



Niklas Kiuru

# Implementation of Product Resourcing in an Employee Scheduling System

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

11 October 2021

## Abstract

Author: Niklas Kiuru  
Title: Implementation of product resourcing in an employee scheduling system  
Number of Pages: 39 pages  
Date: 11 October 2021

Degree: Bachelor of Engineering  
Degree Programme: Information and Communications Technology  
Professional Major: Mobile Solutions  
Supervisors: Peter Hjort (Senior Lecturer)

---

This thesis explores how to support product resourcing by expanding an enterprise resource planning system's employee scheduling feature. The purpose of the work is to discuss the steps and challenges faced when implementing a large feature on a large existing codebase using AngularJS and Symfony frameworks. Furthermore, the technologies used and how they facilitated or hindered the development will be analyzed.

This thesis covers how the project was planned, how the AngularJS and Symfony frameworks operate, and how the new features of the scheduling system were implemented for the client and server. AngularJS is a front-end web framework developed by Google that aims to speed up development. Symfony is a web server framework that especially facilitates enterprise application development.

Although both the AngularJS and Symfony frameworks are both considered legacy, and no longer are under active development, this project showed that the newest web technologies are not always necessary when implementing new features. There were challenges caused by AngularJS' architecture that slowed development. The large codebase also caused issues, as it meant there were numerous features that had to be supported. The employee scheduling system was expanded to allow products and sales packages to be attached to the work shift. The result of the thesis was a feature rich employee scheduling system that the client uses to plan private music events.

Keywords: AngularJS, employee scheduling, ERP, Symfony

## Tiivistelmä

Tekijä:	Niklas Kiuru
Otsikko:	Tuoteresursoinnin toteutus työvuorosuunnittelujärjestelmässä
Sivumäärä:	39 sivua
Aika:	11.10.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaajat:	Lehtori Peter Hjort

---

Insinööriyössä oli tarkoituksena tutkia ja kuvata toiminnanohjausjärjestelmän työvuorosuunnitteluominaisuuden jatkokehitystä. Jatkokehityksen tarkoituksena oli laajentaa olemassa ollutta työvuorosuunnittelua tukemaan tuotteiden resursointia, jotta se tukee viihdealan yritysten tarpeita. Insinööriyössä pohdittiin niitä haasteita ja vaiheita, joita ilmaantui kehitysprojektin aikana ja suuren koodikannan vuoksi.

Toiminnanohjausjärjestelmä kehitettiin AngularJS-käyttöliittymäohjelmistokehystä ja Symfony-verkkosovelluskehystä hyödyntäen. Googlen kehittämä AngularJS on ensimmäinen versio suositusta Angular-ohjelmistokehystä, jonka tarkoitus on helpottaa käyttöliittymäkehitystä. Symfony on palvelinohjelmointiin kohdennettu ohjelmistokehys, joka tarjoaa työkaluja, jotka edesauttavat suurien projektien kehitystä. Työvuorosuunnittelu on vain yksi osa laajaa toiminnanohjausjärjestelmää. Ohjelmisto myös tukee muun muassa työajanseurantaa, palkanlaskentaa ja laskutusta.

AngularJS- ja Symfony-kehiksiä pidetään vanhentuneina eikä niitä kehitetä enää aktiivisesti, mutta tämä projekti osoitti, että uusimmat verkkoteknologiat eivät aina ole välttämättömiä uusien ominaisuuksien kehittämisessä. AngularJS aiheutti kuitenkin haasteita sen arkkitehtuurista johtuen. Suuri koodikanta myös aiheutti ongelmia, koska tuettavia ominaisuuksia oli paljon.

Työvuorosuunnittelua laajennettiin siten, että työvuoroihin pystyy liittämään erilaisia myyntituotteita ja tuotepaketteja. Insinööriyön tuloksena syntyi laaja työntekijöiden resursointijärjestelmä, jota insinööriyön asiakas käyttää yksityisten musiikkitapahtumien suunnitteluun.

Avainsanat: AngularJS, työvuorosuunnittelu, toiminnanohjausjärjestelmä, Symfony

# Contents

## List of Abbreviations

1	Introduction	1
2	Application Architecture	2
2.1	CERP	3
2.1.1	MariaDB	3
2.1.2	PHP, Symfony and DoctrineORM	3
2.2	Client	7
2.2.1	AngularJS Architecture	7
2.2.2	AngularJS templates and styling	8
2.2.3	AngularJS Controllers and Services	10
3	Project Specifications & Planning	11
4	Implementation	14
4.1	Resources	15
4.1.1	Addresses	17
4.1.2	Comments	20
4.2	Resource Packages	20
4.3	User Interfaces	23
4.3.1	Resource Management	24
4.3.2	Employee Resourcing	28
4.3.3	Assignment views	34
5	Analysis	36
6	Conclusion	37
	References	39

## List of Abbreviations

- API: Application Programming Interface. A connection through which applications or computers can share information.
- CRUD: Create, Read, Update, and Delete. Basic operations for the persistent storage of data.
- CSS: Cascading Style Sheets. Language used for determining the presentation of HTML.
- CTE: Common Table Expression. Temporary named result set, that can be referenced by other SQL statements
- DQL: Doctrine Query Language. An object-query-language used by Doctrine.
- ERP: Enterprise resource planning. Applications that are used to manage business processes and store data.
- GUI: Graphical User Interface. Interface that allows users to interact with software through graphical icons.
- HTML: HyperText Markup Language. Language used for displaying documents on a web browser.
- HTTPS: Hypertext Transfer Protocol Secure. Encrypted communication protocol used widely on the internet.
- JSON: JavaScript Object Notation. Human-readable file format often used for communication between web applications and servers.
- MVVM: Model–view–viewmodel. Software architecture that facilitates the separation of user interfaces from business logic.

- ORM: Object-relational mapping. Technique for converting objects so that they can be stored in a database.
- PHP: PHP: Hypertext Preprocessor. General purpose scripting language focused on web development.
- RDBMS: Relational Database management system. Software system used for managing a database.
- SQL: Structured Query Language. Query language used for managing data in a relational database.
- SCSS: Syntactically awesome style sheets. Scripting language interpreted into CSS.
- URL: Uniform Resource Locator. A reference to a resource that specifies its location on a computer network.
- YAML: YAML Ain't Markup Language. Human-readable data-serialization language.

## 1 Introduction

With the rising trend of digitalization more businesses are looking for software solutions to increase the efficiency of their business. One of these solutions is the introduction of an enterprise resource planning system (ERP). This thesis discusses a customer project to expand the employee scheduling system to support their specific use case. The project was done as a part of my work for Linkity Oy. The customer, Popmaster Oy, required a system in which employees and physical resources could be assigned to a job. The core of the features already existed in the Linkity enterprise resource planning system, and this thesis will explore and analyze the process of expanding the employee scheduling system and its ancillary features to support the customer's use case. The entire project consisted of other features as well, but this thesis will focus on the features relating to the scheduling system.

A work shift planning system or also known as an employee scheduling system is a software solution that automates and tracks the scheduling of employees to different jobs. The aim of these systems is to optimize resource allocation, lower the amount of manual work required to create schedules, and to increase employee productivity with the overall goal of reducing costs. [1, pp. 3 – 6.]

The Linkity ERP system is a large application consisting of two parts: a server built using PHP and an AngularJS based web application as the client. The employee scheduling feature of the ERP is only one of the features of the system. The Linkity ERP handles payroll automation, timekeeping, accounting, invoicing, and employee management. This work will focus mainly on the employee scheduling features and only introduce concepts from the other parts of the ERP when they are directly influencing implementation decisions or to give necessary context.

The thesis will cover the architecture and technologies used in the Linkity ERP system. Then the project's specifications are explored in detail, followed by

detailed information on how the employee scheduling features were expanded and implemented. Finally, areas of improvement will be discussed along with which approaches worked and which did not and what challenges arise from working on a large existing codebase. The focus of the thesis will be on the client side of the project to reflect the amount of work it required.

## 2 Application Architecture

The Linkity ERP consists of CERP, the server component and Majakka, the client. CERP is built using the Symfony framework and Majakka using the AngularJS web framework. Majakka can be used either from a browser or a hybrid application that can be installed on a phone as depicted in figure 1. Communication between the two applications occurs over HTTPS using JSON (JavaScript Object Notation) with all requests going to a single URL (Uniform Resource Locator). CERP then determines the operation to be executed, from parameter inside the JSON passed from the client.

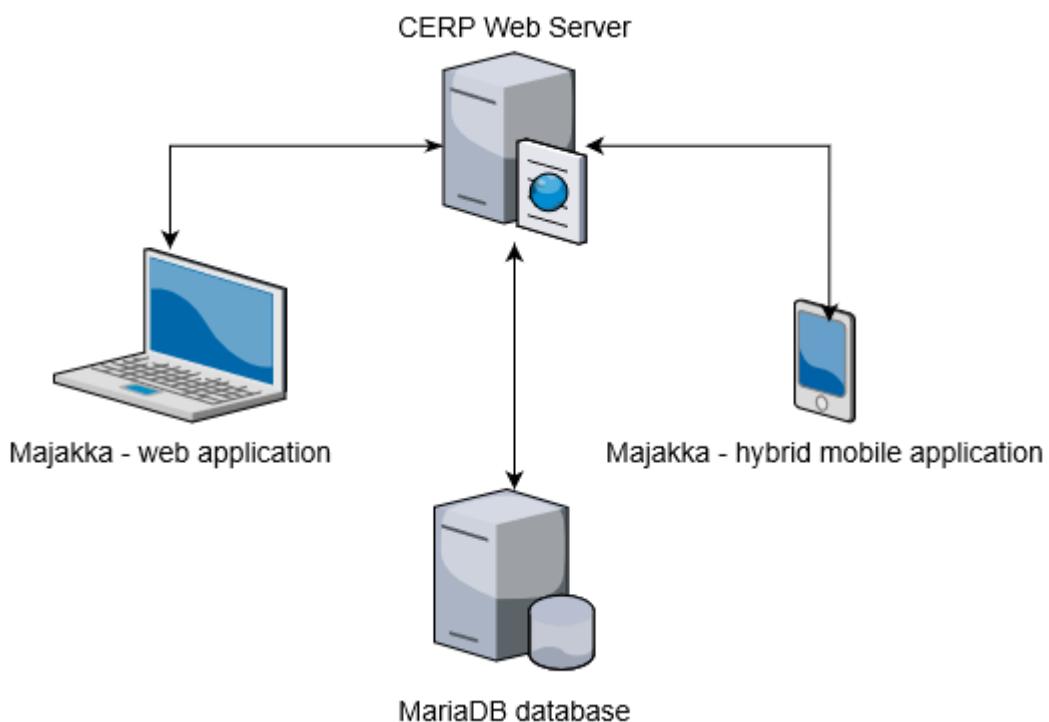


Figure 1, Linkity ERP architecture



## 2.1 CERP

Cushy Enterprise Resourcing and Planning system is the standalone server component of the Linkity ERP. The server component of the Linkity ERP is a web server built using the Symfony framework version 2.7. The Symfony framework and the server are both written in PHP. In short, the server provides an API (Application Programming Interface) with endpoints to complete CRUD (Create, Read, Update and Delete) operations on different data. [2.]

### 2.1.1 MariaDB

MariaDB is the database that is used by the server. MariaDB is a community-developed open-source fork of MySQL. MariaDB is a relational database management system (RDBMS), and it uses SQL for querying and maintaining the database. Relational databases organize the data into tables of columns and rows with each row denoted by a unique identifier. Generally, each table represents a specific type of entity such as “user” with the rows representing an instance of that entity. Columns are different attributes or properties that the entity has like “name”. [3.]

### 2.1.2 PHP, Symfony and DoctrineORM

A common architecture in Symfony projects is separation of code into entities, repositories, managers, and handlers. Entities provide the definition for what properties are included in the entity and have getters and setters for them and must have a unique identifier or primary key. The actual database table is defined in a YAML (YAML Ain't Markup Language) file with the necessary metadata, such as column length and data type. Repositories are for querying and finding records in the database. Each entity has their own repository automatically generated. A typical request travels through a handler and a repository before being returned to the client as shown in figure 2.

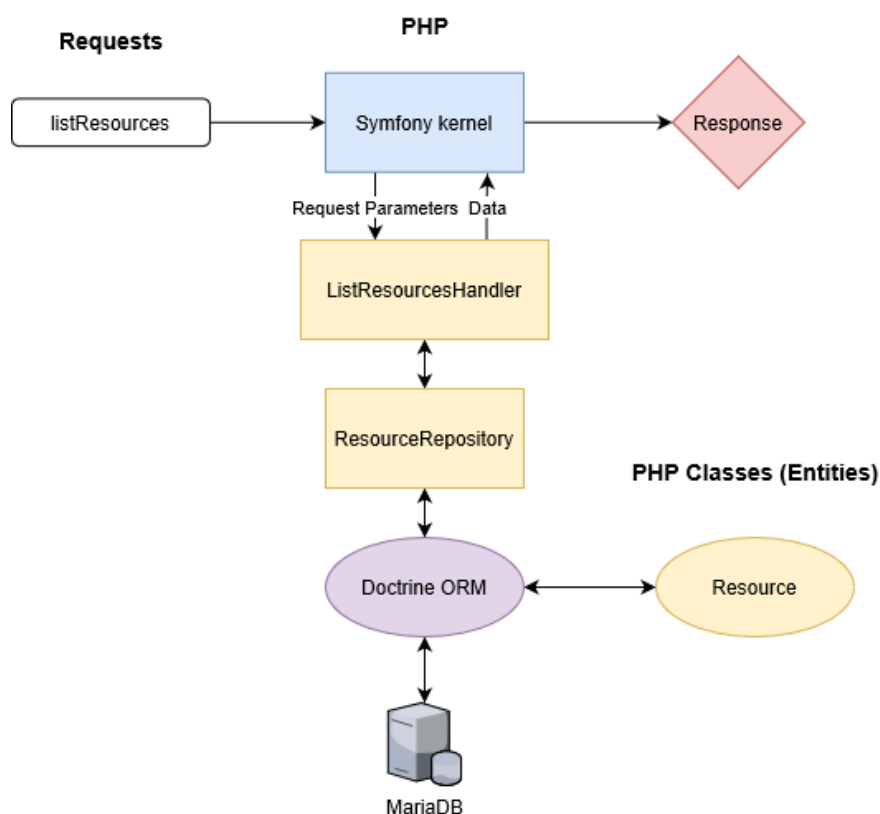


Figure 2, simplified path a request takes through CERP.

CERP uses the Doctrine ORM or object-relational mapper for providing persistence for PHP objects [4]. Object-relational mappers convert data between incompatible type systems. PHP objects cannot just be added to a relational database as the object contains non-scalar values. Relational databases are not object-oriented and can only store scalar values organized into tables. ORMs automatically convert the non-scalar values in the objects into scalars when saving records into a database as depicted in figure 3. The same process also happens when retrieving records from a database, the rows are turned into objects containing non-scalar values according to the map. The ORM also handles persistence, or the saving of objects into the database. In most cases developers do not need to write any SQL by hand to create new rows in the database. The outcome of this is less code and lower complexity. [5.] Furthermore, Doctrine ORM handles relationships between different entities, thus developers do not need to worry about foreign keys and joins in simple cases. The downside to this is poor performance when dealing with thousands of records with many relationships [6].

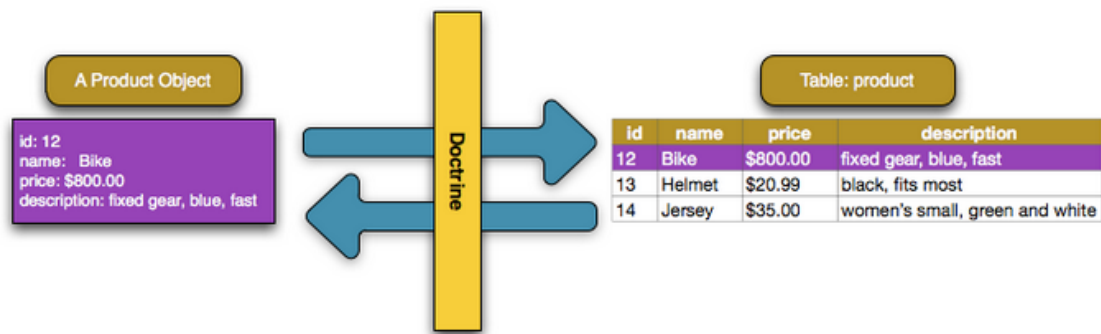


Figure 3, Doctrine mapping between object and database table [4].

Developers can also create their own repositories which extend the EntityRepository where they can separate the query logic from the handlers. This makes the code more reusable as it is no longer tightly coupled with the model as there is a one-way dependency between entities and the ORM. All native SQL or other queries should be placed in each entity's own repository. [5.] A repository is a mediator between the entities and the ORM and acts like a in-memory entity-object collection [7].

DoctrineORM uses DQL (Doctrine Query Language) which is an object-query-language and a dialect of OQL (Object Query Language). Contrary to native SQL queries, DQL does not refer to columns or tables but to the DoctrineORM entities and their properties. [8.] This allows developers to combine the ease of working with objects and the power of native SQL clauses. Furthermore, there is an API for building queries using DQL called QueryBuilder. The same query is shown in listing 1 defined using DQL and the query builder. The builder provides functions such as “select”, “innerJoin”, and “groupBy” that can make building queries simpler. The queryBuilder can be especially useful when the query depends on dynamic conditions as concatenating SQL strings can become very hard to read and maintain. [9.]

```
$query = $em->createQuery('SELECT u FROM Cerp\Model\User u WHERE u.id = ?1');
```

```
$qb->add('select', 'u')
    ->add('from', 'User u')
    ->add('where', 'u.id = ?1')
```

```
->add('orderBy', 'u.name ASC');
```

Listing 1. A simplified user query using DQL and the query builder.

Managers are PHP files that contain utility functions that are related to an entity or concept. It is an implementation of the mediator pattern which is a software pattern which is used to encapsulate logic for how objects interact with each other. For example, a `MessageManager.php` would contain functions for sending push notifications, SMS messages and emails. These are functions that are used in many places; thus, it is logical to separate them from request handlers. [10, pp. 273.]

Request handlers are also PHP files with the responsibility of responding to a request from the client. In CERP, request handlers contain the operation name of the API endpoint, a JSON schema for the request and response, and the code responsible for responding and completing the request.

```
"createResource": {
  "type": "object",
  "title": "Create resource",
  "properties": {
    "name": {
      "type": "string"
    },
    "businessUnitId": { "type": "integer" },
    "isUnlimited": { "enum": [ "limited", "unlimited", "unique" ] },
    "isActive": { "type": "boolean" },
    "amount": { "type": "integer" },
    "description": { "type": "integer" },
    "resourceClassIds": {
      "type": "array",
      "items": {
        "type": "integer"
      }
    }
  },
  "image": {
    "type": "object",
    "properties": {
      "name": { "type": "string" },
      "mimeType": { "enum": [ "image/jpeg", "image/png" ] },
      "imageDataInBase64": { "type": "string" }
    }
  }
}
}
```

Listing 2. Abridged JSON schema of `createResource`

The schema in listing 2 is used for validating and documenting the parameters that can be passed to that handler and what data the request returns to the client. The server returns an error if the client passes an invalid value for a property, which reduces the chances for errors in production code. An example of a request handler is list resources depicted in figure 2, which returns the addresses requested by the client through running a native SQL query which is contained in a repository. Each different API operation has its own handler.

## 2.2 Client

The client portion of the Linkity ERP, Majakka is a single page web application (SPA) built using AngularJS version 1.8.0, HTML and SCSS. Majakka is a hybrid web application, meaning it can be accessed using a web browser or installed on a smartphone. Aside from the AngularJS framework, the client has few external libraries except for jQuery, moment and lodash.

### 2.2.1 AngularJS Architecture

AngularJS is an open-source web framework developed by Google and first released in 2010 [11]. AngularJS aims to simplify and accelerate development by providing a framework for client-side model-view-viewModel (MVVM) architecture. Some of the design goals for AngularJS include decoupling the document object model (DOM) from application logic, decoupling client code from server code and finally to provide structure for the user interface and business logic. [12, pp. 2,3.]

Model-view-viewmodel is a software architectural pattern that organizes different parts of code into separate elements where each part has a specific task. Specifically, it facilitates the separation of the development of the graphical user interface or GUI from the development of the business logic. These three elements are the model, view, and viewmodel. The model typically represents some real-world entity and is at the core of the MVVM. The view is made up of all the functions that directly interact with the user, and this is what they see

when using the application. The viewmodel is the code responsible for maintaining the relationships between the view and the model, so that if the value of the model changes so does the value in the model. [12. pp. 2.] This is represented by the two-way arrows between the view and the view model shown in figure 4.

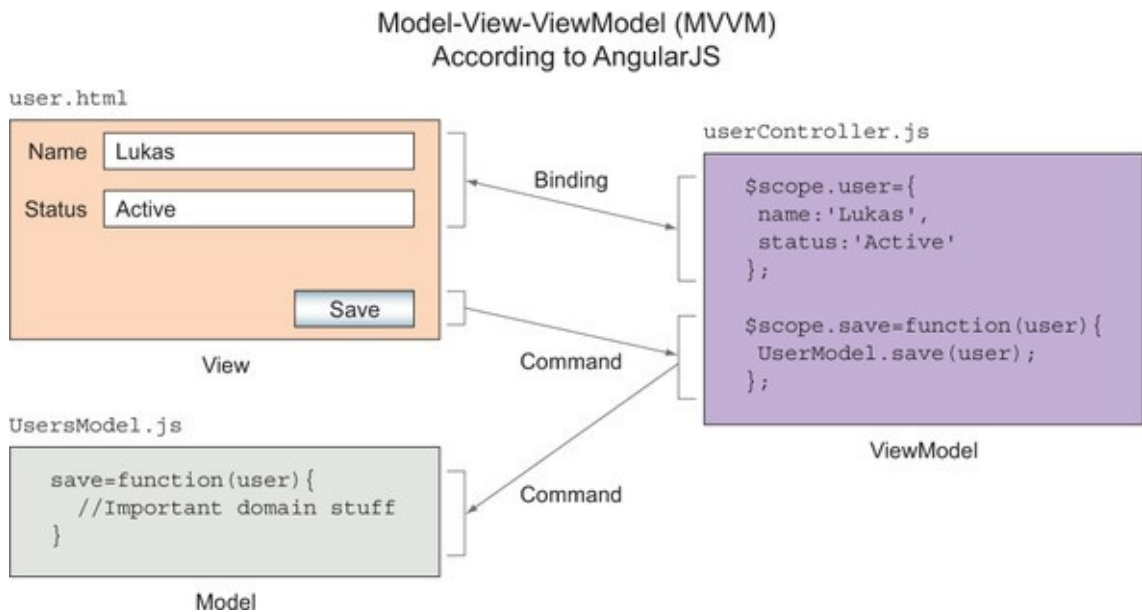


Figure 4, diagram explaining AngularJS' MVVM architectural pattern [13, pp. 37].

### 2.2.2 AngularJS templates and styling

In AngularJS, the view is composed of templates that are written in HyperText Markup Language or better known as HTML, which is the standard markup language for documents meant to be displayed in a web browser. However, the HTML is extended with AngularJS-specific elements and attributes. AngularJS contains 4 types of additional elements and attributes: directives, markup, filter, and form controls.

Directives are markers on a Document Object Model element or DOM element like an attribute, element name or CSS class. These markers tell AngularJS's HTML compiler to attach specified behaviour or to transform the element and its children. Directives can be created, but AngularJS comes with a set of ones built in, like `ngModel`, `ngBind`, `ngClick`, etc. [14.]

Markup refers to the AngularJS's interpolation of content in double curly braces “{{ variable }}”. Any variable or function inside the curly braces will be evaluated and displayed. Whether model is changed in the controller or changed through user input, the mode will be same in both the template and controller. This is the viewModel part of AngularJS and is known as two-way-binding and is depicted in figure 5. The values evaluated in the curly braces can be formatted using built in, or custom AngularJS filters. Usual use cases for filters are formatting dates, currency, or filtering arrays. [15.]

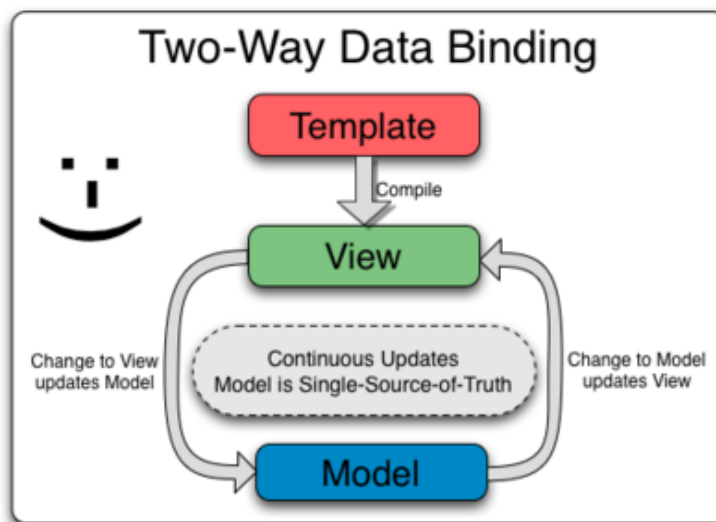


Figure 5, diagram showing how two-way binding updates the model [15].

The final custom set of extensions to HTML are related to forms and their validation. AngularJS allows developers to define a form either using the form HTML tag and name attribute or using the ng-form attribute on any HTML element [16]. Then all inputs that are nested inside the form are bound to it which allows for easy checking of the state of the form. This means its easy to see if a user has touched the form, if its valid, what fields are missing etc. Furthermore, AngularJS has built in CSS classes that the ng-model attribute applies based on the state of the input. For example, if a use inputs an improperly formatted phone number, the ng-invalid class is added to the input. Then the developer can simply write the desired styles to display an appropriate error message to the user. [12, pp. 55.]

### 2.2.3 AngularJS Controllers and Services

The controller contains all the business logic behind the views. Controllers are JavaScript files that are instantiated using a constructor function that create an instance of a controller. The purpose of controllers is to expose variables (the model) and functionality to expressions and directives. Each controller gets their own child scope (`$scope`) through an injectable parameter to the constructor. Any properties or expressions set to the `$scope` are available in the template as well. [13, pp. 39-41.]. The `$scope` acts a regular JavaScript object, thus data can be added to it using dot notation as shown in listing 3.

```
var myApp = angular.module('myApp', []);

myApp.controller('GreetingController', ['$scope', function($scope) {
    $scope.greeting = 'Hello!';
}]);
```

Listing 3. Basic controller initialization with a greeting attached to the scope.

Components are simply a special kind of directive with a simple configuration which facilitates a component-based application structure, such as the one we have in Majakka [17]. These components are independent, modular, and reusable building blocks. A component-based architecture has a strong separation of concerns which speeds up development and encourages code reuse. [18.] In AngularJS components, unlike directives can only be triggered as an HTML element, not an attribute or CSS class. To instantiate a component, the component factory function is used, and a configuration object is passed along to it. An example of a basic component is shown in listing 4. In this configuration object all the attributes (also known as bindings) are defined along with the template, either as a string or the path to the separate HTML file. [17.]

```
angular
  .module('myApp')
  .component('comment', {
    templateUrl: 'views/components/comments/commentComponent.html',
    controller: CommentController,
    transclude: true,
    bindings: {
      comment: '<'
```



```
    },
  });
```

Listing 4. Comment component that accepts a comment as an attribute.

Services are JavaScript files that are wired together with dependency injection (DI) and usually used to share code between different parts of the application. Dependency injection refers to the technique through which objects supply the dependencies of another. [19.] If controller A requires some functionality that is provided in service B, it needs to be injected into the controller. Controller A does need to configure how to configure service B, controller A only needs to worry about how to use service B as the creation of the dependency is handled by AngularJS. [20.] In AngularJS Services are also singletons, meaning that they are shared between all components that refer to it. Services are created using the factory function which rather than creating an instance of the service, it creates the factory function used to instantiate the service when called. A common use case for services is fetching data from the server. This makes it so, that the code used to fetch some specific data must only be written once, instead of multiple times whenever that data is needed which promotes code reuse. [21.]

For the client user interface (UI) components Google's Angular Material UI library is used. These components such as text fields, dialogs, buttons etc. conform to Google's Material Design guidelines and thus are uniform in style and easily reusable. Most of the UI elements used in majakka are material components. The library also provides a service for theming the application through color palettes.

### **3 Project Specifications & Planning**

Prior to actual development, a list of prioritized requirements is agreed with the client. The client for the project is one of the leading DJ and karaoke service providers in Finland. The final product needed to consist of a system using which they could plan their customer's events and bill them according to the services provided.

After the project specifications were agreed upon, the first step was to systematically go through each of the over 90 specifications and requirements from the client and outline whether feature already exists in the Linkity ERP, requires expansion, or is a completely new feature. This discussion was done amongst the senior developers and founder of the company. The outcome was that most of the required features already existed in the Linkity ERP, some required updating and some needed to be completely implemented from scratch.

The second step was to determine in what way the existing features required expansion. This was done by starting at the model level and determining how the data model needs to be expanded. The features were then grouped into five categories and descriptions of how the features should work were written along with what functionality was already implemented. These five categories were:

- Resource management
- Resource as a product
- Venues as a resource
- Shiftable resources
- End-user's user experience

At the core of the project is the work shift. The work shift needs to contain one or more employees, a package which was sold to the customer, and the venue in which the event will take place. The relevant other work shift properties are as follows with an example in parenthesis:

- Project (Customer Oy)
- Job (Wedding)
- Date (5.30.2020)
- Start time (14:00)
- End time (22:00)

The work shift concept needed to be expanded so users can attach a resource package and a venue to the work shift. A resource package is a product which contains multiple work shift requirements e.g., speaker, microphone etc. and

then is visible on the invoice sent to the customer. In the Linkity ERP system this concept exists but it cannot be attached to a work shift, nor does it have the user interface required to create such packages.

Resource classes represent a skillset an employee has, such as DJ. This entity is used as the requirement for work shifts and the resource packages. Each resource needs to have a resource class, so that the system can find the relevant resources to fulfill the requirement. In employee work shift planning, the resource class for employees could be DJ, or presenter. Through this the system can find all employees who are DJs or a presenter which lowers the workload on the planner, as they do not need to know if each employee is a DJ or presenter. For non-employee resources, the resource class would simply be “speaker” or “microphone”, In other words, the resource class is the type of resource.

The resource entity in the Linkity ERP is a high-level concept that represents things that can be booked to a work shift i.e., a shiftable entity. A user and a microphone are both examples of resources, however a user is a child class of resource, while the microphone is simply a resource.

Resource packages are the Linkity ERP representation of a sales product. These packages need to contain any number of requirements and have a price and cost. As the current resource package entity does not support the adding of resources this concept needs to be expanded. Furthermore, the work shift entity will also need to be changed to accommodate the resource package.

Venues are also a new concept that need to be implemented. Venues represent physical locations in which work shifts take place. As resources and venues are very similar, it was prudent to simply make a venue a special type of resource. The type of resource is determined by the resource class, which has a property which marks it as a venue resource class. Venues need to have an address so that distances to workers and the products needed for the shift can be calculated. Likewise, resources need to have addresses as their location is

important so the system can offer relevant resources for any given work shift. The distances are important for efficiency as well, as it makes no sense to go fetch speakers from hundreds of kilometres away if one can be found close by. The same goes for workers as the employer needs to pay for the distance they need to drive to the venue.

There is also a need for the users to be able to comment on venues and resources. This is because there is a need to pass information between the different employees who might do shifts at the venue with the different resources. A description field is reserved for the admin users to edit and for the employees to view. Comments provide a way for employees to make notes about a location or resource and should be visible for everyone. Comments can be edited and deleted by the author or deleted by an admin.

For the venues and resources there needs to be a user interface to create, edit, and delete the created entities. Furthermore, a new interface will need to be created for the worker's view of shifts with venues and products booked to it since the current one does not support it. This new employee work shift interface needs to support adding comments and updating the location of the resources. Due to the addition of booking venues and products to work shifts a completely new interface will need to be implemented for creating them.

The system would be tested by the actual end-users as soon as the features were complete to get feedback as quickly as possible. Once the regular employee's features were implemented the testing would start with them as well. The plan was to use 3-5 regular employees who would be booked to shifts using the Linkity ERP and then feedback and any possible bugs would be reported back.

## **4 Implementation**

Actual implementation of the project begins at the core concept of resource management. After that, packaging of resources into sales products and then

finally integrating everything together in employee resourcing. Most of the development is focused on expanding already existing entities and linking them together so that the client can fully utilize all the features of the server. The only completely new entity are the comment and resource address, everything else already exists or requires expansion. The changes to the server are covered first to introduce all the new entities and features, and afterwards the implementation details of the user interfaces that rely on the new server features are detailed.

## 4.1 Resources

Resources, both worker, product and venue types needed to be able to be assigned to a work shift. The work shift already supports adding resources to on the database level, but the API had no support for it nor did the server support creating simple resources. Thus, request handlers for create, edit, delete, and get resource needed to be created to for resource management. Furthermore, the resource class entity needs to be expanded to support a “venue” resource class.

For server side, the handlers are simple as very little validation is required and the resource entity itself is fairly limited in scope. To create a resource, the client sends a name, the business unit that resource is in, if the resource is unlimited, an array of resource classes and an image in base64. The only required field is the name. The resources created for this project are all the enumerated type “unique” as there exists only 1 of each resource created.

```
$assoc = array(
    "limited" => Resource::ISNOTUNLIMITED,
    "unlimited" => Resource::ISUNLIMITED,
    "unique" => Resource::ISUNIQUE,
);
$isActive = $this->parseBooleanParameter($resourceData, "isActive", false, true);
$isUnlimited = $this->parseEnumParameter($resourceData, "isUnlimited", false, $assoc, 2);
$businessUnitId = $this->parseIdParameter($resourceData, "businessUnitId", false, null);
$amount = $this->parseIntParameter($resourceData, "amount", false, 1);
$resourceClassIds = $this->parseIdArrayParameter($resourceData, "resourceClassIds", false);
```

Listing 5. Parsing request parameters using helper functions.

The parsing of properties send by the client is done through separate functions for each data type. For example, the `parseBooleanParameter` accepts as parameters the whole request object sent by the client, the property name as a string, if the property is required, and finally the default value if it is not present in the request object sent by the client. For enumerated parameters, the list of valid enumerables is provided as an array and the value sent by the client is validated against the array. The resource class ids are validated by checking if the id refers to a valid resource class. If no errors are found in the request, the resource is created by calling the `EntityManager`'s `persist` function, which creates the SQL required for creating the entity and caches it. The actual SQL is executed once the `flush` function is called. The handler responds to the request by returning an JSON object with the resource's id.

Modify resource request works almost exactly as the create handler but it has extra validation for checking if the request's passed `resourceId` exists. The delete handler also validates the id, and as there are lots of references in other entities to the resource, it is not possible to delete it. Instead, the resource is set as inactive and filtered from requests by the server if not specifically requested to list deleted resources as well.

For the list operation, it required changing it to support filtering by resource type. This was required so the client would not have to do the filtering when showing the resources in the system. Finally, the `getResource` operation is created which only accepts the resource id as the parameter and returns, if found, the requested resource.

If the resource exists, it is necessary to convert the data from the PHP object to JSON. Also, the JSON object returned to the client usually has slightly different properties than the ones in the server, so the response object is populated using the getters in the resource entity as shown in the code below.

```
$response->resourceId = $resource->getId();
$response->name = $resource->getName();
$response->isUnlimited = $assoc[$resource->getIsUnlimited()];
if (!is_null($resource->getInventory())) {
    $resourceRow->amount = (int) $resource->getInventory()->getAmount();
}
```

```

}
$response->isActive = $resource->getIsActive();
$response->description = $this->cleanNull($resource->getDescription());

```

Listing 6. Populating response object using resource classes' getter functions.

#### 4.1.1 Addresses

A requirement for resources was that their location should be known.

Resources do not have an address, but the entity already exists in CERP. For the client, there needs to be a convenient way to add either a new address or link an already existing address to a resource. This functionality requires a new list operation. So, the address entity needs to be attached to the resource and added to all relevant resource request handlers. This was achieved by creating the new entity: ResourceAddress. To create the new entity the table needs to be defined in the ResourceAddress.orm.yml file and the ResourceAddress.php class needs to be created. The ResourceAddress class extends the Address class and the variables and their getters and setters for the entity are defined in the PHP class. The YAML file is the mapping file for DoctrineORM and contains all the columns of the table, along with the relationships. The ResourceAddress entity has a one-to-many relationship with the resource entity as shown in the code snippet below.

```

oneToMany:
  resource:
    targetEntity: Cerp\ModelBundle\Entity\Resource
    mappedBy: resourceAddress
    cascade: [ all ]

```

Listing 7. Resource address table's relationship mapping.

DoctrineORM also needs the inverse mapping added to the resource mapping file as shown below. This is naturally a many-to-one relationship.

```

manyToOne:
  resourceAddress:
    targetEntity: Cerp\ModelBundle\Entity\ResourceAddress
    inversedBy: resource

```

Listing 8. Resource table's mapping for resource address.

For listing addresses the `listAddresses` operation is created. The operation needs to be fast, have filters for a multitude of fields and have pagination. This is because there can be tens of thousands of addresses in the database and offering them to the user at once would not be very useful.

```
{
  "listAddresses": {
    "type": "object",
    "title": "List addresses",
    "properties": {
      "addressType": { "enum": [ "resource", "user", "customer", "job", "all"
] },
      "showOnlyPublic": { "type": "boolean" },
      "firstEntry": { "type": "number", "default": 0 },
      "maxEntries": { "type": "number", "default": 9999 },
      "searchTerm": { "type": "string" },
      "userId": { "type": "number" },
      "jobId": { "type": "number" },
      "sortDirection": { "enum": [ "asc", "desc" ], "default": "desc" },
      "sortField": { "enum": [ "id", "street", "city", "zipCode", "country",
"type" ], "default": "id" }
    }
  }
}
```

#### Listing 9. Abridged `ListAddressesHandler.php` request JSON schema.

The list operation was created to be very generic and able to be used for lots of different uses in the client, not just for offering possible addresses for the resource. The generic nature of the request is reflected in the schema, as there are filters for a variety of fields. One of the most important filters is the `addressType`. This allows the client to request only addresses that are set to either a resource, user, customer, or job. The handler itself can be very simple as it only needs to parse the parameters from the request and pass them to the address repository's `searchAddresses` function.

```
WITH CTE AS
( SELECT CASE
      WHEN ua.id and c.id THEN 'customer'
      WHEN ja.id THEN 'job'
      WHEN ra.id THEN 'resource'
      WHEN u.id THEN 'user'
      ELSE 'any'
    END type
  from cerp_address a
    left join cerp_useraddress ua on (a.id = ua.id)
    left join cerp_jobaddress ja on (a.id = ja.id)
    left join cerp_resourceaddress ra on (a.id = ra.id)
    left join cerp_user u on (ua.user_id = u.id)
    left join cerp_job j on (ja.job_id = j.id)
    left join cerp_resource r on (ra.id = r.resourceAddress_id)
```



```

        left join cerp_customer c on (u.id = c.id)
    where a.domain_id = :domainId
        and a.street is not NULL)
SELECT * FROM CTE
WHERE CTE.street not like '%deleted_at%';

```

Listing 10. Part of SQL used for address query.

The SQL query for addresses is quite complex as the address type is not a column in the address table. Instead, it is inferred from the table of the address, e.g., if the joined resource address table has id, then it must be of the resource type. To be able to use the calculated address type in the where clause in the query a common table expression (CTE) was used. In the base query there is no other conditions except that the street name should not contain 'deleted\_at'. This is to filter out deleted and anonymized users, whose addresses must also be anonymized.

```

if (!is_null($searchString)) {
    $searchString = strtoupper($searchString);
    $sql .= " and (UPPER(CTE.street) like :searchString or UPPER(CTE.city)
like :searchString or UPPER(CTE.country) like :searchString or
UPPER(CTE.zipCode) like :searchString or UPPER(CTE.description) like
:searchString)";
    $params['searchString'] = "%".$searchString."%";
}
if (!is_null($type) && $type != "all") {
    $sql .= " and CTE.type = :addressType ";
    $params['addressType'] = $type;
}

```

Listing 11. Code that adds conditions for address type and search term.

To allow for searching, the function checks if the search string passed to it exists and if it does checks if the string matches some of the properties of the address. Namely, the street, city, zip code or description. The type generated by the CTE is used for filtering if it is not "all". The filters added to the query are simply concatenated onto the existing query with each being a new condition in the WHERE clause. Pagination is handled by using SQL's LIMIT and OFFSET clauses, and the sorting handled using the SORT BY clause. All of these are implemented in the server so that the client can simply display the data requested by the user, which both speeds up development and is more performant.

### 4.1.2 Comments

In the requirements for the product resourcing, employees needed to have an ability to mark down extra information about venues and devices. The solution to this was to implement a comment entity in which the employees could write down any notes or remarks about the resources. The comment entity would be made generically as well, to allow for it to be added to other features in the ERP in the future if necessary. Two new entities were created: comment and resource comment. To implement this, single table inheritance is used in which both entities are in the same table: `cerp_comment`.

```
discriminatorColumn:
  name: _discriminator
  type: string
  length: 50
discriminatorMap:
  Comment: Cerp\ModelBundle\Entity\Comment
  ResourceComment: Cerp\ModelBundle\Entity\ResourceComment
```

Listing 12. Comment table's discriminator column definition.

The type of comment is determined using a discriminator column. The discriminator map defines all the types of comments that exist and declares the corresponding PHP class. This pattern of single table inheritance allows for new types of comments to be added with little effort and virtually no performance impact. [22.] The request handlers for listing, creating, deleting, and modifying comments are very similar to the handlers for resources and follow the same conventions.

## 4.2 Resource Packages

The resource package entity is a legacy feature that mainly is used in billing. The entity needed further development to allow for it to contain any number of requirements, which allows it to act as a template for new work shifts which streamlines the process of creating new shifts. The resource package and the job date, which is the system name for work shift, are both sub classes of `ResourceAllocation`. The resource allocation entity is responsible for containing

the requirements to a work shift, and the resources matching those requirements. This means that the resource package already has support for containing the requirements necessary to create sales packages. The work therefore is to add support for adding a resource package to a work shift and modify the resource package response handlers to support adding resources to them.

```
{
  "requirements": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "resourceClassId": { "type": "integer" },
        "requiredAmount": { "type": "integer" }
      },
      "required": [ "resourceClassId", "requiredAmount" ]
    }
  }
}
```

Listing 13. The createResourcePackage operation schema for requirements.

As can be seen in listing 13 above, the requirements contained in a resource package are simply a list of resource classes and the required amount of them. These resource classes represent the requirements for a work shift. For example, the basic package sales product could require one of following resource classes: a microphone, speakers, sound equipment, and a table. This resource package is then added to a work shift which takes the requirements from the given resource package.

Work shifts encapsulate a lot of information as they are very configurable and detailed. The changes necessary for allowing a work shift to contain the resource package are querying the resource package in the list and get jobdate operations and adding, modifying, or removing a resource package in the create and modify jobdate operations.

```
left join jobdate_resourcepackage jdrp on (jdrp.jobdate_id = jd.id)
left join cerp_resource_package rp on (rp.id = jdrp.resourcepackage_id)
```

Listing 14. Joins needed to add resource package to job date.

The resource package was added to the list and get operations simply by joining the `jobdate_resourcepackage` join table and the `cerp_resource_package` tables to the original query. For the create and modify operations the client would send all the requirements and resources booked to that work shift along with the resource package id.

```
//CreateJobdateHandler.php
$requirements = $request->createJobdate->requirements;
if (property_exists($request->createJobdate, "resourcePackageId")) {
    $rpRepository = $em->getRepository('CerpModelBundle:ResourcePackage');
    $resourcePackage = $rpRepository->findOneBy(array(
        'domain' => $domainId,
        'id' => $request->createJobdate->resourcePackageId,
    ));
    $jobDate->addResourcePackage($resourcePackage);
    $requirementsByRCId = array();
    if (is_null($requirements)) {
        $requirements = array();
    } else {
        foreach ($requirements as $overrideRc) {
            $requirementsByRCId[$overrideRc->resourceClassId] = $overrideRc;
        }
    }
    foreach ($resourcePackage->getRequirements() as $requirement) {
        $rcId = $requirement->getResourceClass()->getId();
        if (!array_key_exists($rcId, $requirementsByRCId)) {
            $req = new \stdClass();
            $req->resourceClassId = $rcId;
            $req->requiredAmount = $requirement->getAmount();
            $requirements [] = $req;
        }
    }
}
```

Listing 15. Create job date handler's code for adding a resource package.

As shown in the code snippet above, an array is created from all the requirements passed to the handler with the key set as the id of the requirement. Then the requirements contained inside the resource package are queried and checked if the client has passed it. If not, then it is added to the work shift's requirements. This extra check makes sure that even if the client does not send the full requirements array to the server, it still contains all the requirements that are present in the resource package. This is important for data integrity.

```
if (is_null($request->modifyJobdate->resourcePackageId) && !is_null($jobDate->getResourcePackage())) {
    $jobDate->deleteResourcePackage($jobDate->getResourcePackage()[0]);
} else {
    $rpRepository = $em->getRepository('CerpModelBundle:ResourcePackage');
```

```

$resourcePackage = $rpRepository->findOneBy(array(
    'domain' => $domainId,
    'id' => $request->modifyJobdate->resourcePackageId,
));
$existingRP = $jobDate->getResourcePackage();
if (count($existingRP) > 0) {
    $contains = false;
    foreach ($existingRP as $rp) {
        if ($rp->getId() === $resourcePackage->getId()) $contains = true;
        else $jobDate->deleteResourcePackage($rp);
    }
    if (!$contains) $jobDate->addResourcePackage($resourcePackage);
} else $jobDate->addResourcePackage($resourcePackage);
}

```

Listing 16. Modify job date handler's code pertaining to resource packages.

For the modify operation, the server expects if the property should be modified, then it is present in the request. This follows the same pattern as the majority of the other modify operations in CERP. If client passes the resource package id as “null”, then it is deleted from the job date object. Otherwise, the code checks if the package is already present in the job date and if not then it is added. In the modify handler it does not override the requirements passed by the client, using the resource package's requirements.

The final necessary feature in the server is the generation of a cost and billable items from the resource package attached to a work shift. Cost items are the costs incurred to the company for selling a product and the billable item represents the item the customer sees on the invoice. The new cost items are generated when creating a new work shift and then again updated when modifying an already existing one. Implementation of the generation of cost items is straight forward as the functionality already exists in the server.

### 4.3 User Interfaces

The user interface work required for the project started with implementation of the views necessary for managing the inventory of resources. Then, the complete overhaul of the work shift creation modal and finally the employee assignment view. The largest of the tasks was the work shift modal, as it needed to be completely re-implemented due to the complexity of the data

model and the amount of configurability needed to be supported in work shift creation.

### 4.3.1 Resource Management

For the admin user to manage their business' resources, a separate user interface for creating, viewing, and deleting them was necessary. In the list it was important for the users to see where the resource was, the latest comment alongside the actual name and the type of sales product it was. Seeing the comment was especially important as the comment is one of the ways employees could relay information about the condition of the resources they take to their gigs. Furthermore, the admin user can immediately see if the resource is deleted by the colored border on the left side of the table.

Resource management interface showing the Resources tab selected. The table displays a list of resources with columns: ID, Name, Business unit, Address, Products, and Newest comment. The table is filtered to show 209 resources. The interface includes a search bar, a 'Show deleted' toggle, and a table with 6 columns and 20 rows of data. A green border on the left side of the table indicates active resources, while a red border indicates deleted resources.

ID	Name	Business unit	Address	Products	Newest comment
27104	22-tuumainen TV 1	Tampere	-	22" TV	-
27113	22-tuumainen TV 10	Tampere	-	22" TV	-
27114	22-tuumainen TV 11	Tampere	-	22" TV	-
27115	22-tuumainen TV 12	Tampere	-	22" TV	-
27116	22-tuumainen TV 13	Tampere	-	22" TV	-
27117	22-tuumainen TV 14	Tampere	-	22" TV	-
27118	22-tuumainen TV 15	Tampere	-	22" TV	-
27119	22-tuumainen TV 16	Tampere	-	22" TV	-
27120	22-tuumainen TV 17	Tampere	-	22" TV	-
27105	22-tuumainen TV 2	Tampere	-	22" TV	-
27106	22-tuumainen TV 3	Tampere	-	22" TV	-
27107	22-tuumainen TV 4	Tampere	-	22" TV	-
27108	22-tuumainen TV 5	Tampere	-	22" TV	-
27109	22-tuumainen TV 6	Tampere	-	22" TV	-
27110	22-tuumainen TV 7	Tampere	-	22" TV	Jalka rikki
27111	22-tuumainen TV 8	Tampere	-	22" TV	-

Figure 6, resource list component, resources tab selected.

As illustrated in figure 6, resource management list consists of a row of filters and actions. The user can select from the list of resources by pressing on a row in the table. This enables the actions in the top right of the action row. The only action that can be done in bulk is deletion. There are two main views to the resource management screen, the resources, and venues tables. This can be toggled using a selector in the top row of the page. This at the same time

configures the create resource modal between creating regular resources and venues.

The sorting, pagination, and filtering of the data is handled in the client. This was done because the already implemented list resources operation did not have support for sorting and pagination. This meant that the complexity of the table is increased somewhat.

```
function sort(ev) {
  var sortBy = ev.sortBy;
  var sortDirection = ev.sortDirection;
  var res = getCache('filteredResources');
  if (sortBy === 'resourceClasses') {
    res = _.sortByOrder(res, [function(resource) {
      return resource.resourceClasses ? resource.resourceClasses[0].name :
undefined;
    }], [sortDirection]);
  } else if (sortBy === 'comment') {
    res = _.sortByOrder(res, [function(resource) {
      return resource.comments.length ?
resource.comments[resource.comments.length - 1].updatedAt : undefined;
    }], [sortDirection]);
  } else {
    res = _.sortByOrder(res, [sortBy], [sortDirection]);
  }
  setState({ sort: { sortDirection: sortDirection, sortBy: sortBy } });
  var startIndex = getData('pagination.startIndex');
  var endIndex = getData('pagination.endIndex') + 1;
  setData({ resources: res.slice(startIndex, endIndex) });
}
```

Listing 17. Sorting function for resource list table.

As depicted in listing 17, sorting the items in the table is implemented using the Lodash function `sortByOrder`. It handles the actual sorting by being given the value, and sort direction using which it will sort the array and return it. When sorting by the resource class of the product, the first resource class of the resource is used and when sorting by comments, the latest comment is used. The sorting function also needs to paginate the data again as the order of the resources has changed. To achieve this, the pagination start and end indexes are fetched from the component's global data object using a getter function, and then the corresponding portion of the resource array is set to be the ones visible to the user. See figure 7 below.

Figure 7, resource modal with resource component.

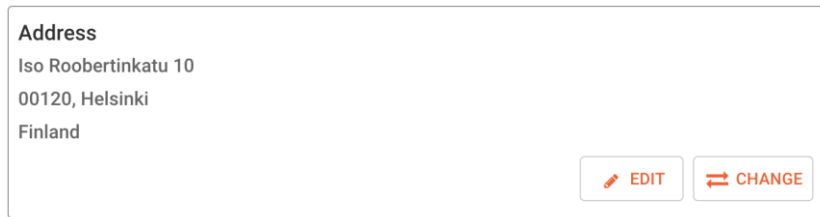
To create a new resource a modal was used which wrapped the resource component. This is because creating resources does not require too much information and it needs to be as easy and quick as possible. If a separate page would have been created, it would have looked empty and slowed down the process too much. The user just must fill in the name of the product and the business unit in which the product is. The exact same component is also used for creating venues and this is done by using the “place” toggle. The only untraditional part of the form is the address input which is implemented as its own component as it could be used elsewhere in the application.

```
<address-component address="$ctrl.address"
  original-address="$ctrl.resource.address"
  on-update="$ctrl.update($event)"
  flex>
</address-component>
```

Listing 18. HTML for displaying address component and it's bindings.

As can be seen in the code above, the data bindings for the component are the resource's address, the original resource address which is necessary when editing an existing resource and the on-update function which handles updating the state of the parent component's address. The address component has multiple states: address display, new address, and address search.



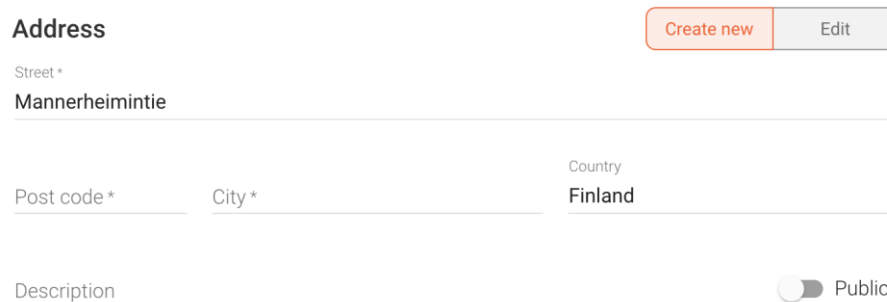


Address  
 Iso Roobertinkatu 10  
 00120, Helsinki  
 Finland

[EDIT](#) [CHANGE](#)

Figure 8, address component's address display.

The default UI for the presented to the user when the resource has an address is the display state shown in figure 8. The component also supports editing the addresses, which presents the address form shown in figure 9. This editing can be dangerous as it will then update the addresses of each resource that have that exact same address. To mitigate this, the selector in the top right of figure 8, allows toggling between editing the original address or simply creating a new one for the resource.



Address [Create new](#) [Edit](#)

Street \*  
 Mannerheimintie

Post code \*      City \*      Country  
 Finland

Description  Public

Figure 9, address creation form.

Another feature that was added with the addition of resource addresses was the ability to create public and non-public addresses. Public addresses can be suggested when searching for an address, but private ones cannot. This is important if the resource is stored at an employee's home address and therefore it is important to not suggest that address to other employees. This is the reason why all addresses are private by default, as slight inconvenience when creating addresses is less troublesome than possibly exposing someone's home address without permission.

### 4.3.2 Employee Resourcing

The most complex part of the project was re-implementing the work shift creation modal. The old one shown in figure 10 was too simple and small to allow for any more additions without becoming difficult to use. The additions of the venue, multiple employees to a single shift and the resources and packages could not be reasonably implemented into the original modal. Furthermore, the modal was created using a legacy AngularJS controller with very few components which lead to a bloated controller. Due to these reasons, the only reasonable solution was to completely rewrite the modal.

Figure 10, old work shift modal.

The first steps before beginning the implementation were to systematically investigate how the original modal worked and what features it supported as the new modal would also need to support at least most of the old features.

Through discussions with other members of the development team, it was decided that as there were many distinct steps to create a work shift, it would be best to separate each step into its own tab inside the modal. This also made it easy to decide to make each separate tab into its own component, with one parent component which handles passing and updating the tabs' data. Furthermore, any features which would be used elsewhere or had easy to separate logic, would also be made into components as shown in figure 11.

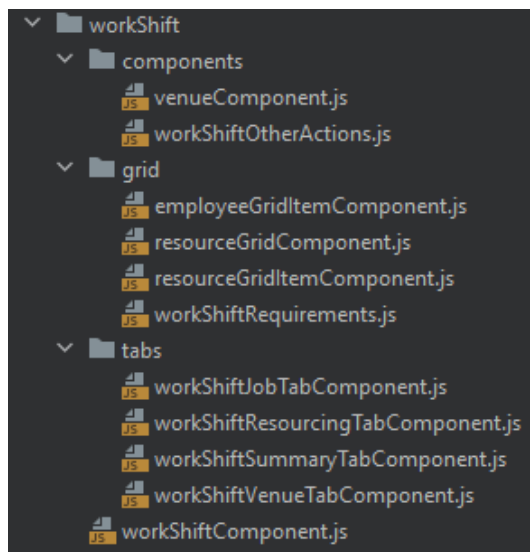


Figure 11, work shift modal components.

The parent component of the modal is responsible for passing data to all the child components and making sure the jobDate object is synchronized between all tabs. The object changes are kept in synchronization by the handleChange function shown below.

```
function handleChange(obj) {
  if (obj && _.isObject(obj)) {
    _.assign(vm.jobDate, obj);
  }
}
```

Listing 19. Handle change function using the lodash assign function.

The function is called from the child components with an object that has the properties that have been updated. This is then merged into the original jobDate object. As all the child components have been passed the reference to the

jobDate object, they all have the changes applied to the object. For instance, the workShiftJobTab component shown in figure 12, calls the handleChange function when the user has selected a job using the selectors in the modal. The parent component also calls functions inside the child components when the user changes tabs.

```
function tabSelected(selectedTab) {
  if (selectedTab === (vm.settings.venueResourcing ? 2 : 1)) {
    if (vm.childApi && _.isFunction(vm.childApi.updateVenue) &&
      _.isFunction(vm.childApi.updateSelectedRequirements)) {
      vm.childApi.updateVenue();
      vm.childApi.updateSelectedRequirements();
    }
  }
}
```

Listing 20. Function that is executed when tab is changed.

As not all businesses need to track the work shift venue it is configurable. Thus, the selected tab index changes and needs to be checked. Tab selected function, in listing 20, is called by the Angular material tabs component whenever user changes tabs. When the resourcing tab is selected, the resourcing component needs to recalculate distances to the venue as it could have changed and initialize the selected requirements.

The screenshot shows a web application interface for creating a new job date. The title is "New Jobdate". There are four tabs: "PROJECT / JOB" (active), "PLACE", "RESOURCING", and "SUMMARY".

The "PROJECT / JOB" tab contains two main panels:

- Jobdate information:**
  - Input field: IHKU - Helsinki
  - Input field: 1 IHKU Helsinki - DJ
  - Notes: (empty text area)
- Jobdate date and time:**
  - Calendar: Shows October 2021. The 9th is selected.
  - Work shift type: Add time manually
  - Time range: 08:00 - 16:00

At the bottom of the form, there is a summary: "IHKU - Helsinki - 1 IHKU Helsinki - DJ" and "01.10.21, 08:00 - 16:00". Buttons for "CLOSE", "SAVE", and "NEXT" are visible.

Figure 12, project tab with project, job, date and time selection.

In the venue tab shown in figure 13, the work shift planner is shown all the relevant details relating to the venue. The component also shows other relevant work shifts that take place in the same venue. To speed up the creation of one-off venues, there is an option to create a new venue by opening the same resource modal as in the management page.

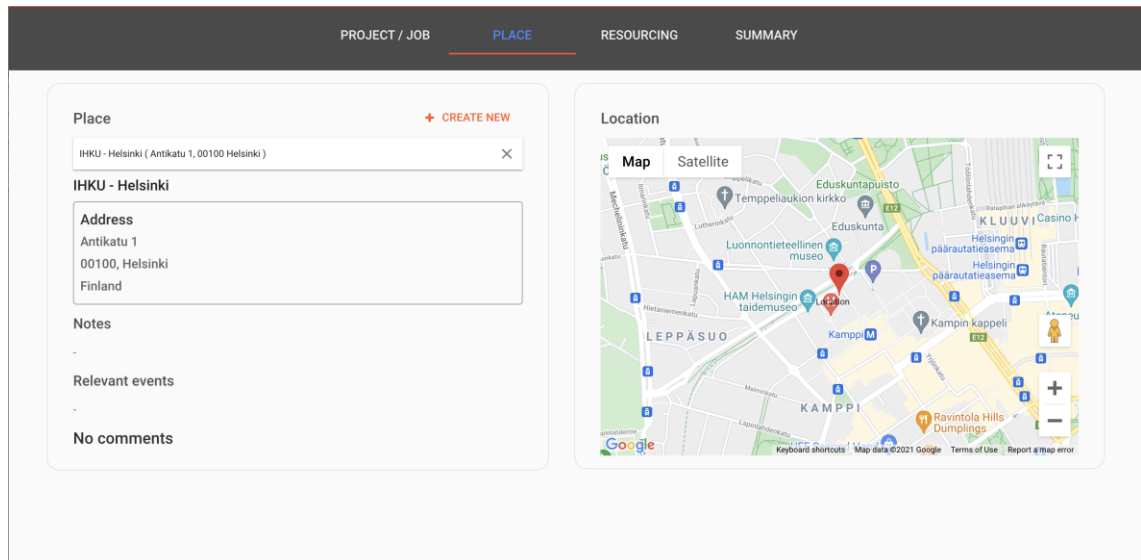


Figure 13, venue tab with venue component and map.

The resourcing tab is the most complex part of the whole modal in respect to interface and programming. The complexity is a result of the many ways the employees can be filtered and the way the server returns data to the client. As can be seen in figure 14, the user first selects the requirement and then can choose additional criteria. The criteria can be a resource attribute, which represents something that the employee has, like a driver's license, or another resource class. In figure 14, the requirement is a DJ who is a part of team 1. The user can select to offer the work shift to an employee or assign them.

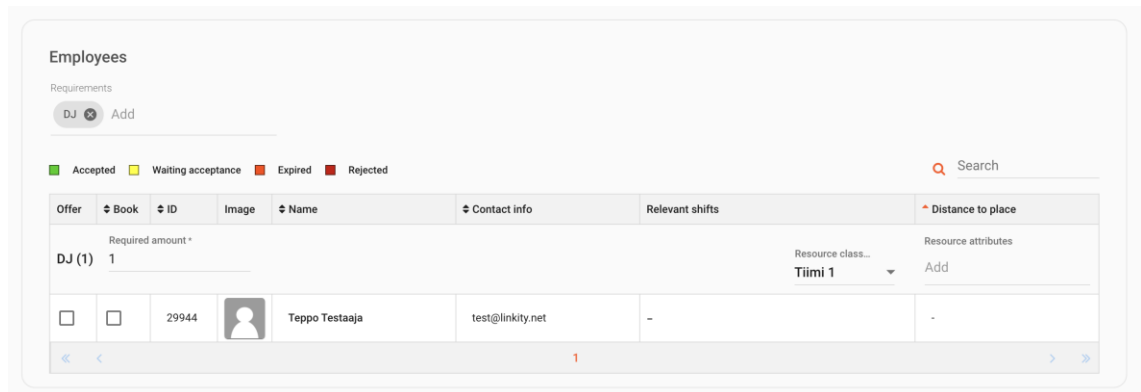


Figure 14, resourcing tab's employee grid table component.

If the user selects a resource attribute, it is added as a column to the grid. As the table is constructed using the CSS grid, the columns will need to be recalculated.

```
var extra = resourceAttributes.length;
requirement.columnStyle = '3.5rem 4rem 5rem 4.05rem 1fr 1fr 1.25fr 0.6fr
repeat(' + extra + ', 0.5fr)';
```

Listing 21. Code for setting CSS grid template columns.

As can be seen in the code above, the additional columns are added using the CSS repeat function. The columnStyle string is used as the value to the grid-template-columns CSS property and is bound to it using ng-style.

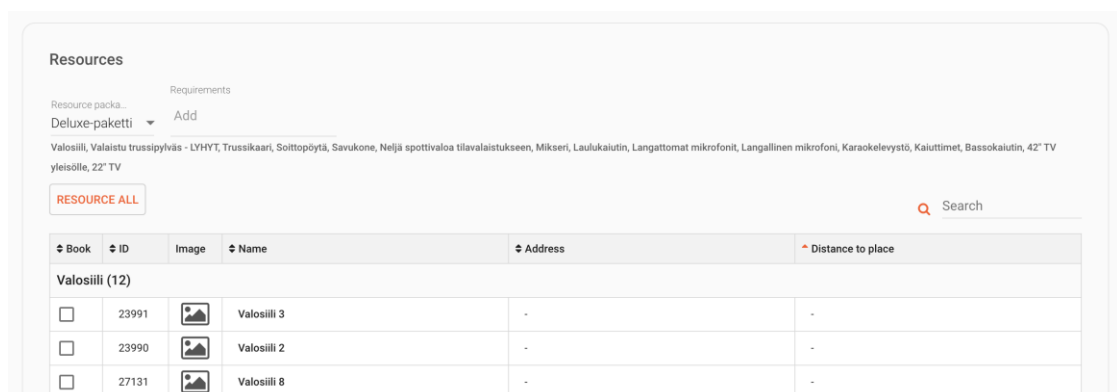


Figure 15, resourcing tab's resource grid table and resource package selector.

The resources grid uses the same grid as the employees, but with a different list item component. This was possible as the employees and product resources are both resources and contain the necessary properties. The resource

package is used as a requirement prefill, and selecting a package, like in figure 15, fetches the resources matching the package's requirements. As clicking on a resource for each requirement would be pointless, a button for selecting the closest resource for each requirement was added.

```
function getDistanceFromLatLonInKm(latitude1, longitude1, latitude2, longitude2) {
  var radius = 6371; // Radius of the earth (km)
  var dLat = deg2rad(latitude2 - latitude1);
  var dLon = deg2rad(longitude2 - longitude1);
  var a =
    Math.sin(dLat / 2) * Math.sin(dLat / 2) +
    Math.cos(deg2rad(latitude1)) * Math.cos(deg2rad(latitude2)) *
    Math.sin(dLon / 2) * Math.sin(dLon / 2);
  var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

  return radius * c;
};
```

Listing 22. Function that calculates distance as a straight line.

The function used to calculate the distance between the venue and resources is shown above and implemented using the Haversine formula. The distance is calculated as a straight line; thus, it cannot be used for very accurate distances and employees would most likely drive to fetch the resources required for a gig. But the cost of using an API Google's Distance API would be very high. Furthermore, it is more important to get a general idea of the distance rather than the accurate driving distance.

The screenshot displays a user interface for managing event resources. It is divided into two main sections: 'General information' and 'Resurssit' (Resources).

**General information:**

- Date: October 1, 2021
- Time: 08:00 - 16:00
- Venue: IHKU - Helsinki
- Location: 1 IHKU Helsinki - DJ

**Place:**

- Venue: IHKU - Helsinki
- Address: Antikatu 1, 00100 Helsinki

**Messaging:**

- Send notification about the work shift
- Re-send work shift notification

**Other actions:**

- SET AVAILABILITY
- HISTORY
- DELETE

**Resurssit (Resources):**

Image	Resource	Offer status	Required amount
<b>DJ</b> Required amount: 1			
	Teppo Testaaja	-	<input type="checkbox"/>
<b>Peruspaketti</b>			
<b>Soittopöytä</b> Required amount: 1			
	Soittopöytä 11	Booked	<input checked="" type="checkbox"/>
<b>Mikseri</b> Required amount: 1			
	Mikseri 3	Booked	<input checked="" type="checkbox"/>
<b>Langattomat mikrofonit</b> Required amount: 1			
	Langattomat mikrofonit 21	Booked	<input checked="" type="checkbox"/>
<b>Langallinen mikrofoni</b> Required amount: 1			
	Langallinen mikrofoni 11	Booked	<input checked="" type="checkbox"/>
<b>Karaokelevyistö</b> Required amount: 1			
	Tietokone 3	Booked	<input checked="" type="checkbox"/>

Figure 16, summary tab showing assigned resources and a collapsible package.

With the large number of details that could be in the work shift, it was decided that a summary tab would be a good solution. The summary tab would also be the default tab when opening a created work shift. The summary tab would also be the right place to check if the shift was created correctly without having to navigate through each tab. This meant that it was the right place to add other actions such as delete or view the shifts history. On the right side of figure 16, the client requested that if a package was added to a work shift it would be shown as an expanding list. This reduced visual clutter and hid any extraneous information from the user.

#### 4.3.3 Assignment views

Similarly, to the work shift modal, the employee's assignment modal had to be reimplemented. The old modal had no support for venue or resources which were crucial for the employee, so they know where to go, and what to take with them to the event. In addition, employees needed a way to add comments to resources and venues and update the addresses of resources. The modal also needed to be designed mobile-first as regular employees use the mobile application on their smartphone instead of the web application on a browser.



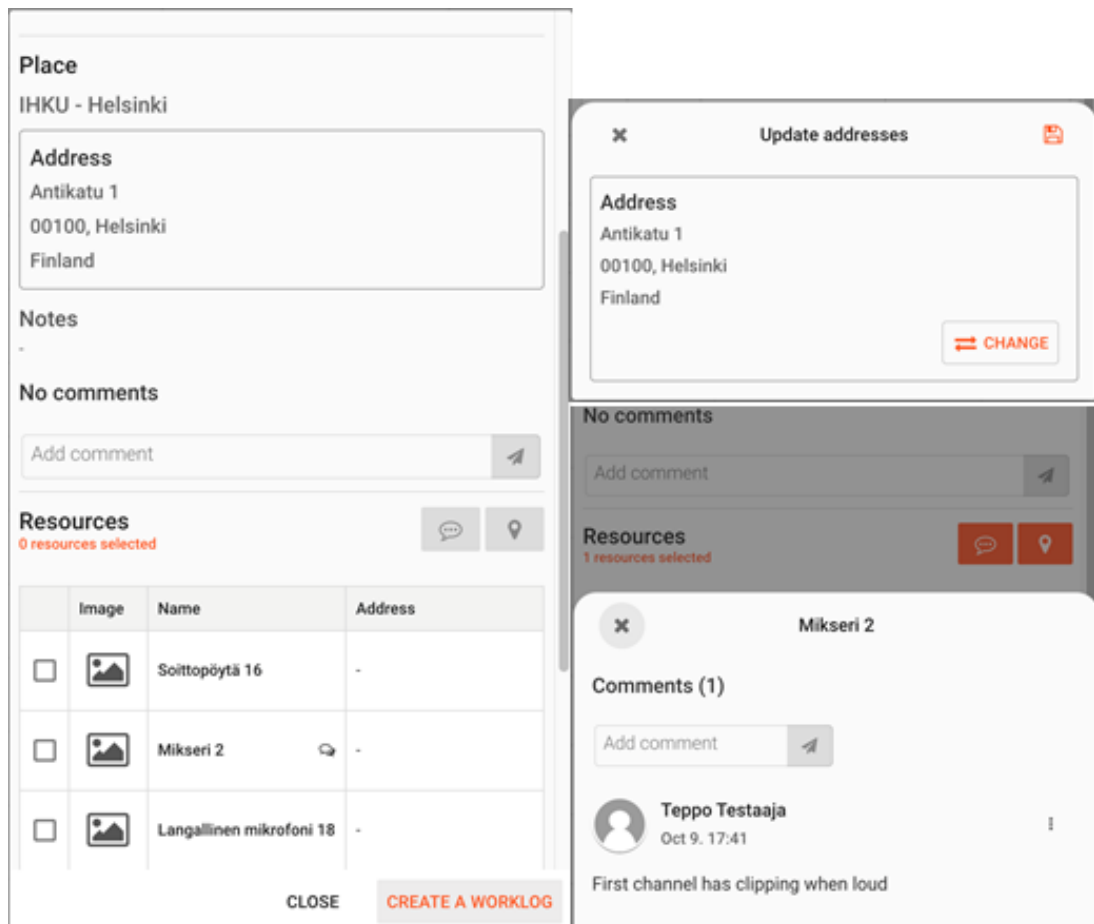


Figure 17, assignment component with address and comment components in the bottom sheet.

The assignment component was created by extending the existing assignment modal and converting it to a component. Then it was simply a case of adding the venue component and creating a list of resources. The resource list can be seen in the left in figure 17 and users can select them by pressing on a row, which enables the actions in the table. The user can either add comments to a single resource or change the addresses in bulk. It was important to be able to change the resources' location easily as repeating the same action multiple times is inefficient and annoying for the user. The actions appear partly on top of the existing content along the bottom of the screen which makes them easy to access on mobile.

## 5 Analysis

The extensions to the employee scheduling and resourcing features proved to be straightforward to implement especially in the server-side. This was because the most complex features, such as booking employees, listing available resources for a shift, and generating bills were already implemented. On the contrary, parts of the front-end were much more complex to implement. The complexity derived chiefly from the data structure of the job date.

The server-side architecture had a strong separation of concerns between different operations. Those that were strongly linked, such as the job date entity and resource, were implemented in a generic fashion which allowed both employees and product resources to be treated the same way. In other words, the foundation for feature expansion was very good as the focus was on creating new features rather than struggling to convert existing code to support them.

The front-end had several simple user interfaces such as the resource management and assignment pages. Those interfaces could leverage already existing code and components, which sped up development and lowered the chances for bugs. However, the work shift modal was the biggest hurdle by far. As the schema for the job date supported an unlimited number of requirements for a shift, then it was decided that the new modal should also support any number of requirements, both employee and non-employee. Then inside these requirements is any number of resources. This nested structure made for complex code which became very difficult to debug. This is where the limitations of the AngularJS' component became evident. Due to the amount of boilerplate required to create components and the difficult inter-component communication, dividing the resource grid tables into small components was challenging.

In respect to the implementation of the features and code structure there is also room for improvement. The grid tables used in both the resource management and the resourcing tab of the work shift modal, could have been made into grid

table component. This would have meant that that features such as sorting and parts of the HTML templates would not have been needed to implement twice. In addition, it would have lowered the code complexity in the resourcing tab of the work shift modal. The resourcing tab required a complete refactoring once the resource package feature was completed as the code simply became too complex. It would have been much wiser to finish all features relating to the new work shift before beginning its implementation.

Some development time was wasted when it was realized that users did not want to very intensive inventory management. This meant that the resource management and address features could have been simpler and therefore easier to use for the users. This could have been avoided if some of the specifications regarding resource management would have been discussed in greater details and other possible solutions shown to the client. Another point of improvement was planning and coordinating delivery order of features better with the client. It became evident that the customer's workflows required the resource packages, and it was one of the last features to be added. This delayed the initial testing phase which could have offered some very valuable feedback before full feature completion. Then at that point it becomes more work to change existing features than to implement them in the wanted manner in the first place.

## **6 Conclusion**

This thesis demonstrates the process of planning and implementing of a product resourcing system on top of an existing employee scheduling system using AngularJS and the Symfony framework. In a broader scope, it also showed the challenges of extending existing software features, especially ones that could be considered legacy at this point. ERP systems are often very large and usually require some amount of configuration and development to fit into new organizations' workflows and use cases. In this project, the client Popmaster Oy was strongly present to offer feedback and development ideas so that the finished product would fit their field.

Symfony framework and AngularJS at this point are both at the end of their lifecycle but as established through this development project, they are very capable even though they lack the support for the latest features like TypeScript or newer versions of PHP. This thesis covered what and how the features of the frameworks facilitated development and how they were used to complete the projects. The frameworks at this point are feature complete and very stable, which made development faster. With basic JavaScript and PHP, development would have taken significantly longer. However, older frameworks lack certain features such as full support for component-based architecture, which can be a hinderance.

The project was mostly a success, and the employee scheduling system is in active use by the client. The features outlined in this thesis are integrated into the Linkity ERP software and make the resourcing features of the system more extensive.

## References

- 1 Ernst A; Jiang H; Krishnamoorthy M & Sier D. 2004 Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*. 2004;153(1):3-27.
- 2 Symphony and HTTP Fundamentals. Online. Symfony SAS. <[https://symfony.com/doc/2.6/book/http\\_fundamentals.html](https://symfony.com/doc/2.6/book/http_fundamentals.html)>. Accessed 10 August 2021.
- 3 Introduction to Relational Databases. Online. MariaDB foundation. <<https://mariadb.com/kb/en/introduction-to-relational-databases/>>. Accessed 10 August 2021.
- 4 Databases and Doctrine. Online. Symfony SAS. <<https://symfony.com/doc/2.6/book/doctrine.html>>. Accessed 10 August 2021.
- 5 Getting Started with Doctrine. Online. Doctrine. <<https://www.doctrine-project.org/projects/doctrine-orm/en/2.4/tutorials/getting-started.html>>. Accessed 12 August 2021.
- 6 Pivetto, Marco. Doctrine ORM Hydration Performance Optimization. 2015. Online. <<https://ocramius.github.io/blog/doctrine-orm-optimization-hydration/>>. Accessed 10 August 2021.
- 7 Rob Mee & Edward Hieatt. Repository. Online. <<https://martinfowler.com/eaCatalog/repository.html>>. Accessed 14 August 2021.
- 8 Doctrine Query Language. Online. <<https://www.doctrine-project.org/projects/doctrine-orm/en/2.4/reference/dql-doctrine-query-language.html>>. Accessed 16 August 2021.
- 9 The QueryBuilder. Online. <<https://www.doctrine-project.org/projects/doctrine-orm/en/2.4/reference/query-builder.html>>. Accessed 16 August 2021.
- 10 Gamma, Erich; Helm, Richard; Johnson, Ralph & Vlissides, John. 1995. *Design Patterns*. Electronic book. Addison-Wesley Professional.
- 11 What is AngularJS?. Online. <<https://docs.angularjs.org/guide/introduction>>. Accessed 6 August 2021.
- 12 Seshadri, Shyam; Green, Brad. 2014. *AngularJS: Up and Running*. Electronic book. O'Reilly Media Inc.

- 13 Ruebelke, Lukas. 2015. AngularJS in Action. Electronic book. O'Reilly Media Inc.
- 14 Creating Custom Directives. Online. <<https://docs.angularjs.org/guide/directive>>. Accessed 16 August 2021.
- 15 Data Binding. Online. <<https://docs.angularjs.org/guide/databinding>>. Accessed 17 August 2021.
- 16 Forms. Online. <<https://docs.angularjs.org/guide/forms>>. Accessed 9 August 2021.
- 17 Understanding Components. Online. <<https://docs.angularjs.org/guide/component>>. Accessed 9 August 2021.
- 18 Afonso, Francisco. 2020. Component Architecture: 3 Reasons to Invest in One. Online. <<https://www.outsystems.com/blog/posts/component-architecture/>>. 22 October 2020. Accessed 20 August 2021.
- 19 Sankar, Anand Mani. 2014. AngularJS Dependency Injection Demystified. Online. <<http://anandmanisankar.com/posts/angularjs-dependency-injection-demystified/>>. 8 September 2014. Accessed 30 August 2021.
- 20 Karia, Bhavya. 2018. A quick intro to Dependency Injection: what it is, and when to use it. Online. <<https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/>>. 18 October 2018. Accessed 30 August 2021.
- 21 Services. Online. <<https://docs.angularjs.org/guide/services>>. Accessed 30 August 2021.
- 22 Inheritance Mapping. Online. <<https://www.doctrine-project.org/projects/doctrine-orm/en/2.9/reference/inheritance-mapping.html>>. Accessed 3 October 2021.

