

Carers Scheduling Prototype Software Application

John O'Grady

April 2017

SUBMITTED IN PART FULFILMENT
FOR THE HIGHER DIPLOMA IN COMPUTING
SCHOOL OF INFORMATICS AND ENGINEERING,
INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN,
DUBLIN, IRELAND

Author:
JOHN O'GRADY

Supervisor:
DR. MATT SMITH



Abstract

This project is concerned with requirements gathering, planning and delivery of a prototype rostering and scheduling software application for the Irish Wheelchair Association (IWA), a large non profit service provider which needs a new organisation wide software process to manage the planning of care appointments for people with disabilities. This project begins by setting the organisational context for the planned new software initiative. Next, it progresses to examine the body of current academic research in software delivery to provide a brief synthesis of best practice in the planning, implementation and deployment of modern software applications.

Based on this research, this author selects the Systems Development Life Cycle (SDLC) methodology as an appropriate toolset to determine the requirements of IWA for a new rostering and scheduling solution which the organisation plans to deploy to its 1,500 Personal Assistants. Having gathered the requirements, the project progresses to define the organisational context for the software deployment.

Using the SDLC methodology the author, who is responsible for leading the Information Technology function at IWA, works collaboratively with key project stakeholders across the organisation to define a broad functional requirement specification and produces various key design artefacts.

As the development work on the project progresses, a suite of SDLC mandated project management documents are iteratively refined and finalised. A high level test plan is developed and implemented to ensure code quality and alignment with user expectations and requirements. Various rework is undertaken to address issues and shortcomings identified through the testing process.

Finally, in conclusion, the author offers some insights from his experience of the software planning and development process together with consideration of some areas for further investigation and ongoing improvement of the prototype which are outside the scope of the initial project build.

Contents

1	Introduction	4
2	Organisational Context	5
2.1	History of Irish Wheelchair Association	5
2.2	Overview of Assisted Living Service	6
2.3	Current Scheduling Process	7
2.4	Overview of Organisational Impacts	8
2.5	Procurement and Tendering Approach	9
3	Literature Review	10
3.1	Software Development Paradigms	10
3.2	The Waterfall Methodology	10
3.3	Systems Development Life Cycle Methodology	11
3.4	Spiral Model	12
3.5	Agile	14
4	Initiation and Concept Development	17
4.1	Scope of Final Rostering and Scheduling Solution	17
4.2	Scope of Prototype Solution	18
4.3	Feasibility Review	19
5	Requirements Analysis	20
5.1	Functional Requirements	20
5.2	Availability Requirements	21
5.3	Security Requirements	21
5.4	Use case specification	22
5.5	Detailed Use Case Analysis	22
6	Design Specification	35
6.1	System Design Document	35
6.1.1	Hardware Architecture- Prototype Application	35
6.1.2	Hardware Architecture- Production Application	35
6.2	Software Development Document	35
6.2.1	Design considerations	35
6.2.2	Model View Controller	35
7	Testing Specification	37
7.1	Testing Approaches Used	37
7.1.1	Unit Testing	37
7.1.2	Functional Testing report	39
7.1.3	User Acceptance Testing report	44
7.1.4	Overview	44
7.1.5	Outcomes from User Acceptance Testing	45
7.1.6	Final comments	45

8	Implementation	47
8.1	Implementation Plan	47
8.1.1	Evaluation of Different Web Development Frameworks . .	47
8.1.2	Choice of a Web Development Framework	48
8.1.3	Examples of some code efficiencies available in Symfony .	49
8.2	Entity Relationship Diagram	53
8.3	Code Organisation for Project	55
8.4	Detailed Implementation Plan	56
8.5	Web Design Approach	57
8.5.1	Step1 : Symfony templating	57
8.5.2	Step2 : Base Template Inheritance	59
9	Evaluation	67
9.1	User Feedback	67
9.2	Further Enhancements	67
10	Conclusions	67
10.1	Review of material covered	67
10.2	Further Development Opportunities	67
10.3	Outstanding Issues/Continuous Improvement Plan	67
11	Appendices	67
11.1	Code Listing	67
11.1.1	Database creation scripts	67
11.1.2	Entity Classes	67
11.1.3	Mapping Extension Classes	67
11.1.4	Repository Classes	67
11.1.5	Form Type	67
11.1.6	Controller Classes	67
11.1.7	Twig Templates	67
11.1.8	List of Vendor tools used	67

1 Introduction

This project examines the requirements for a prototype Rostering and Scheduling solution for which the ultimate client, the Irish Wheelchair Association (IWA) has a real world business requirement. It is envisaged that the finished software deliverable at the conclusion of this project will not be a fully enterprise ready software solution but rather the project deliverable will be employed as a prototype for user acceptance testing by representative end users. This software development process will be used to examine and further refine IWA's existing assumptions in relation to the imminent deployment of a critical real life deployment whose successful implementation is considered to be fundamental to the competitive position of the IWA.

The scope of this project is to establish the detailed software functionality requirements for the new application and to utilise this information to deliver a prototype software solution for evaluation purposes which will cover the primary use cases and functional requirements identified. This will enable end users to test, assess and give feedback on the prototype and it is envisaged that this process will be constructive in mitigating the risk of any significant functionality components being omitted from the final production system or the usability of the ultimate solution not meeting user requirements in full.

Following this iterative process, it is planned to document a more comprehensive set of final functional requirements for the production deliverable will fundamentally underpin a competitive tender exercise and vendor engagement through which IWA will select, configure and implement a live rostering and scheduling solution for its 1,500 Personal Assistants across Ireland.

2 Organisational Context

2.1 History of Irish Wheelchair Association

The Irish Wheelchair Association (IWA) is a vibrant independent organisation which was founded in 1960 by a group of people with disabilities. IWA is governed by a Board of Directors elected from its 20,000 membership base. It is the largest provider of Assisted Living Services in the Ireland, employing over 2,600 employees and delivering over 2 million hours of service annually. It is substantially funded by the Irish Government, primarily through the Health Service Executive, from whom it receives funding as a Section 39 agency under the Health Act 2004, although it also receives funding from various other statutory and non statutory sources and raises funds directly from the general public. A challenge which is directly pertinent to the planned software project is that IWA's funding revenue streams are primarily remitted in respect of service level agreements for the delivery of front line services and it receives no direct funding for Information Technology or indeed other Shared Services activities.

During the recent economic downturn and its impact on the public finances, IWA, like many non service providers, has had seen the government apply substantial cumulative funding cuts between 2008 and 2014. Against this backdrop, IWA has managed to maintain and in some cases expand its level of service activity through pursuing internal efficiencies and implementing a series of pay reductions which have been agreed with its workforce. However, maintaining service delivery in the face of significant funding reductions has progressively diminished the financial reserves that the Association holds.

These funding reductions have also significantly impacted on the state of IWA's Information Technology infrastructure which is now in need of attention following a sustained pattern of historic under investment. The organisation's strategic plan for the years 2017 to 2010 now views Information Technology initiatives as a core enabler of driving overall efficiency and value for money in its model of service delivery, on the strict understanding that approval of all capital investment decisions must be clearly linked to a defined and fully costed business case which will deliver an identifiable and measurable return on investment in net financial terms to the organisation.

IWA has been at the forefront of developing person centred service provision in Ireland, based on international best practice, and IWA is now somewhat unique among large charities in Ireland in that it remains wholly owned by, and accountable to its 20,000 members who are made up of people with disabilities, active volunteers and other IWA supporters. In all areas of its activities, IWA advocates for independence and quality of life for all people with disabilities in Ireland. In this regard, the stated mission of IWA has recently been updated in its new Strategic Plan for 2017-2020 as follows:

Irish Wheelchair Association has a vision of an Ireland where people with disabilities enjoy equal rights, choices and opportunities in how they live their lives, and where our country is a model worldwide for a truly inclusive society.

In addition to Assisted Living Service, which is IWA's largest service employing 1,700 of IWA's 2,600 strong workforce, IWA provides a range of other services including a network of 57 community based Resource and Outreach Centres, respite services, driving tuition and a variety of member led youth and local branch projects. IWA Sport is a subsidiary division of IWA which is a recognised National Governing Body of Sport by the Irish Sports Council and IWA operates a network of volunteer led Sports clubs throughout Ireland which are an vibrant component of Ireland's Paralympic movement, particularly with regard to the development of young disabled athletes. IWA also has developed specialist teams in the specialist areas of Accessibility, Housing and Transport where it is widely acknowledged as an expert in providing expertise and advocacy in ensuring public and private developments are configured to meet the needs of people with disabilities.

2.2 Overview of Assisted Living Service

IWA's Assisted Living Service (ALS) provides significant individual supports to people with disabilities which are tailored to the needs and wishes of each person to enable them to live independently. IWA, with funding support from the Health Service Executive, provides a Personal Assistant (PA) to assist with tasks that the person with a disability might find difficult or impossible to do in their daily lives. Traditionally, people with disabilities have been treated under the medical model of care, with negative impacts for their independence and the degree of control they exercised over their own lives. As (**brisenden**) has outlined

The medical model of disability is one rooted in an undue emphasis on clinical diagnosis, the very nature of which is destined to lead to a partial and inhibiting view of the disabled individual.

Originating in the international Independent Living movement, Assisted Living is an alternative, person centered philosophy which postulates that people with disabilities are best placed to make determinations on their own needs. In keeping with this, IWA's Assisted Living Service (ALS) provides support to individuals in their homes and communities facilitating community participation, access to education, employment and improved quality of life.

The ALS model of service delivery comes in two main strands

- **Self-directed or leader-managed package.** In a self-directed or leader-managed package, the person with the disability acts as the leader or service manager for IWA. This involves recruiting their own personal assistants, organising their weekly rosters, returning their timesheets, arranging holiday cover, etc. The leader can consult the service coordinator when necessary.
- **Supported package.** In the supported package, the service coordinator takes responsibility for some or all of the management, delivery and operation of the service.

2.3 Current Scheduling Process

IWA does not currently have a unified scheduling, rostering and time attendance system in place across all of its ALS locations. A custom static (non calendarised) solution for roster and timetable planning functionality has been developed on its Microsoft Dynamics CRM environment and is currently in use across all IWA offices and a calendar based extension of this is currently in a pilot phase in a small number of IWA locations. It is likely that IWA will migrate management of schedules and rosters to the new Rostering and Scheduling platform however it should be noted that IWA does envisage continuing to use Dynamics CRM for other a variety of other key service management functions such as on-boarding service users, employee recruitment, risk assessments, evaluations and logging contact and activity information with employees and service users.

While Microsoft Dynamics CRM has provided some basic functionality in relation to static rosters to date, in the absence of a fully calendarised roster solution across all IWA offices, local teams in IWA have developed a variety of long-standing and ad-hoc local solutions to managing planning rosters, all of which IWA wishes to discontinue in favour of the proposed new solution.

The current IWA process for capturing time and attendance information for payroll and billing purposes involves each PAs completing paper timesheets which are progressively signed off by the service user or a family member throughout the month at each service visit.

At the end of the payroll month, the timesheets are delivered to the local coordinator at the local IWA office and are then data entered into the Focal-Point system. These timesheets are initially entered as pending approval by an ALS Administrator within the system and routed to the ALS Coordinator for the service for approval. The coordinator reviews the timesheet against the expected visits and service budget and approves or amends the timesheet.

Once the Focal-point process is complete, an export of the approved timesheet data is taken from Focal Point and used to feed the Mega pay payroll system. The Mega pay application in turn feeds the Access Accounts system for invoicing/billing purposes. In the context of implementing a new rostering and time attendance system, IWA wishes to cease using Focal-point for the capture of Time and Attendance information and approval of same in favour of the new rostering and scheduling system directly managing the capturing and approval of time and attendance data which can then be exported directly to Access Accounts for billing purposes and Megapay for payroll processing.

2.4 Overview of Organisational Impacts

The casual reader of the preceding section will quickly appreciate the inherent inefficiencies of the current paper centric process in an organisation of the size of IWA. Indeed, IWA employs a service management cohort of 26 Service Coordinators, 8 Service Support Officers and 20 Administrators across 15 ALS offices around Ireland and it has been estimated that across these employee groups, approximately 25% of their working time is spent managing rostering and scheduling functions in relation to the ALS service to ensure all service visits are covered and a further 25% of their time is currently consumed in manual data entry and approval tasks in relation to timesheet and payroll information.

This inefficient process has a direct impact on IWA's cost base and also has an indirect opportunity cost impact by limiting the available time of Service Coordinators to spend on other essential functions such as planning, evaluation, training and supervision activities. In this context, it should be noted that each IWA Service Coordinator is responsible for managing a large number (between 50-100 per Coordinator) of service packages for individual service users and each also supervise a similar number of Personal Assistants for whom the Service Coordinator acts as line manager. This also has an impact on the ratio of required back office personnel to service delivery hours and due to the somewhat manual rostering and scheduling process currently being operated, IWA's service coordination and administration costs per service delivery hour are somewhat higher than some of its competitors, placing it at a competitive disadvantage, particularly in comparison to some of the private sector commercial operators who have recently entered the Irish home care market and who in many cases currently have more sophisticated technological solutions in place to handle this key internal process.

2.5 Procurement and Tendering Approach

IWA has recently launched the first phase of the competitive tender process for its new rostering and scheduling solution through the Office of Government Procurement's E-Tender's website which is available at this link [The first stage of the procurement process requires prospective vendors to complete a short pre-qualification questionnaire which provides an opportunity for vendors to demonstrate the capabilities of their software platform against the high level requirements summarised in the preceding section, as well as providing an overview of their organisational capability and customer references where they have already deployed their solution in a similar usage context. Following assessment of the pre-qualification questionnaires, IWA will shortlist a small number of interested vendors for the second phase of the tender process which will require vendors to submit a more comprehensive response against a detailed Request for Tenders \(RFT\) document provided to the vendors by IWA. Once a successful platform/vendor meeting all of IWA's mandatory requirements as set out in the RFT document has been appointed through the procurement process, IWA will work with that vendor to configure and test the system to align the chosen platform which IWA's requirements. It is envisaged that IWA will initially implement the solution for a small group of 50-100 Personal Assistants who will take part in a pilot exercise to confirm, test and sign off on the chosen solution as fit for purpose prior to its rollout to the wider group of 1,500 Personal Assistants and 25 Service Coordinators.](#)

3 Literature Review

3.1 Software Development Paradigms

In this section, we complete a rapid tour through the body of academic literature to review different software development paradigms and select an appropriate methodology for the prototype project. We begin by looking at the Waterfall Approach to software development before examining some alternatives. Many computer scientists regard the Waterfall methodology as the classical approach to software development while the Agile methodology and related approaches have been gaining increasing traction in recent times and can be reasonably positioned as the more modernist or fashionable approach, especially for projects which are likely to need the capacity for rapid adaptation.

3.2 The Waterfall Methodology

In the 1960s and early 1970s, the largest customer worldwide for software development was the U.S. Government's Department of Defense which managed a sophisticated and critical portfolio of software systems and projects which was probably without rival in terms of its scale at the time. (**royce**) is often credited as being the first to use the Waterfall development as a term to describe a planned top down approach to software development which moves incrementally through a series of rigid steps which cannot be revisited once completed. However, others have disputed that Royce was the first to use the term and also suggested that Royce did not envisage a rigid approach to development, noting that Royce was in favour of an iterative approach which involved the capacity for the rework of outputs based on the experience of using outputted artefacts in subsequent development steps, though only within successive steps. Royce also felt that there was considerable risk in the fact that the Waterfall model only envisaged testing of the outputs of the project as a penultimate activity when the project was close to completion. Royce's model of Software Development suggests the following incremental steps in developing new software systems

- Systems requirements
- Software requirements
- Analysis
- Program design
- Coding
- Testing
- Operation

3.3 Systems Development Life Cycle Methodology

More recent commentators have adapted the foregoing model to add some additional steps and the following framework is frequently used in establishing a structure for project teams engaged in software development, though there are various alternate models also in use which use similar overall concepts:

- **Initiation.** In this stage, the project's life begins when a project sponsor recognises the need or opportunity for a software project to take place.
- **System Concept Development.** At this juncture, the team defines the scope for the project and engages in risk analysis activities, defining boundaries for the project and undertaking feasibility studies.
- **Planning.** Here, the team develops a project management plan and sets about acquiring the resources needed for the project.
- **Requirements Analysis.** Now the project is underway, and the project team work to establish what the user requirements are, using this knowledge to define a detailed functional requirements document.
- **Design.** This stage sees the requirements transformed into a detailed Systems Design Document which begins to examine and plan the technical approach which will be employed to deliver the required functionality.
- **Development.** This begins the actual technical delivery, where the team convert the design documents into realisable code and system resources including database creation, preparing test cases and beginning the installation, coding and assembly/ compilation of a live software deliverable
- **Integration and Test.** The focus in this stage is to ensure that the system under construction aligns with and meets user expectations in relation to the system's functional requirements. These assumptions can be tested through user acceptance testing and users undertaking quality assurance activities. It is best practice here that the test activities should be formally documented.
- **Implementation.** Now that we have a system in place, it needs to be deployed to a production environment and where necessary integrated with other relevant software and hardware resources.
- **Operation & Maintenance..** This stage defines a series of operational tasks and procedures to operate, maintain and fine-tune the information system in a production scenario.
- **Disposition.** This stage at the end of the project focuses on the end of system activities, with a particular emphasis on the data aspects of projects.

The Waterfall model gained wide acceptance as a framework for software development, though controversy raged continually about its effectiveness and appropriateness to large scale projects. In order to increase the speed of delivery and in response to increasing complexity and project risk, the Department of Defense looked to computer scientists in academia in order to assist it in developing more reliable and predictable software development methodologies.

In an influential publication, (**defensesscienceboard**), reporting to the U.S. Military as the Defense Science Board Task Force on Military Software, highlighted increasing risks associated with more traditional software development approaches including the Waterfall approach. This report noted that the cost of military software contracts had been steadily increasing, while the time to delivery was also becoming longer. It highlighted a variety of mitigation strategies which it recommended to the Department of Defense including standardising programming languages (using the DoD mandated Ada language) as well as a renewed focus on requirements gathering and an more iterative approach to managing software projects.

A more recent study (**peterson**) found that the major issues with the Waterfall model arise in relation to the requirements gathering activities, and also present in relation to the verification of same. Based on this, it concluded that even though it was still commonly in use, the Waterfall model was fundamentally unsuitable for large scale or complex development projects

3.4 Spiral Model

Noting that the Waterfall approach had become the de facto standard for military (and general) software development, and echoing the shortcomings noted by the Defence Science Board report, (**boehm**) proposed an alternate framework which he termed the Spiral model.

Because the Waterfall approach placed a strong emphasis on the finalisation and sign off of detailed specification criteria prior to moving on to the development and implement stages, Boehm noted that this highly structured approach frequently served to act as a barrier to early prototyping. Boehm contrasted this with the Evolutionary Development approach (where a user might declare ‘I cant tell you what I want, but Ill know it when I see it.’), which had become popular as an antidote to the rigid strictures of the Waterfall approach. He also observed that using Evolutionary Development also presented some significant challenges in that it could frequently be difficult to distinguish this approach from the older and less structured ‘Code and Fix’ or ‘Spaghetti code’ implementations which have been repeatedly found to have performed very poorly as they scaled to larger implementations. Figure 1. below shows an overview of the Spiral model of development

To address the shortcomings of both methodologies, Boehm proposed the Spiral model which he noted as evolving based on the insights he had gained from applying the Waterfall model in large government projects. It suggests a flexible approach based on the risk profile of each project which adopts suitable tools from one or more process based methodologies such as incremental development, the waterfall model, or evolutionary prototyping. A challenge with the Spiral model is that its core concepts have often been oversimplified leading to misconceptions and what Boehm terms ‘hazardous spiral look-alikes’. This term refers to implementations of the Spiral approach which appear to have the all the required core components but in fact diverge and inviolate the one or more of the key principles of the model.

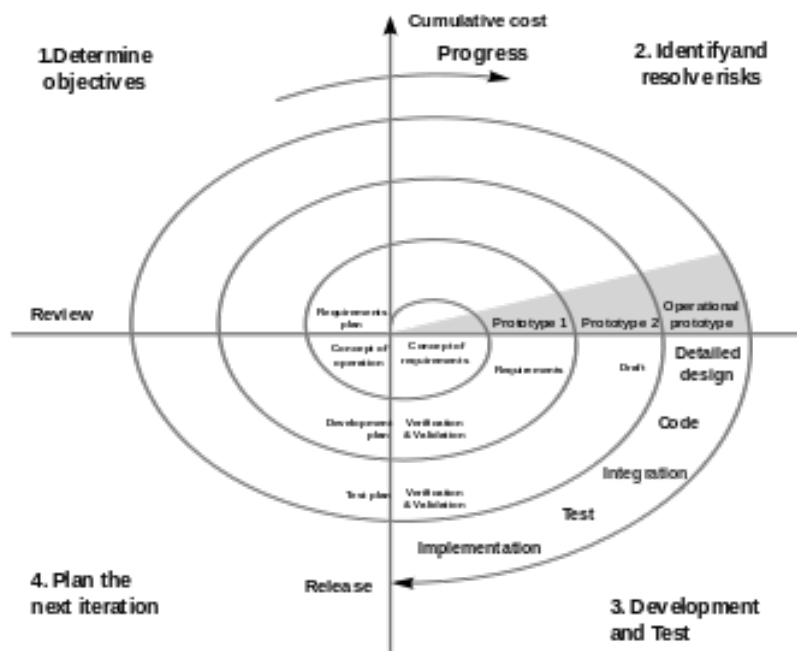


Figure 1: The Spiral model of development

To counteract this approach, Boehm stresses that projects using the spiral model should always display six invariants (items which are always required in every project) and he usefully provides examples where each invariant could be subject to an incorrect interpretation which would invalidate its meaning.

- **Define artefacts concurrently.** This invariant notes that concurrency is a better approach here as defining project artefacts sequentially increases the risk that the project does not meet stakeholder expectations.
- **Perform four basic activities in every cycle.** At its core, the Spiral model emphasises four basic activities in each cycle of development as follows:
 - 1. Determine the objectives
 - 2. Identify and resolve risks
 - 3. Development and test
 - 4. Plan the next iteration
- **Risk determines level of effort.** It is the responsibility of the project team to determine how much effort should be spent on each project area, based on the perceived level of risk for the area. This decision should always be made based on a strategy of reducing the overall level of risk to the project.

Table 1: The Agile Manifesto

Individuals and interactions	over	Processes and tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contract negotiation
Responding to change	over	Following a plan

- **Risk determines degree of details** Based on a risk based approach, the team must make a determination as to how much detail needs to be gathered for each project artefact. For example, they should ensure they gather sufficient detail in relation to requirements areas where a detailed specification helps to reduce unpredictability and contribute to lowering the overall level of risk, for example in precisely defining the integration approach to be taken between hardware and software components. Conversely, project teams should feel entirely comfortable producing much less detail in relation to front requirements specifications in other areas which have a lower overall level of project risk, for example in relation to the design of the graphical user interface.
- **Use anchor point milestones.** In the original specification of the spiral model, it did not initially include any project milestone assessments but from practical experience- again with a risk focused approach- these were introduced to anchor project delivery to communicate progress updates to stakeholders.
- **Focus on the system and its life cycle** This invariant recommends that project managers and stakeholders take a long term view of the project life cycle and avoid an excessive focus in the initial stages of the project on the development of software code.

3.5 Agile

The Agile manifesto emerged as a strong counterpoint to the various process driven approaches such as the Waterfall methodology and echoed many of the concerns and ideas of earlier iterative and incremental software development approaches.

The manifesto was written by 17 experienced software developers who convened in 2001 and drawing on their own experience of efficient software development principles, sought to define the guiding principles for the Agile Alliance. Central to their thinking was the notion that while they valued the items on the right hand column of the table above, they placed an even higher value on the items in the left hand column. For example, while they did see a value in developing documentation, they did not take this to the level of generating copious manuals for form's sake which would gather dust, not be kept up to date, and ultimately deliver little in terms of value to end users.

The twelve principles of the Manifesto for Agile Software Development, which are reproduced verbatim below, have found wide acceptance among software developers, particularly in recent years as the time to market between releases in new areas of software development including mobile apps and on-line web applications needs to be much shorter than in traditional application development.

The **Manifesto for Agile Software Development** is based on twelve principles:

- Customer satisfaction by early and continuous delivery of valuable software
- Welcome changing requirements, even in late development
- Working software is delivered frequently (weeks rather than months)
- Close, daily cooperation between business people and developers
- Projects are built around motivated individuals, who should be trusted
- Face-to-face conversation is the best form of communication (co-location)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Continuous attention to technical excellence and good design
- Simplicitythe art of maximizing the amount of work not doneis essential
- Best architectures, requirements, and designs emerge from self-organizing teams
- Regularly, the team reflects on how to become more effective, and adjusts accordingly

Where traditional software houses could plan for new versions of their software to be released perhaps on an annual basis, which allowed sufficient time for testing, feedback and rework of code from testing to eliminate bugs, modern application development needs to focus on regularly adding new functionality. The 'bite-sized' approach of Agile is a good fit for these rapid development requirements, allowing software development houses to iteratively plan and develop and test new releases of working software while allowing for frequent application updates to be responsive to customer needs, competitor actions or market conditions, often completing a development cycle within a short period of a few weeks.

These Agile principles have inspired many further iterative innovations in the field of software development giving rise to other approaches such as Extreme Programming, Lean Software Development (which has many parallels with Toyota's Lean Manufacturing System for vehicle manufacturer) and also resources and tools such as Kanban Boards, Scrum and DevOps tools. Some of the core project management methodologies such as PMBOK (The Project Management Book of Knowledge) and PMI (Project Management Institute) have also adapted to the emergence of Agile principles by defining specific Agile oriented project management methodologies. There are also mainstream training and certification options available for those who wish to pursue careers as Agile practitioners or Scrum masters.

4 Initiation and Concept Development

4.1 Scope of Final Rostering and Scheduling Solution

In order to address and remedy the deficiencies noted in the earlier Organisational context section in relation to the inefficient processes currently being used to manage the organisation's rosters and scheduling processes, the IWA's Senior Management Team have mandated the IWA ICT team to work with internal stakeholders to define functional requirements and implement a competitive tender process to select and deploy a new software solution to enable IWA to manage rosters and the capture of time and attendance information in a more efficient manner. The proposed solution will deliver the following functionality components to IWA:

- **A Rostering and Scheduling solution** for use by ALS Administrators, Coordinators and Support Officers. The proposed solution should also interface with the IWA Megapay payroll system. Finally, it should manage the customer billing process in relation to the care services provided to statutory and individual customers.
- **A Mobile Application** to be used by PAs employed by IWA which should be capable of running on the iOS, Android and Windows Phone and thereby be suitable to run on the personal mobile devices of employees to avoid IWA having to supply company owned and funded devices on the corporate account.
- **Attendance Verification Mechanism via Mobile App.** A key requirement for the mobile application is that it provides a reliable and independent real-time verification of an employee's attendance at a service visit location, together with timestamped confirmation of the length of time that was spent at the location. This is required to satisfy IWA service level agreement obligations with its funders and to minimise the possibility of fraud.

-
- **Quotation/Proposals Generation.** The proposed enterprise solution should be capable of generating detailed and personalised Service proposals/ quotations where the coordinator can plan the service schedule for the service user, referencing each visit to the appropriate price card item to generate a completed quotation for the customer which shows the provisional service schedule and the projected weekly invoiced cost.
 - **Employee, Service User and Customer Portals.** IWA would also like to implement Service user and Employee portals which includes an authentication layer so that a service user or their family member can view upcoming service visits. Similarly, employees can view their upcoming roster visits to various service users including information such contact information, tasks to be completed etc. and a Customer portal where a funder can view invoices, schedules for upcoming service and validate completed visits by viewing validation timestamps of attendance.

4.2 Scope of Prototype Solution

As the prototype application is being fast tracked for user acceptance testing, this application has more limited scope and is primarily focused on the backend Rostering and Scheduling solution and in particular on examining in detail the user experience and optimal process and validation checks required by Coordinators and Administrators in handling various common service delivery scenarios. The prototype will also attempt to model in a simplistic fashion the experience of portal layer users who will interact using various security limited roles such as employees, service users and customers although it will not attempt to fully implement the data privacy restrictions to be granted to each type of role- for example employees being unable to access the planned roster records for other employees to the fully robust extent that would be required in an enterprise level application. The aspects of creation of a mobile app, integration of the mobile app with the backend rosters and scheduling and the verification of attendance via the mobile app are all considered as out of scope for the prototype application.

4.3 Feasibility Review

Risk Register for Project and Mitigation Actions

Risk: It may be challenging to find a solution which meets all of IWAs requirements.

Mitigation: In this regard, it will be necessary to prioritise those essential aspects of the new systems functionality over nice to have aspects

Risk: Required project resources are not available or their input delays key project phases

Mitigation: Agreement of IWA management to release key resources when required and provide backfill for the other work normally assigned to those resources

Risk: The project takes longer to deliver or software/customisation costs are higher than anticipated.

Mitigation: Detailed project planning and agreement of deliverable, costs and timelines with vendor. Prioritisation of scope and deliverables to ensure key functionality is delivered on time.

Risk: Solution does not fully meet IWA's requirements or becomes outdated as IWA's requirements change.

Mitigation: Detailed agreement on deliverable/scope /cost for initial build and sign off of same with vendor. Comprehensive user acceptance testing and user sign off on agreed and tested functionality at key milestones. Design solution so that it is flexible to adapt to the key areas where IWA requirements may change e.g. payroll payment rate structure, billing rate structure, customer invoicing/reporting formats, additional functionality/ logic in the mobile app.

Risk: Less than 100% adoption by ALS teams, current manual processes persist at local level.

Mitigation: Dedicated project manager for rollout phase working with local teams. Tight project planning and monitoring/direction from local ALS management teams to ensure project phases and adoption take place to plan. Structured training plan for staff using system and identification of staff who are struggling to adapt to the new process and provision of support by local ALS team leads to ensure they are brought up to speed.

Risk: The prototype has limited value as it has marked differences in functionality or user interface from available production platforms which might be considered by IWA

Mitigation: Focus on confirmation of required functionality over specific user interface features. Research commercial offerings and build some aspects of their user interface into prototype so that users get a realistic sense of what the finished product might look like

5 Requirements Analysis

5.1 Functional Requirements

In this functional requirements section, we set out in simple terms what the prototype application will do.

- **Managing Service Appointments.** For this application, the core objective of the system is to manage a series of appointments for care visits and to make sure that each one is covered, taking account of the relationships between service users and personal assistants.
- **Documenting service user, customer, employee and office location and contact information.** The application should record (and show in the index and show views for each entity) the address and contact information for each employee, service user or office.
- **Displaying service user roster as calendar plan.** The application should display the roster plan for each service user in a calendar plan (similar to an Outlook diary) format and allow the user to interact with and update each visit from that calendar view.
- **Assigning service user, employee and customers to office locations.** The allow a user to indicate which IWA office is responsible for managing the relationship with a given service user, employee or customer.
- **Geocoding addresses and Displaying Google Maps.** For each service user, employee and office. the system should integrate with Google Maps to obtain valid geocoordinates for the address and display this as a map on the view screen for each record.
- **Assigning a Personal Assistant to a Service User.** A Personal Assistant (employee) will frequently be assigned as the ‘preference’ (or regular) resource to be sent to a given service user, based on a positive experience of working together.
- **Assigning a Personal Assistant as ‘Do Not Send’ to a Service User.** Conversely, a negative experience from a previous visit may give rise to the need to designate that a given resource should not be sent to fulfil care visits for a particular service user. In this case, we would indicate this relationship by creating a ‘Do Not Send’ association between the service user and the employee.
- **Employee Unavailable Times.** The system needs to document times when a Personal Assistant (employee) is not available due to other commitments e.g. second job, family, college attendance and not show this employee as available when finding an employee to fill an unfilled slot

-
- **Employee Absence.** The system needs to document times when a Personal Assistant (employee) is on leave and unavailable to work and not show this employee as available when finding an employee to fill an unfilled slot
 - **Employee Working.** The system needs to document times when a Personal Assistant (employee) is assigned to a roster and therefore unavailable to work on other rosters. It should not show this employee as available when finding an employee to fill an unfilled
 - **Roster required employees .** The system should store for each visit how many employees are required.
 - **Roster status.** When the number of assigned employees is less than the required quotient, the system should show a warning to this effect. When enough employees are assigned, the message should be updated to reflect this.
 - **Find an employee for an unassigned roster.** When the number of assigned employees is less than the required quotient, the system should allow a user to find a suitable employee to fulfil the visit, bearing in mind ‘do not send’ relationships and filtering out employees who are working elsewhere, designated as unavailable at that time or recorded as being on an approved absence (holidays, sick leave etc.) at the time of the roster. slot

5.2 Availability Requirements

The system should be available over a web browser once the user has internet access. It should not rely on a client install of software on the user’s machine nor should it require the user to use a VPN connection.

5.3 Security Requirements

- **Encryption** The system should use an encrypted (https, SSL) connection to the server to ensure privacy of the data in transmission.
- **Role Based Security** The system tag each user to the IWA office that they are assigned to, and restrict that user to only be able to see service users, employees and customers associated with the office that the user is associated with (Note: out of scope for prototype deliverable)
- **Service User Portal User- Security** The system should allow user who are attached to the Service User security role to login in and to only be able to see rosters for that service user (Note: included in scope for prototype deliverable)
- **Employee Portal User- Security** The system should allow user who are attached to the Employee security role to login in and to only be able to see rosters to which that employee is assigned (Note: out of scope for prototype deliverable)

- **Customer Portal User- Security** The system should allow user who are attached to the Customer security role to login in and to only be able to see rosters for which that customer is designated as the customer of the Roster visit (Note: out of scope for prototype deliverable)

5.4 Use case specification

The high-level use case specification aims to identify all the actors who will use a system and what actions they can take in that system.

5.5 Detailed Use Case Analysis

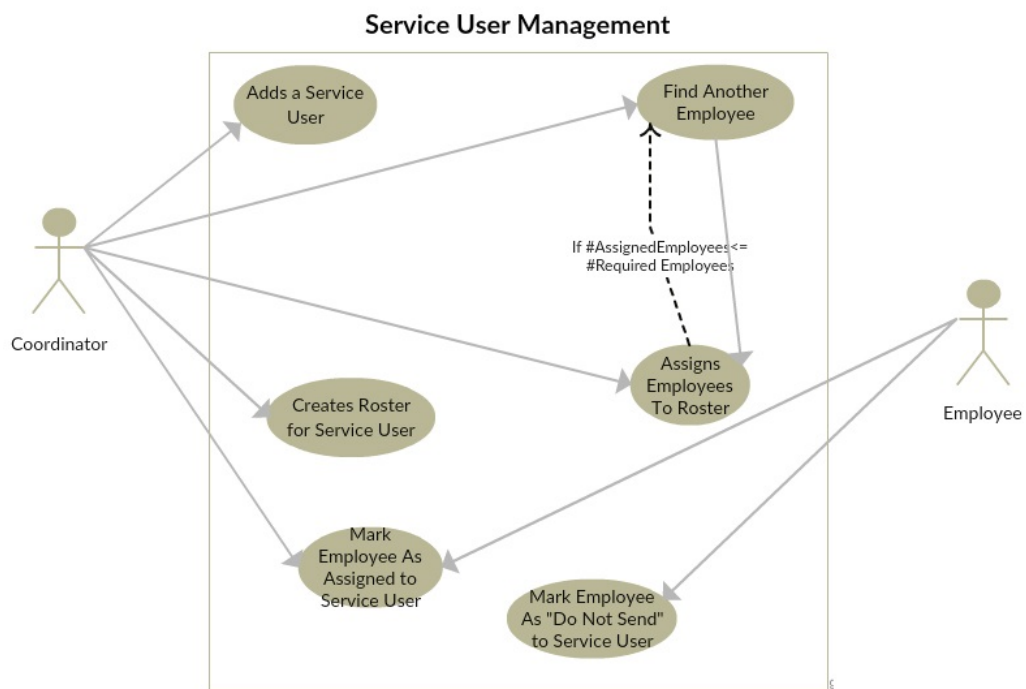


Figure 2: Roster Use Case

The following detailed use cases have been identified for the prototype application. As many of the use cases relate to standard CRUD (Create, Read, Update, Delete) functionality for each entity for which it is envisaged that the development approach will use the Symfony skeleton templating facility to initially scaffold the generic controller and related views and form type classes, a generic use case for each of these four standard functions has been created to avoid duplication and repetition. As noted, this generic use case- for example new() entity record- is therefore applicable to all of the individual entities noted, and any particular variations (for example passing parameters to identify a related entity context) are noted in the use case where appropriate

Title	User login
Description	User supplies credentials to login
Actor	User
Precondition	User must not be logged in ie is currently authenticated as anonymous user
Postcondition	User is authenticated having supplied correct password and holds one or more security role
Basic Flows	<ol style="list-style-type: none"> 1. User navigates to login area via base template button which is visible on all pages. . . 2. User enters username and password and clicks login. . . 3. Controller method checks username and password against user repository. If credentials are matched to repository logs user in and set the user as authenticate, allocating any roles held by that user 4. If credentials are not matched to repository then display login failed message and redisplay the login screen ldots
Alternate tion Flows	<div>Excep-</div> <ol style="list-style-type: none"> 1. User who is not logged in attempts to access a secured resource and is automatically rerouted to the login screen. . .

Title	User registration
Description	User a username and password and creates a new user account
Actor	User
Precondition	User must not be logged in i.e. is currently authenticated as anonymous user
Postcondition	User is authenticated having supplied correct password and holds one or more security role. User credentials are updated in the database for future visits
Basic Flows	<ol style="list-style-type: none"> 1. User navigates to register area via base template button which is shown on all pages when the user is not currently logged in ... 2. User enters username and password and clicks login... 3. Controller method checks username and password against user repository. If credentials supplied are valid, and do not duplicate an existing user account the user is set to a logged in stage and the database is updated 4. If credentials are matched to an existing user record in the repository display the registration failed message and redisplay the registration form screen ldots
Alternate Exception Flows	<ol style="list-style-type: none"> 1. None...

Title	User accesses index view of entity records	
Description	User can view index list of one of type Customer, Employee, Roster Assignment, Service User, Service User Assignment, Employee Absence Times, Employee Unavailability, Do Not Send, Office records. This is a reusable use case mapped to multiple Entities,all created using the Symfony templating approach	
Actor	User	
Precondition	User must be logged in and hold an appropriately authorised security role	
Postcondition	N/A.	
Basic Flows	<ol style="list-style-type: none"> 1. User navigates to index view and can see all customer records. . . 2. User clicks on Edit link and can view a single page view screen of the selected record. . . 	
Alternate Flows	Exception	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role. . . 2. User attempts to route to a record that doesn't exist and is shown error 404 not found. . . 3. Attempt to retrieve entity records failed at database level. Re-display previous screen and advise user that an error was encountered and suggest that they retry the effort. . . 4. User authenticates but does not hold an appropriate role to access the index view. Display 403 forbidden exception message. . .

Title	User views individual entity item
Description	User can select view an individual item of type Customer, Employee, Roster Assignment, Service User, Service User Assignment, Employee Absence Times, Employee Unavailability, Do Not Send, Office records
Actor	User
Precondition	User must be logged in and hold an appropriately authorised security role
Postcondition	N/A.
Basic Flows	<ol style="list-style-type: none"> 1. User navigates to that record and can see all attributes of that entity record on a single page...
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User attempts to route to a record that doesn't exist and is shown error 404 not found... 3. Attempt to retrieve selected record failed at database level. Redisplay index and advise user that an error was encountered and suggest that they retry the effort... 4. User authenticates but does not hold an appropriate role to view the entity item. Display 403 forbidden exception message...

Title	User can create a new entity record
Description	User can create a new record of type Customer(s), Employee, Roster Assignment, Service User, Service User Assignment, Employee Absence Times, Employee Unavailability, Do Not Send, Office records. This is a reusable use case mapped to multiple Entities,all created using the Symfony templating approach
Actor	User
Precondition	User must be logged in and hold an appropriately authorised security role
Postcondition	The newly created record is now saved to the database
Basic Flows	<ol style="list-style-type: none"> 1. User navigates to index view and clicks the new button or navigates from a related entity (see Alternate Flow below)... 2. User clicks on New link and can view a single page form screen with empty attribute controls for the selected record type...
Alternate Flows	<ol style="list-style-type: none"> 1. from service user record, user clicks on a button for new roster or assign employee or mark as do not send. Or from employee, user clicks on add unavailability time or employee absence period. User is routed to special version of the New() action for that entity which accepts either an employee or service user object as a parameter and method binds new entity record to have service user or employee context as an associated foreign key when saving newly created record)...
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User authenticates but does not hold an appropriate role to access the index view. Display 403 forbidden exception message... 3. Attempt to save new record failed at database level. Redisplay new record form with previously entered form data and advise user that an error was encountered and suggest that they retry the effort...

Title	User can edit an entity record
Description	User can retrieve and edit an existing record of type Customer(s), Employee, Roster Assignment, Service User, Service User Assignment, Employee Absence Times, Employee Unavailability, Do Not Send, Office records. This is a reusable use case mapped to multiple Entities,all created using the Symphony templating approach
Actor	User
Precondition	User must be logged in and hold an appropriately authorised security role
Postcondition	The updated record is now saved with changes made to that single entity record updated to the database
Basic Flows	<ol style="list-style-type: none"> 1. User navigates to index view and clicks the edit button ... 2. User is routed to a single page form screen with attribute controls for the selected record type prepopulated with the previously saved values retrieved from the database... 3. User makes changes to the field values and clicks update or save. The changes are validated and then saved to the database...
Alternate Flows	<ol style="list-style-type: none"> 1. The user cancels the attempt to edit the record and turns to the (non editable) form version of the entity...
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User authenticates but does not hold an appropriate role to access the index view. Display 403 forbidden exception message... 3. Attempt to save edited record failed at database level. Redisplay edit record form with previously entered form data and advise user that an error was encountered and suggest that they retry the effort...

Title	User can delete a new entity record
Description	User can delete a single entity record of type Customer(s), Employee, Roster Assignment, Service User, Service User Assignment, Employee Absence Times, Employee Unavailability, Do Not Send, Office records. This is a reusable use case mapped to multiple Entities,all created using the Symfony templating approach
Actor	User
Precondition	User must be logged in and hold an appropriately authorised security role
Postcondition	The deleted record is removed from the database.
Basic Flows	<ol style="list-style-type: none"> 1. User navigates to index view and clicks the delete button ... 2. the selected record is removed from the database...
Alternate Flows	<ol style="list-style-type: none"> 1. The user clicks the delete button when in the view/show mode and having previously selected an individual record to view/show...
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User authenticates but does not hold an appropriate role to access the index view. Display 403 forbidden exception message... 3. Attempt to delete record failed at database level. Re-display current record and advise user that an error was encountered and suggest that they retry the effort...

Title	Check for a Do Not Send relationship between an employee and a service user
Description	When saving an assignment of an employee to a roster, check first that a do not send relationship has not already been defined for that employee.
Actor	User
Precondition	User must be logged in and hold an appropriately authorised security role. This method is called when assigning an employee to a roster record
Postcondition	The roster assignment record is updated with the relevant employee entity from the database.
Basic Flows	<ol style="list-style-type: none"> 1. User assigns an employee to a record. ... 2. If no do not send relationship exists between the service user and the employee, save the selected roster assignment record to the database...
Alternate Flows	<ol style="list-style-type: none"> 1. If a do not send relationship does exists between, do not save the roster assignment and advise the user that the change has not been saved because of the do not send relationship...
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User authenticates but does not hold an appropriate role to access the index view. Display 403 forbidden exception message... 3. Attempt to save new record failed at database level. Redisplay current record and advise user that an error was encountered and suggest that they retry the effort...

Title	Display available employees for a roster
Description	When a roster does not have enough employees assigned, show available employees to the user and allow them to select one.
Actor	User
Precondition	User must be logged in and hold an appropriately authorised security role. The roster must not have sufficient employee resources attached
Postcondition	None.
Basic Flows	<ol style="list-style-type: none"> 1. User clicks on the Find An Employee To Assign button on a roster record . . . 2. The controller method retrieves all employees and then filters out employee who have a do not send relationship to that service user. . . 3. From the residual array of employee objects, the controller method filters out employee who are not available at that time due to being assigned elsewhere. . . 4. From the residual array of employee objects, the controller method filters out employee who have recorded that they are unavailable at that time. . . 5. From the residual array of employee objects, the controller method filters out employee who have an absence recorded which overlaps with the roster time. . . 6. Finally, the residual array of available employee objects are displayed to the user and the user can select an individual employee record from the displayed list and click the assign button. . .
Alternate Flows	<ol style="list-style-type: none"> 1. There are no available employees and the user navigates away from the page. . .
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role. . . 2. User authenticates but does not hold an appropriate role to access the view. Display 403 forbidden exception message. . . 3. Attempt to assign the selected employee to the roster record failed at database level. Redisplay current record and advise user that an error was encountered and suggest that they retry the effort. . .

Title	Get Mapping coordinates for address
Description	Use call to Google Maps API to retrieve geo coordinates for an address. Reusable factory method which accepts either an employee, service user or office object and geocodes the address for that object
Actor	User
Precondition	User must be logged in and hold an appropriately authorised security role. The object to be geocoded must have an address
Postcondition	The latitude and longitude of the object's address are saved to the database.
Basic Flows	<ol style="list-style-type: none"> 1. User clicks on the Geocode button from the show screen of an employee, service user or office entity record... 2. The controller method sends an appropriate API call to the Google Maps location service which returns a valid set of the coordinates for the address...
Alternate Flows	<ol style="list-style-type: none"> 1. If Google Maps cannot locate the coordinates return and save an empty of coordinate values as latitude and longitude...
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User authenticates but does not hold an appropriate role to access the geocoding view. Display 403 forbidden exception message... 3. Attempt to assign the newly obtained coordinates failed at database level. Redisplay current record and advise user that an error was encountered and suggest that they retry the effort...

Title	Assign user role(s)
Description	Assign role(s) to a user
Actor	Administrator
Precondition	User must be logged in and hold an appropriately authorised security role.
Postcondition	User role is updated against the appropriate user record.
Basic Flows	<ol style="list-style-type: none"> 1. Administrator clicks on Assign User Role button from the Admin button shown only to Administrator role... 2. Administrator selects the required role and clicks update...
Alternate Flows	<div>Exception</div> <ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User authenticates but does not hold an appropriate role to access the Administrator view. Display 403 forbidden exception message... 3. Attempt to assign the newly obtained coordinates failed at database level. Redisplay current record and advise user that an error was encountered and suggest that they retry the effort...

Title	Password reset
Description	Reset the password of a user
Actor	Administrator
Precondition	User must be logged in and hold an appropriately authorised security role.
Postcondition	New password value is updated against the appropriate user record.
Basic Flows	<ol style="list-style-type: none"> 1. Administrator clicks on Reset Password button from the Admin button shown only to Administrator role... 2. Administrator selects the user whose password needs to be reset and then enters in a new password... 3. New password is saved to the database...
Alternate Exception Flows	<ol style="list-style-type: none"> 1. User is not logged in. On routing to area, system routes User to user login screen and denies permission to resource until they have successfully authenticated using an account holding the appropriate role... 2. User authenticates but does not hold an appropriate role to access the Administrator view. Display 403 forbidden exception message... 3. Attempt to save the new password failed at database level. Redisplay current record and advise user that an error was encountered and suggest that they retry the effort...

6 Design Specification

6.1 System Design Document

6.1.1 Hardware Architecture- Prototype Application

Considerations for the Prototype Application: It is envisaged that the prototype system will be hosted initially on the author's own personal web hosting for prototyping purposes and with a limited initial user base. This will incorporate a web front end, with a successful user login required to access most parts of the application. The system will utilise a MySQL back end database and the web application will interact with this database layer through Symfony/Doctrine repository classes (see below for more information)

6.1.2 Hardware Architecture- Production Application

Considerations for the Production Application: It is envisaged that the production system will incorporate a load balanced web application server with a backend database though a private cloud hosted system is considered to be ideal for the usage scenario. Further consideration of the hardware/ infrastructure environment will depend on the specific solution chosen as a result of the procurement process and is deferred here until this has been completed.

6.2 Software Development Document

6.2.1 Design considerations

Given the requirement to rapidly complete and deliver a prototype application for user acceptance testing, the ability to deliver a basic working application in a short time frame becomes a paramount consideration. A second consideration is to ensure an appropriate separation of concerns between the front end application (the view that is presented to the user through the user interface), controller methods where the application business logic is applied (for example determining which employees are actually available to cover an unfilled roster) and lower level data level operations such as retrieving, adding or updating a record from/to the back end database.

6.2.2 Model View Controller

The **Model View Controller** approach as shown in Figure 3 below is a widely used application development approach which meets the above requirements and also helps to enforce a degree of rigour and structured design to the code base for the application.

In this approach there are three main components in the architectural pattern each of which serves a different purpose but works in conjunction with the other two components, while achieving the requirement for separation of concerns.

- **Model** represents an object with each entity's attendant data attributes
- **View** represents the presentation layer to the user of the data which is stored in the model

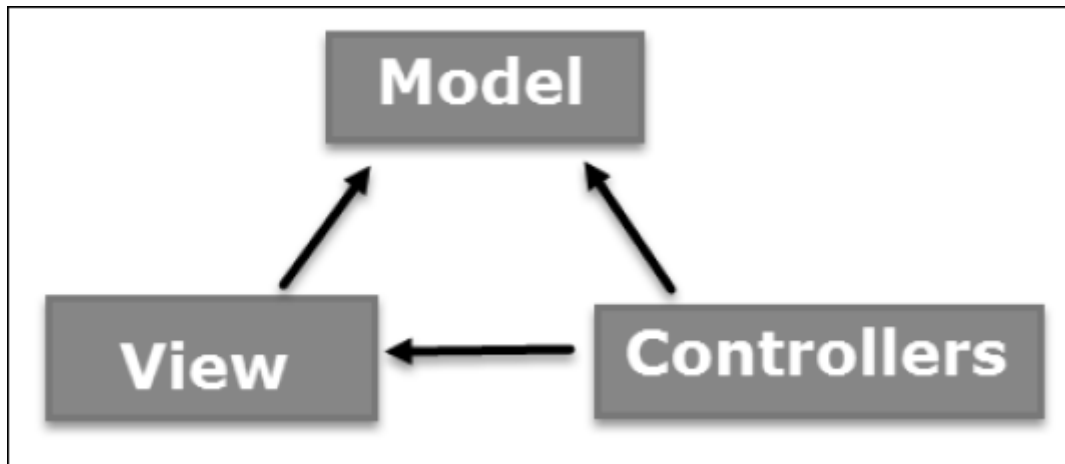


Figure 3: The Model View Controller

- **Controller** represents the connecting layer between the model and the view. It allows the data flow into the model object and keeps the presentation of the data in the view up to date when the actual data in the object changes. It can act on both the model and the view and keeps the view and the model separate from one another.

The discussion of choosing an appropriate project implementation methodology is continued in the Implementation plan later in this document.

7 Testing Specification

7.1 Testing Approaches Used

A variety of different approaches were used in testing the application to ensure that all functionality was working as per the specification and also to confirm the usability of the application through end user acceptance testing by subject matter experts within the business.

7.1.1 Unit Testing

Firstly, a comprehensive series of automated tests were developed using PHPUnit, which is a third party automated testing tool which has been integrated into the Symfony framework. This allows for a test driven approach to development where the automated series of tests can be developed and run repetitively as changes to the application are made to test that all code modules continue to successfully pass the tests and have not been impacted by the most recent changes made. This is considerably more efficient and lower in risk than relying solely on human driven functional testing. Indeed, a cornerstone of Test Driven Development is to write the automated tests first, in the knowledge that these will initially fail to run successfully and then proceeds to write the required implementation code to pass the test cases (**maxwilliams**). (**janzen**) note that the use of the term *unit* is the smallest possible testable software component, outlining that there is debate about the actual definition of the term. Some experts argue that it should be an individual model entity while others prefer to focus on individual methods within a class as the unit.

For the purposes of this project, the author has focused on grouping each unit test at an entity model test. Some other classes- for example the reference entity for employee absence reason and which was very basic in nature and whose structure continues to be closely based on the original boilerplate created by the Symfony templating system- have been excluded but otherwise detailed test classes in PHPUnit have been created for all of the core classes in the application. The following test classes were created

- OfficeTest (discussed in more detail below)
- CustomerTest
- DoNotSendTest
- RosterTest
- EmployeeTest
- ServiceUserTest
- ServiceUserAssignedEmployeeTest

The full code library for the tests is included in the Appendix One but it is useful to delve into one class to demonstrate the testing approach taken.

This section discusses the test units implemented for the Office entity for which the seven individual test units have been created. Two are discussed in detail to outline the testing approach taken.

testShowSecondOfficeItemfromIndex. this test tests that the View link to second item (id=2) in the index view is routes correctly. This link is publicly available therefore the client can be called without supplying a user context.

```
public function testShowSecondOfficeItemfromIndex()
{
    $client = static::createClient();

    $client->followRedirects(true);
    $crawler = $client->request('GET', '/office');
    $link= $crawler->selectLink('View')
        ->eq(1)
        ->link();

    $this->assertEquals(

        $link->getUri(), 'http://localhost/office/2');
}
```

testEditFormSubmit. This test uses the edit facility on the office entity. Because this link is not publicly available the client class here must be supplied with a valid user context otherwise the test will fail. The button crawlerNode below creates a new form object and because a web page can have more than one form, the particular one to be used is identified by the text value of its submit button text value, in this case the Edit link. Having called up the edit form, the test submits a form with valid data and then validates that following submission of the form that the context is back on the show page as per the redirect in the edit controller.

```
public function testEditFormSubmit()
{
    // this link is only available to authenticated users
    with the ROLE_ADMIN role

    $client = static::createClient(array(), array(
        'PHP_AUTH_USER' => 'testuser@gmail.com',
        'PHP_AUTH_PW'   => 'pa$$word',
    ));
    $client->followRedirects(true);
    $crawler = $client->request('GET', '/office/1');
    // follow the Edit link
    $crawler = $client->click($crawler->selectLink('Edit')->link());
    $buttonCrawlerNode = $crawler->selectButton('Submit Changes');
    //fill in new form values
    $form = $buttonCrawlerNode->form(array(
        'AppBundle\Office[officeName]' => 'Clane',
        'AppBundle\Office[addressLine1]' => 'Ballinagappa Road',
    ));
}
```

```

        'appbundle_office[addressLine2]' => 'Clane',
        'appbundle_office[addressLine3]' => 'Kildare',
        'appbundle_office[eirCode]' => 'D05W987',
        'appbundle_office[landlineTelephone]' => '04556555',
        'appbundle_office[mobileTelephone]' => '12456',
        'appbundle_office[isActive]' => true,
        'appbundle_office[countyPostcode]' => 1001009));
        // submit the form
        $client->submit($form);
        $client->followRedirects(true);
        // check that we're back on the show page
        $this->assertEquals(

        $client->getRequest()->getUri(), 'http://localhost/office/1');

    }
}

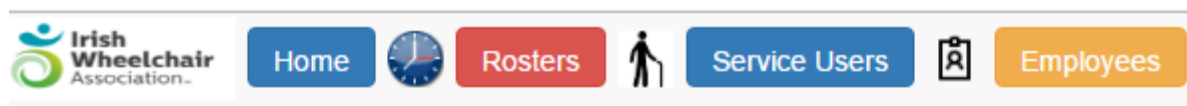
```

7.1.2 Functional Testing report

A significant amount of functional testing was completed by the author throughout and at the conclusion of the build of the initial prototype application. It would be difficult to document the operation and outcome of every individual functional test completed in an application of this size, but a small representative sample is included here.

Title	Create Service User with valid Data
Description	User can navigate to the index view area of Service Users and a single entity record of type Service User. On Clicking Add New Service User, the user should be presented with a blank form to complete the input of a new Service User Record. If the data is valid, once the user clicks submit, the page should route back to the show page for the newly created service user. If they used a valid address, the system should have retrieved coordinates from Google Maps, saved these as part of the create action and should now be displaying the address in the map pane
Expected Outcome	Service user is shown in the Index view with the required property values
Actual Outcome	As expected

Title	Attempt to Create Service User with missing Data (tests x3)
Description	User can navigate to the index view area of Service Users and a single entity record of type Service User. On Clicking Add New Service User, the user should be presented with a blank form to complete the input of a new Service User Record. User submits form without completing mandatory fields- first name, last name and address line 1 .
Expected Outcome	If mandatory data fields are left blank , once the user clicks submit, the page should highlight these to the user giving them an opportunity to amend the form and resubmit.
Actual Outcome	As expected
Notes	Test repeated on other mandatory fields



Serviceuser creation


First name	
Last name	
Address line1	<div>  Please fill out this field. </div>
Address line2	
Address line3	
Eir code	

Figure 4: Test Create Service User with missing Data

Title	Attempt to Create Service User with invalid date data
Description	User can navigate to the index view area of Service Users and a single entity record of type Service User. On Clicking Add New Service User, the user should be presented with a blank form to complete the input of a new Service User Record. User enters an invalid start date and end date
Expected Outcome	If mandatory data fields are left blank , once the user clicks submit, the page should highlight these to the user giving them an opportunity to amend the form and resubmit.
Actual Outcome	As expected
Notes	In this instance, the JQuery datepicker prevents an invalid date being possible

Title	Add a Roster to the service user
Description	From the show page for a service user, the User can click the Add New Roster button. Then the user should be presented with a roster form to input a new Roster Record. User enters an valid start date, time and end date and end time for roster
Expected Outcome	Roster should be saved, user redirected to show service user r page and able to see roster in the calendar view.
Actual Outcome	As expected
Title	Add a Roster to the service user with empty date or time (tests x4)
Description	From the show page for a service user, the User can click the Add New Roster button. Then the user should be presented with a roster form to input a new Roster Record. User clicks create without selecting a valid start date, time and end date and end time for roster
Expected Outcome	The page should highlight the missing form data to the user and allow them to complete and resubmit the data
Actual Outcome	As expected
Notes	In this instance, the JQuery datepicker prevents an invalid date being possible but a blank date is possible and correctly handled by highlighting this to the user. Test was repeatedly successfully sequentially with start time missing, end date end time

Roster creation

Service user id	Bob Chilcott ID:164 ▼
Roster start time	dd/mm/yyyy --:--
Roster end time	dd/mm/yyyy --:--
Number resources needed	Choose How Many Resources Are Needed ▼
Customer id	▼
Roster status	▼

Create

Back to the list

Figure 5: Test Create Roster with missing date

Title	Set the employee to being on leave and recheck availability
Description	Enter an employee absence and navigate to an unfilled roster during the period of the absence to verify that the absent employee is not shown as available
Expected Outcome	The find an employee to assign screen should exclude the absent employee from the list of available employees
Actual Outcome	As expected
Title	Remove the absence for the employee and recheck availability
Description	Remove the absence and navigate to an unfilled roster during the period of the absence to verify that the absent employee is not shown as available
Expected Outcome	The find an employee to assign screen should now show the previously absent employee from the list of available employees
Actual Outcome	As expected
Title	Create roster and leave as unassigned
Description	Create a new roster with a requirement for one employee resource and save
Expected Outcome	Roster should show a warning that not enough employees have been assigned
Actual Outcome	As expected
Title	Assign roster to employee and recheck roster status
Description	Locate an unfilled roster and assign an available employee to the roster
Expected Outcome	Roster should now show a success message that the roster has sufficient employees, roster should no longer show the Find an Employee to Assign
Actual Outcome	As expected
Title	Mark Employee as not to be sent to a service user
Description	From the show page for a service user, the User can click the Mark as Do Not Send button. Then the user should be presented with a Do Not Send form to create a new association between a service user and an employee record. User clicks Mark as do not send.
Expected Outcome	Association should be saved, user redirected to show service user page and able to see that the employee is now marked as do not send
Actual Outcome	As expected
Title	Check Employee is not shown as available for a service user who they have DO NOT SEND association
Description	From a roster which is not fully assigned service user, the User can click the FIND AN EMPLOYEE TO ASSIGN button. The user should be presented with of available employees excluding the Do Not Send employee
Expected Outcome	The user should be presented with of available employees excluding the Do Not Send employee
Actual Outcome	As expected

Title	Check Employee is restored as as available employee for a service user once their DO NOT SEND association is removed
Description	From the show page for a service user, the User can click the Remove button in the Do Not Send button beside the relevant employee. On navigating to an unassigned roster for that service user, the User can click the FIND AN EMPLOYEE TO ASSIGN button. The user should be presented with of available employees which now includes the previously missing employee while they were marked as Do Not Send
Expected Outcome	The user should be presented with of all available employees excluding the employee who has missing in the previous step
Actual Outcome	As expected
Title	Verify Google Maps geocoding
Description	Check coordinates against Google Maps and verify map
Expected Outcome	On changing an address or creating a new record, the co-ordinates should be updated to reflect the correct location of the record which should be consistent with Google Maps
Actual Outcome	As expected
Notes	Some addresses in rural areas are not always recognised by Google Maps
Title	Verify Google Maps distance calculation
Description	Check distance shown between available employees and a given service user against Google Maps and verify distance
Expected Outcome	The distance shown for available employees should be consistent with that available in the directions area of Google Maps
Actual Outcome	As expected
Notes	Some addresses in rural areas are not always recognised by Google Maps
Title	Create Service User with valid Data
Description	User can navigate to the index view area of Service Users and start to create a single entity record of type Service User. Clicking Add New Service User, the user should be presented with a blank form to complete the input of a new Service User Record. If the data is valid, once the user clicks submit, the page should route back to the show page for the newly created service user. If they used a valid address, the system should have retrieved coordinates from Google Maps, saved these as part of the create action and should now be displaying the address in the map pane
Expected Outcome	Service user is shown in the Index view with the required property values
Actual Outcome	As expected

7.1.3 User Acceptance Testing report

User acceptance testing is an integral and essential part of any software development project, but frequently this important area of the implementation of a new system does not get enough resources and attention, usually to the detriment of the final software deliverable. As (**davis**) notes:

Lack of user acceptance has long been an impediment to the success of new information systems.

A number of recent research efforts note the ongoing pattern of failed software projects, with a number of contemporary analysts estimating the failure rate to be in the between 50% (**florentine**) and 75% (**gartner**). It is therefore crucial to devote sufficient time and resources to this important activity and to have a robust and structured way to feedback the results of the testing into the overall development process so that all issues identified are resolved before the go-live.

For this project three different internal IWA users completed detailed user acceptance to attempt to complete a wide range of functions in the application independently with minimal instruction. The three users (who are all IWA employees involved in the Rostering project in various ways) willingly participated and offered highly constructive and useful feedback to the project.

Testing Team

- KL who works as an Assisted Living Coordinator
- MB who works as Payroll Manager
- MK who works as an ICT support analyst and web designer

7.1.4 Overview

The three users' individual task data entry sheets are included in Appendix 3. In summary the users were required to complete a series of sequential tasks which involved both working with existing preset service users and employee records as well as creating new service user and employee records and working to create a roster and assign an employee record to it. The value of end user testing clearly became evident to the author as one reasonably serious bug was identified through the testing process having been completely overlooked in both the development stage's functional and also in the automated unit testing. This issue primarily arose from one of the end users traversing through the steps in a different sequence to the well travelled route the author had been using, revealing in the process a hitherto overlooked bug in the code which only came to light as a result of the detailed user acceptance testing.

7.1.5 Outcomes from User Acceptance Testing

Title	Bug: when editing a roster
Description	when editing a roster, the roster reverted from the previously associated service user and showed an empty service user drop down control.
Resolution	The RosterType form class had an incorrect code entry which worked fine for new entries but dropped the service user when editing. As a workaround, the code in this RosterType class was split into two FormType modules one which was called from the Create() context and one which was called from the Edit() context
Status	Resolved
Title	Issue: Confusion with Assigning Rosters
Description	Some users found it a little confusing navigating from the service user screen into a roster and then clicking Find An Employee to Assign and finally clicking Assign
Resolution	This approach was taken to avoid calling the Google Maps distance API in a scenario where the roster was already filled with sufficient assigned employees and there was no need to suggest additional employees
Status	Open/Under review
Title	Suggestion: Lock menu items at the top of the screen
Description	The static menu navigation items at the top of the screen did not stay at the top once you began to scroll down the page
Resolution	This was resolved using the navbar navbar-default navbar-fixed-top class in Bootstrap which pins a fixed navigation bar at the top of the screen
Status	Resolved
Title	Suggestion: Lock the Add new employee/service user button at the top of the index pages
Description	As soon as users scroll down the index pages for service users, employees and rosters, they had to scroll back up to the top to locate the Add New button
Resolution	This was resolved by creating a new .divide-nav class to include buttons and other navigation items which needed to be pinned at the top of the screen and also by adjusting the body css class so that it created padding at the top of the screen so that this second class was positioned below the navbar-fixed-top class
Status	Resolved

7.1.6 Final comments

While each user made some minor suggestions in navigating through the different screens, in particular in relation to assigning an employee to a roster, they generally reported that the application was easy to use and straightforward to navigate around in. Some useful suggestions were made in relation to pinning

the navigation menu at the top of the screen, pinning the add new item button at the top of the index views for rosters, service users and employees and these have already been implemented in an updated release of the application which will be retested by the users concerned. Crucially, a significant and hitherto unidentified bug was identified through the testing process, where on editing a roster, the association to the service user was removed and the user needed to re-select the appropriate service users in the user interface. The root cause of this bug has been identified and resolved. Finally, a common theme identified by all users was that the process to assign an employee to a roster took too many steps and needed to be made more user friendly. This is now under review and options for improvement will be scoped further with the users concerned.

8 Implementation

8.1 Implementation Plan

8.1.1 Evaluation of Different Web Development Frameworks

The following different web development frameworks- each of which are suitable to produce dynamic, data driven web pages- were evaluated in the search to find a suitable platform on which to build the prototype application. All three platforms examined offer the ability for server side logic within an application.

- **ASP.NET MVC** is the Microsoft led development stack based on the .Net framework which is a widely used set of web development tools . It offers a strong integrated development environment (IDE) in Visual Studio which is freely available in the Community edition for non commercial use cases. It uses a combination of HTML, Razor, a proprietary Microsoft syntax used for developing views and C# for controller and repository and other non view related classes. It supports an MVC led approach with scaffolding of views and controllers based on having first creating a model with the relevant entity classes. It also integrates well with Azure web services, Microsoft's cloud stack where Visual Studio allows you to easily deploy web applications from a development environment. Asp.Net also includes the Entity Framework which essentially allows a code free, scaffolded approach to working with the database, driven entirely from the data model in your application. It also supports a test driven approach and integrates with a wide variety of third party components through a vibrant Nuget package marketplace.
- **Symfony** is a widely used web development framework developed by Fabien Potencier and distributed as open source tools which also supports the MVC approach to separation of concerns. It uses the very flexible Twig templating engine (which in the author's opinion is easier to use than Razor!) and allows users to adopt a flexible approach to selecting which Symfony components they choose to use in a given application. As Symfony has reached a wide degree of developer adoption, achieving 500 million downloads of the framework in 2016, it is widely supported with a significant choice of different add-on components (termed bundles in the Symfony domain) which are available to its developer community. Symfony uses PHP as its main code language and allows developers to easily scaffold views and controllers based on the creation of an entity model. It also supports a scaffolded approach to interacting with databases, similar in approach to Entity Framework through the use of command line tools which validate and update your database as required. Because Symfony uses the Doctrine ORM (Object Relational Mapping) which abstracts database layer operations from the rest of the application it supports MySQL and various other database platforms including SQLite, PostgreSQL and Microsoft SQL server.

An interesting twist on the Symfony approach, which is essentially a full stack framework, is the Silex approach, which uses elements of the Symfony framework (and a related library called Pimple) without requiring the developer to fully utilise all of the Symfony components and instead begin-

ning with a skeleton framework which is much smaller and lighter in size than the full Symfony framework. When using Silex, a developer still has the flexibility to add in other components from the larger Symfony stack as the requirements of their application grow. One aspect about Symfony to note here is that the author notes some commentary from the developer community online which would suggest that Symfony is an ideal approach for small to medium web applications but its highly component driven approach can result in some performance impacts when used in very large scale deployments. Symfony also supports a test driven approach, both through using the test classes built into the application framework but also through its tight integration with the PHPUnit testing framework.

- **Node.js** is a third very commonly used web development open source framework developed by Ryan Dahl which runs on a Javascript run time environment which allows for Javascript code to be executed server side. Again, Node.js also supports the MVC approach to separation of concerns. The innovations provided by Node.js allow Javascript, traditionally a client side scripting language, to be run server side and has become one of the core components driving the "Javascript everywhere" development paradigm. Node.js supports a similar ORM driven approach to database operations resulting in a wide variety of relational and non relational databases being supported. Similar to ASP.Net and Symfony, Node.js has a number of different templating languages including Jade, blade, Mustache and Handlebars JS. It has also achieved wide acceptance among developers and has an extensive library available of reusable components, which serve to significantly reduce the requirement for developers to write their own code for commonly used operations. Node.js is perceived as an advanced level language with a significant learning curve to be overcome for developers migrating to the platform. An advantage of Node.js is that it does not create a lock on server resources and scales very well to large scale applications, working particularly well in scenarios which require real time processing. Corporate users of Node.js include LinkedIn, Microsoft, Netflix and Paypal.

8.1.2 Choice of a Web Development Framework

Though it was very instructive to learn more about Node.js as part of the research for the project, the perceived difficulty level associated with this platform and the author's lack of previous exposure to it meant that it was not an ideal choice for a student project, particularly as there was a requirement for a rapid delivery of a prototype application for user testing. The author had some previous exposure to ASP.NET MVC and viewed this as viable choice for the application. However, using PHP/Symfony was a better fit given that the syllabus for the Higher Diploma in Computing that had been undertaken over the past 2 years had included significant exposure to PHP and the project/prototyping requirement afforded the author an opportunity to gain a familiarity with Symfony. It was also very helpful that the author's supervisor (**smith**) on this project had produced some excellent learning materials on the Symfony platform which greatly assisted in getting to grips with the new platform. A minor criticism of the Symfony platform is that while the Symfony documentation

(**weaver**) is generally excellent and extremely comprehensive, it can be somewhat challenging for developers migrating to the platform for the first time to get over the initial learning curve and get up and running in developing their very first Symfony application.

In fact, the first draft of the prototype application for this project was initially built on the Silex platform and as some of the requirements for implementation of a robust security model and the advantages of the scaffolding approach offered by Symfony came to light, it was determined to be useful to rebuild the entire prototype application on the full Symfony stack. It was most instructive to the author to note that having built the application on the Silex platform originally, the second build on the Symfony platform, which was completed from scratch without reuse of any of the Silex application, took approximately 30% of the time originally taken to build the Silex application, due in no small part to the automated scaffolding approach which Symfony offers to automate the construction of CRUD controller functions as well templated views.

8.1.3 Examples of some code efficiencies available in Symfony

The author was very impressed with the following Symfony capabilities which among others were noted to substantially save time in manual coding tasks:

Twig Templating allows calling server side code in a format which is less verbose and easier to manage than embedding PHP code among the HTML code in your presentation view. A simple example is included here. Note also the concise way that the Annotation of related classes referred to below allows you to access attributes of a related class, in this case the mobile telephone property of the related employee class.

```
{% for assignedEmployee in assignedEmployees %}

    <td>{{ assignedEmployee.employeeId }}</td>
    <td>{{ assignedEmployee.employeeId.mobileTelephone }}</td>
{% endfor %}
```

The **Annotation capabilities** allow foreign keys to be defined in annotations to a given class or attribute which are automatically created in the underlying database, as follows:

```
/**
 * Roster
 *
 * @ORM\Table(name="roster")
 * @ORM\Entity(repositoryClass=
 "AppBundle\Repository\RosterRepository")
 */
class Roster
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
```

```

    * @ORM\GeneratedValue(strategy="AUTO")
    */
    private $id;

    /**
     * @ORM\ManyToOne(targetEntity=
     "AppBundle\Entity\ServiceUser")
     * @ORM\JoinColumn(name="serviceUserId",
     referencedColumnName="id")
     */
    private $serviceUserId;

```

The **Form Type/ Builder** classes allow an entity's data entry/update form to be automatically scaffolded based on the entities data attributes as defined in its model. For example rather than having to create each an individual label and attribute for each property of the customer class, Symfony automatically scaffolds this using very simple syntax as follows:

```

    public function buildForm(FormBuilderInterface $builder,
    array $options)
    {
        $builder->add('customerName')->add('addressLine1')->
        add('addressLine2')->add('addressLine3')->
        add('eirCode')->add('landlineTelephone')->
        add('mobileTelephone')->add('isActive')->
        add('mainContact')->
        add('countyPostcode')->add('managingOffice');
    }

    /**
     * {@inheritdoc}
     */
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Customer'
        ));
    }

```

Then in the actual presentation view, the syntax required to build the form, in this case an Edit form for a customer record, is extremely concise and the scaffolding approach automatically updates the form as the model of the entity class changes, An example of calling the formbuilder from a view is shown in the Twig template as follows:

```

{ form_start(edit_form) }}
    {{ form_widget(edit_form) }}
    <input type="submit" value="Edit"/>
    {{ form_end(edit_form) }}

```

Where it's necessary to extend the logic of an individual entity's form type, Symfony provides an extensive library of FormTypes which can be used and overridden as required. For example, the EntityType class allows you to wire the contents of a dropdown control to the list of objects in another entity- or in this case, the ChoiceType allows you to define in your FormType class the contents of a drop down control together with both the values and labels for the dropdown control. An example of how you can extend the logic of the formtype is this instance where the form needed to include or not include a roster date field depending whether the submitted form data included a value in the posted `_REQUEST` attribute which represented the posted form data. In this scenario, the form builder needed to handle both a new record being created which routes to the generic form without a roster date prepopulated, and a new record being created from a calendar with the specific date in question already set by the user clicking +new in the on a specific date. This can be handled using logic in the formbuilder as follows:

```
public function buildForm(FormBuilderInterface $builder,
    array $options)
{
    $timezone = new \DateTimeZone("Europe/Dublin");

    if (isset ($_REQUEST['rosterDate'])) {
        $builder->add('serviceUserId', EntityType::class,
            array('class' => 'AppBundle:ServiceUser',
                'data' => $options['serviceUser']))
        ->add('rosterStartTime', DateTimeType::class,
            array('date_widget' => "single_text",
                'time_widget' => "single_text",
                'data' => new \DateTime($_REQUEST['rosterDate'],
                    $timezone)))
        ->add('rosterEndTime', DateTimeType::class,
            array('date_widget' => "single_text",
                'time_widget' => "single_text",
                'data' => new \DateTime($_REQUEST['rosterDate'],
                    $timezone)))
        ->add('numberResourcesNeeded',
            ChoiceType::class, array(
                'choices' => array(
                    '0' => '0',
                    '1' => '1',
                    '2' => '2',
                    '3' => '3'),
                'required' => true,
                'placeholder' => 'Choose How Many
                    Resources Are Needed',
                'empty_data' => null
            ))
        ->add('customerId');
    } else {
```

```

$builder->add('serviceUserId', EntityType::class,
array('class' => 'AppBundle:ServiceUser',
'data' => $options['serviceUser']))
->add('rosterStartTime', DateTimeType::class,
array('date_widget' => "single_text",
'time_widget' => "single_text",
))
->add('rosterEndTime', DateTimeType::class,
array('date_widget' => "single_text",
'time_widget' => "single_text",
))
->add('numberResourcesNeeded',
ChoiceType::class, array(
'choices' => array(
'0' => '0',
'1' => '1',
'2' => '2',
'3' => '3'),
'required' => true,
'placeholder' => 'Choose How Many Resources Are Needed',
'empty_data' => null
))
->add('customerId');
}
}

```

The **Dynamic Query** capabilities of Symfony via Doctrine ORM can obviate the need to create specific views at a database level of entity classes where a (presentation level) view needed to present de-normalised data to the user. As an example of this capability, where a **rosterassigned** entity exists which has a foreign key attribute which relates to a roster entity via a many to one entity relationship. This relationship allows for one or more employees to have a roster assignment association to a single roster entity. Once this relationship exists, it's possible to dynamically query the entity to retrieve only assignments which have a relationship to a specific roster, as follows:

```

$assignedEmployees = $em->
getRepository('AppBundle:RosterAssignedEmployee')
->findByRosterId($roster->getId());

```

The **Query builder** feature offers a powerful and flexible range of default Doctrine ORM queries similar to the `findByRosterId()` one noted above. These cater for most scenarios but it is also possible to create your own queries which resemble an SQL type logic (there's also a Doctrine Query Language which resembles the T-SQL query approach to database queries even more closely, but the author did not find a need for this in this project) which allows you to create your own custom queries when the need arises.

For example, the query shown below accepts a datetime object and a service user object as input parameters and checks to see if there are any rosters for the service user starting at any time in the 24 hour period of the datetime object provided. This query is used in a section of the Show view for a service user

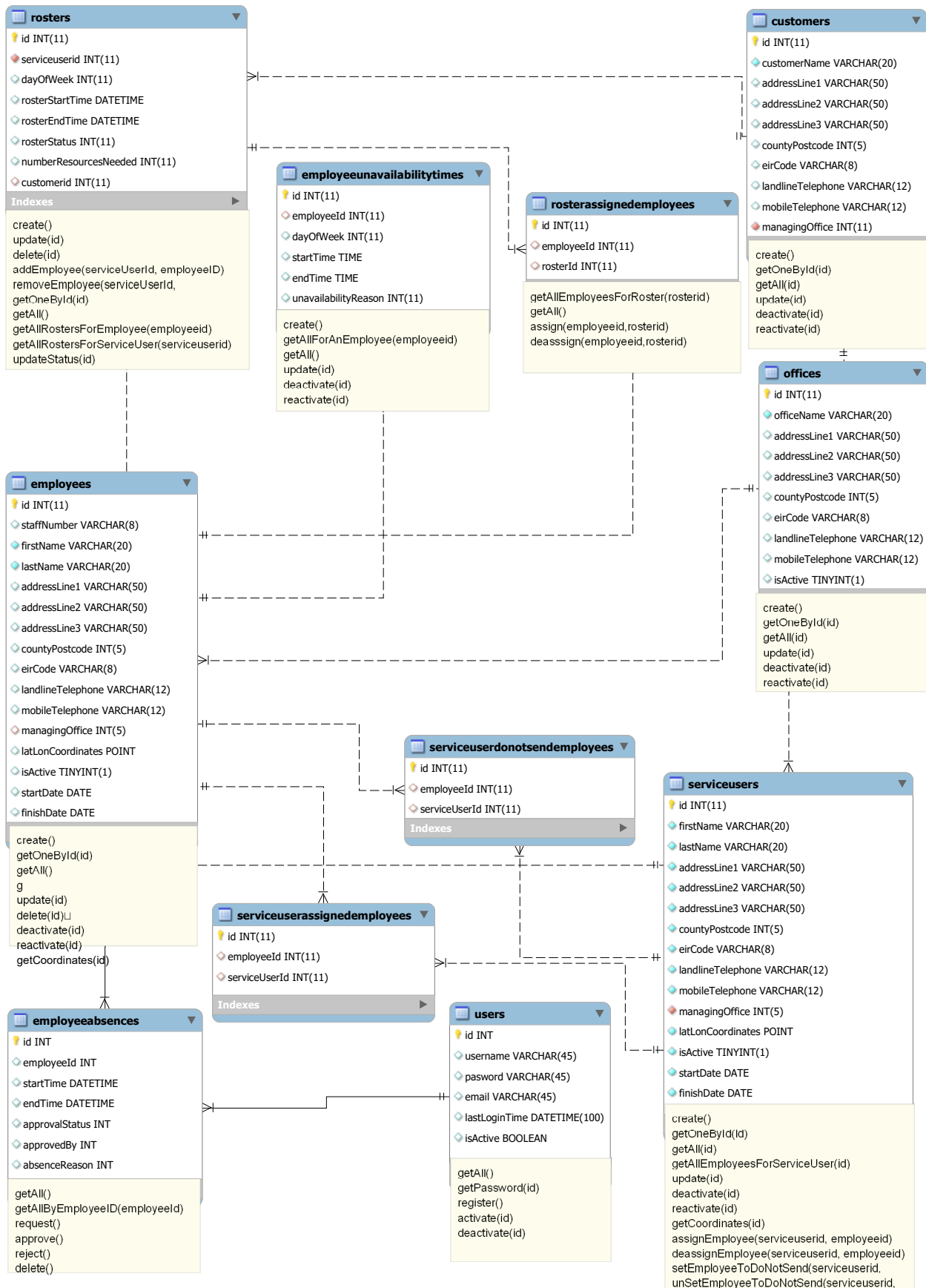
which displays a calendar control in a table control showing each as a line of table data. It then returns a true or false value to the calling method depending on whether or not a roster was found for that day. The logic of the custom `getByDate()` query is as follows:

```
// this doctrine filter is used by the for loop in the show
// method of the service user controller.
//for each checked date in the for loop it checks to see if
//that service user has any rosters starting in that 24 hour period.
public function getByDate(\DateTime $date, ServiceUser $serviceUser)
{
    $from = new \DateTime($date->format("Y-m-d") . " 00:00:00");
    $to = new \DateTime($date->format("Y-m-d") . " 23:59:59");

    $qb = $this->createQueryBuilder("roster");
    $qb->andWhere($qb->expr()
        ->between('roster.rosterStartTime', ':date_from', ':date_to'))
        ->andWhere("roster.serviceUserId= " . $serviceUser->getId());
    $qb->setParameter('date_from', $from,
        \Doctrine\DBAL\Types\Type::DATETIME);
    $qb->setParameter('date_to', $to,
        \Doctrine\DBAL\Types\Type::DATETIME);
    $result = $qb->getQuery()->getResult();
    return $result;
}
}
```

8.2 Entity Relationship Diagram

Based on the review of the use cases together with workshops with the subject matter experts provided within the Assisted Living Services team a map of the required classes for the application together with the properties and methods required for each class was created. This was then modelled into an initial Entity Relationship Diagram which is included on the next page.



8.3 Code Organisation for Project

As the prototype application, though less full featured than the anticipated production version, utilised a reasonable number of entity (model) classes, views and controller methods, an organised and consistent approach to code organisation within the project was important. The following strategy was adopted.

- **Presentation views** were grouped, as is standard in under the Symfony project structure under the
`\.app\Resources\views` node of the project. Within this folder a subfolder for each entity was created with a standard naming convention used, based on the scaffolded names created by Symfony of `index`, `show`, `edit` and `delete`.
- **Custom Error template views** were grouped, as is standard in under the Symfony project structure under the
`\.app\Resources\TwigBundle\views` node of the project.
- **Controllers classes** for each object were grouped in under the
`\.src\AppBundle\Controller` node of the project. Within this folder a separate controller file was created with a standard naming convention used e.g. `EmployeeController`, `RosterController`. Within each controller file, the method names mapped to the related views e.g. `index()`, `new()` etc.
- **Entity Models** classes for each object were grouped in under the
`\.src\AppBundle\Entity` node of the project. Within this folder a separate controller file was created with a standard naming convention used e.g. `Employee`, `Roster`. As noted the Annotation facility within Symfony was used to implement database level requirements such as primary keys, foreign keys, foreign key constraints etc.
- **Form types classes** for each entity were grouped in under the
`\.src\AppBundle\Form` node of the project. Within this folder a separate controller file was created with a standard naming convention used e.g. `EmployeeType`, `RosterType`. These were extended to implement linkages between different entity classes in the view- for example contents of drop down controls related to other entities.
- **Repository classes** for each entity were grouped in under the
`\.src\AppBundle\Repository` node of the project. Within this folder a separate controller file was created with a standard naming convention used e.g. `EmployeeRepository`, `RosterRepository`.
- **Test classes** for each object were grouped in under the `\.src\AppBundle\Test` node of the project.
- **Reusable or Factory Methods** which did not relate to a specific entity, for example the Google Maps integration classes were grouped in a specific folder for that type for example the Google Maps classes are under the `\.src\AppBundle\Mapping` node of the project.

-
- **Third party components** were installed as default by Composer in the `\..src\Vendor` node of the project.

8.4 Detailed Implementation Plan

The following implementation plan was developed and progressively put into practice as work on the project progressed. See also the project diary in Appendix X which progressive build up and testing of the code base for the project.

- **Complete and confirm Entity Relational Design** including models and anticipated methods and views for each entity.
- **Complete and confirm Use cases** for different functionality of the application
- **Install Symfony Project** and supporting components e.g. Twig, Doctrine etc.
- **Create Entity model classes for main areas of application** e.g. Employee, Service User, Office, Roster, Customer etc.
- **Create supporting Entity model classes for other areas of application** e.g. ServiceUser Assignment, ServiceUserDoNotSend, Employee-Absence, RosterAssignment as well as lookup list classes e.g. AbsenceReason, County etc.
- **Annotate** data classes to create appropriate entity relationships and primary, foreign keys
- **Scaffold initial views and controllers** using Symfony/Doctrine commands to create initial default views and controllers with standard Create/Read/Update and Delete Classes
- **Test and Confirm resulting Database schema** and populate with sample data
- **Extend views and controllers** to show related entities where required for example Service User views to be facilitate user creating viewing, editing and deleting related records e.g. Rosters, Assignments etc. and automatically associating these with the underlying service user from whose context the action was initiated.
- **Update Form Types** so that the forms correctly show related entities
- **Update Repository Classes** for custom queries as required
- **Google Maps integration** build and implement to controllers and views on various entities using this functionality.
- **Implement Find Available Employees** adapt subsidiary methods in the service user controller so that the system has the capacity to suggest available employees for a given roster slot, progressively completing validation on initial list of employees and filtering out do not sends, employees on absence, employees working elsewhere, employees who are not available at time of the roster.

-
- **Implement Security** Extend user class and create roles and annotate controller methods to restrict access where needed.
 - **Calendar view of Rosters**Show monthly view of calendars using third party framework such as Bootstrap Calendar or Full Calendar.
 - **Calendar Add/Delete/Update** Update so that users can add/delete or update individual rosters when shown in the calendar view.
 - **First Pass testing** on above methods and logic
 - **Automated Unit testing** on above methods and logic by creating unit test classes on the main code modules
 - **Work on Application Visuals and aesthetics** to ensure a user friendly and consistent approach to styling throughout the site.
 - **Bootstrap.** Investigate the capabilities of Bootstrap and their potential applicability to the site
 - **Bootstrap.** Update the show view for the employee and service user to display a responsive multi column view which exposes a range of different entities related to service users and employees.
 - **User Acceptance testing** . Prepare testing script sheets to that users can work through a series of task administrative tasks which touch on both manipulating existing data and creating new records.
 - **Address issues from User Acceptance testing** . Work through the results of the user acceptance testing and address any bugs, issues or user suggestions.

8.5 Web Design Approach

8.5.1 Step1 : Symfony templating

This project started by using the Symfony facility to generate boiler plate CRUD (Create/Read/Update/Delete) templates and associated controllers. This approach yields considerable time savings for developers, obviating the need for much repetitive code generation and it is clear that a command line bundle can complete this more accurately than a human developer. In fact, as noted earlier in the Design section, the author had the experience of initially creating a basic initial prototype version of the Scheduler application initially in Silex, the light-weight version using a bare-bones approach to the Symfony framework. In this iteration, the author initially create templates and controllers to reflect the requirements in the model for each entity. Even when using the initially created template and controller for the first entity (service user) as a template to generate controllers and view templates for the other dozen or so entities, this was a very time consuming exercise, taking many days of development. At this point a decision was made to rebuild from scratch on the full Symfony framework to leverage the power of the Security model, skeleton templating system and various other features including the very flexible Dynamic Doctrine queries.

The second iteration of the application was built using the Symfony skeleton bundle which allows for the use of a command line driven approach to generating boiler plate templates and controllers. This took considerably less time to accomplish, estimated by the author to have delivered time savings in the order of 70-80% over the original and more manual Silex approach.

This powerful approach uses the following command which generates the controller templates in a few seconds based on an already existing model class for the referenced entity is already in place.

This command yields the following initial templated code which can be generated in a few seconds once the model is already in place.

```
$ php app/console generate:doctrine:crud
--entity=AppBundle:Employee--format=annotation --with-write
--no-interaction
```

Templated Controller methods

When this routine completes the following five controller actions will be created

- **indexAction()** This action lists all records. It can be further extended to accept an incoming parameter, for example and when linked with an appropriate Doctrine query can quickly achieve filtering of the records that are returned.
- **showAction()** This action shows one given record identified by its primary key which is passed by reference to the underlying object.
- **newAction()** This action creates a new record.
- **editAction()** This action edits an existing record identified by its primary key which is passed by reference to the underlying object.
- **deleteAction()** This action allows the user to deleting an existing record identified by its primary key which is passed by reference to the underlying object. This approach uses the FormBuilder class via calling a special `createDeleteForm()` method to build the Delete Form.

View Templates

The routine will also create the following template forms which are equivalent to the controller methods and are based on the standard templates forms in the `Resources/skeleton/` directory. It is also useful to note that these skeleton templates can be customised to achieve templating of individual entities from your own modified skeleton template but this approach was not used for this project.

- **index**
- **show**
- **new**
- **edit**

Two points are worth noting here:
Firstly, a sample of the new view is included below in order to demonstrate the power of the Symfony templating engine.

```
{% block body %}
    <h1>Absencereason creation</h1>

    {{ form_start(form) }}
    {{ form_widget(form) }}
    <input type="submit" value="Create"/>
    {{ form_end(form) }}

    <ul>
        <li>
            <a href="{{ path('absencereason_index') }}">
                >Back to the list</a>
        </li>
    </ul>
{% endblock %}
```

In a case where the entity in question had a large number of individual properties, the power of the above concise code becomes obvious where the

```
{{form_widget(form)}}
```

command will automatically create form labels and input controls for each property, choosing form controls appropriate to the underlying datatype and automatically updating the template as the underlying model changes over time.

Secondly, it can be noted that there is one fewer template view than controller method because the `deleteAction` creates its form within the `createDeleteForm` method which means that a template view for the Delete Action is not needed.

8.5.2 Step2 : Base Template Inheritance

The work in Step 1 above yielded a set of basic templates and controllers which were powerful in terms of functionality but were unfortunately quite plain and uninteresting in their initial styling. This can be seen in the following screenshots of the initial list and show templates on the next page.

In this second step of development, the author sought to leverage the power of the capacity within the Twig framework for template inheritance. Here, we can define a base template which contains all of the html code, navigation cues and references the style sheets (CSS) which are common to every page in the site.

Once this is in place, it is easy to achieve a professional and consistent look and feel throughout the application which is vital to achieving user adoption. In a prototype or 'proof of concept' application such as the one being developed in this project, these considerations are less critical than in a full production scenario nevertheless the author attempted to achieve a professional look, within the significant limitations of his visual design talents and styling knowledge, which would give the test users confidence in navigating through the application through the use of various colour coded cues.

Absencereasons list

Id Absencereason		Actions
<u>1</u>	Annual Leave	<ul style="list-style-type: none">showedit
<u>2</u>	Sick Leave	<ul style="list-style-type: none">showedit
		<ul style="list-style-type: none">Create a new absenceReason

Figure 6: Absence Reasons List View

Absencereason

Id 1	
Absencereason Annual Leave	
<ul style="list-style-type: none">Back to the listEdit<input type="button" value="Delete"/>	

Figure 7: Absence Reason Show View

Even more importantly, it was vital to give the users a chance to contribute to the design and aesthetics of the site at this early stage so that these suggestions could be factored into the vendor discussions in building the final production version.

A base template was developed with the following objectives

- **CSS classes.** Include the custom CSS classes which apply to site, these are used to style buttons and define the look and feel of tables, headings and text as well as the layout of navigation controls.
- **External JQuery and CSS libraries.** The application touched lightly

upon the use of JQuery libraries for areas such as calendar controls when selecting dates and also leveraged the now ubiquitous Bootstrap framework in its layout design for more complex pages. The author had hoped to use an external JQuery library to show an interactive control to manage CRUD functions for viewing and updating of rosters but encountered some difficulties and reverted to a simpler dynamic HTML control to display calendars instead.

- **Navigation bar.** The application sought to use a series of icon driven navigation cues at the top of the screen which are present consistently as a user navigates through the site.
- **Navigation bar.** The application also sought to display the user's login information when a user had authenticated to the application together with a **logout** button or in the case where they were not yet logged in to display a **login** button which would link the user to the login authentication page. Each icon and button contained an embedded anchor tag to link the user to that section of the application.
- **Flash messages.** Finally, the application used some embedded and colour-coded flash messages which read from the global Session variable. In the case where a standard CRUD event completed an action successfully, a message was appended to the Notice array in the FlashBag() class which the session uses to store alerts. In the event that an error was encountered, for example a save event did not complete successfully, a message was appended to the Error array in the FlashBag() class. Then as the page was refreshed in the browser the base template template shows these to the user, colour-coded as red if the new message was an error or in green if the message was a notice.

A sample of the user interface that was achieved using this approach can be seen in the figure below

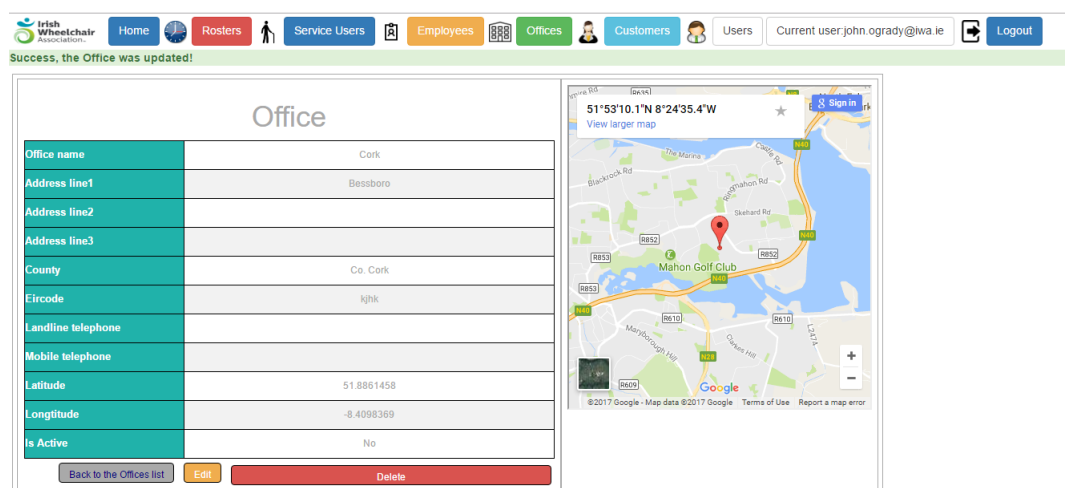


Figure 8: Sample page with underlying Base template

The approach for using an inherited base template was sufficient for most pages in the application. A reworked edit page, developed originally from the boilerplate code which was generated by the Symfony but now extended using the Base page is shown in screenshot in the figure below, this time for the customer entity.

Customer edit

Customer name	Codec DSS
Address line1	Hyde House
Address line2	Hyde
Address line3	Adelaide Road
Eir code	
Landline telephone	6045000
Mobile telephone	
Is active	<input checked="" type="checkbox"/>
Main contact	Shane Lilly
County postcode	Dublin 2
Managing office	Cork

Submit Changes

Back to the list Delete

Figure 9: Edit page extending underlying Base template

As the reader can determine from the above screenshot, the links have now been restyled using css stored in the button class which represents anchor links styled to look similar to buttons, closely modelled on the css used in the Bootstrap model but remaining as links and not buttons. A colour coded approach was used throughout the application as shown in the screenshot below.

Service UserDetails

Back to the Service User list Edit Service User Record Delete Service User

First name	Joe
Last name	Mangel

Rosters for this Service User

Add New Roster

Figure 10: Button colour scheme

- **Blue buttons** were used to show additional detail on the selected records.
- **Green buttons** were used to indicate either a step to add a new record or the commit/save action following the editing of an existing record .
- **Grey buttons** were used to navigate backwards from the context of an individual record to a list which shows an index view of multiple records.
- **Amber buttons** were used to indicate a step to edit a record, implying changing the saved values.
- **Red buttons** were used to indicate the deletion of a record or the removal of an association between two records.

Where the entity in question contained address data, the underlying controller implemented the generic Mapping class which was created as part of the application to include the functionality to automatically geocode addresses using integration with the Google Maps API. The resulting coordinates were then stored individually as latitude and longitude and shown in the detail/show view of each entity which used an I-Frame using standard Google Maps code to display a map on a HTML page based on including two coordinates in the URL passed to the I-Frame.

This approach to simply extend base pages and use the generic Symfony templates was deemed sufficient for most simple templates throughout the application. However, some more complex pages, such as the show view for the employee and service user entity needed to expose a number of underlying entities which relate in a many to one relationship back to the parent entity.

For example, the view action for an employee record needed to show a number of different panes of information to the user, which all needed to be accessible to user from the show page for that employee, as follows:

- **Employee details** displayed on the form and editable using the edit employee button which routes to an edit page.
- **Rosters for the employee.** This area was of central importance to viewing employee details. (A similar approach was taken to the Show page for a service user record.) In this case, as already noted above, it had been intended to use an available JQuery library such as Full Calendar to display a custom control which represented the dates in the roster calendar for an individual service user or employee in an Outlook or Google Calendar style planner view, with CRUD functionality available to the user in a drag and drop approach where a user could click on a time in the calendar and drag the control to the finish time of the appointment. However, some difficulties were encountered in this area, mainly related to the author's lack of experience with handling the JSON events that arise from such JQuery driven controls. As a compromise a simpler HTML driven calendar was implemented as an interim measure with the ability to display and edit (each item displayed acted as an anchor link to the underlying roster record) existing roster. Users also have the ability to click the add button on a given date. The approach that was taken was to include three separate calendar controls which automatically updated based on the date context when the page is rendered. The first one shows the current calendar month, then the next calendar month and finally the previous calendar month, all shown in a top down view, with the current calendar month. The user can edit roster items in the current and next month and can only view items in the previous month.
- **Association to other related records.** It made sense for the user to be able to add a related record associated with the employee, for example a planned temporary employee absence (period of leave) or a time in the week when the employee was not permanently noted as unavailable, for example due to family commitments, or a second job held. Similar requirements existed on the service user record where it was needed to be able to add an association to an employee as either permanently assigned

to that service user, or alternatively to mark an employee as flagged not to be sent to that service user.

- **Google maps.** There was a requirement identified to display the Google maps representation of the address coordinates in a static map on the employee and service user view (Show) page.

These challenges called for a more structured approach to the user interface design than for most other pages in the site.

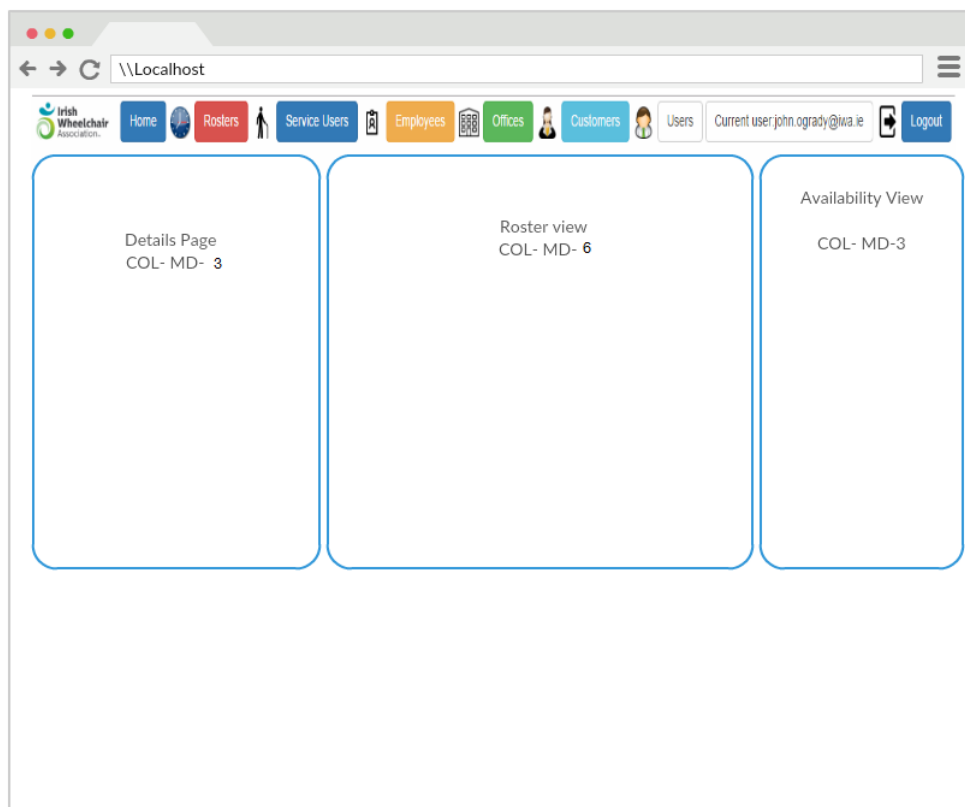


Figure 11: Bootstrap template approach

The final design involved extending the base template to include the top navigation ribbon and then to apportion the page to different sized panels left, centre and right. For this the Bootstrap CSS template, originally developed by the team at Google, but now open sourced and widely used across the modern web, was an ideal fit.

This approach allows you to define the amount of screen real estate to be assigned to different controls on the page and the page is then automatically resized to allow for the different resolution of different devices, including allowing for a responsive rendering of the page on mobile devices. In practice, this may mean that a series of controls are rendered left to right on wide screen devices such as a laptop or large tablet with 1080pixels available in the horizontal

dimensions but will be rendered top to bottom on a narrower screen device such as smartphone. The Bootstrap template uses a grid of 12 parts so a control mapped to the the col-md-3 class would be allocated three twelfths (or one quarter) of the overall width of the screen. The approach that was taken, bearing in mind the amounts of data to be displayed in each area was to allocate three screen units (out of 12) to the leftmost details pane, six units to the roster area control, and three units to the availability view for employees (which also includes the Google Maps control further down in the pane).

A sample of the final Service User design using this methodology is shown on the next page.

66

9	Evaluation
9.1	User Feedback
9.2	Further Enhancements
10	Conclusions
10.1	Review of material covered
10.2	Further Development Opportunities
10.3	Outstanding Issues/Continuous Improvement Plan
11	Appendices
11.1	Code Listing
11.1.1	Database creation scripts
11.1.2	Entity Classes
11.1.3	Mapping Extension Classes
11.1.4	Repository Classes
11.1.5	Form Type
11.1.6	Controller Classes
11.1.7	Twig Templates
11.1.8	List of Vendor tools used