My Consultancy Services

Delta Ready Technology Pty Ltd

Damian Nolan

Daniel Verdejo

B.Sc.(Hons) in Software Development

April 17, 2018

Final Year Project

Advised by:
Daniel Cregg (GMIT)
&
Patrick Nolan
(Delta Ready Technology Pty Ltd)
Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

About this project

Abstract Any person providing an online personal consultancy services always face the challenge of acquiring, managing and retaining clients and delivery of the service over a disparate array of tools. The primary problem is fragmentation of these tools. Social media platforms such as Facebook are great in providing the networking and promotional aspect of things and while it is possible to run a business this way it is not the most effective and efficient means to an end. Furthermore they are not designed to be business focused. The goal is to streamline the experience and allow a service provider to focus on the service delivery and growing their consultancy rather than having a inordinate amount of time managing clients.

Authors Damian Nolan, Daniel Verdejo

Chapter 1

Introduction

With the commencement of their final year in Software Development at GMIT, the authors were presented with the offer of collaborating on the My Consultancy Services project as junior software developer interns for an external company, Delta Ready Technology Pty Ltd[1]. As a small consultancy company itself, Delta Ready Technology Pty Ltd would use this opportunity to work closely with the interns in creating a start-up product for small business or sole traders which allowed them to streamline the management of the services in which they provide. The project, My Consultancy Services would be a Greenfield project that uses cutting edge technology to solve an interesting business problem and would be directed by the company chief technology officer. As part of an onboarding process to allow the interns to get up and running with some of the latest technologies, research in certain fields was advised prior to the commencement of work and tutoring would be provided throughout the course of product development. A Microservice Architecture would be employed as to separate the concerns of various components and to allow them to be a reusable, maintainable, independently deployable suite of modular services which can be used for the future developments of the services provided by the company.

1.1 User Stories

Consider the following user stories[2]:

- When deciding to offer an online service that involves interpersonal communication, Users want to be able to setup a personal page describing my services, so that I can manage the communication from the point of purchase right through to delivering my online service.

- When users setup personal pages describing their services, Users want to be able to set their pricing and the times they are normally willing to provide services, so that a user can feel like they are in control of their consultancy and their client knows when they are ordinarily available.
- When a user decides to avail of an online service that involves interpersonal communication, they want to be able to pay and have their account created if they do not already have one, additionally have their first session scheduled for them, so that they can manage their schedule accordingly.
- A consultant will be able to setup an online consultancy within ten minutes without training and receive their renumeration for work completed within thirty days without having to chase up with clients for outstanding invoicing. Additionally a client won't have to waste any time chasing up with the consultant on asking how to pay for the service. Double booking of clients should not be a problem as the system should provide adequate feedback when this is likely to occur.

1.2 Objectives

The project is concerned with a number of objectives to be completed in order to reach a minimal viable product or MVP. The following will be created as a suite of containerised Docker[3] services and deployed using a Jenkins continuous delivery (CD) pipeline[4].

- A RESTful web service for account management including all user account based business logic and serving as the primary base of the application.
- A PostgreSQL[5] relational database for user account management.
- A CouchDB[6] per user model to facilitate user content storage using a NoSQL database.
- A Radicale Caldav Server[7] for management of calendar services.
- A RESTful web service for calendar management that handles CRUD functionality of user calendar events and act as a JSON wrapper for the Radicale Caldav Server.
- A client application designed using Stencil Ionic - Progressive Web Application using Web Components.

1.3 Overview

The subsequent chapters of this paper will showcase various considerations of software development and design and aspects of software technology regarding the My Consultancy Services project.

The Methodology includes a deep look into the project management workflow used throughout the entire software development life cycle and what approaches were taken. A Technology Review is included to give an overview and analysis of different languages, frameworks, libraries and standards employed within the project. Finally, System Design and Evaluation provide a briefing on the system architecture, how it was designed and a reflection on the outcomes of the project including robustness, stability and testing.

1.4 Project Resources

As a result of working with an industrial parter, Delta Ready Technology Pty Ltd has request that the source code repository remain private. For any queries regarding this please do not hesitate to email:

- **G00324947@gmit.ie**
- **G00282931@gmit.ie**

Links

- My Consultancy Services - Mono Respository:
  `https://bitbucket.org/deltaready-mcs/mcs-microservices-mono-repo`
- My Consultancy Services - Minor Dissertation:
  `https://github.com/Verdagio/Final-Year-Minor-Dissertation`

Chapter 2

Methodology

2.1 Kanban Management Approach

Throughout the development life-cycle for this project we implemented an Agile Kanban task management approach. Kanban is a management method that enables teams and organizations to visualize their work, outline and reduce or completely remove bottlenecks, and achieve significant operational advancements in terms of quality and throughput[8]. The methodology aims to gradually improve any aspect of an organization, whether it be IT/operations, software development and engineering, staffing, marketing and so on. The core concept of Kanban are the following:

- Visualize Workflow
  - Separate the complete workload into descriptive segments or states, visualized as named columns on a wall.
  - Write each user story (piece of work) on to a card and place in a column to indicate how far along the workflow the piece of work is
- Limit Work In Progress
  - Allocate specific limits to how many pieces of work can be in progress within each workflow segment or state.
- Measure the Lead Time
  - Lead Time, also referred to as cycle time, is the average time required to complete one piece of work. Measure Lead Time and optimize the process to make the Lead Time as predictable and small as possible.

This may look somewhat familiar if you are knowledgeable in the Lean Pull Scheduling System, as Kanban is a direct implementation of this[8]. A piece of work can only progress to the next segment or state when it acquires a slot in there.The implementation of Kanban and other Lean Manufacturing Methods, can significantly benefit workflow in some of the following ways:

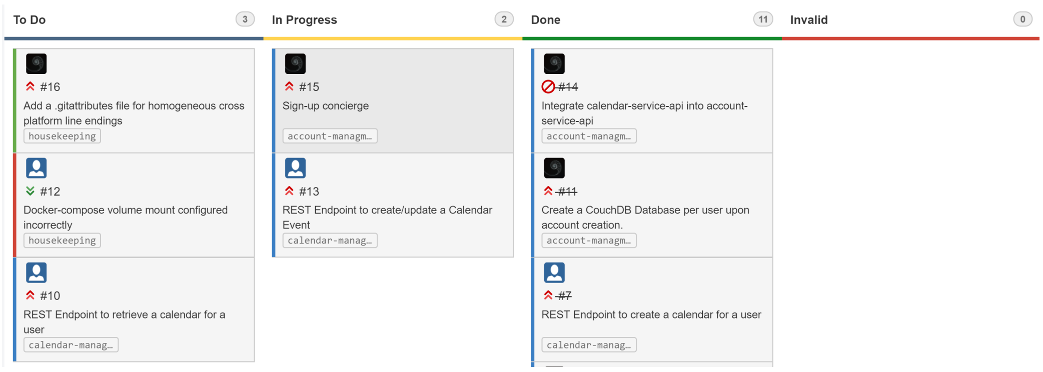
- Visibility of bottlenecks become very apparent in real-time. This promotes collaboration amongst people to optimize the entire value chain as opposed to just their own part.
- Tends to Naturally grow throughout all aspects of the organization, resulting in higher visibility of all goings on within the organization.
- Reduces company costs via reduction of inventory within the range of 25%-75%.
- Continuous support, and increased speed of all pieces of work within the workflow due to visibilty and organization.

Kanban supports continuous workflow, termed Value Stream, when it is applied as a project management approach to software development projects. ”The Value Stream consists of all actions required to bring a project from creation to completion.”[8]. Actions can either add value to the project, or add no value, but may or may not be avoidable. When an action is avoidable it is termed as waste, the elimination of waste is facilitated by the Kanban methodology. In software development terms, there are three types of waste:

- Waste in code development, due to
  - Partially completed work, Defects.
- Waste in project management, due to
  - Extra processes, code hand offs, extra functions.
- Waste in team potential, due to
  - Task switching, waiting for information or instructions.

2.2 Agile Kanban with Git Workflow

Agile Kanban is a Kanban approach to Agile Software Development. In Agile Kanban, the Kanban board is used as a visualization of the workflow. The board will display the status and progress being made on individual tasks, where they can be tracked. Consider Figure 2.1



To Do 3

#16
Add a .gitattributes file for homogeneous cross platform line endings
housekeeping

#12
Docker-compose volume mount configured incorrectly
housekeeping

#10
REST Endpoint to retrieve a calendar for a user
calendar-manag...

In Progress 2

#15
Sign-up concierge
account-managm...

#13
REST Endpoint to create/update a Calendar Event
calendar-manag...

Done 11

~~#14~~
Integrate calendar-service-api into account-service-api
account-managm...

~~#11~~
Create a CouchDB Database per user upon account creation.
account-managm...

~~#7~~
REST Endpoint to create a calendar for a user
calendar-manag...

Invalid 0

Figure 2.1: Kanban Board

Git workflow as a version control tool is a system that tracks modifications to a file or files over the lifetime of a project so that a developer or team can recall specific versions at a later stage. A very common version-control method is to copy files from one directory to another directory. This method is common because it is easy, but is incredibly error prone. It is easy to mistake the directory that is currently active and accidentally write to the incorrect file or copy over files that were not intended to be.[9] The main difference between Git and other version control software tools is the way Git handles its data. Where other systems use what is often referred to as delta-based version control, they store information as a set of files and the changes made to each file over time, Git does not store its data this way. Instead, Git stores its data more like a collection of snapshots of a miniature file system. In Git, whenever a commit is made, a picture of what all the files look like at that point in time is taken and stored as a reference to that snapshot.[9] If any of the files have not been changed, Git does not store the file again, just a pointer to the previous file it has already collected and stored. ”Git thinks about its data more like a stream of snapshots.”[9]

In the Figure 2.1 example a couple of tasks have been created, each given their priority as well as the component they relate to and displayed on the board in the To Do state. A developer can come and move a task from the To Do state into the In Progress state to signify that this task is now being worked on. In order to protect the main development / master branch from becoming corrupted by merge conflicts, where numerous developers may be editing the same segment of work, when a task is started a new branch is created off of the development branch[10]. See 2.2

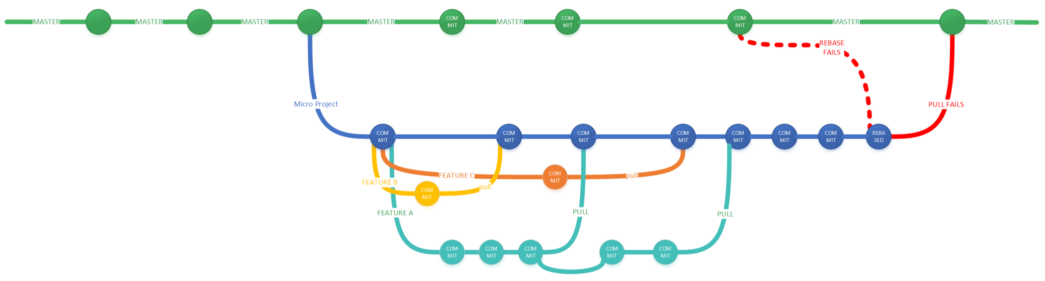


Figure 2.2: Git branches

Once the task is finished passing all integration and acceptance tests, moved into the DONE state. Before a task can be put into the Done state the developer assigned to the task, must create a Pull Request for the task, which in turn will be reviewed by their colleagues / management to confirm that the task meets the specification of the task, conforms to the quality requirements expected, and works as expected[10].

2.3 Slack

Slack is a team collaboration and services tool used by Software Engineering teams world-wide in the development process of software products. With the use of Slack, team members can stay in touch and comment in numerous different chat rooms for sharing code, reporting bugs and general conversation which may be valuable to product development. The use of Slack throughout the software development life cycle in My Consultancy Services proved to be a beneficial and truly invaluable piece of communication software for working remotely and interacting online. Weekly or fortnightly meetings were held in place between the interns and lead product manager. This helped in providing clarity of arising issues and tutoring of new technologies which were implemented throughout development.

Chapter 3

Technology Review

3.1 Docker

3.1.1 What is Docker?

Docker is a platform that enables containerization of applications. Containerization provides standardized units for development, shipment and deployment. Container images are lightweight, isolated, executable packages of software that contain all the components to run the specified; the code, tools, dependencies, and settings. With availability on Linux and Windows based containers, both will run seamlessly regardless of the environment. Containers running in isolation means that regardless of how the environment or infrastructure which the container will run on, the application within the container will be unaffected by discrepancies across the environments, and other applications. Docker provides containers that run on a single machine share the operating system-kernel, resulting in rapid starting and less consumption of the central processing unit (CPU) and random access memory (RAM). Based on open standards, Docker containers can run on all major Linux distributions, Microsoft Windows, Apple MAC OS, and any infrastructure like: virtual machines, bare-metal, and via cloud services.

3.1.2 Docker Containers v Virtual Machines

What are the benefits or differences of using Docker containers over Virtual Machines? While Docker containers and virtual machines have relatively similar resource isolation and allocation benefits, they function differently. Where a virtual machine will mimic the behavior of an operating system running on a physical computer, Docker containers visualizer just the operating system, rather than the operating system running on the hardware

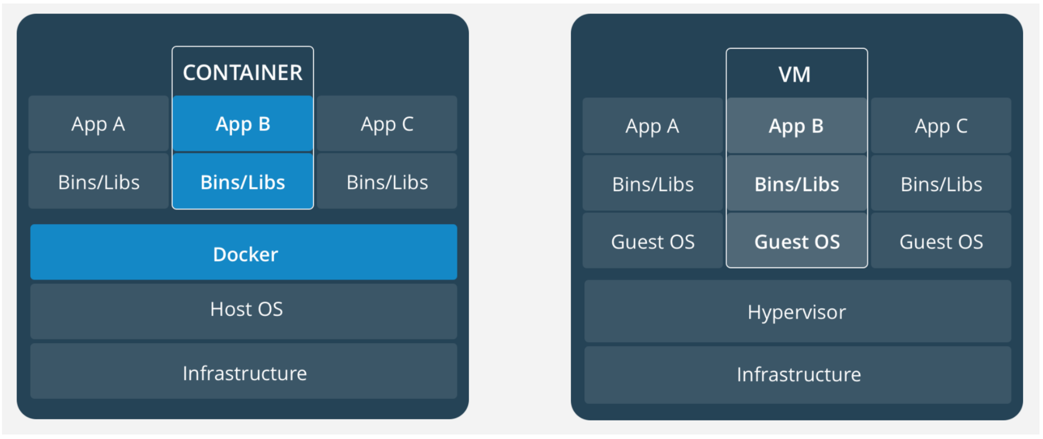


Figure 3.1: Docker Containers vs Virtual Machines

itself, resulting in higher efficiency and portability. Consider figure 3.1

Containers are an abstraction at the application layer which packages the code and dependencies together. Numerous Docker containers can be running on the same machine and share resources such as the operating system kernel with other Docker containers running on the same machine, even though each of these Docker containers are running in isolation in the user space. Docker containers also require less space than virtual machines, with a container image typically using less than one hundred megabytes of storage space in size, alongside their near immediate start time. On the other hand Virtual machines, are an abstraction of physical hardware allowing one server to appear or act as if it were many servers. The hyper-visor, also known as virtual machine monitor, produces any guest operating systems with a virtual operating platform and in turn manages the execution of all guest operating systems on the host machine. Each virtual machine will have a full copy of an operating system, one or many applications, any required dependencies, libraries and binaries, all of which would consume up to or more than one hundred gigabytes. Virtual machines can also be slow to launch.

Docker is a container platform to construct, secure and administer a large array of applications from the early stages of development to deployment and production whether they are local or “in the cloud”. Docker Community Edition provides tools to build applications to developers for free, whereas Docker Enterprise Edition provides a Containers-as-a-Service platform for IT companies with multi-architecture operations at scale.

3.1.3 Dockerizing Existing Applications

An application does not have to be built from the ground up with Docker, as existing applications can be containerized. With potential of reducing cost, start up time, and provide a layer of security, Docker can aid in time and resource management when building new applications or even modernizing legacy applications. Using the Modernize Traditional Applications [MTA] kit via Docker Enterprise Edition, developers can package existing applications into containers, which will enable the application to be portable. This alteration implements modern properties to legacy applications, without the need to change any of the source code. It will also reduce the overhead required to run the legacy application, increase operational efficiency, and consolidate infrastructure.

Because Docker packages applications and their dependencies together into an isolated container, this makes them portable to any infrastructure, making them suitable for Cloud migration, multi-cloud or hybrid cloud infrastructures. Thus eliminating the age-old “works on my machine” problem. The Docker Certified Infrastructure guarantees that the containerized applications work consistently. It provides an integrated environment for multiple distributions of enterprise Linux, as well as on Cloud providers such as Amazon Web Services, Azure and IBM. Docker Certified containers provide trusted independent software vendor products packaged and distributed via Docker containers. Docker Certified plug ins provide easy to download and install containers for an environment, as well as networking and volume plug ins.

3.1.4 Continuous Integration and Services

Docker also provides tools and strategies to Developer Operations (Dev Ops) teams. John Willis’ White paper on “Docker and the Three Ways of DevOps“, outline three principles known as “The Three Ways of DevOps”, and describes the purpose of these principles, how to apply these principles using Docker within organizations, and the results and benefits from doing so.

“The First Way: System Thinking”

This is referred to as the pipeline; the flow and direction from left to right, outlining the importance of understanding the system as a complete value stream. Managing this concept is often referred to as bottleneck reduction or global optimization, or “Lead Time” in the Lean project management methodology[11]. This is the equivalent of the time taken to get from rough diagrams to the delivery of a product to paying consumers, or in simpler terms, the time from starting a project, and the project becoming a product of value[11]. To ensure System Thinking is effective, Dev Ops teams need to able to apply the following:

- Escalate “velocity” by hastening each process component within the pipeline.
- Diminish “variation” by removing wasteful or time intensive sub processes within the pipeline.
- Ascension of processes by isolating functionality, therefore improving visualization and understanding of the overall flow.

Docker can aid the above in the following ways:

- **Velocity:**
  - In regards to *Development flow*, developers using Docker can create Docker environments locally on their machine to develop and test containerized applications. In contrast, the alternative of using virtual instances running as numerous hosts could consume a lot of time during start up time and convergence, depending on the overall complexity[11]. To summarize, Docker delivers less context switching time for testing and retesting, resulting in significantly higher velocity.
  - In regards to *Integration flow*, Docker can simplify continuous integration with use of Dockerized build slaves. A continuous integration system can be designed in such a way that it would allow multiple virtual instances to run in isolation, each as individual hosts. Environments can run Docker hosts within a Docker host, termed “Docker-in-Docker” for build environments. This provides elegant isolation of the build and breaks down the environments for the services in test, meaning that the original instances are never debased due to the embedded host having the ability to be recreated for each slave instance[11].
  - Finally with regards to *Deployment flow*, to achieve higher velocity in Continuous Delivery of software, there are numerous approaches that may be hit by using Docker. One challenge of production deployments is ensuring the seamless and quick changeover time from version to version. ”A Blue Green deploy is a technique where one node of a cluster is updated at a time (i.e., the green node) while the other nodes are still untouched

(the blue nodes).”[11] This approach requires a continuous process where each node is updated and tested, one at a time. The fundamental outcomes of this are:

1. The time required to update all nodes needs to be fast.
2. If a cluster requires roll back, this must also be done fast.

To summarize, Docker containers execute the roll back and roll forward processes more efficiently, and also are a lot cleaner due to container isolation during changeover.

- **Variation:** A fundamental pro of using Docker images throughout the software delivery pipeline is that the environment and application can both be packaged within the container image. With Docker, a developer can package the actual environment (e.g OS, dependencies, other requirements) within the same image. This convergence reduces the possible variation at all stages of the delivery pipeline; i.e. development, integration, and deployment[11]. If a developer were to test a collection of Docker images as a service on any machine, the services in question can be the exact same during integration testing and right up to deployment. ”A Dockerized pipeline approach delivers converged artifacts as binaries and therefore are immutable starting from the commit”[11].

- **Visualization** Containerized Microservices is a model which has been generating a buzz within the software development world lately. In an architecture of micro services, “services” are defined as bounded context. When these services are connected as Docker containers and used within the delivery pipeline, they become visible without delay on a domain. Services become more visible in the pipeline when bounded by their business context and then isolated as Docker containers. With more visibility an organization can isolate, and determine control rapidly; thus reducing overall Mean time to repair[11]. ”The “First Way”” and Docker can provide global optimization around software Velocity, Variation and Visualization. Dockerizing the development pipeline, organizations can reduce the cost and risk of software delivery while increasing the rate of change.”[11]

”The Second Way: Amplify Feedback Loops”

”The ”Second Way” is defined as amplifying and shortening feedback loops such that connections can be made fast and continuously.”[11] also known as the right to left flow.

- **Velocity** In the second way velocity means the same as Velocity from the first way. The flow in this case may not always be going in the same direction. Defect cause interruptions in the flow and changeover time. Dev Ops need to manage, feedback, changeover speed, the adaptiveness of the system in relation to defect detection, and reimplementation. Dockers streamlining of packaging allows organizations to utilize shorter changeover times because of defects, making it easier to halt processes if any defects are indeed identified.[11]

- **Visualization** An advantage of the immutable delivery process is most artifacts are conveyed throughout the pipeline as binaries[11]. This means that service delivery teams are able to generate meta-data from the source that is maintained, enabling visualization at any stage within the pipeline. With that meta-data the time required to resolve defects is reduced, therefore the overall Lead Time of the service being delivered is also reduced.

"The Third Way: Continuous Learning"

"In this “Third Way”, the symbol of a complete loop is used because it ties the first two ways together through a rigorous implementation of a learning process."[11] The word Kaizen is often seen as an approach of continuous improvement within an organization. This approach being fulfilled through a culture of continuous experimentation and learning in all parts of the organization[11]. The speed to market is essential in the deliverance of software and services. It is also the speed at which reaction, and reproduction to customers validation of the delivered service. The major principles of the Dev Ops mindset all point to outcomes like rapid innovation, increased quality and a feedback loop of continuous learning. "The Docker platform uniquely allows organizations to apply tools into their application environment to accelerate the rate of change, reduce friction and improve efficiencies."[11].

3.2 Jenkins

Jenkins[12] is an Open Source Continuous Integration and Continuous Delivery automation server that has a vibrant community and is widely used across both Small and Medium Enterprises and Large Enterprises alike. It has a rich plugin ecosystem and can be deployed both on premises and in the cloud. The recent addition of the Pipeline mode has gained massive support within the community and softened the attitude of hard core Continuous Delivery preachers towards the product.

3.3 Node.js

3.3.1 About Node.js

Node.js or more commonly referred to as Node is designed as an asynchronous event driven JavaScript runtime for scalable network applications built off Google Chrome V8 Engine. Google Chrome V8 Engine will translate the JavaScript, which Node Application Programming Interface's are written in, into machine readable code. Node provides the ability to handle many connections concurrently using callbacks, but if there is no work pending, it will sleep[13]. This is in opposition to the current common concurrency model that employs operating system threads, as thread-based networking can be relatively difficult to use and inefficient. Node users need not worry of processes dead-locking, since no locks exist. I/O processes never block, because very few functions in Node directly perform I/O, as a result of this scalable systems are feasible to build using Node. Node presents an event loop as a runtime construct rather than as a library. Node will enter this event loop after execution of an input script, and exit the event loop once all callbacks have been performed[13].

3.3.2 JavaScript to Machine Code via the Google Chrome V8 Engine

Overview

The V8 Engine is an open source high-performance JavaScript engine from Google, written in C++ that is used in Google Chrome, Node, and more. The V8 Engine can be embedded into any C++ application or run standalone[14]. *"V8 implements ECMAScript as specified in ECMA-262, 5th edition, and runs on Windows (XP or newer), Mac OS X (10.5 or newer), and Linux systems that use IA-32, x64, or ARM processors."*[14]

Technical details

The V8 Engine is capable of compiling and executing JavaScript source code, handling allocation of memory for objects, as well as garbage collection for redundant objects that are no longer needed. This garbage collector is key to the V8 Engines performance. Designed specifically for quick execution of large JavaScript applications, there are three primary areas tied to the V8 Engines performance:

1. Fast Property Access

- With JavaScript's dynamic nature, the properties of objects can be added or removed while in progress, which means that they are likely to change. To reduce the time complexity of accessing JavaScript properties, the V8 Engine dynamically creates hidden classes in the background. When a property is added or removed this hidden class will be updated, rather than using a dynamic lookup to resolve the property's location in memory[14]. *"There are two advantages to using hidden classes: property access does not require a dictionary lookup, and they enable V8 to use the classic class-based optimization, inline caching."*[14]

2. Dynamic Machine Code Generation

   - The V8 Engine will determine the state of an object's current hidden class at initial code execution for property access of any given object. Property access is optimized by prediction of future use of a class for any future objects accessed in the same code section and usage of information from the class to patch inline cached code used to the hidden class[14]. *"The combination of using hidden classes to access properties with inline caching and machine code generation optimises for cases where the same type of object is frequently created and accessed in a similar way. This greatly improves the speed at which most JavaScript code can be executed."*[14]

3. Efficient Garbage Collection

   - The V8 Engine will garbage collect objects that are no longer needed, reclaiming the memory used by said objects. The V8 Engine uses a *"stop-the-world, generational, accurate, garbage collector"*[14] to make sure that object allocation is fast, garbage collection is short, and no memory fragmenting occurs. The V8 Engine will stop the program execution during a garbage collection cycle, only partially processes the object heap during garbage collection cycles, and always knows where all objects and pointers are in memory.[14]

3.3.3 The Event Loop

Overview

The event loop allows Node to carry out non-blocking I/O operations, even though JavaScript is single-threaded via offloading of operations to the system kernel when possible[13]. As most modern system kernels are capable of multi-threading, this means that multiple operations can be executing in the background. On completion of an operation a callback is added to a poll queue that will be executed over the course of time[13].

Callbacks

Despite the fact that callbacks look similar to events, the key difference between the two are that callbacks are called once an asynchronous function returns, as opposed to an event which works using an observable. A Callback is similar to an asynchronous function, upon completion of a task, a callback function is called. Node uses and supports callbacks throughout all of its application programming interfaces, thus callbacks combined with asynchronous function calls maintains concurrency in Node.

Technical explanation

On start Node initializes the event loop, processes the provided input script which can make asynchronous application programming interface calls, schedule timers, or call the next tick method on the process, and begins the event loop which can be seen in 3.2.
For each stage of the event loop there is a first-in-first-out queue of callbacks waiting to be executed. Upon entry of a stage within' the event loop, operations which are specific to that stage are carried out, then callbacks are executed on the stage's queue until either it has been fulfilled entirely, or the callback limit has been reached, once that is done the event loop will proceed to the next stage, et cetera[13]. As these operations can schedule other operations or new events to be processed in the poll stage, which can carry out events as new events are polled. The outcome of this is that the poll phase may go on much longer than a timer's inception. The stages are as follows:

1. **Timers:**

    - Controlled by the poll stage the timer outlines an opening at which point a provided callback can be executed, as opposed to an exact time someone desires it to be done. Once a certain time frame has gone by, timer callbacks will run. Although they could be delayed by other callbacks or Operating system scheduling. *"Note: To prevent the poll phase from starving the event loop, libuv (the C library that implements the Node.js event loop and all of the*

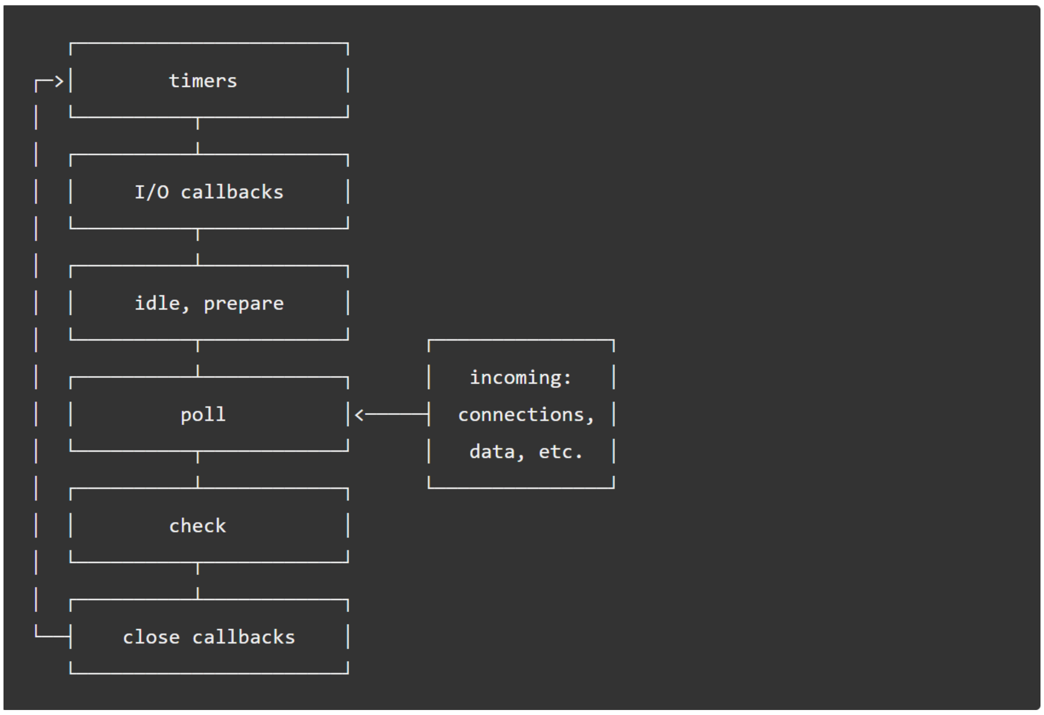


Figure 3.2: Node Event Loop stages

asynchronous behaviors of the platform) also has a hard maximum (system dependent) before it stops polling for more events.”[13]

2. **I/O Callbacks:**
    - In the I/O Callbacks stage, system operations callbacks like TCP errors for example, are executed, most callbacks are executed except close callbacks, those scheduled by timers, and set immediate calls .[13]
3. **Idle, prepare:**
    - This is only used internally.[13]
4. **Poll:**
    - The poll stage will execute scripts for timers where their opening has closed, and process events within the poll queue. If there are no timers scheduled upon entry of the poll stage either the event loop will go over its callbacks queue, executing each one after another in a synchronous manner until all callbacks have been

executed, or a limit is reached, if the poll queue is populated. Alternatively if the poll queue is not populated either the poll stage will be ended by the event loop and the check stage begin, or the event loop will wait until callbacks are added to the queue then immediately execute them.[13] When the poll queue has been emptied the event loop will then check for timers whose openings have ended. If this is the case for one or many, the event loop will then return to the timers stage and execute its callbacks.

5. **Check:**

   - In this stage any `setImmediate()` callbacks will be invoked. Also: *"This phase allows a person to execute callbacks immediately after the poll phase has completed."*[13].

6. **Close callbacks:**

   - Here `process.nextTick()` will be called, but what is that exactly? In Node, each iteration of the event loop is called a "tick". The `process.nextTick()` function is specific to the Node event loop and when it is used a callback function runs immediately after the events in an event queue have been processed by an iteration of the event loop. `process.nextTick()` is similar to JavaScript's `setTimeout()` function but operates much faster as it does not have to use JavaScript's runtime to schedule its queue of events.

3.4 TypeScript

3.4.1 What is TypeScript?

TypeScript[15] is a superset of JavaScript which compiles to plain JavaScript. It delivers everything and more that JavaScript has to offer and is a multi-paradigm language that brings flavours of Object-Oritentated Programming and the modern JavaScript Functional Programming techniques together. TypeScript provides a number of benefits when working in a JavaScript environment such as:

- Classes:
  ECMAScript 6 or ES6 introduced class definitions officially to the JavaScript ecosystem. TypeScript classes are similar however garnished with type checking bring a cleaner, more consistent and robust codebase.

- Interfaces:
  The ability to define contracts in one's code and conform to a specified interface provides structure and shape that values adhere to. While a Class may offer a factory or singleton[16] pattern providing initialisation to values TypeScript interfaces simply define a structural contract that defines what the properties of an object should have as a name and as a type.

- Inheritance:
  With the use of TypeScript Classes and Interfaces, Inheritance is also provided and gives developers a more comfortable object-orientated approach in designing software and modeling data. Using the extends keyword in object-orientated programming languages such as Java can often be taken for granted and is truly a powerful mechanism in the way in which software is designed.

- Static Typing:
  The use of Static Typing within TypeScript is a massive bonus for developers both coming from common JavaScript and indeed other statically typed languages such as Java. Static typing or strong typing means that the type of a variable is known at compile time. In contrast, weakly typed or dynamically typed languages associate the type with runtime and as a result types do not need to be specified.

And indeed perhaps one of the biggest benefits is enabling IDEs to provide a richer environment for spotting common errors as you write code, which can be one of the most frustrating aspects of writing common JavaScript. Anders Hejlsberg, lead architect of the C# programing language at Microsoft has worked a lot on the development of TypeScript and therefore both languages syntactically share many similarities[17].

3.4.2 TSLint Configuration

A linter or lint refers to a tool that is used to analyze source code and flag bugs, stylistic or programmatic errors. TSLint[18] is an extensible static analysis tool very similar to ESLint for JavaScript. TSLint checks TypeScript code for readability, maintainability and functionality errors and makes for a much cleaner and systematic codebase. This helps developers and co-workers to maintain both small and large projects, keeping consistency throughout.

TSLint employs a simple `tslint.json` file in order to work with the TSLint NPM package. The JSON file defines a basic set of rules to enforce upon TypeScript code and make sure it conforms to a standard. For example, in a JavaScript or TypeScript environment it is natural to conform the use of quotation marks to either double or more commonly single quotes. This becomes helpful as it reduces small annoying problems that can be encountered when working in teams. It is also very common to utilize git hooks[19] as a pre-commit rule to make sure all code is passing the linter without throwing any errors.

Example `tslint.json`

```
{
    "extends": "tslint:latest",
    "rules": {
        "quotemark": [true, "single", "avoid-escape"],
        "ordered-imports": [true, {
            "import-sources-order": "any",
            "named-imports-order": "case-insensitive"
        }]
    }
}
```

3.4.3 Integration with Node.js

The base scaffolding for a Node.js server application using TypeScript can be handled using a series of NPM scripts[20] defined in an application `package.json`. Microsoft offer a number of samples for getting started with using TypeScript and various libraries such as React and Express[15].

TypeScript source code is contained in a directory `src` and transpiled to JavaScript contained in a directory `lib`. Using the tools provided by NPM for managing scripts, the aim is to employ a number of scripts for various tasks such as:

- `"clean"` - Remove contents of the **lib** directory
- `"build"` - Transpile TypeScript to JavaScript
- `"lint"` - Check for readability, maintainability & functionality errors
- `"watch"` - Run the application in development mode
- `"postinstall"` - Rebuild upon install
- `"start"` - Run the application

- "test" - Run the application unit tests

Combining the usage of a number of the scripts used for various tasked documented above it can be made relatively straight forward to direct and manage a TypeScript development environment accompanied with unit tests, linting and debugging using Chrome dev tools[21].

Example Snippet `package.json`

```
"scripts": {
    "clean": "rimraf lib",
    "lint": "tslint --force --format verbose \"src/**/*.ts\" &&
    ↪  tslint --force --format verbose \"test/**/*.ts\"",
    "build": "npm run lint && echo Using TypeScript && tsc
    ↪  --version && tsc --pretty",
    "postinstall": "npm run build",
    "start": "npm run build && node lib/server.js | bunyan",
    "test": "npm run build && cross-env NODE_ENV=test mocha
    ↪  --nolazy --require source-map-support/register
    ↪  --require ts-node/register --recursive
    ↪  'test/**/*.spec.ts'",
    "watch": "concurrently -rk \"npm run build -- --watch\"
    ↪  \"nodemon --nolazy --inspect=0.0.0.0:9229 -d 20
    ↪  lib/server.js | bunyan\"",
    "watch:test": "npm run test -- --watch-extensions",
    "watch:test:debug": "npm run watch:test --
    ↪  --inspect-brk=0.0.0.0:9229"
}
```

3.4.4 Async/Await

The use of Async/Await is relatively new in the world of JavaScript and was introducted in the 2017 version of ECMAScript or ES8[22]. The use of async functions is said to be the answer to 'callback hell'. The previous options for writing asynchronous code in JavaScript were callbacks and promises. Async/Await is based on promises. Before taking a look at Async/Await one must first understand promises.

Promises are a placeholder or proxy that represent the value or result of an asynchronous operation. Promises are considered to be *thenable*, using the `.then()` function. The `then()` function takes two arguments which are callback functions to be run for either success or failure of the Promise and also returns a Promise. This is for the use of what is known as Promise chaining. A Promise has 3 states it can be in:

- **Pending** - where the asynchronous operation hasn't completed yet
- **Fulfilled** - where the asynchronous operation is resolved with a result
- **Rejected** - where the asynchronous operation is rejected with a reason for failure

Async/Await makes asynchronous code look and behave a little more like synchronous code[23]. For those familiar with languages such as C# the syntax will look very familiar.

Take for example a function `fetchData` that returns a promise and that promise resolves with some JSON object:

Using Promises

```
const request = () =>
  fetchData()
    .then(data => {
      console.log(data)
      return "done"
    })

request()
```

Using Async/Await

```
const request = async () => {
  console.log(await fetchData())
  return "done"
}

request()
```

3.5 Restify

Restify is Node.js framework for building RESTful web services[24]. Similar to Express.js[25], Restify allows for creating Node.js server applications. However, in contrast to Express, Restify comes as more lightweight and performance optimized framework, without aspects like templating or serving static resources such as HTML and CSS files.

Restify acts solely as a server application with any front end technology involved and comes equipped with a suite of middleware plugins[24] for processing incoming HTTP requests. Some of the world's largest technology companies use Restify as part of their applications technology stack such as NPM, Netflix and Pintrest[24].

3.6 Microsoft Bot Framework

The Microsoft Bot Framework[26] offers a service for creating conversationalist bots through the Bot Builder SDK for both the C# programming language through .NET and Node.js. The Bot Service allows developers to build, deploy and manage intelligent bots to interact naturally with users through websites, apps or third party clients such as Skype, Slack and Facebook Messenger[26]. The Bot Builder SDK for Node.js has been developed to work with popular server frameworks such as Express.js[25] and Restify[24] and makes it easy to create bots and test them using the Bot Framework Emulator[27].

The Bot Builder SDK for Node.js is written entirely in TypeScript and therefore provides developers with the beauty of types and intellisense in IDE's such as Visual Studio Code[28]. As is common with Microsoft Frameworks, API documentation is given in a set of examples and an SDK reference, however due to the source code being written in TypeScript it becomes relatively easy to read through the source and see exactly what is going on behind the scenes. Having well defined and easy to read source code is a powerful tool for developers and allows them to dig deep and get a true flavour for a given framework or library.

Conversations with bots are managed throughout a series of dialogs. Dialogs are what make up the essence of a conversational between human interactivity and indeed this is modeled throughout the Bot Builder SDK. Dialogs flow through in a waterfall sequence and decisions can be made by the bot given a set of results from a user or client.

Microsoft's Bot Framework also provides the ability of LUIS[29] integration. LUIS - Language Understanding Intelligent Service is Microsoft's machine learning based natural language processing service. LUIS can be integrated into a number of Microsoft Frameworks including the Bot Builder SDK. Building bots to interact with users is a powerful technique and is becoming popular throughout the industry. Providing intelligent or smart bots can provide a seamless user experience which may leave users questioning whether it may be a bot they are interacting with at all.

3.7 Databases

3.7.1 PostgreSQL

PostgreSQL is an object-relational database system. Comically PostgreSQL is often referred to as being around since the days of dinosaurs by the members of the software engineering community. And indeed somewhat rightfully so, as it has more than 20 years of active development going back as far as 1996. Its name Postgres even comes from the very earliest of databases Ingres[30] However, it has not always been the most popular and has often been neglected in favour of Oracle and MySQL. Oracle and MySQL relational databases are the top 2 most widely used databases[**dbrankings**] however in the last number of years PostgreSQL has made a resurgence. This may be somewhat to do with the proven architecture that has gained a very good reputation for reliability, data integrity and correctness[5]. PostgreSQL is completely open-source, running on all major operating systems, fully ACID compliant and includes MVCC[5].

3.7.2 CouchDB

CouchDB[6] is an open-source database developed by the Apache Software Foundation. It is a Document Store[31] which employs JSON (JavaScript Object Notation) as its mechanism for modeling and storing data. CouchDB operates similarly to MongoDB however MongoDB has more widespread use[32] with the primary differences between the two being how data is accessed. A CouchDB Document Store is accessed and communicated with via a HTTP REST api[33] which allows CRUD actions on documents and their internal structures.

Security Documents are provided for each database by CouchDB in the form of `db_security` which allows for the definition of specific users of a document. This becomes necessary when implementing a 'Couch Per User' database model, whereby each user of a system is given their own document store database.

While MongoDB provides faster queries and may be easier to grasp coming from an SQL background, CouchDB also has a lot to offer with features such as mobile support[34].

3.8 Data Objects / Structures

3.8.1 JSON

The JavaScript Object Notation (JSON) object is a text-based, versatile data interchange format, derived from ECMAScript Programming Language Standard. JSON describes formatting rules for portable representations of data structures[35]. It can neither be called nor constructed, and does not contain any major functionality[36]. JSON is capable of representing objects, arrays, and other primitive data types such as boolean, numbers, strings, or null[35]. A JSON object would take a form similar to the following example:

```
{
        "student" : {
                "name" : "Jake Johnson",
                "course" : {
                        "name" : "Computer Science",
                        "year" : 3,
                        "modules" : [
                                {"name" : "O.O.P."},
                                {"name" : "O.S."},
                                {"name" : "Web Apps"}
                        ]}
        }
}
```

Functions

The JSON object contains functions *parse* and *stringify*, used to parse or construct JSON text.

1. **parse** This function will parse a JSON formatted string and return its ECMAScript value. *"The JSON format is a restricted form of ECMAScript literal. JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript arrays. JSON strings, numbers, booleans, and null are realized as ECMAScript Strings, Numbers, Booleans, and null."[35]*

2. **stringify** This function will take a ECMAScript value and return a string representation of the object in JSON format. [35]

Grammer

JSON and ECMAScript source text are relatively similar. JSON is built up of a series of characters which adhere to the guidelines of SourceCharacter. There are two forms of JSON grammer:

1. **Lexical** *"JSON Lexical Grammar defines the tokens that make up a JSON text similar to the manner that the ECMAScript lexical grammar defines the tokens of an ECMAScript source text."[36]*

2. **Syntactic** *"JSON Syntactic Grammar defines a valid JSON text in terms of tokens defined by the JSON lexical grammar. The goal symbol of the grammar is JSONText."[36]*

3.8.2 XML

Extensible Markup Language (XML) is a derivative of Standard Generalized Markup Language (SGML)(ISO 8879). It partially describes behaviors of applications which process a class of data objects known as XML documents [37]. These documents are built up of units named entities, an entity can contain either parsed or unparsed data. *"XML provides a mechanism to impose constraints on the storage layout and logical structure."*[37]. XML was developed with the following goals intended:

"

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

"[37]

XML Example

```
<student>
        <name>Jake Jackson</name>
        <course>
                <name>Computer Science</name>
                <year>3</year>
                <modules>
                        <module><name>O.O.P.</name></module>
                        <module><name>O.S.</name></module>
                        <module><name>Web Apps</name></module>
                </modules>
        </course>
</student>
```

3.8.3 JSON and XML unified

As standard, the data object being used by HTTP requests throughout the project is the JSON object. Not all of the components would work with this as standard, therefore a wrapper of sorts needed to be built in order to interface with the Radicale XML document store. JSON data coming into the calendar service api would validate that certain required fields were populated with conforming data, such as a date which would need to conform to **ISO8601** standards in order for Radicale to accept incoming requests for creation of calendars and events. The elements would be validated from JSON data and then extracted building an event object, this event object would then be parsed into the XML version of said event. Once the event had its minimum required data it would then be ready to push to Radicale, who in turn would run its own validation to ensure that the data received did indeed contain all necessary data required to build an event.ics file, and store it within its respective calendar. Once this process was successful Radicale would return a status code informing that a calendar or event had been created or updated, alongside a location to said calendar.

3.8.4 Radicale

What is Radicale?

Radicale is a free and open-source solution for storing and manipulating calendar and contact information, with capabilities of storing numerous address books and calendars. It is written using Python, and will run on most UNIX-like platforms such as: (Linux, *BSD, macOS) and Windows. Manipulation is available for both calendar and contact information from local or remote access, which is limited through authentication policies. Radicale aims to be a lightweight solution, which is easily installed, used, and configured, thus as a result of this, it is pre-configured for immediate usability, and does not require many software dependencies[7]. Radicale is a server, it does not have a client, nor will a client be created by the development team behind the project as they claim: *"No interfaces will be created to work with the server, as it is a really (really really) much more difficult task."*[7]. Radicale uses the open standards available CalDAV and CardDAV as they are widely used by clients and servers. Some of the technical choices made before the project was built were as follows:

- **Oriented to Calendar and Contact User Agent** - Using defined protocols a calendar or contact server may interact with a calendar or contact client. As CalDAV and CardDAV have many use cases and features, these were the chosen protocols.

- **Simplicity** - Radicale is designed in such a way that it be easy to install, configure and then use.
- **Lazy** - Where the CalDAV RFC describes what can or cannot be done. Radicale will assume that the everything is fine and that protocol violations are non-existent. The consequence of this is the server may reply with meaningful answers, meaningless answers, or in some cases not answer at all.

Radicale itself appears to have originally been a simple school assignment or project which took on a life of its own afterwards.[7] Nonetheless, its free, it works for its intended purposes, and what is required for a portion of this project.

Chapter 4

System Design

A Microservice Architecture is employed to create the application as a series of distinct interactive components. A Mono Repository Model[38] has been adopted as to increase developer productivity and mitigate the complexities of code sharing across various different repositories.

4.1 What are Microservices?

The term ”Microservice” does not hold a true or set definition it is merely a term introduced by a number of software architects at a workshop near Venice in May, 2011 to provide context to a number of reoccurring design principles and patterns that seemed to be emerging more frequently. Just under a year later, James Lewis presented at 33rd Degree, conference for Java Masters in Krakow where he spoke about building systems composed of systems and focusing on the Unix philosophy of minimalist and modular. [39] This gave rise to the hot new term ”Microservices” and began to get more and more attention.

Martin Fowler describes the Microservice Architectural style as a approach to developing a single application as a suite of independently deployable services each running in its own process [40]. Each service should be independent in its our right and provide functionality based around a single responsibility. Services should be loosely coupled, provide high cohesion and adhere to the single responsibility principle. All of a sudden it can be seen that many of the traits mentioned are key concepts in Object Orientated Programming which are also common in the design principles of Service Orientated Architecture. Stubbings and Polovina [41] explore and contrast the how Object Orientated expertise can be leveraged in the design of Service Orientated Architecture (SOA). Employing fundamental aspects of the

Object Orientated Programming paradigm such as abstraction and encapsulation can provide a significantly more proficient approach to designing Microservice based applications.

4.2 Are Microservices just SOA?

This immediately sparks the debate - Is Microservice Architecture really just SOA? Upon the rise of the term ”Microservices” into the community, many SOA architects and engineers were coming forward, saying ”We’ve been doing this for years!”. And as Martin Fowler mentions in his keynote[42] - ”Service Orientated Architecture is a very broad term”. It encompasses a vast landscape of design concepts, principles, patterns and implementation standards. And it is often considered controlled by vendors who release commercial solutions Fowler goes on to mention that in essence Microservices can be really be considered a subset of SOA.

Indeed, this is a popular and well defined statement and is backed up by Adrian Cockcroft, the man responsible for pioneering the architectural style at Netflix as they moved towards cloud based and Microservice orientated design. He describes the architecture as ”fine-grained SOA”, in his keynote at Dockercon in 2014 [43].

Netflix are widely considered to be part of the pioneers of the Microservice Architectural style and have helped pave the way for it to flourish, introducing such libraries and tools as Hystrix[44] and Chaos Monkey[45] for testing and strengthening systems.

4.3 Monolith vs MicroServices

Josh Evans of Netflix provides the analogy of the human body to describe the architectural style in his keynote [46]. He describes Microservices as the organs which make up the human body and work together as one organism. Perhaps the best way to put Microservices into context is by comparing the differences in contrast to a traditional monolithic architecture. Martin Fowler uses a similar example in his Resource Guide to Microservices where he outlines a potential structure of a traditional enterprise application. [40] Take for example a simple application that exists in 3 tiers.

4.3.1 Presentation Layer

A Presentation Layer sits at the top of the application. This is the entry point of an application from a customer or user perspective, more commonly referred to as the front end. For example, this layer may consist of a web application composed of HTML, CSS and Javascript presenting a user interface to the client.

4.3.2 Logic Layer

A Logic Layer lies behind the user interface and provides functionality to carry out business operations. This could be a wide variety of functionality and responsibilities could be anything from user accounts to instant messaging to calendar events. For this purpose, lets take a Java server application as the example. Requests may be issues over HTTP via REST[47] or RPC and handled accordingly by a number of object orientated classes to perform the requested operation.

4.3.3 Data Layer

The Data Layer lives at base of the application stack and is responsible for persistence of the raw application data. All of the data needed and in use by the application exists here. A data store such as a relational database could be employed to persist and manage data in this layer. The server application may employ a framework such as Hibernate[48] JPA for object relational mapping in order to perform actions on data.

The monolith in this example is the server application. The services it provides run on a single process and are not independently deployable. A monolith is a software application whose modules cannot be executed independently [49]. In the case of the example defined above we may need to make improvements, changes or simple bug fixes to any one of the services we described. Changes made to the source code of the application imply - rebuild and redeploy. But what if the changes are small and confined to one service? There is an overhead associated with this as it becomes expensive to build a new release for production and this may only be affordable every couple of weeks. Continuous integration and continuous deployment are considered hidden dividends of Microservices [50]. Although not exclusive to the world of Microservices they may in theory be more easily orchestrated as each service has a single responsibility, however require careful orchestration. This allows developers to focus on developing and can speed up productivity across teams.

4.4 Microservices Characteristics

To migrate the example above to a Microservice based architecture the functionality described for handling user accounts, instant messaging and calendar events would be refactored into individual services that are independently deployable and each run in an isolated process. This provides a more loosely coupled and highly cohesive design, adhering more so to the single responsibility principle mentioned previously. However, that being said this does not mean it is an easy task to move from one architecture to another. Particularly in the case of Microservices there a wide range of aspects to be taken into consideration. Knoche writes of using modernization paths to predict performance impact on systems migrating from monolithic software applications towards microservices [51]. Microservices may not hold a definition set in stone however, it is clear that there is a number of definitive common characteristics that can be used to identify with architectural style.

1. Independent Deployment

Independent deployment is a key characteristic and indeed perhaps the most important principle of Microservice architecture. The idea of having the ability to deploy services as independent applications implies a number of benefits.

Language Neutrality

A collection of Microservices operating together for the purpose of a single application do not have to be confined to one programming language. Sam Newmann uses the term - Technology Heterogeneity [52]. A service can use the best tools for the job and there may be various services written in completely different languages using different technologies.

Specialised Teams

Teams working on their respective services can become experts in the area and obtain a great understanding of how to carry out their jobs to the best of their ability. Having teams focus on their respective services may speed up productivity [52]. Why be a jack of all trades and master of none?

Appropriate Scaling

Independent Deployment lends itself to scaling in the right places. In terms of a traditional monolithic piece of software, the monolith may be replicated numerous times to achieve desired scaling. This infers the question - is every aspect of the system being used enough to justify increased scaling of the entire system. Independent deployment offers scaling of individual services where appropriate, the term horizontal scaling is used [46].

2. Bounded Context

The term ”Bounded Context” is used when describing the characteristics of Microservice Architecture. This pattern is key in Domain Driven Design [53]. Bounded context in Microservices refers to organisation and creation of services based upon their business logic. A service should have a single responsibility and that responsibility should be entirely encapsulated within that service. This also lends itself to the idea of specialised teams as previously mentioned. Fowler speaks in his keynote about how Amazon divided themselves into teams responsible for a section of business logic all the way through to the end-user experience [42].

3. Communication

In a cloud or distributed environment service interactions with one another can be carried out via IPC (inter-process communication) or protocols such as HTTP are commonly applied. Many Microservices communicating together in a chain through HTTP may suggest a certain amount of latency being involved, depending on what operations are being carried out. To combat situations like this, Google introduced gRPC[54]. gRPC is as open source RPC framework that employs Protocol buffers[55] as a method of serializing structured data. It is highly efficient and is designed to target speed and performance. Binary data is sent through the wire at high speeds, alongside rapid serialisation/deserialization [56].

In order to provide communication via services or processes it is standard to use light-weight mechanisms [49]. This also lends itself to the idea of smart endpoints and dumb pipes[40]. The services themselves become the endpoints where business logic lives and pipes refer to the mechanisms used for them to communicate which are kept simple.

4. Decentralisation

Decentralisation of services infers a substantial amount of additional complexities within an application ecosystem. How does a client know what service it needs to contact in a cloud or distributed network? API Gateways[57] and Service Discovery[58] are two critical concepts in Microservice

Architecture. An API Gateway uses a Facade pattern[16] from object orientated design to encapsulate how requests are made to a number a services. This minimises code complexity on the client side and eliminates direct calls to services made by the client. The API Gateway is used in conjunction with a Service Discovery pattern. This may be either client side or server side. Service Discovery employs the use of a service registry, a database of network locations of available service instances[58].

4.5 Containerisation with Docker

Docker [3] is a containisation platform that utilises the host operating system using lightweight images at the application layer to build and ship applications effectively and quickly. Containers can be considered as instances of images running in isolated processes. They are often compared to virtual machines however they are more lightweight and can share the host OS kernal with many other containers. Virtual machines often take a couple of minutes to start whereas containers launch in a number of seconds. Container images are substantially smaller than virtual machines and are typically tens of MBs in size. In the last number of years Docker has become an industry leader in the world of Microservice Architecture. Sinnott and Korzhirbayev carried out a comparison of container-based technologies for the cloud[59] where they analyse Docker and evaluate pros and cons of container-based technologies based on in depth research of a number of performance metrics such as CPU, Disk I/O and Network I/O performance.

My Consultancy Services aims to provide containerisation of all services throughout the application. For example, housed in the mono repository of the project is a number of sub directories each containing a distinct containerised service. Such as:

- Accounts Service
- Calendar Service
- Accounts Database
- CouchDB Per User
- Calendar Server

Each service is distinct in its own right and available throughout the Docker network at its own service URL.

4.5.1 Dockerfiles and Docker Compose

The system utilizes Dockerfiles and Docker Compose for container orchestration.

Containers are built with Docker using Dockerfiles[60]. A Dockerfile is a declarative file in which instructions for images containing applications or services are defined. For example, a basic Dockerfile may include:

- A base image
- Installation of Dependencies
- Adding of source code
- Exposing ports to the host OS
- Commands for starting an application

Example Dockerfile

```
# Base Image
FROM node:alpine

# Specify a working directory
WORKDIR /app

# Add package.json and install dependencies
ADD package.json /app/package.json
RUN npm install

# Add application source code
ADD . /app

# Expose port 3000 on host machine
EXPOSE 3000

# Start the server
CMD ["node", "server.js"]
```

More detailed and complex configurations can also be defined such as environment variables and volumes. Volumes provide a way in which containers can persist data. Smart, Nguyen and Jaramillo provide excellent analysis of how Docker technology can be leveraged in Microservice Architecture [61].

Docker compose can be used for the orchestration of multiple containers specifying their build paths to each service's respective Dockerfile. Similar to Dockerfiles a number of configuration settings can be defined here. John Zaccone[62], a Docker Captain, provides an excellent example using Docker Compose in his keynote at Dockercon 2017, the source code of his demo application can be found on his Github [63].

4.5.2 Containerised Development Environments

Through the use of volumes, a `docker-compose.debug.yml` file can be defined which allows development in a containerised environment. It is essential to have a good understanding of how an application behaves in a containerised environment for production purposes and therefore taking the extra time to setup a development environment which closely mimics this is truly a beneficial endeavour.

Example `docker-compose.debug.yml`

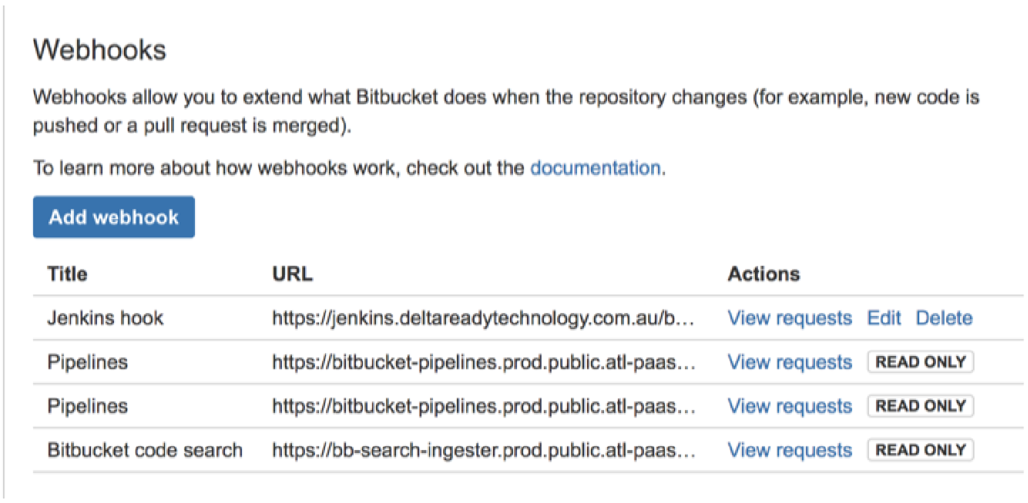
```
version: "3"

services:
    account-service-api:
        image: "account-service-api"
        build: .
        user: "node"
        working_dir: /home/node/app
        volumes:
            - .:/home/node/app
        ports:
            - "9000:9000"
            - "9229:9229" # Expose ports for debugging via Chrome Inspect
        depends_on:
            - "account-database"
            - "calendar-service-api"
            - "couchdb"
        command: "npm run watch"

    account-database:
        image: "account-database"
        build:
            context: ../account-database
        volumes:
            - db-data:/var/lib/postgresql/data
        ports:
            - "5432:5432"
        restart: always

    calendar-service-api:
        image: "calendar-service-api"
        build:
            context: "../calendar-service-api"
        ports:
            - "9001:9001"
        depends_on:
            - "radicale-caldav-server"
        command: "npm run start"

    couchdb:
        image: "couchdb"
        build:
            context: ../couchdb-per-user
        volumes:
            - couchdb-data:/opt/couchdb/data
        ports:
            - "5984:5984"
        restart: always

    couchdb-configurator:
        image: "couch-configurator"
        build:
            context: ../couchdb-per-user/configurator
        depends_on:
            - couchdb
        command: ["/wait-for-couchdb.sh", "couchdb", "5984", "/configure-couchdb

    radicale-caldav-server:
        image: "radicale-caldav-server"
        build:
            context: "../radicale-caldav-server"
        ports:
            - "5232:5232"
        volumes:
            - cal-data:/home/radicale/data
            - config:/home/radicale/config
        restart: always

volumes:
    db-data:
    couchdb-data:
    cal-data:
    config:
```

4.6 Jenkins Pipeline

A Jenkins Pipeline[4] has been setup for build stages and deployment. Jenkins can be deployed on bare metal, VPS (Virtual Private Server) or as a Docker container and can be configured as a standalone server or in a cluster with Jenkins Swarm[64]. The modus operandi employed by Delta Ready Technology is that of a Jenkins Swarm running in a Docker Swarm[65]. Essentially, Jenkins is deployed as a global service on a three node Docker Swarm which provides a degree of fault tolerance and redundancy. Additionally, three Jenkins Agents which are used to perform the build are run globally across the cluster. A custom Docker Registry is used to store the docker images built in the CI (Continuous Integration)[66] process and resides at `https://registry.deltareadytechnology.com.au` and is protected by User Access Control.

A build is triggered when a commit is made to the either of the Git branches `master` or `develop` in BitBucket. The branches that trigger builds are configurable and currently a regular expression that tests for develop and master achieves the desired effect. Rather than polling the BitBucket repository a Webhook (see Fig 4.1) is installed in BitBucket Cloud when configuring the Bitbucket Branch Source Plugin.

Once the build pipeline has completed successfully the My Consultancy Service suite of Microservices are deployed based on a Docker Stack[67] configuration (see Fig 4.2).



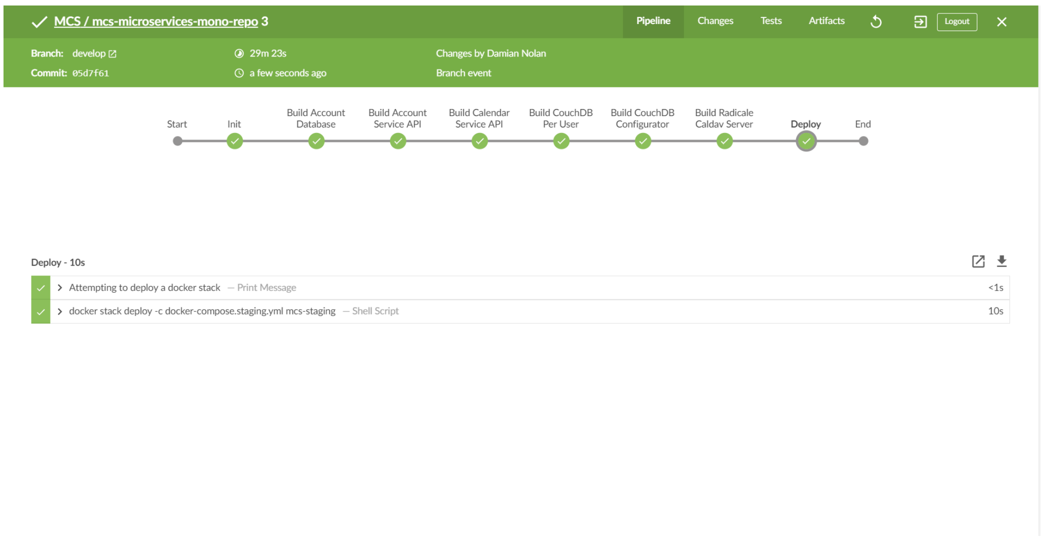
Webhooks

Webhooks allow you to extend what Bitbucket does when the repository changes (for example, new code is pushed or a pull request is merged).

To learn more about how webhooks work, check out the documentation.

Title	URL	Actions
Jenkins hook	https://jenkins.deltareadytechnology.com.au/b…	View requests Edit Delete
Pipelines	https://bitbucket-pipelines.prod.public.atl-paas…	View requests READ ONLY
Pipelines	https://bitbucket-pipelines.prod.public.atl-paas…	View requests READ ONLY
Bitbucket code search	https://bb-search-ingester.prod.public.atl-paas…	View requests READ ONLY

Figure 4.1: Webhooks



MCS / mcs-microservices-mono-repo 3
Pipeline
Changes
Tests
Artifacts
Logout

Branch: develop
Commit: 05d7f61
29m 23s
a few seconds ago
Changes by Damian Nolan
Branch event

Start
Init
Build Account Database
Build Account Service API
Build Calendar Service API
Build CouchDB Per User
Build CouchDB Configurator
Build Radicale Caldav Server
Deploy
End

Deploy - 10s

```
Attempting to deploy a docker stack — Print Message                              <1s
docker stack deploy -c docker-compose.staging.yml mcs-staging — Shell Script     10s
```

Figure 4.2: Jenkins Deployment

Chapter 5

System Evaluation

5.1 Validation

5.1.1 Overview

In relation to data, the process of validation used throughout the project ensured that incoming or outgoing data has gone through checks to ensure that the data has quality, also that it is correctly formatted and meaningful in relation to the requested api[68]. Its intent is to provide defined guarantees for acceptability, consistency, and accuracy in the data which is persisting within the api, application, and databases. Rules are defined for any incoming data, which the data must conform to (e.g. an email must be in the following format user@domain.com). There are numerous types of validation which can be carried out based on its intent, scope, or complexity:

- **Type validation** This is a relatively simple form of validation which will check, and verify that the value that is input to a key conforms to the expected value, whether it is a specific value or primitive type (e.g. for key name, value should be a string, for key age, value should be an integer, et cetera)[68].

- **Range and constraint validation** This type of validation will check consistency of input within a minimum or maximum range, or evaluating sequences against a regular expression (e.g. for key phone number, value should contain x amount of only digits between 0 and 9)[68].

- **Code and cross-reference validation** This type of validation tests that user-supplied data conforms to one or more rules or requirements within the scope of the intent (e.g. for key password, value should contain an uppercase character, a lowercase character, a digit, a symbol, and must be at least 8 characters long). When validating against constraints cross-referencing of a directory or table would validate that x cannot be the same as y (e.g. for key new user email, value cannot already exist within the table of user emails)[68].

To enable this behavior the joi library is used, this allows the creation of a blueprint or schema named a ”validator” for objects with defined limitations of what can be input into the values for each key value pair. This ”validator” would then be validated using the restify-api-validation library. Upon validation the library would either allow the function to carry on or throw a validation error which notifies of specific non-conformances within the data being transferred to the api[69].

5.2 Testing

5.2.1 Overview

In relation to testing, tests are carried out to assess whether a system or its components satisfy predefined requirements or not[70]. Throughout the development process of the project, components were built with a Behavior Driven Development methodology, as tasks were nearing a branch merge state each route or function would then have a test written to verify that it meets expectations of its intended functionality. Using the Mocha testing framework and Chai assertion library, an asynchronous suite of unit tests are written for each service[71][72]. A health check is the initial test to ensure that the service itself is functioning, further unit tests are written for each route in different situations to ensure that each component of the service is working and assert the correctness of each components response[71][72]. With any function or component which would create data structures to be stored in a database or server the Nock library is used, this works by intercepting htttp requests and mocks the expected response in the given situation, thus no redundant testing data is being stored upon each run of a test suite[73]. Finally the proxyquire library allowed for seizing control of functions in the effort to negate dependencies during testing of each routes source code[74].

System Under Test

The term system under test refers to testing a system for correctness in its operation. Using docker compose test files, entire services would be tested all at once to ensure the system itself was working as expected. This means that every aspect of the environment is tested from the container itself being built, the environment being created and configured with dependencies and requirements, the overall health check being carried out, all components unit tested in numerous different situations as listed below.

5.3 Robustness

With the combination of validation, system, and unit testing this resulted in a more sturdy system overall. The tests carried out would cover numerous different situations and answer questions such as:

- Is there a valid authentication key present in the request?
- Is it possible to hit a protected route without a valid authentication key or no authentication key whatsoever?
- What would happen if the function is sent data which it does not have defined rules to handle?
- What would happen if the function is sent incomplete data?
- What would happen if the function is sent data which is in the correct format for validation and the authentication key is also valid?
- But most importantly, does this function do exactly what we had originally intended it to do every time it is called no matter what the situation may be?

5.4 Expectations vs Reality

5.4.1 Expectations

In the introduction the following initial objectives were outlined:

- A RESTful web service for account management including all user account based business logic and serving as the primary base of the application.
- A PostgreSQL[5] relational database for user account management.
- A CouchDB[6] per user model to facilitate user content storage using a NoSQL database.
- A Radicale Caldav Server[7] for management of calendar services.

- A RESTful web service for calendar management that handles CRUD functionality of user calendar events and act as a JSON wrapper for the Radicale Caldav Server.
- A client application designed using Stencil Ionic - Progressive Web Application using Web Components.

5.4.2 Reality

In reality the following objectives were either fully or close to completion:

- A RESTful account service which enables the management of accounts.
- A relation database to store all user accounts.
- A NoSql database per user account for account content storage.
- A server successfully managing calendar data.
- A RESTful calendar service for management of user calendars and events, with parsing of JSON to XML for compatibility with Radicale Caldav server.

5.4.3 Shortcomings

Of the initial six objectives set out, the primary shortcoming was the client application via Ionic Progressive Web Application (PWA) using Web Components. The technology itself is due to be introduced in Ionic 4, which at time of writing is still currently in BETA. The PWA Toolkit when released would deem the construction of PWAs relatively easy according to the team behind Ionic[75]. To ensure that the system itself would not require copious hours of bug fixing and updating once coming out of BETA to full release, the decision to wait for the Ionic 4.x stable release was made.

Chapter 6

Conclusion

6.1 Microservice Architecture

While Microservice Architecture is a fantastic solution for many reoccurring problems in an application design, it should be considered by no means a silver a bullet [52]. Microservice Architecture certainly adds a number of complexities and is often considered very risky to begin with as adopting this style can be very difficult to get right. Every application has different requirements and perhaps a simpler approach should be taken in the beginning. It is not uncommon to start with a monolith style architecture and later refactor to a Microservice based design. CTO of NPM Laurie Voss explains in his keynote how NPM began as a monolith and later refactored to a Microservice Architecture and is now the world's largest software registry[76]. So while Microservices and indeed Docker may not be the cure to address every aspect of software design and architecture, they both definitely have a lot promising and helpful tools to offer.

6.2 Docker as a development tool

Docker as a development tool is relatively easy to pick up and implement, due to the fact that it is well supported and documented by its community. However, it comes with a steep learning curve as you traverse deeper into its more advanced features and functionality. The isolated environments that Docker containers provide, enables work across different machines regardless of the underlying infrastructure, which in the case of this project, provided an easy work flow during cross examination, merging, and service integration. Going forward Docker will continue to make this project more and more versatile, secure, quick, and easy to maintain because of the Docker ethos. Long after the project has been deployed and it is in use by the product owner, integration engineers and developer operations teams should find processes like: onboarding new services, or analyzing and repairing problematic services easy to identify, then address, seamlessly and fast. Finally with the exponential growth in adoption rate of Docker over the last couple of years, virtual machines may very well be a thing of the past in years to come.

6.3 Asynchronous Typescript code via Node.js runtime environment

The popularity of Node.js is not surprising whatsoever, throughout the lifetime of the project the task of building services using this tool was a relatively smooth learning process. With excellent documentation, support, and a wide variety of libraries to use for any task imaginable, developers using the framework should find the server-side logic somewhat easy and quick to pick up and get running. When paired with TypeScript the framework becomes even better, because TypeScript provides a rich type safe code base when using JavaScript for server-side logic development. The ability to create a type using two existing libraries means that what once were limitations are now reduced to a minor task of creating a type. The ability to use the Object Spread Operator also allows for some pretty nifty tricks such as tagging the payload onto the back of the options (headers) before they are sent out in a HTTP request. The act of writing asynchronous functions also enhances the code a great deal, by simply replacing a chain of callbacks with one keyword `await`, this reduces the overall source lines of code (SLOC), reduces complexity, and for the most part, makes the code more readable.

6.4 Authentication, Testing, and Validation

As discussed in previous sections, the addition of authentication, testing, and validation add a professional level of robustness to the overall project, without adding to much extra complexity. With authentication adding a layer of security to each service, validation adding a layer of reliability to each component of each service, and testing adding the overall assurance that each service, each component, no matter what the situation may be, will return what is expected of it. These three aspects add quality to the project as a whole.

6.5 Future Development

As highlighted in the system evaluation, building a responsive, client application via Ionic 4 Progressive Web Application ToolKit using Web Components will be a task when the full stable release becomes available.

6.6 Thoughts and Observations

Working as a professional development team using an Agile Kanban workflow along with feature branches and pull requests has provided real world experience for moving forward into the software engineering industry. The following conclusions can be deduced from working on My Consultancy Services:

- Unit Testing proves the robustness of code and provides a validation mechanism for proving its legitimacy and reasoning.
- Unit Testing can easily highlight and identify bugs.
- Writing high quality code does not require a user interface or client application for the means of demonstration or testing.
- Setbacks are common in any development environment and as a result push back features that were initially scheduled for earlier dates.
- Producing an industry standard piece of software over the course of two semesters proves difficult when working on several other modules.

Chapter 7

Appendix

- **My Consultancy Services - Mono Respository:**
  `https://bitbucket.org/deltaready-mcs/mcs-microservices-mono-repo`
- **My Consultancy Services - Minor Dissertation:**
  `https://github.com/Verdagio/Final-Year-Minor-Dissertation`

Bibliography

[1] URL: https://deltareadytechnology.com.au/.

[2] Agile Modeling. *User Stories: An Agile Introduction*. URL: http://www.agilemodeling.com/artifacts/userStory.htm.

[3] *Docker*. URL: https://www.docker.com/what-docker.

[4] Jenkins. *Jenkins Pipeline*. URL: https://jenkins.io/doc/book/pipeline/.

[5] PostgreSQL. *About PostgreSQL*. URL: https://www.postgresql.org/about/.

[6] Apache Software Foundation. *CouchDB*. URL: http://couchdb.apache.org/.

[7] Kozea. *Radicale documentation*. URL: http://radicale.org/about/.

[8] TutorialsPoint. *Kanban*. 2016. URL: https://www.tutorialspoint.com/kanban/kanban_tutorial.pdf.

[9] Ben Straub Scott Chacon. *Pro Git*. Second Edition. Apress, 2014. URL: https://git-scm.com/book/en/v2.

[10] Vincent Driessen. *A successful Git branching model*. Blog post, date accessed Jan/2018. Jan. 2010. URL: http://nvie.com/posts/a-successful-git-branching-model/.

[11] Docker John Willis. *DOCKER AND THE THREE WAYS OF DEVOPS*. 2015. URL: https://www.docker.com/sites/default/files/WP_Docker%20and%20the%203%20ways%20devops.pdf.

[12] *Jenkins*. URL: https://jenkins.io/.

[13] NodeJS. *Documentation*. Dates accessed, Oct-2017 to Apr-2018. URL: https://nodejs.org/en/about/.

[14] Google. *Chrome V8 Engine Documentation*. Date accessed, 08-Apr-2018. URL: https://github.com/v8/v8/wiki.

[15] Microsoft. *TypeScript*. URL: https://typescriptlang.org/.

[16] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, p. 175. ISBN: 0-201-63361-2.

[17] Anders Hejlsberg. *Introducing TypeScript*. URL: https://channel9.msdn.com/posts/Anders-Hejlsberg-Introducing-TypeScript.

[18] Palantir Technologies. *TSLint*. URL: https://palantir.github.io/tslint/.

[19] Git. *8.3 Customizing Git - Git Hooks*. URL: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks.

[20] NPM. *NPM Scripts*. URL: https://docs.npmjs.com/misc/scripts.

[21] Node.js. *"Debugging Guide"*. URL: https://nodejs.org/en/docs/guides/debugging-getting-started/.

[22] ECMAScript International. *ECMAScript 2017 Language Specification*. URL: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[23] Mostafa Gaafar. *Hackernooon - 6 Reasons Why JavaScript's Async/Await Blows Promises Away*. URL: https://hackernoon.com/6-reasons-why-javascripts-async-await-blows-promises-away-tutorial-c7ec10518dd9.

[24] Restify. *Restify*. URL: http://restify.com/.

[25] Node.js Foundation. *Express.js*. URL: https://expressjs.com/.

[26] Microsoft. *Bot Framework*. URL: https://dev.botframework.com/.

[27] Microsoft. *Bot Framework Emulator*. URL: https://github.com/Microsoft/BotFramework-Emulator.

[28] Microsoft. *Visual Studio Code*. URL: https://code.visualstudio.com/.

[29] Microsoft. *LUIS*. URL: https://www.luis.ai/.

[30] Craig Kerstiens. *PostgresGuide*. URL: http://www.postgresguide.com/.

[31] DB Engines. *Document Stores*. URL: https://db-engines.com/en/article/Document+Stores.

[32] DB-Engines. *DB-Engines Ranking*. URL: https://db-engines.com/en/ranking.

[33] Apache Software Foundation. *CouchDB - HTTP API Reference*. URL: http://docs.couchdb.org/en/2.0.0/http-api.html.

[34] Matan Sarig. *CouchDB vs MongoDB*. URL: `https://blog.panoply.io/couchdb-vs-mongodb`.

[35] D. Crockford. *RFC 4627 JSON)*. July 2006. URL: `http://www.ietf.org/rfc/rfc4627.txt`.

[36] ECMA International. *Standard ECMA-262 5.1 Edition, ECMAScript® Language Specification*. June 2011. URL: `https://www.ecma-international.org/ecma-262/5.1/#sec-15.12`.

[37] W3C. *Extensible Markup Language*. Nov. 2008. URL: `https://www.w3.org/TR/2008/REC-xml-20081126/`.

[38] Atlassian Stefan Saasen. *Monorepos in Git*. URL: `https://developer.atlassian.com/blog/2015/10/monorepos-in-git/`.

[39] James Lewis. "Microservices, Java - the Unix Way". In: Mar. 2012. URL: `https://vimeo.com/74452550`.

[40] Martin Fowler James Lewis. *Microservices Resource Guide*. 2014. URL: `https://martinfowler.com/microservices/`.

[41] Gary Stubbings and Simon Polovina. "Levering object-oriented knowledge for service-oriented proficiency." In: *Computing* 95.9 (2013), pp. 817–835. ISSN: 0010485X. URL: `http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=90053029&site=eds-live`.

[42] Martin Fowler. "GOTO Conference Berlin 2014, Microservices". In: 2014. URL: `https://www.youtube.com/watch?v=wgdBVIX9ifA`.

[43] Adrian Cockcroft. "State of the Art in Microservices". In: 2014. URL: `https://youtu.be/nMTaS07i3jk`.

[44] Netflix. *Hystrix*. URL: `https://github.com/Netflix/Hystrix`.

[45] Netflix. *Chaos Monkey*. URL: `https://github.com/Netflix/chaosmonkey`.

[46] Josh Evans. "Mastering Chaos - A Netflix Guide to Microservices". In: QCon San Francisco, 2016. URL: `https://www.infoq.com/presentations/netflix-chaos-microservices`.

[47] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". AAI9980887. PhD thesis. 2000. ISBN: 0-599-87118-0.

[48] RedHat JBoss Middleware. *Hibernate ORM*. URL: `http://hibernate.org/orm/`.

[49] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: (June 2016).

[50] TOM KILLALEA. “The Hidden Dividends of Microservices.” In: *Communications of the ACM* 59.8 (2016), pp. 42–45. ISSN: 00010782. URL: `http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=117173592&site=eds-live`.

[51] Holger Knoche. “Sustaining Runtime Performance While Incrementally Modernizing Transactional Monolithic Software Towards Microservices”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE ’16. Delft, The Netherlands: ACM, 2016, pp. 121–124. ISBN: 978-1-4503-4080-9. DOI: `10.1145/2851553.2892039`. URL: `http://doi.acm.org/10.1145/2851553.2892039`.

[52] Sam Newmann. *Building Microservices*. O’Reilly, 2014. ISBN: 9781491950357.

[53] Martin Fowler. *Bounded Context*. URL: `https://martinfowler.com/bliki/BoundedContext.html`.

[54] Google. *gRPC*. URL: `https://grpc.io/`.

[55] Google. *Protocol Buffers*. URL: `https://developers.google.com/protocol-buffers/`.

[56] Joey Clover. *Microservices are hard - an invaluable guide to microservices*. URL: `https://hackernoon.com/microservices-are-hard-an-invaluable-guide-to-microservices-2d06bd7bcf5d`.

[57] Nginx. *Building Microservices: Using an API Gateway*. URL: `https://www.nginx.com/blog/building-microservices-using-an-api-gateway/`.

[58] Nginx. *Service Discovery in Microservices Architecture*. URL: `https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/`.

[59] Zhanibek Kozhirbayev and Richard O. Sinnott. “A performance comparison of container-based technologies for the Cloud”. In: *Future Generation Computer Systems* 68.Supplement C (2017), pp. 175–182. ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2016.08.025`. URL: `http://www.sciencedirect.com/science/article/pii/S0167739X16303041`.

[60] Docker. *Dockerfile reference documentation*. URL: `https://docs.docker.com/engine/reference/builder/`.

[61] Robert Smart David Jaramillo Duy V Nguyen. “Leveraging Microservices Architecture by using Docker technology”. In: (Mar. 2016).

[62] *John Zaconne, Docker Captain*. URL: `https://www.docker.com/captains/john-zaccone`.

[63] John Zaccone. *Office Space Dockercon Demo.* URL: https://github.com/jzaccone/office-space-dockercon2017.

[64] Jenkins. *Jenkins Swarm.* URL: https://wiki.jenkins.io/display/JENKINS/Swarm+Plugin.

[65] Docker. *Docker Swarm.* URL: https://docs.docker.com/engine/swarm/.

[66] Martin Fowler. *Continuous Integration.* URL: https://www.martinfowler.com/articles/continuousIntegration.html.

[67] Docker. *Docker Stack.* URL: https://docs.docker.com/engine/reference/commandline/stack_deploy/.

[68] Wikipedia. *Data validation.* URL: https://en.wikipedia.org/wiki/Data_validation.

[69] hapijs. *Documentation.* URL: https://github.com/hapijs/joi.

[70] Wikipedia. *Software testing.* URL: https://en.wikipedia.org/wiki/Software_testing.

[71] Mocha. *Mocha Documentation.* URL: https://mochajs.org/.

[72] Chai. *Chai Documentation.* URL: http://www.chaijs.com/.

[73] node-nock. *Documentation.* URL: https://github.com/node-nock/nock.

[74] thlorenz. *Documentation.* URL: https://github.com/thlorenz/proxyquire.

[75] Ionic Justin Willis. *Announcing the Ionic PWA Toolkit Beta.* Jan. 2018. URL: https://blog.ionicframework.com/announcing-the-ionic-pwa-toolkit-beta/.

[76] Laurie Voss. "How npm split a monolith and lived to tell the tale". In: 2016. URL: https://www.oreilly.com/ideas/how-npm-split-a-monolith-and-lived-to-tell-the-tale.