efforts on the maritime collision module, so treated this module more as a prototype than a finished product. In the words of Eric Raymond, what I've created is a "plausible promise" of a maritime collision module. [30]

I had hoped to complete the core ODR platform more quickly than I actually managed it. In general, I did find that progress was slower than anticipated, and I'd like to briefly examine the reasons why, especially as I actually managed to negotiate some simpler requirements (removing formal resolution "offers", etc) to help reach my deadlines.

It is partly down to my busy schedule: company work and administration, travel, visiting family and friends. My self-imposed deadline was always going to be an ambitious milestone to adhere to. However, going beyond just external commitments, I believe that coding in a test-driven way has slowed me down.

Having to write integration and unit tests, being informed by Travis several minutes later that I've broken the build, having to go back and fix tests, having to refactor my unit tests when I refactor my codebase - these have all factored into a more drawn-out development process. I think I probably would have finished the core platform sooner had I not disciplined myself to write tests throughout.

Of course, if I hadn't developed in a test-driven way, there's no way of knowing how much time I would have spent manually testing or fixing complicated bugs that start at one point and proliferate throughout the system. It is very possible that my development progress might have fallen further behind or even ground to a halt. Regardless of time, I'm extremely satisfied to have this collection of tests that cover every aspect of my codebase, as they allow me to refactor hundreds of lines of code and automatically validate that everything still works. The development overhead on writing tests is easily worth it for the ability to refactor without anxiety.

TDD aside, "requirements creep" set in over the course of the project with stakeholders clarifying new requirements such as a file upload facility, the ability to view user profiles, and so on. These were in addition to my own self-imposed requirements, such as automated AWS deployment. I'd estimate that each new requirement adds at least two days to the development phase: one day to build and test, and the equivalent of a day in ongoing maintenance when refactoring the code.

Overall, I think I was just too ambitious in aiming to complete the core platform, in a fully test-driven way, to the Gantt schedule that I created. My only regret in terms of time management is

that I spent too long preparing for the project, spending around 2-3 weeks clarifying requirements which have since evolved naturally anyway, as well as reading around the subject of maritime law and gathering historical cases: these ended up playing much less of a role in my project than the dissertation title might suggest.

6.5 Appropriateness of the design

6.5.1 Choice of language and framework

I believe that the choice of PHP as the implementation language was correct. SmartResolution is easily deployable as almost all servers support PHP; server support for Ruby, Node and other languages is less common.

In hindsight, the choice of F3 as a framework was an interesting one. To recap what I said in the design section: "F3 is fundamentally different to its competitors because I could slot F3 into my code, rather than slotting my code into F3." I emphasised the need to be agile and the disadvantage of being locked into specific directory structures in large-scale frameworks such as Symphony or Zend.

Now that the project is complete, its directory structure has actually become quite complex, encompassing a rich and deep MVC separation with additional directories for database-querying middleware classes, classes that handle the module API, classes representing dispute states, and so on. F3 copes well with this, but perhaps a more heavyweight framework would have enforced additional advantages, such as namespaced classes or handling autoloading[1].

Although SmartResolution has developed into a project that would have benefited from the features provided by a larger framework, F3 definitely had a lower learning curve. This meant that I was able to start implementing features the day I started developing, rather than spending days or weeks getting to grips with a heavyweight framework. I don't regret my choice, especially since frameworks tend to be moving towards a modular style anyway, as discussed in subsection 3.1.4.

6.5.2 Appropriateness of implementation

Figure 4.2 contains the class diagram showing the object-oriented nature of the system and the interaction between the models. I'm quite happy with the final design, but certain elements could still be improved.

For instance, the `MediationState` class encapsulates a lot of business logic, representing the mediation state of a dispute right from mediation proposal, to choosing a mediation centre, to the mediation centre offering a list of available mediators, etc. The state pattern worked well for the dispute itself, and should maybe have been extended to incorporate this mediation logic too as it would have made state querying more consistent.

I believe that, in hindsight, more time spent on an up front design would have benefited the final design. It may also have meant I'd have side-stepped some of the difficult periods of the project,

[1] Composer generates `vendor/autoload.php`, but `webapp/autoload.php` is manually created and must explicitly pull in various files and directories Frameworks such as Zend provide ways of hooking into the autoloader.

such as when I refactored all of the models to take an array of data instead of a database record ID in the constructor. I may have spotted the database/model coupling sooner and implemented the system correctly in the first place.

At the higher level, I'm very pleased with the MVCR structure of the application. This pattern made it easy to assign routes to different handlers and to separate the concerns of the model with the wider concerns of the controllers and the data-querying in the application itself.

6.6 Future improvements

I've delivered three major, fully-functional components totalling around 10,000 lines of code, but there is so much more I'd have liked to have added if time allowed for it. These improvements are separated by component below.

6.6.1 SmartResolution

The admin account must be created manually in the database. In future, it would be nice if this were a part of a user-friendly installation process complete with a nice GUI. This process would prompt the user for information such as admin email and password and which kind of database the installation should use (such as SQLite or MySQL).

Assuming an admin account has been created, when the admin logs into their account they're presented with three options: Marketplace, Modules and Customise. The latter option is a placeholder and contains no functionality. It was hoped that this might later allow the admin to customise their installation by changing the SmartResolution logo or enabling/disabling mediation. I would also like to have added the ability to switch themes, as WordPress allows bloggers to easily change the look and feel of their site by activating a new theme. This is something that is perfectly possible in SmartResolution, thanks to its MVC architecture.

6.6.2 Maritime collision module

The original requirements suggested that the module could pull in similar historical cases, firstly as a useful reference for the lawyers but secondly as a heuristic that directly affects the module's suggested resolution. Naturally, this is something I'd have liked to have added if the time was available.

The research discussed in subsection 1.2.3 suggested that maritime law is different depending on the geographical location of the collision. In future, my maritime collision module could be expanded to apply different laws depending on the location of the collision, or else multiple different modules could be created and the choice of which module to use made the responsibility of the agents.

6.6.3 SmartResolution Marketplace

Currently, there is nothing clever regarding putting modules on the SmartResolution Marketplace: the process of zipping up modules, deploying them to the marketplace and editing the marketplace

JSON feed is manual.

This is something I would want to automate in the future, perhaps providing an upload interface which takes a GitHub repository URL and zips up the contents automatically. However, the manual process currently in place does at least give me the advantage of being able to check the code and make sure that there is nothing insidious about its contents before making it available on the marketplace.

A more exciting feature I would have liked to have added is inspired by Modria: hosted ODR. It is perfectly feasible to have a corporate ODR service hosted on smartresolution.org where corporations pay a subscription for their SmartResolution installation to be accessible at their own subdomain of the SmartResolution website.

6.7 Relevance to degree scheme

My degree scheme is *Software Engineering* and I believe I've developed this project to a high standard in true software engineering style, using the following best-practice guidelines:

- Almost fully object-oriented, but follows a functional paradigm where appropriate.
- Uses PDO for database interaction, to protect against SQL injection attacks.
- Uses F3 as a framework, delegating the heavy lifting to third-party modules which handle business logic such as password encryption and HTTP routing. Software reuse is a key part of software engineering.
- Uses Bootstrap at the front-end, to minimise the wastage involved in duplicating well-established UI design and to enable mobile responsiveness by default.
- Clean, semantic HTML markup, W3C-validated and friendly to screen readers.
- Appropriate use of numerous well-established design patterns, including the MVCR composite design pattern for separation of concerns, state pattern for encapsulation of business logic and publish-subscribe pattern for decoupling the core platform with any installed modules.
- Javadoc-style documented comments throughout codebase.

At a higher level, I've utilised industry-standard tools and best practices to ensure my code stays at the highest quality:

- The entire SmartResolution core platform is covered by low-level unit tests and high-level Cucumber tests, testing all aspects of functionality. If any functionality breaks as the result of an ill-thought-out refactor, at least one test should fail.
- Dependency management through Composer and RubyGems, both monitored remotely through the Gemnasium dependency monitoring service.

- Travis: a continuous integration platform configured to automatically run all of my project's tests whenever a commit is pushed to the repository. This way, even if I forget to run the test suite, the tests will still be run and I'll be warned. The build status is also automatically pulled into smartresolution.org and the SmartResolution repository README, so anybody can see at a glance whether or not the latest version is stable.
- CodeClimate: an automated code review service which analyses the quality, style and security of every line of code, giving a helpful second opinion. The suggestions from this service were directly fed back into the design of my code.

At the highest level, I've used services to keep on top of my project management:

- GitHub - a version management system allowing me to develop branches and then merge into the master branch following a line-by-line code-review. GitHub also provides a place to log issues including bugs, possible enhancements, and so on. I used these in conjunction with GitHub's 'milestone' feature, which allows me to group issues into related milestones.
- JIRA - I used my own installation of JIRA to plan out the project management aspects of the major project, including planning for the mid-project demonstration.
- Gantt chart - an industry-standard technique for planning and monitoring progress.
- PHPDoc - a tool to generate HTML documentation from my DocBlock comments, which has been a useful reference when refactoring.

6.8 Summary

SmartResolution has grown into quite a substantial platform which has scaled well to the challenges of a large codebase by being engineered in a disciplined and sustainable way.

I believe that the platform developed in this project fulfils a real need and has benefited from taking inspiration from WordPress for its API design and its marketing strategy. Though online dispute resolution doesn't have the wider appeal that blogging might have, SmartResolution could easily form the basis of a successful ODR provider's platform in future and become something of a household name.

More than just offering a feature-complete base to build upon, SmartResolution supports modular extensibility for any features that it does *not* have. The core software would probably require further modification to support the events and global functions that were not required by the maritime collision module - it is impossible to know in advance what kind of API might be required by the developers of future modules - but the core infrastructure is now in place.

The maritime collision module implementation is simple but well designed. It demonstrates what SmartResolution's modular build makes possible, providing a plausible promise of what the module could be if it were given a little more attention. To implement everything I originally wanted to implement would be another major project in itself and is something worth considering for one of next year's students.

Finally, I feel that the development of the SmartResolution website and the accompanying developer documentation has solidified a collection of fragmented concepts in a tangible way.

Building the platform website and encouraging developer activity through providing a marketplace has steered this ODR platform in a direction where it may otherwise have been lost at sea, if you can pardon the pun.

10,000 lines of code, 20,000 words and 400 hours of effort later, and the project is ready to be deployed to production, used by people, and further enhanced according to their feedback. I was recently able to demo the final version of the software with the customer and they were very pleased. Dr. Constantina Sampani will now be presenting what we've built at the Maritime Arbitrators' conference in Hong Kong in May 2015.

There has already been some discussion regarding taking this project further, perhaps as a major project for one of next year's final-year students to make the maritime collision module more sophisticated. It will be really interesting to see what happens to SmartResolution in this field which is rapidly gaining significance and acceptance in the eyes of the law. I hope I've created a solid foundation to build upon.

Appendices

Appendix A

Choosing a Project Development Methodology

The industry is moving towards an agile approach which stresses the importance of being able to embrace change. Agile is about "deferring design decisions until the last possible moment so that they can be made in light of experience" gained through spike work, talking to the on-site customer, and so on.

The idea of agile is that the cost-of-change curve is made shallower. Customers are happy because it's never too late to tweak a feature, and developers are happy because they're not expending lots of effort into writing up requirements specifications, designing UML diagrams, and the like.

Numerous high-profile examples exist of multi-million pound software systems going exponentially over budget or failing to deliver at all as a result of following the Waterfall model, which some consider to be antiquated and not suited to the world of software engineering. In comparison, there is this image of agile developers being able to nimbly build incredible systems without being held up by dull and costly processes such as documentation.

Though I am a keen advocate of agile, we should be able to adapt our choice of methodology to the project at hand, not the project at hand to our choice of methodology. Looking at this project impartially, I had a few reasons to consider a plan-driven approach:

- This project requires building an Online Dispute Resolution system. Systems like this have already been established, meaning that there is already a list of fairly straightforward features that can be formalised in advance to be taken into account in the design.
- In ODR, there is a heavy emphasis on law and following processes correctly. For example, the ODR platform cannot be allowed to give either party an unfair advantage through being able to exploit lifespan negotiation. It is absolutely critical that the platform does not violate any of the rules of ODR - and therefore it is essential to document these rules in the requirements specification, for traceability and accountability.
- This project's customer is very busy and has only been able to meet a couple of times over the course of the project. This is not nearly often enough to constitute as an 'on-site customer' - one of the pre-requisites of an agile project - again suggesting that plan-driven is the best approach.

- Finally, the process of gathering requirements, creating a design, and implementing and testing a substantial software system is still somewhat new to the sole developer on this project. Having a design and a plan is a comforting safety net.

Of course, the above points don't disqualify agile practices from the project. Agile principles of TDD, continuous integration (CI), regular releases and merciless refactoring are all very worthwhile activities and are not necessarily mutually exclusive from the Waterfall model. However, they do not lend themselves particularly well to the traditional workflow of implementation and then testing.

Thus, it was decided that the approach should be a hybrid one of Waterfall and Agile. The project should begin with a strong set of requirements and there should be some up front design for parts of the project that are unlikely to change, such as the database schema. At the implementation stage, the project should switch to a business-driven, test-driven approach that utilises the best of the agile processes.

Appendix B

How does SmartResolution work?

Online Dispute Resolution is aimed at lawyers (hereafter known as Agents), representing their clients through the ODR platform. Optionally, we have mediators, whose aim is to intervene as an independent third-party if the two opposing lawyers are struggling to reach a resolution. SmartResolution supports both of these roles.

1 Roles in the system

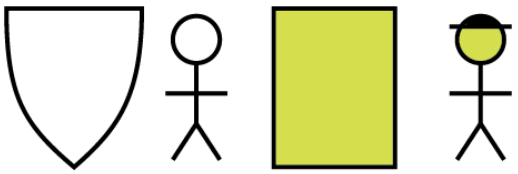

Figure B.1: Roles in the system

In figure B.1, from left to right, we have:

- Law Firm - an Organisation account.
- Agent - an Individual account, belonging to a Law Firm.
- Mediation Centre - an Organisation account.
- Mediator - an Individual account, belonging to a Mediation Centre.

2 Setting up

Before a dispute can be opened on SmartResolution, both Agents must be registered to the system. For each Agent to be registered, their Law Firm must also be registered.

1. Law Firm A registers an account and logs in.
2. Law Firm A creates an account for Agent A.
3. Law Firm B registers an account and logs in.
4. Law Firm B creates an account for Agent B.

3 Creating a dispute

3.1 Assigning the dispute to the relevant parties

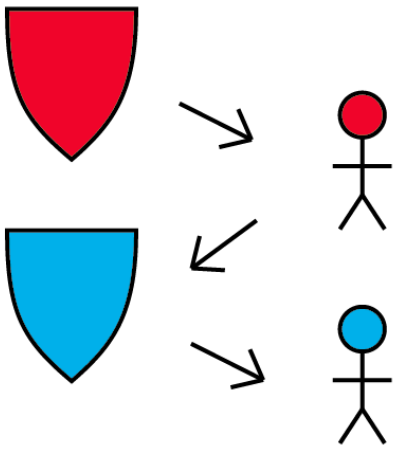

Figure B.2: Creating a dispute

Figure B.2 demonstrates:

- Law Firm A creates a dispute, assigning it to Agent A.
- Agent A fills in their summary for the dispute and opens the dispute against Law Firm B.
- Law Firm B assigns the dispute to Agent B.
- Agent B fills in their summary to accept the dispute.

3.2 Negotiating a lifespan

The agents need to agree on a start and end time for the dispute. A dispute can last for hours, weeks or months, depending on the preferences of the agents involved.

A lifespan can be re-negotiated at any point in the dispute process.

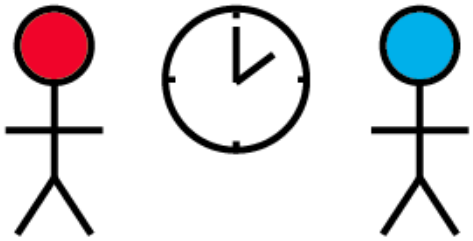

Figure B.3: Negotiating a lifespan

Figure B.4: The dispute begins

3.3 The dispute begins

With a lifespan negotiated and a dispute fully assigned, the agents are now able to communicate through the SmartResolution chat facility and upload documents as evidence.

4 Mediation

Sometimes, disputes may not be solved without the aid of an independent mediator. SmartResolution supports Mediation Centres and Mediators: at any point in an active dispute, either agent can propose mediation.

- An agent proposes to use Mediation Centre 'MC' to mediate the dispute.
- The other agent accepts. 'MC' is notified that they are now the Mediation Centre of the dispute.
- 'MC' provides a list of all available Mediators.
- The agents are free to peruse the profiles of the Mediators, reading their CVs and contacting them externally if necessary.
- An agent proposes to use Mediator M to mediate the dispute.

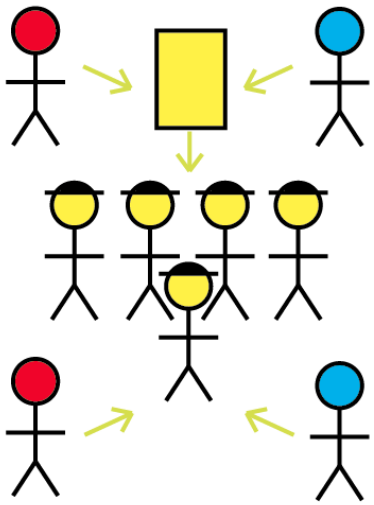

Figure B.5: Mediation

- The other agent agrees. The dispute is now in mediation.

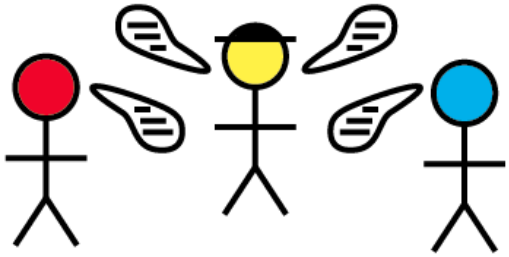

Figure B.6: Mediation in action

With the dispute in mediation, all communication is done through the mediator, unless the mediator proposes "Round-Table Communication" and both agents accept. In this case, a three-person chat is unlocked. At any time, either agent can revoke Round-Table Communication.

5 Modules

If there is a relevant SmartResolution Module installed to the system, that may help the Agents to reach a resolution.

For example, let's say we have a 'Maritime Collision' module. This adds a 'Maritime Collision' dispute type to the dispute-creation screen. With the dispute type set to this type, an additional option is unlocked: Agents are able to let the module ask them structured, maritime-law-specific questions. The module then interprets their answers according to the law, and automatically suggests a resolution. Modules can help to cut out mediators altogether, which is what sets SmartResolution apart from the competition.

Appendix C

Requirements Gathering

My plan was always to use behaviour-driven development in my project, and I believe passionately in keeping all documentation as closely in line with the code as possible. I used Gherkin syntax to describe the features, and linked these up to an executable Ruby and Cucumber test suite.

These are the original requirements, agreed and signed off by all of the project's stakeholders.

1 Account Creation

```
Feature: Account Creation
    I should be able to register an account of a certain type, e.g.
       Company/Agent
    And I should be able to log into said account

  Scenario: Company registration
    Given I am an authorised representative of the Company
    When I attempt to create a new Company account
    Then the account should be created
    # @TODO - as discussed in the supervisor meeting, we could add an
       admin verification stage at this stage.
    # This could be as complicated as we want to make it, so for now,
       let's add a boolean in the database that
    # says if the Company is verified or not. Make it verified by
       default, but carry out isVerified checks at login.
    # We can add the verification steps later and make Companies
       unverified by default.
    # At that stage, we need to add additional features, e.g. Given I
       am unverified, When I try to log in, Then I should
    # Not be allowed to do anything.

  Scenario: Agent initiates Case against a Company not registered to
     the system
    Given I have not yet registered a Company account
    And a Dispute has been initiated against my Company
    When I attempt to create a new Company account
    Then the account should be created
    And the my Company should be automatically linked to the dispute
```

```
  Scenario: Company login with valid credentials
    Given I have registered a Company account
    When I attempt to log in with valid credentials
    Then I should be logged into the system

  Scenario: Company login with invalid credentials
    Given I have registered a Company account
    When I attempt to log in with invalid credentials
    Then an authentication error should be displayed
    # @TODO - as discussed, correct and incorrect login attempts should
        be logged so that we can later add
    # additional security such as locking out accounts after a
       threshold of unsuccessful attempts is reached.

  Scenario: Agent login
  Scenario: Mediator login
  Scenario: Mediation Centre login

  Scenario: Create Agent account
    Given I have logged into a Company account
    Then I should be able to create an Agent account
    And the Agent should be sent an email notifying them they've been
       registered

  # Exactly the same process as for Company registration. There should
     be a drop-down list that lets registering users
  # select whether they're registering as a Company or a Mediation
     Centre
  Scenario: Mediation Centre registration
  Scenario: Create Mediator account
```

2 Dispute Creation

```
Feature: Dispute creation
    Given I am logged into an authorised account
    Then I should be able to create a Dispute

  Scenario: Creating a Dispute
    Given I am logged into a Company account
    Then I should be able to create a new Dispute

  Scenario: Allocating a Dispute to an Agent
    Given I have created a Dispute
    And I have created an Agent
    Then I should be able to allocate the Agent to the Dispute # this
       should also be possible AT the Dispute creation stage

  Scenario: Submitting a Dispute
    Given a Dispute has been assigned to me # by my Company, regardless
        of who instigated the Dispute
    When I write a Dispute summary
    And I choose to submit the Dispute to the system
```

```
    Then the Dispute should be submitted

  Scenario: Initiating a Dispute against a Company
    Given I have submitted a Dispute
    Then I should be able to initiate it against another Company
    # We pick from a drop-down list of Companies in the system
    # or provide a Company email address inviting them to register.

  Scenario: Being initiated a Dispute
    Given a Dispute has been initiated against my Company
    And I have created an Agent
    Then I should be able to allocate the Agent to the Dispute
```

3 Dispute

```
Feature: Dispute (pre-Mediation)
    The Dispute is underway, both Agents are free to communicate with
       one another,
    propose offers, attach evidence, etc.

  Background:
    Given the Dispute is fully underway
    And the Dispute is not in Mediation

  Scenario: Free communication
    Then I should be able to communicate with the other Agent freely

  # The "Propose Resolution" mechanism outlined below is a separate
     facility to above.
  # Think of the free communication as a private messaging system (
     which gets blocked when
  # we go into Mediation, then re-opened with the additional Mediator
     person when entering
  # round-table communication).
  # The offer mechanism is a more formalised communication, where you
     offer a certain amount,
  # under X conditions - Accept | Deny | Propose Counter Offer

  Scenario: Make an offer
    Then I should be able to send the other Agent an offer

  Scenario: Accept the offer
    Given I have been sent an offer
    Then I should be able to accept the offer
    And the Dispute should close successfully

  Scenario: Propose counter-offer
    Given I have been sent an offer
    Then I should be able to propose a different offer

  # Note:
  # I can also Decline an offer by taking the Case to Court - see
     dispute_independent.feature "Take the Dispute to Court".
```

```
  # I can also propose Mediation. See dispute_independent.feature "
     Start the Mediation Process"
```

4 Dispute-Independent Features

```
Feature: Processes relevant to the Dispute but that are not dependent
    on the current state of the Dispute
     There are various functionalities that do not depend on the current
         state of the Dispute
     And should be accessible at any point in the Dispute

  Scenario: Start the Mediation process
     Given the Mediation process has not begun
     Then I should be able to start the Mediation process

  Scenario: Take the Dispute to Court
     Given the Dispute has not yet been resolved
     Then I should be able to Take the Dispute to Court
     And the Dispute should close unsuccessfully
```

5 Dispute Lifespan Negotiation

```
Feature: Negotiating a Dispute lifespan
     When a Dispute is opened and each Company has allocated an Agent
     The Agents need to negotiate a Dispute lifespan
     i.e. the maximum length of time the Dispute can continue without
        resolution
     before being automatically taken to Court.

  Scenario: Creating a Dispute lifespan offer
     Given both Agents have submitted the Dispute
     Then I should be able to make a lifespan offer # regardless of who
        submitted the Dispute first

  Scenario: Accepting a Dispute lifespan offer
     Given the other Agent has sent me a Dispute lifespan offer
     Then I should be able to accept the offer
     And the Dispute should start

  Scenario: Create a counter Dispute lifespan offer
     Given the other Agent has sent me a Dispute lifespan offer
     Then I should be able to make a lifespan offer
     And therefore decline their original offer

  Scenario: Renegotiating the Dispute lifespan mid-Dispute
     Given the Dispute is fully underway
     Then I should be able to make a lifespan offer
     And the Dispute should continue normally despite the renegotiation
        offer
```

6 Putting a Dispute into Mediation

```
Feature: Mediation
    At any point in a confirmed Dispute
    Either Agent can propose Mediation
    Whereby a Mediator is introduced to help to resolve the Dispute
    If complications arise during the Mediation Creation process - e.g.
        a
    list of Mediators being provided to the Agents but are not suitable
       ,
    then either Agent can restart the Mediation process.

  Background:
    Given the Dispute is underway and a lifespan has been agreed

  Scenario: Choosing a Mediation Centre
    Given both Agents have agreed to start the Mediation process
    Then I should be able to select the Mediation Centres I'm happy
       with
    # We do this by choosing the Mediation Centres we want AND an order
        of preference.

  Scenario: No mutually chosen Mediation Centres
    Given both Agents have selected the Mediation Centres they want
    And there are no matches in their choices
    Then both Agents should have the opportunity to choose again

  Scenario: One mutually chosen Mediation Centre
    Given both Agents have selected the Mediation Centres they want
    And there is only one match in their choices
    Then that should be the chosen Mediation Centre

  Scenario: Multiple mutually chosen Mediation Centres
    Given both Agents have selected the Mediation Centres they want
    And there are several matching choices
    Then one Mediation Centre must be chosen upon by both Agents
    # It's been suggested we could make Agents choose (before this step
       )
    # the order of preference for the Mediators, then the system could
       suggest
    # a Mediator based on a points system.

  Scenario: Mediation Centre is notified of the Agents' decision
    Given my Mediation Centre has been chosen by both Agents of a
       Dispute
    Then I should be notified that my Mediation Centre has been chosen
    And I should have the facility to offer a list of Mediators to the
       Agents

  Scenario: Mediation Centre provides list of Mediators
    Given a Mediation Centre has provided a list of available Mediators
    Then I should be able to view the details of each Mediator #
       including CV, etc
    And I should be able to select the Mediators I'm happy with

  Scenario: No mutually chosen Mediators
```

```
    Given both Agents have selected the Mediators they want
    And there are no matches in their choices
    Then both Agents should have the opportunity to choose again

  Scenario: One mutually chosen Mediator
    Given both Agents have selected the Mediators they want
    And there is only one match in their choices
    Then that should be the chosen Mediator

  Scenario: Multiple mutually chosen Mediators
    Given both Agents have selected the Mediators they want
    And there are several matching choices
    Then one Mediator must be chosen upon by both Agents

  Scenario: Mediator is notified of the Agents' decision
    Given I am a Mediator
    And I have been chosen by both Agents of a Dispute
    Then I should be notified that I have been chosen
    And I should be made to sign a confidentiality agreement

  Scenario: Mediator signs confidentiality agreement
    Given I am a mutually-chosen Mediator for a given Dispute
    And I sign the confidentiality agreement
    Then the Dispute should now be in Mediation Mode
```

7 Dispute in Mediation

```
Feature: Dispute (under Mediation)
    The rules of the Dispute have now changed. All communication must
       be done through the Mediator.
    It is at this point that the business logic specific evidence-
       gathering can be applied, so that
    the artifical intelligence in the module can provide a second
       opinion to the Mediator.
    The Mediator, being a specialised and trained individual, can
       choose to ignore or amend the given
    advice.

  Background:
    Given the Dispute is fully underway
    And the Dispute is in Mediation

  Scenario: Block Agent A and B from communicating with one another
    Given we have not activated round-table communication
    Then I should not be able to communicate with the other Agent

  Scenario: Mediator requires further information
    Given a Dispute type was selected at the beginning of the Dispute #
        e.g. "maritime collision"
    Then the type-specific module should offer custom forms to the
       Agents to fill in

  # business logic specific stuff relating to Maritime Collisions etc
```

```
    MUST be put into a separate feature file
# (in the plugin directory). This set of features must be as abstract
    and generic as possible.

Scenario: Filling in the type-specific forms
  Given I am an Agent
  And I have filled in the forms provided by the type-specific module
  Then there should be no more forms to fill in
  And I should see that the system is awaiting a response from the
     other Agent

Scenario: Filling in the type-specific forms as the second Agent
  Given I am an Agent
  And I have filled in the forms provided by the type-specific module
  And the other Agent has also filled in the forms
  Then there should be no more forms to fill in
  And I should see that the system is awaiting a response from the
     Mediator

Scenario: AI logic is applied
  Given I am a Mediator
  And both Agents have completed the type-specific module forms
  Then I should see the results of the AI in the type-specific module
  #And I should be able to advise each Agent individually
  # Commented out the above line because it isn't testable.
     Essentially, the Mediator can send a
  # private message to either Agent, negotiating a resolution. It is
     up to the Agents to formally
  # send an offer through the "Propose Resolution" facility.

Scenario: Accepting the Mediator's offer
  Given the Mediator has given me an offer
  Then I should be able to accept the offer
  And the Dispute should close successfully

Scenario: Declining the Mediator's offer
  Given the Mediator has given me an offer
  Then I should be able to decline the offer
  And the Dispute should remain open

Scenario: Sending an offer for round-table communication
  Given I am a Mediator
  Then I should be able to offer round-table communication
  # The Mediator should (through a dedicated facility) be able to
     propose round-table negotation,
  # whereby the free communication of all parties is enabled.

Scenario: Accepting the offer for round-table communication
  Given the Mediator has suggested round-table communication
  Then I should be able to accept the offer
  And the Dispute should go into Round Table Mediation mode

Scenario: Declining the offer for round-table communication
  Given the Mediator has suggested round-table communication
```

```
Then I should be able to decline the offer
And the Dispute should remain open and under Mediation
```

Appendix D

Commercial Viability

The below extract is taken directly from my blog, available at:

`http://ashton.codes/blog/commercial-viability/`

As a result of using the Fat-Free Framework, my project is forced to adhere to the terms of the GPLv3 license, which stipulates that if I distribute my software I must make the source code easily available. In theory, this shouldn't force me to go open-source. As my project is web-based, I am within my rights to deploy my code to a server, and any users of a website that uses that code do not have the right to access the source code, since I'm not deploying the program itself to them: only the results of the program. The AGPL license would be a different story, as it closes that loophole.

Nevertheless, I've chosen to make my project open-source. Why? Well, partly because GitHub charges for private repositories, and all of the CI tools that I'm using in my project (Travis, Code-Climate, Gemnasium) are free for open-source projects, but charge money for private repositories. In addition to all this, I want people to be able to follow my progress. I want something I can show future employers. I also found it very useful to be able to show my supervisor my feature files so that they could be double-checked and signed off.

All these reasons led to me open-sourcing my project: but does that hinder its commercial viability? One's instinct is to say "yes" - if people can download, modify, run and re-distribute your software for free, why would they ever want to pay for it? I believe my project does hold commercial promise, and the reason came to me while I was trying to plan out what I would say in my Mid-Project Demonstration, which is happening this time next week.

1 Where is the cool stuff?

I feel I have to justify the amount of time spent building the core ODR platform. After all, the exciting bit about my project, surely, is all about the maritime collision logic; the AI and the neural networks and the natural language processing that form the virtual court simulation and replace the physical court. That's some pretty awesome stuff right there.

Though the maritime collision logic is seemingly the exciting, ground-breaking stuff, there's

also been a heavy emphasis on keeping the underlying system as abstract as possible, so that any module of business logic can be plugged into the system. And that, ladies and gentlemen, is where the truly exciting stuff is.

The core ODR platform is a large, extensible, robust and skilfully constructed framework. It needs to be, if developers are to have any confidence in building plugins for it. After completing the core platform, I expect to not have all the time in the world to work on the maritime collision module. I hope that, through the module, I can at least develop a plausible promise as to what the system is capable of. The maritime module is a prototype, a placeholder... something that other developers can spend a lot more time expanding upon in the future. The real meat of this project is in the underlying platform.

The analogy I'm hoping to use on Tuesday is this: the ODR platform is like WordPress. The Maritime Collision module is like a WordPress plugin. The former is, by necessity, a huge, well-tested framework which has had contributions from hundreds of developers. The latter could be as large in scope as the former, or it could be a one-script plugin that scratches a lone developer's personal itch. Both can be developed independently of one another.

2 Investigating the WordPress pricing model

I realised that my project could learn many lessons from WordPress. Both projects do completely different things, but I can utilise and copy certain things like how WordPress fires events that plugins can be hooked into, and how it allows site administrators to easily install, upgrade and uninstall plugins directly through the admin dashboard provided. But going beyond the software level, I can also learn lessons from how WordPress makes money, and why they continue to do what they do.

With all the goodwill in the world, WordPress would not continue to be developed if money wasn't being made somewhere. So where is it made? WordPress itself is free to download and install on any server as you wish. You can even save yourself the effort of doing that, and create your own WordPress subdomain directly through wordpress.com, at no cost.

The truth is, there are a lot of companies making a lot of money off the back of WordPress, and I'm sure some of the WordPress contributors are getting in on the action themselves. Developers can create plugins and themes, and though most of these are free, many of them are not... and some of them are really rather expensive.

Though these themes and plugins are covered by the same GPLv2 license WordPress is bound by, people are willing to pay for them since they often include developer updates and support in the price. In theory, anyone who purchases one of these plugins is free to redistribute the plugin for free, but in practice this tends not to happen and people still buy the original plugin directly from the developer (through the WordPress platform) for the benefits outlined above.

3 How this can be applied to my project

I think that keeping my ODR platform open-source is a smart move. Anyone is free to download the project, install it on their own server, and even start charging money for their online dispute resolution services. I think it's also a good idea to make the maritime collision module free, and

perhaps a handful of other dispute type modules too (such as divorce cases, unfair dismissal claims, etc), to make the system more useful and show people what is possible with the core platform.

If I were to continue with the project beyond university, my plan would be to develop even more dispute type modules: say, tenancy agreement breaches, slander, etc - and to charge money for these modules. Anyone would be able to set up a website with the core platform and some dispute types for free, but to rise above the competition and have the most comprehensive ODR system, they'd have to pay for additional modules. Think of it as The Sims' expansion packs!

4 Building my brand

WordPress is as big as it is today because not only is it open-source, but it is also a household name. It has built up a reputable brand and is now the market leader in blogging software. My ODR platform clearly needs an identity.

As a placeholder, I created a logo a couple of weeks ago, entitled "SmartResolution". It seemed a fairly good and appropriate name: my platform offers online dispute resolutions, but in a smart way, i.e. input analysis and a heuristically-driven suggestion as to the outcome of the dispute.

Today I bought a domain name to help reinforce that branding: smartresolution.org. At the time of writing there is nothing to see yet, since I have no server - but I intend to set up AWS integration in the next couple of weeks so that I can easily and automatically deploy the latest version of my project to a cloud server, accessible through smartresolution.org.

I'm not expecting to make money from this (for a start there are other issues I haven't discussed, such as my project technically being the property of Aberystwyth University), but at the very least it is an easy way to give my supervisor and stakeholders access to the beta version of my project. I'd like to develop my project as if it's going somewhere, because if I don't, it never will.

Appendix E

Rationale For My Database Design

I'll explain the rationale behind my decisions, starting from the most logical beginning: account registration.

1 Accounts

I recognised that all accounts, be they organisations or individuals, agents or mediators, all require the ability to log in using an email address and a password. Rather than duplicating that information across several tables representing each user type, I encapsulated it into its own table: `account_details`.

Most websites these days now have some form of email verification process, requiring new registrants to click on a unique link sent to their email address. I knew that SmartResolution would likely require such a feature in the future, so added a `verified` boolean to the table. For now, it defaults to true, but as and when email verification becomes a requirement, SmartResolution will be ready for it.

Given that I've removed duplication by creating the `account_details` table, what was stopping me from simply combining all other account-related fields to that table, from name to type to description?

Well, to begin with, the `name` property is subtly different between organisations and individuals. Individuals have a first and second name, e.g. Chris Ashton, whereas organisations tends to have just one, e.g. Webdapper Ltd. I could have made one generic `name` field and stored individuals' names in the field, i.e. "Chris Ashton" rather than "Chris" and "Ashton", but this would have restricted the software in other ways, e.g. searching for agents by surname. There are ways around this, of course, such as using regular expressions to extract a surname from the end of the string, but this would be in violation of first normal form as the database table would not be atomic.

Furthermore, there are other differences between individuals and organisations. The former needs to be able to edit their CV, whereas the latter needs to be able to edit their organisation description. Though the implementation of these functions - going to the account settings page and editing a HTML textarea - is identical, it's possible in future that this might not be the case. What happens if CV editing is later changed so that a PDF must be uploaded? What if additional fields are added to the edit screen, such as age and gender? To future-proof the application, it

made sense to keep the `organisations` and `individuals` tables separate.

I could have opted for an even more finely granulated table structure and separated `individuals` into `agents` and `mediators`. However, I could see no differences between the two types in terms of the data required of them, so these tables would only be duplicates of one another, save for the table name. To avoid an overly complicated structure, I kept the higher-level account type as its own table and created a `type` field as a modifier with SQLite CHECK constraints limiting the values to either "agent" or "mediator".

This has the advantage of allowing more account types in future, by adding another allowed option to the CHECK constraint. All of my production-code queries, such as `"SELECT * FROM individuals WHERE some condition AND account_type = :account_type"`, should still work right out of the box. If I'd gone for separate tables, however, I might have to go through all of my production code and add additional queries, e.g.

```
if type == "mediator"

    "SELECT * FROM mediators WHERE some condition"

else if type == "agent"

    "SELECT * FROM agents WHERE some condition"

else if type == "something_else"

    "SELECT * FROM somewhere_else WHERE some condition"
```

2 Disputes

Every dispute is created by a law firm, assigned to an agent, opened against another law firm and finally assigned to one of that firm's agents. Every dispute, therefore, has four accounts associated with it.

In addition, either "party" (consisting of a law firm and an agent) must be able to create and continue to edit their summary of the dispute, viewable to all parties. Therefore we have a dispute linked to two parties, each of which is linked to a law firm, an agent and a summary.

This relationship is implemented in the `dispute_parties` table, which I've kept separate from the main `disputes` table. I didn't want to over-complicate the `disputes` table with too many fields, but this was not the only reason for my decision. I felt that it was quite possible a future requirement might be for multi-party disputes, where we have three parties representing clients A, B and C. The `dispute_parties` table decision taken would be able to support such a requirement.

I had hoped to use this table for storing the dispute mediation centre and mediator information, but the mediation requirements were more complex than I'd first hoped, and required separate tables. I'll discuss these a little later. For now, let's look at the `disputes` table.

Each dispute has a primary key, `dispute_id`, to uniquely identify it, and is linked to two parties. I soon found that I'd be rendering lists of disputes in my application, e.g. an agent checking on the status of all of their disputes at once, so needed a visual way of distinguishing

between the disputes. It made sense to add a `title` field, and this was later clarified by my supervisor as a requirement.

Each dispute should also have a type, which is how SmartResolution would allow modules to hook into the core platform. The module of the corresponding type would automatically be loaded for the given dispute. As such, a `type` field representing the unique ID of the module was required.

Disputes can be in one of many states, but any of these states can be put into a higher-level category as either "ongoing" or "closed". These states are mutually exclusive. More importantly, neither state can be inferred from other fields in the database alone, so there is no risk of redundancy. [1]

To explain further: for a dispute to be closed, it needs to have had an agreed lifespan which subsequently comes to an end, closing the dispute automatically. This is a complicated query, to have to make just to get the status of a dispute, but this is not the only reason I created the status field. A dispute can have an active lifespan but then be manually closed by one of the agents - there needs to be a way of storing that closure.

For finer granularity, and to allow for data analysis further down the road, I expanded the possible states slightly to indicate whether a dispute was resolved successfully, or closed because it had to be taken to court. Therefore, the three possible states are "ongoing", "resolved", or "failed", and these are enforced with SQLite CHECK constraints.

Finally, there is a `round_table_communication` boolean. I'll explain this in the "Mediation" section later on.

3 Dispute lifespan

Disputes have a lifespan: a start and end date defining the time in which a dispute must be resolved before it is closed automatically, giving agents an incentive to focus.

One's instinct is to put this directly in the `disputes` table, but this would not allow for the fact that lifespans are negotiated, not pre-medidated. Agents should be able to *propose* a lifespan, and the other agent should be free to accept it or decline it and respond with their own proposal.

Given that lifespans are not automatically applicable, the intentions are muddled if this information is stored directly in the `disputes` table. We'd now need to add a `lifespan_status` describing whether or not a lifespan has been accepted or merely offered.

As an agent is required to propose an offer and the other is required to respond to it, the identity of the proposer must also be stored, so we'd need an extra field again, e.g. `agent_proposer`. This complicates the `disputes` table even more, but even this would still be a valid solution.

The problem arises when an agent wishes to *renegotiate a lifespan mid-dispute*, as is a functional requirement. How can we differentiate between what is a current, active lifespan, and what is a new offer?

All of these complicated scenarios meant that the best solution was to extract the lifespan information out of the main `disputes` table, and into its own table: `lifespans`. For reasons

[1] Redundancy is where information in the database becomes misaligned. If the same bit of information is captured in multiple places, there's a risk the two might conflict - then which one do you go for?

I've just explained, the corresponding `dispute_id` is stored in the `lifespans` table, rather than the `lifespan_id` being stored in the `disputes` table. The remaining fields are fairly self-explanatory.

4 Mediation

Mediation is approached in a similar way to lifespans, as again it is something which one agent proposes and the other accepts. An agent must propose a mediation centre, the other agent must accept, and then an agent must propose a mediator which the other agent must accept.

The process for selecting a mediation centre and a mediator is the same: the only difference is how the options are populated. Under the current business logic, all mediation centres are available for selection, whereas only the mediators that the chosen mediation centre has marked as available are able to be proposed by the agents.

Given that the process for proposing and accepting both a mediation centre and a mediator is the same, it makes sense to combine the two in one table: `mediation_offers`. The `proposer` is the ID of the agent who made the proposal, and the `proposed_id` is the ID of the mediation centre or mediator that they wish to propose to mediate the dispute.

Whether the proposal is for a mediation centre or a mediator is marked by the `type` field, which has a CHECK constraint of either "mediation_centre" or "mediator". This risks introducing redundancy to the database (as, for example, a mediator ID may be provided even if the `type` has been set to "mediation_centre") but saves the application from having to make a complicated and inefficient table-join to extract that information.

When a mediation centre is chosen, the mediation centre must provide a list of available mediators, after which the agents must select one mediator. This is a many-to-many relationship[2] and thus must be represented by its own table: `mediators_available`.

It may seem odd to have the 'round_table_communication' property in the `disputes` table, since it is a property that can only be changed by the mediator and when a dispute is in mediation. However, it is an attribute of the dispute itself and can be said to default to 'false'. It is not an attribute of mediation, but can be affected by the actions performed whilst in mediation.

5 Miscellaneous

Finally, we have miscellaneous tables such as `notifications`, `messages` and `evidence`. These are all very separate concepts to what has been discussed previously, though they are also intrinsically linked to tables we've already discussed.

Notifications are linked to accounts, evidence is linked to disputes, and messages are linked to both. There can be zero of each, or there can be many of each. For these reasons, it's self-explanatory why these should be in their own tables.

[2] An unlimited number of mediators may be available for a dispute, and any mediator may be available for an unlimited number of disputes. This is known as a many-to-many relationship.

Appendix F

Third-Party Code and Libraries

I will begin by listing the third-party code, libraries, frameworks and images used by the three components of this dissertation. The full description for each dependency can be found at the end of this appendix, in alphabetical order.

1 SmartResolution website

smartresolution.org uses a number of third-party frameworks and services:

- Bootstrap: this is used for the front-end styling and as a UI framework.
- Fat-Free Framework: this powers the back-end, allowing me to define HTTP routes and their handlers.
- phpDocumentor: this is used to generate API documentation from the Docblock comments in the core SmartResolution software.
- SlickNav: responsive mobile menu.
- jQuery: a dependency for the plugin above.

In addition to the SmartResolution website, I developed a demo video showing the workings of the SmartResolution software and the maritime collision module; this video is embedded on the homepage of the SmartResolution website. This video is hosted on YouTube and makes use of YouTube's annotation facility to give useful commentary throughout the video. I overlayed a music track by Eric Matyas, which is free under the Creative Commons by Attribution license[1]. The track is entitled 'GAME MENU_v001' and is available at `http://soundimage.org/looping-music/`

The design and creation of the installation icons[2] were outsourced to a friend of mine, Lani Cossins. The workflow images on the website[3] and in appendix B were my own design, but were later digitised by Rosie Bettles.

[1] Licensing is discussed on the provider's website at `http://soundimage.org/`
[2] Available at `http://smartresolution.org/documentation`
[3] Available at `http://smartresolution.org/workflow`

It is worth explaining that, though smartresolution.org is hosted on AWS, it is server-agnostic and no AWS-specific assets exist in the repository. Therefore, I will not be discussing the use of AWS here.

2 SmartResolution

As SmartResolution is more substantial than the website which hosts it, it is necessary to split its dependencies into certain categories.

2.1 Front-end

- Bootstrap: for front-end styling and as a UI framework.
- jQuery Date and Time picker: for a user-friendly way of negotiating dispute lifespans.
- jQuery: a dependency required by the plugin above.
- The dashboard icons used in the core platform are designed by Freepik.

2.2 Back-end

- Fat-Free Framework: routes HTTP requests to controllers and also handles database interaction and password encryption.
- SQLite: the RDBMS used by SmartResolution for persistence.

2.3 Development dependencies

- Rubygems: for handling Ruby dependencies.
- Composer: for handling PHP dependencies.
- Cucumber: the BDD framework in use for end-to-end testing.
- RSpec, Capybara and Poltergeist as the Cucumber drivers.
- PHPUnit: for providing methods for (and subsequently executing) PHP unit tests.
- Symfony's YAML component: for handling PHP's parsing of YAML files, used by SmartResolution's PHP unit tests.

3 Maritime collision module

No third-party libraries were used in the development of the module. All code is native PHP/HTML and all assets (such as the anchor image) were taken from the public domain (in this case, pixabay.org).

4 Descriptions

All of the following were used without modification.

Bootstrap (v3.2.2) - MIT - `http://getbootstrap.com/`

Bootstrap defines sensible browser defaults (instantly improving unstyled webpages), a responsive 12-column layout and all manner of useful component classes. These classes help represent standard program 'states' such as success, error, new information, and so on. "Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web.".

Capybara (v2.4.4) - MIT - `https://github.com/jnicklas/capybara`

Used in conjunction with Ruby and Cucumber, "Capybara helps you test web applications by simulating how a real user would interact with your app. It is agnostic about the driver running your tests"; in our case, we use Poltergeist.

Cucumber (v2.0.0) - MIT - `https://github.com/cucumber/cucumber`

"Cucumber is a tool for running automated tests written in plain language." Cucumber features are the way I textualised my requirements at the beginning of the project. They're incredibly useful as they are written in plain language yet are linked to tests, so serve as tests and documentation all at once.

Dashboard icons - Freepik license - `http://www.flaticon.com/packs/web-pictograms`

These icons are free for use but the Freepik license requires that the phrase "designed by Freepik" is displayed on the webpage that uses the icons. I have fulfilled this by putting an attribution to them in the footer of the SmartResolution software. License URL: `http://cdn.flaticon.com/license/license.pdf`

Fat-Free Framework (v3.4) - GPLv3 - `https://github.com/bcosca/fatfree`

F3 provides a solid foundation including "an easy-to-use Web development tool kit, a high-performance URL routing and cache engine, built-in code highlighting, and support for multilingual applications." I particularly found the database interaction and routing modules useful.

jQuery (v2.1.3) - MIT - `https://jquery.org/`

jQuery is the most popular JavaScript library in the world, providing useful methods on DOM elements in a cross-browser and backwards-compatible way. Many JavaScript libraries and plugins require jQuery as a dependency, such is its ubiquity. License URL: `https://jquery.org/license/`

jQuery Date and Time picker (plugin for jQuery) (v2.4.1) - MIT - `https://plugins.jquery.com/datetimepicker/`

A user-friendly date and time picker, used by the SmartResolution core software to make it easier for agents to propose start and end dates for their disputes.

phpDocumentor (v2) - MIT - `https://github.com/phpDocumentor/phpDocumentor2`

Having marked up SmartResolution with detailed, semantic comments, I needed a program to parse those comments and generate API documentation in HTML. phpDocumentor was exactly what I needed.

PHPUnit (v4.6) - BSD 3-Clause - `https://github.com/sebastianbergmann/phpunit/`

"PHPUnit is a programmer-oriented testing framework for PHP". I use PHPUnit to unit-test the SmartResolution core software. BSD license information: `http://opensource.org/licenses/BSD-3-Clause`

Poltergeist (v1.6.0) - MIT - `https://github.com/teampoltergeist/poltergeist`

Poltergeist is a driver for Capybara which allows one to run Capybara tests on a headless WebKit browser (provided by PhantomJS). I chose it because it is quick to run and supports JavaScript, which is required by some SmartResolution features.

RSpec (v3.2.0) - MIT - `https://github.com/rspec/rspec/`

RSpec is used for defining assertions, exposing methods such as `assert(some_condition)`. It provides the very foundation of my Cucumber feature tests.

Rubygems (v2.2.2) - MIT - `https://github.com/rubygems/rubygems`

RubyGems is a package management framework for Ruby. It can be used to automatically download all of SmartResolution's Ruby dependencies, including Cucumber, Capybara and RSpec.

SlickNav (v1.0.3) - MIT - `http://slicknav.com/`

Duplicates a given menu and styles it in a mobile-friendly (i.e. dropdown) style. The application using it simply has to add a media query to their CSS to hide the 'mobile' menu and show the 'desktop' menu when an arbitrary breakpoint is reached.

SQLite (v3.7.13) - Public domain - `http://www.sqlite.org/`

A simple, fast relational database management system. It is important to note that this choice of RDBMS is not particularly important, as another RDBMS can be swapped into SmartResolution with relative ease. Copyright information: `http://www.sqlite.org/copyright.html`

Symfony's YAML component (v2.6) - MIT - `https://github.com/symfony/Yaml`

Parses a string containing YAML-formatted data. YAML (a recursive acronym, standing for "YAML Ain't Markup Language") is a terse way of representing structured data, and has gained popularity for use with languages such as Ruby and Python. PHP has no built-in YAML parser, so I used Symfony's YAML component, which is an industry-standard YAML parser for PHP.

Appendix G

Comments: a Philosophy

Comments are often a contentious issue. *Clean Code* (Robert Martin) has an entire chapter dedicated to them. [26] I think his and my opinion on comments can be summed up in one line, taken from that chapter: "comments are, at best, a necessary evil." That is, they should be avoided where possible, only used "to compensate for our failure to express ourself in code".

Tom Maslen, technical lead at BBC Visual Journalism, once told me his views on writing code comments. I may be paraphrasing somewhat, but his philosophy was something like this:

"There are three stages to a developer's commenting ideology: the first stage is where the developer comments every line of code, thinking they're being a good programmer. They soon learn that their comments are muddling the code and make it harder to read."

"They then turn to Javadoc-style API commenting, writing parameter and return-type descriptions above function definitions, marking up their comments in a way that lets them run a tool that can generate documentation from their comments. However, they'll learn that even this isn't sustainable: all comments eventually become misaligned with the code that they originally referred to. All comments are extra effort for the developer, requiring them to duplicate the intentions of their code in multiple places, violating the DRY principle. Comments can lead to sloppy code, since you don't need to be able to explain yourself fully through your code alone: you can augment it with textual descriptions."

"The best programmers write no comments because their code is self-documenting. The reader should be able to read their code top-to-bottom, like a book, and should be able to easily understand what the code is doing. There's no risk of having outdated documentation, since there's no documentation in the first place. Let the code speak for itself. If you feel the need to explain your code, then your code should probably be rewritten."

He wasn't as averse to comments as I've just made out: I know he used to put the odd comment in his code to clarify things for the reader. But he and I agree that a comment in your code is a code smell [17]; an indication that the code needs to be refactored to become more readable and remove the need for that comment.

It is difficult, then, to return to university after my industrial year and find that many of my lecturers are pro-comments and will mark you down for the absence of them. On the one hand, industry is telling me to avoid them, and on the other, academics are telling me to apply them. I had, toward the end of my industrial year, leaned heavily towards that third stage of comments

ideology, only writing comments where I felt my code was not self-documenting enough and making a mental note to refactor later.

In this major project, I've decided to provide Javadoc-style API comments throughout my code. This was not a decision I took lightly. On the one hand, it would appease my lecturers and provide an additional package of documentation explaining the low-level workings of my code. However, I wrote these comments because I thought I'd find them useful, not just because I thought it would impress the markers.

At the BBC, we worked on somewhat small, self-contained applications: items of work that could be started and finished in around 4-6 weeks. We'd often be the sole developer on such features. As a result, creating documentation felt a bit fruitless: we already know our own code (and have tried our best to make it self-documenting), so why go to the effort of providing and then maintaining comments throughout the codebase?

The SmartResolution project is different in scope. It encompasses various front-end and back-end components, as well as rigorous unit and integration tests throughout. Unlike BBC special features, it is *designed* to be free and open-source software, and designed to encourage the open-source community to contribute towards it.

A collaborative project of this scale would fall apart without comments. Even as the sole developer on the project, I found myself forgetting where components were and how I could use them. Far from being a hindrance, I found API-style comments an aid to refactoring, as it allowed me to more clearly see whether my API for a particular class was consistent throughout. This is particularly important in a dynamically typed language such as PHP, where types cannot always be inferred from looking at the function definition alone.

The most important component of the project to be thoroughly commented is the module hook API: developers need to know how they can develop a module to interact with the core platform. As such, this part of the project contains the most detailed comments.

If I were to tackle a small, in-house project in future, I'd still lean towards that third stage of comments ideology. For bigger projects, however, and especially for projects that are likely to be developed collaboratively, API comments definitely have their place.

Appendix H

Cucumber feature granularity

During my BBC placement last year, I attended a Ruby & Cucumber training course that provided an overview of the Ruby language, training in test-driven development, and explained how to define good Cucumber tests. Chris Parsons, who is a major contributor to the Cucumber framework, taught the course.

He taught me that Cucumber features should be abstracted away from the implementations. For example, consider the following scenario:

```
Scenario: Logging in
  When I go to http://localhost:3000/session/new
  And I fill in the 'Username' field with 'admin'
  And I fill in the 'Password' field with 'taliesin'
  And I click the submit button
  Then I should be redirected to the home page
  And the home page should say 'Welcome admin'
```

The scenario is very tightly coupled to the actual implementation of the system, requiring the business analyst to know a lot about low level parts of the system such as the URL of the login page, the presence of a submit button, and so on. If a feature file contained a long list of scenarios like this, the reader would be tempted to skip over lines deemed to be unimportant, defeating the purpose of what should be a 'living' form of documentation. [21]

This style of feature is very constraining: what happens if an additional field is required for verification, or if the user should instead be redirected to a special landing page for registered users, or if the site supports internationalisation and the user has set their default language to Welsh (changing the expected welcome message)?

All of these changes would require updating the feature definition, however the feature itself hasn't changed: the feature still encapsulates the functionality of "logging in". Consider instead this scenario:

```
Scenario: Logging in
  When I go to the login page
  And I fill in the form with valid credentials
  Then I should be logged in
```

We now have a scenario that does not tie itself closely to the implementation. The credentials, the composition of the form or the welcome message might change, but the feature file would

remain the same. This makes sense as the concept of 'logging in' as a feature itself has not changed either.

The first example is written in an imperative style, whereas the second is written declaratively; the style is "more aligned with User Stories in the agile sense, having more of the 'token for conversation' feel to it". [24] Granted, the second example does not provide as much detail. However, in this example, I think Chris Parsons has the right approach. My difficulty in following the declarative style came when:

1. Scenarios required testing multiple inputs.
2. I had to maintain some sort of state.

One feature I struggled to test was the user search functionality:

```
Scenario: Searching the users list on a browser without JavaScript
  Given I have JavaScript disabled
  When I attempt to visit the Users list
  And I search for users
  Then I should see the results
```

My step definition contained hashes of search inputs in the "And I search for users" method that would be mapped to expected results in the "Then I should see the results" method. I had to maintain state between the two steps, iterating back and forth between them, in order to test all of the inputs and outputs.

I'd tried my best to keep the feature file as abstract as possible but it was making my step definition difficult to read and maintain. I started a refactor branch and tried to simplify my step definitions by, essentially, complicating the feature file:

```
Scenario Outline: Searching the users list on a browser without
   JavaScript
  Given I have JavaScript disabled
  When I attempt to visit the Users list
  And the following boxes are checked: <checkboxes>
  And I search for the following term: <search_term>
  Then I should see the following results: <expected_results>
```

I put the question to Twitter: if the choice is a verbose feature file or a verbose step definition, which is the lesser of two evils? A number of fully qualified testers and developers responded and argued the case for and against. I made a decision that moving the assertions to the feature file was probably the best decision, therefore merged my refactor branch. My reasoning was that the inputs and outputs dont refer to implementation details: they require knowledge of the model, but not of the view.

Steven Atherton, a Senior Web Developer at the BBC, made a good point about it being "wrong [to] dip into the definition file to see WHAT'S being tested... it's just HOW". [5] He is suggesting that verbose feature definitions provide more useful documentation from the developers' perspective.

A benefit of feature files is that they clarify requirements, at both a high level and, in this case, quite a low level. We can get business analysts to explicitly confirm the expected application behaviour, rather than leaving it open to interpretation.

Annotated Bibliography

[1] "Convention for the Unification of Certain Rules of Law with respect to Collisions between Vessels," 1910. [Online]. Available: http://www.austlii.edu.au/cgi-bin/sinodisp/au/other/dfat/treaties/1930/14.html

> A simple set of maritime laws; 17 articles only a sentence or two in length. This Convention was used as the basis for the SmartResolution Maritime Collision module.

[2] "Convention on the International Regulations for Preventing Collisions at Sea, 1972," 1972. [Online]. Available: http://www.admiraltylawguide.com/conven/collisions1972.html

> A comprehensive maritime law convention, split into five large sections and containing 38 detailed rules. Given more time, this would have been a logical progression for an 'advanced' maritime collision module for the platform.

[3] Amazon Web Services, Inc, "Aws — amazon linux ami," accessed April 2015. [Online]. Available: http://aws.amazon.com/amazon-linux-ami/

> Amazon's EC2 instances use a Linux image supported and maintained by Amazon, tailored towards providing a "stable, secure, and high performance execution environment" and containing packages that enable easy integration with AWS.

[4] ——, "Global infrastructure," accessed April 2015. [Online]. Available: http://aws.amazon.com/about-aws/global-infrastructure/

> "Amazon Web Services serves over a million active customers in more than 190 countries." This global infrastructure page shows the locations of all of AWS' server farms. At time of writing, Amazon has 13 'availability zones' in the US, 3 in South America, 5 in Europe and 8 in the Asia Pacific.

[5] S. Atherton, "@chrisbashton @tmaslen @jakedchampion its wrong if you dip into the definition file to see whats being tested.. its just how," 28 Nov 2014, 13:13 UTC. Tweet. [Online]. Available: https://twitter.com/Tarqwyn/status/538319811856859136

> Steven Atherton, senior web developer at the BBC, responded to my question "Cucumber people - which is the lesser of two evils? Verbose features, or over-complex step definitions?", suggesting that Cucumber features should be more verbose to make it clear *what* is being tested.

[6] S. Ballmer, "Microsoft ceo takes launch break with the sun-times," 2001, accessed April 2015. [Online]. Available: http://web.archive.org/web/20011211130654/http://www.suntimes.com/output/tech/cst-fin-micro01.html

Steve Ballmer, ex-CEO of Microsoft, famously said of the GPL license (which he refers to indirectly via Linux): "Linux is not in the public domain. Linux is a cancer that attaches itself in an intellectual property sense to everything it touches. That's the way that the license works." Richard Stallman, founder of the Free Software Foundation, later defended the license (*Free as in Freedom*, 2002, Sam Williams): "A spider plant is a more accurate comparison; it goes to another place if you actively take a cutting."

[7] BBC, "Amazon web services 'growing fast'," 2015, accessed April 2015. [Online]. Available: http://www.bbc.co.uk/news/business-32442268

Amazon announced that in the first quarter of 2015, its web services business generated sales of $1.57bn. According to the BBC, "the figures for the first time confirm that Amazon's cloud business is the biggest of its kind in terms of revenue."

[8] BlueSeas, "Accident Investigations." [Online]. Available: http://www.offshoreblue.com/safety/accident-studies.php

A collection of 19 maritime collision case studies.

[9] J. Brien, *Management information systems*. New York: McGraw-Hill/Irwin, 2011, 978-0073376813.

This book suggests that OODBM systems are geared towards multimedia presentation or organizations that utilize computer-aided design. This conclusion factored into my decision to use an RDBMS for SmartResolution.

[10] D. Carneiro, P. Novais, F. Andrade, J. Zeleznikow, and J. Neves, "Online dispute resolution: An artificial intelligence perspective," *Artificial Intelligence Review*, vol. 41, pp. 211–240, 2014.

This paper highlights the relevance of ODR in relation to e-commerce disputes: "We argue that if a transaction occurs online, then disputants are likely to accept online techniques to resolve their disputes". It suggests that current ODR platforms are 'first generation' systems, providing technologies such as instant messaging, video and phone calls, and so on. "Second generation systems extends its first generation with new intelligent and autonomous artefacts", making it "possible to present services with more added value" by developing intelligent software agents, case-based reasoning mechanisms or neural networks. It could be argued that SmartResolution is the first 'second generation' ODR platform.

[11] Civil Justice Council, "Online dispute resolution for low value civil claims," 2015, accessed April 2015. [Online]. Available: https://www.judiciary.gov.uk/wp-content/uploads/2015/02/Online-Dispute-Resolution-Final-Web-Version1.pdf

A report suggesting that settling non-criminal cases of less than £25,000 online would reduce the expenses generated by a court.

[12] M. Daly, "Testing php web applications with cucumber," 2012, accessed April 2015. [Online]. Available: http://matthewdaly.co.uk/blog/2012/11/03/testing-php-web-applications-with-cucumber/

Matthew Daly, a web developer in Norfolk, discusses testing PHP applications with Cucumber. The interesting thing is that he uses Ruby to define the Cucumber steps, rather than PHP's de facto standard Behat. This blog post outlines his reasons for doing so.

[13] Drupal, "Licensing faq — drupal.org," accessed April 2015. [Online]. Available: https://www.drupal.org/licensing/faq/#q7

WordPress actually links to this explanation from their 'About' page (https://wordpress.org/about/license/), such is its relevance to WordPress' own interpretation of the GPL. "The GPL on code applies to code that interacts with that code, but not to data. That is, Drupal's PHP code is under the GPL, and so all PHP code that interacts with it must also be under the GPL or GPL compatible. Images, JavaScript, and Flash files that PHP sends to the browser are not affected by the GPL because they are data."

[14] European Maritime Safety Agency, "Collision," http://emsa.europa.eu/marine-casualties-a-incidents/casualties-involving-ships/collision.html.

A collection of 100 maritime collision case studies.

[15] F3 Community, "Sql — fat-free framework," accessed April 2015. [Online]. Available: http://fatfreeframework.com/sql#Constructor

F3 supports all of the following engines in its Sql layer: MySQL, SQLite, PostgreSQL, Microsoft SQL Server, ODBC, Oracle

[16] M. Feathers, "A set of unit testing rules," 2005, accessed April 2015. [Online]. Available: http://www.artima.com/weblogs/viewpost.jsp?thread=126923

"Michael has been active in the XP community for the past five years, balancing his time between working with, training, and coaching various teams around the world." His experience with XP has given him strong TDD skills, which he's been able to translate into a useful set of unit test rules I tried to adhere to in the development of my project.

[17] M. Fowler, "Codesmell," 2006, accessed April 2015. [Online]. Available: http://martinfowler.com/bliki/CodeSmell.html

Martin Fowler discusses the definition of a code smell: a "surface indication that usually corresponds to a deeper problem in the system".

[18] Free Software Foundation Inc, "What is free software?" accessed April 2015. [Online]. Available: https://www.gnu.org/philosophy/free-sw.html

The 'Free Software Definition' lists the famous four essential freedoms for software to be classed as "free software". This is important to consider as the GPL licenses were designed to help force adherence to these freedoms. As most of my project is covered by the GPL, anyone can apply any of these freedoms to my software.

[19] C. J. Giaschi, "Collisions." [Online]. Available: http://www.admiraltylaw.com/collisions.php

A collection of over 30 maritime collision case studies.

[20] J. Goodman, "The Pros and Cons of Online Dispute Resolution: An Assessment of Cyber-Mediation Websites," pp. 1 – 16, 2003. [Online]. Available: http://scholarship.law.duke.edu/dltr/vol2/iss1/2

This paper explores the process of cyber-mediation in terms of fully automated cyber-negotiation, cyber-mediation using sophisticated software and a neutral third party facilitator, and traditional mediation using online technologies. It shows the existence of platforms which uses AI to indicate when both parties are 'in range' of a settlement, and also AI which "attempts to generate improvements in order to maximize the benefits to both parties."

[21] D. Leal, "How you can implement living documentation (part three in a series)," 2014, accessed April 2015. [Online]. Available: http://blog.mojotech.com/how-you-can-implement-living-documentation-part-three-in-a-series/

David discusses the use of Cucumber features as a living documentation for the codebase. They key phrase here is 'living documentation', which is a philosophy that I've really taken to and try to apply to my projects wherever possible.

[22] Legal Information Institute, "Alternative dispute resolution," accessed April 2015. [Online]. Available: https://www.law.cornell.edu/wex/alternative_dispute_resolution

Alternative Dispute Resolution ("ADR") refers to any means of settling disputes outside of the courtroom.

[23] C. Lema, "What are you paying for when you buy gpl themes and plugins?" 2014, accessed April 2015. [Online]. Available: http://chrislema.com/gpl-themes-plugins/

Chris Lema helped to answer my internal battle of 'why would anyone PAY for a GPL-licensed plugin?'. He suggests that people are not paying for the plugin, but for the latest updates and the continued support of the developer.

[24] B. Mabey, "Imperative vs declarative scenarios in user stories," 2008, accessed April 2015. [Online]. Available: http://benmabey.com/2008/05/19/imperative-vs-declarative-scenarios-in-user-stories.html

Ben Mabey compares and contrasts two different user story styles. Imperative style is very detailed, so much so that it actually loses clarity. Declarative style is less specific, thus easier to read, but often means muddling the step definitions.

[25] D. Martic, "Online Dispute Resolution for Cloud Computing Services," vol. Vol-1105, 2014. [Online]. Available: http://ceur-ws.org/Vol-1105/paper2.pdf

Dusko Martic discusses the EU parliament's decision to vote in favour of the proposal to regulate consumer disputes using ADR and ODR. Though the paper refers mainly to disputes arising out of cloud computing services, it also sets the scene in a wider context for ODR itself, discussing the two main types of ODR: adjudicative (online arbitration) and consensual (mediation and assisted

negotiation). A question is raised as to how to value disputes from an ODR aspect where disputes arise from free online services, since frivolous claims should not be catered for but legitimate claims where there is "considerable economic exploitation of users" are not yet being properly addressed.

[26] R. C. Martin, *Clean Code - A Handbook of Agile Software Cratsmanship*. Prentice Hall, 2008, pp. 53–74, 978-0-13-235088-4.

Clean Code is essential reading for any software engineer in the making. I particularly enjoyed and agreed with the section on comments, formatting, and the 'Don't Repeat Yourself' section.

[27] T. Maslen, "@chrisbashton other repos yes, public not i'm afraid. ping us an email to remind me about this monday. maybe something we can do," 18 Apr 2015, 13:49 UTC. Tweet. [Online]. Available: https://twitter.com/tmaslen/status/589425600114388992

I know that we used the header-manipulation technique to run the Cucumber tests for the In-Depth Toolkit, but the repository for that is private. Tom Maslen, tech lead at BBC Visual Journalism, was able to confirm this via Twitter.

[28] MediatorAcademy.com, "Modria - the operating system for odr mediator - colin rule," 2015, accessed April 2015. [Online]. Available: https://www.judiciary.gov.uk/wp-content/uploads/2015/02/colin_rule_modria_os_for_odr.pdf

This is an interview between the founder of MediatorAcademy.com and the founder of Modria.com, Colin Rule. Colin discusses the history behind Modria, the kind of online disputes it solves today, and where ODR might be heading. In terms of the kind of disputes they're solving today, it's mainly high-volume, low-value disputes that arise from e-commerce, but they also resolve divorces and property tax assessment appeals. He claims that the best application of the technology is high volume, low value cases, since multi-million dollar cases are valuable enough to justify getting a lawyer and going through the court process.

[29] modria.com, "Home - modria," accessed April 2015. [Online]. Available: http://modria.com/

Modria's website has the following quote: "The team behind Modria already helped companies like eBay and PayPal solve more than 400 million cases." Modria's Resolutions Console "automates your business rules for every kind of customer, from high value to high risk. You can even change your policies on the fly and see how those changes will affect your bottom line."

[30] E. S. Raymond, *The Cathedral and the Bazaar*. O'Reilly Associates.

Eric Raymond discusses the world of open-source in detail, including the motivations behind open-source, how open-source can still provide commercial revenue streams, and so on. I was introduced to it through the *open-source Development Issues* module at university, and many of its concepts have stuck with me.

[31] J. Shore and S. Warden, *The Art of Agile Development*. O'Reilly Media, 2010. [Online]. Available: http://www.jamesshore.com/Agile-Book/test_driven_development.html 0596527675.

This excerpt from James Shore's book outlines a good strategy to adopt when developing code in a test-driven way. James provides an example implementation of TDD in a very in-depth way, which helped to clarify certain steps in TDD and show the benefits of applying it.

[32] SQlite, "Features of sqlite," accessed April 2015. [Online]. Available: https://www.sqlite.org/features.html

Contains suggested uses for SQLite, including: "Stand-in For An Enterprise RDBMS. SQLite is often used as a surrogate for an enterprise RDBMS for demonstration purposes or for testing. SQLite is fast and requires no setup, which takes a lot of the hassle out of testing and which makes demos perky and easy to launch."

[33] A. Venkatraman, "Amazon web services 'growing fast'," May 2014, accessed April 2015. [Online]. Available: http://www.computerweekly.com/news/2240219866/Case-study-How-the-BBC-uses-the-cloud-to-process-media-for-iPlayer

The article discusses the transfer of BBC's iPlayer to the cloud using a system called VideoFactory, which is built on AWS architecture including Amazon Simple Queuing Service and Amazon Simple Notification Service.

[34] W3Techs, "Usage of server-side programming languages for websites," 2015, accessed April 2015. [Online]. Available: http://w3techs.com/technologies/overview/programming_language/all

W3Techs statistics suggest that PHP is the server-side language of choice for 82% of all websites.

[35] ——, "Usage statistics and market share of wordpress for websites," 2015, accessed April 2015. [Online]. Available: http://w3techs.com/technologies/details/cm-wordpress/all/all

W3Techs statistics suggest that WordPress is the CMS of choice for 23% of all websites which use a CMS.

[36] M. S. A. Wahab, E. Katsh, and D. Rainey, Eds., *Online Dispute Resolution: Theory and Practice*. The Hague, Netherlands: Eleven International Publishing. [Online]. Available: http://www.ombuds.org/odrbook/Table_of_Contents.htm

Online Dispute Resolution: Theory and Practice is a comprehensive sourcebook about ODR and the chapter 'Artificial Intelligence and Online Dispute Resolution' is particularly relevant to this project. An interesting problem raised from AI solutions, particularly those that use "game theory as a basis for providing intelligent negotiation support", is that the algorithms employed are usually only fair in terms of each disputant's desire being met, and not in terms of justice. An example provided by the paper is that in a Family Law dispute, if the parents were only interested in their own desires and not the interests of their children, game theories typically would not promote the interests of the children. The limitations of SmartResolution's maritime collision module and from any modules that might employ game theory as its core AI are hopefully overcome by the fact that each agent in the dispute is a lawyer, rather than the aggrieved disputants themselves.

[37] R. Young, “Interview with linus, the author of linux,” 1991, accessed April 2015. [Online]. Available: http://www.linuxjournal.com/article/2736

Linus Torvalds, creator of the Linux kernel, gives his views on the future direction of Linux in this interview. The relevant part for the purposes of this paper is his justification for using the GPL as his choice of license.