uFlexi Technical Architecture

September 2017

Authors

Polymorphism: Tom Baldwin, Richard Popple

Ultraflexi: Wingham Rowan

Polymorphism Limited

Electric Works, Sheffield S1 2BJ

+44 (0) 114 286 6356

www.polymorphism.co.uk

polymorphism
enterprise software development

CONTENTS

1 uFlexi purpose, overview and direction

uFlexi is a horizontal (all types of work) marketplace for hourly labour based on a *Stored Availability* mechanism. *Stored Availability* was developed for the travel sector, allowing suppliers to trade individual assets at a precise location for precise blocks of time. Commonalities with hourly labour run deep.

For an overview of the need, context, uniqueness and non-technology challenges of our markets: BeyondJobs.com.

1.1 Overview of Stored Availability

As the travel industry moved online, airlines, hotel chains, hire car companies and other providers progressively collaborated to enable a Global Distribution System (GDS). This underlying database stores (a) details of each seat on a future journey, each hotel room and each vehicle (b) current availability of each of those assets (c) how each is to be priced to maximize the seller's return.

When we book through Expedia, TripAdvisor, Booking.com or other consumer sites, they are drawing data from the GDS then amending its availability records as we purchase. Consumer sites add their own mark-up. But the GDS is so ubiquitous, low overhead and dependable it makes sense to build services on top of it rather than countless start-ups trying to build a market from scratch[1].

1.1.1 Overview of Stored Availability

The GDS is a high-tech seller co-operative in which individual providers set their terms. Because it handles the entire process of searching, matching, pricing and contracting online it captures granular details of each transaction, and each unfulfilled need. Actionable data on utilization, pricing opportunities, buyers to target and trends flows constantly to the supply side.

1.1.2 Benefits for users

A Booking.com user seeking a weekend in Hawaii is not offered vague adverts for hotels that *could* have availability: the best online could offer pre-GDS. Drawing on GDS data, Booking.com lays the market bare; showing every genuinely available option, fully priced, and ready to book. A regular user can click instantly on a branded option knowing she will get what she's paid for. A price-sensitive traveller wanting to understand the market can make a more leisurely and informed selection.

Consumer travel sites have limited power over the supply-side. They can't individually dictate prices because sellers are exposed to such a wide market. Nor do they own sellers' data or trading records. If one of these intermediaries fails it may barely ripple for sellers.

[1] The GDS is an interoperable group of systems, primarily Sabre, Galileo, Travelport, Worldspan. Description of the GDS has been simplified for this analogy. The model is twenty years old and resource allocation in the travel sector is increasingly nuanced. For a fuller introduction to the GDS see: https://en.wikipedia.org/wiki/Global_Distribution_System

But the GDS has also been good for buyers. It instantly exposes a deep, fully priced, market for any travel destination and dates. Booking is immediate and assured, a hotel's record of reliably honouring bookings in such a deep database is crucial to future earnings. The GDS has driven up convenience, quality, range, and responsiveness as informed suppliers constantly align with buyers' needs.

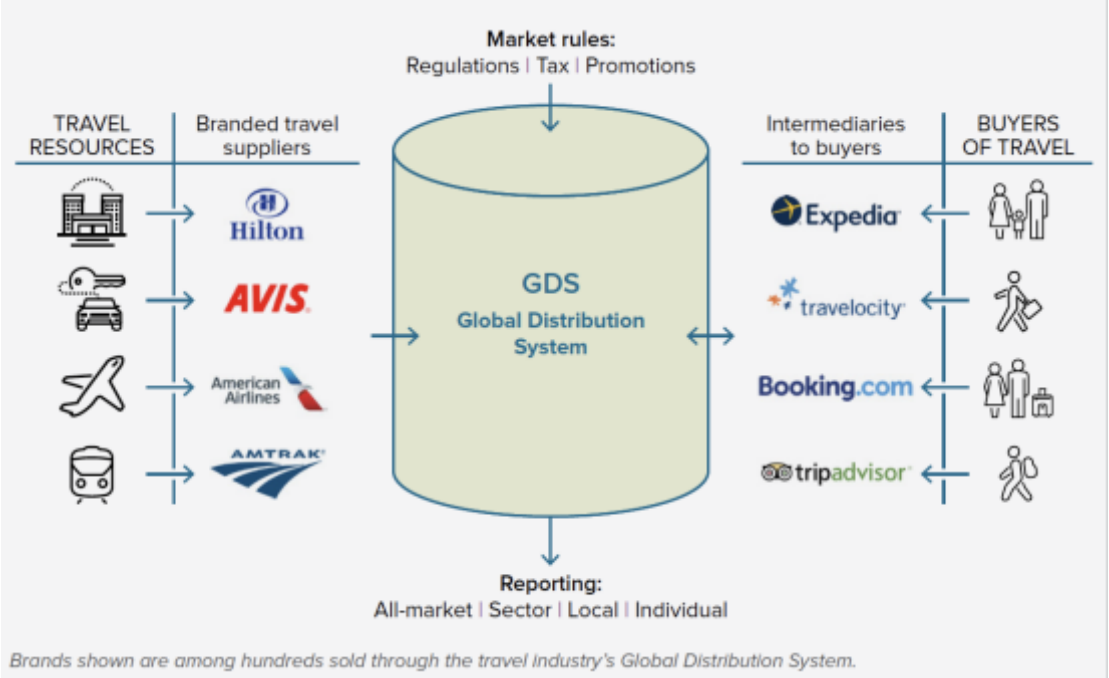

Figure 1

The GDS does not necessarily offer lowest prices. They are typically found by phoning round hotels where a manager can have discretionary pricing power, or even by booking a first night legitimately then suggesting a longer stay off-the-books at reception[2]. Nor has it created a race-to-the-floor by improving market visibility. Tell Expedia you seek an overnight in Orlando then order returns by price. There will be numerous youth hostels vying for business. But they have not killed off mid-market or top-end hotels. The GDS retains its hold on users by being so convenient and risk-free. It then fosters precision pricing, we pay premium for a Marriott or even a Travelodge because we trust the system when it says the facilities will be better.

1.1.3 Stored Availability in labour markets

Hourly labour markets are exponentially more complex than travel bookings. Technology aside, a healthy market must incentivize engagement by a spectrum of stakeholders each able to add value at appropriate points.

uFlexi is our name for the software. An installation for any given city/region is called a CEDAH (Central Database of Available Hours). Like the GDS, it's white-label with users' experience

[2] See for example: http://loyaltylobby.com/2013/05/02/how-fat-are-expedias-hotel-margins/

governed by the intermediary through which they access it. It's useful to understand the role played by different bodies in a CEDAH.

- **Market Commissioner**: This is the body that initiates a market. Because the ideal CEDAH is universal, accountable, ultra-low overhead and neutral the commissioning body is likely to be a local government entity such as (in the US) a public workforce board. The MC sets rules for the market such as who can be an intermediary. In technology terms, the MC probably won't use the system beyond having back office frontline access to reporting data.
- **Market Operator**: This is the entity operating as back office for the city/region. It could be Beyond Jobs or someone else operating our open sourced software reporting to the MC.
- **Agencies**: Employers and Workers enter the market through the website of an intermediary *Agency*. Staff approve bona fides and confirm to the system what certification *checks* each worker has. This determines the roles they can perform.
    - Intermediaries in a CEDAH may be commercial recruiters, employment charities, public bodies or a mix. Some will provide workers *Sellers*; payrolling, insuring and vetting them in return for a mark-up built into each hour sold. Their brand acts as quality assurance, as does a known hotel chain in the GDS. Other intermediaries can sell hours from the database to buyers, adding their own tools, value proposition and charges. Some organizations may want to be intermediaries on both sides of the market.
    - Intermediaries are encouraged to partner: Agency A allowing their workers to be booked by Agency B's employers *buyers* with an agreed margin split. This grows the market.
    - To keep our markets manageable on one underlying system: agencies are clustered in *pools*. They can only partner within their pool and back office frontline staff can be limited to oversight of one pool. The likely application of this structure is to give each city a pool.
- **Sellers**: Each worker manages a list of roles (types of work) for which they qualify. They can turn individual roles on and off, charge higher rates for in-demand or high skilled work and accept or reject employer-specific roles and any attached payrate. They also set availability: the hours they wish to sell.
- **Buyers**: A CEDAH stores, potentially, millions of specific hours of availability. It knows the roles against which each hour can be offered, rules of the person offering, how to price it and the relevant legal controls. Like Expedia, it can instantly marshal and price - then reliably enforce - eligible workers for a given requirement.

1.1.4 Overview of a city CEDAH

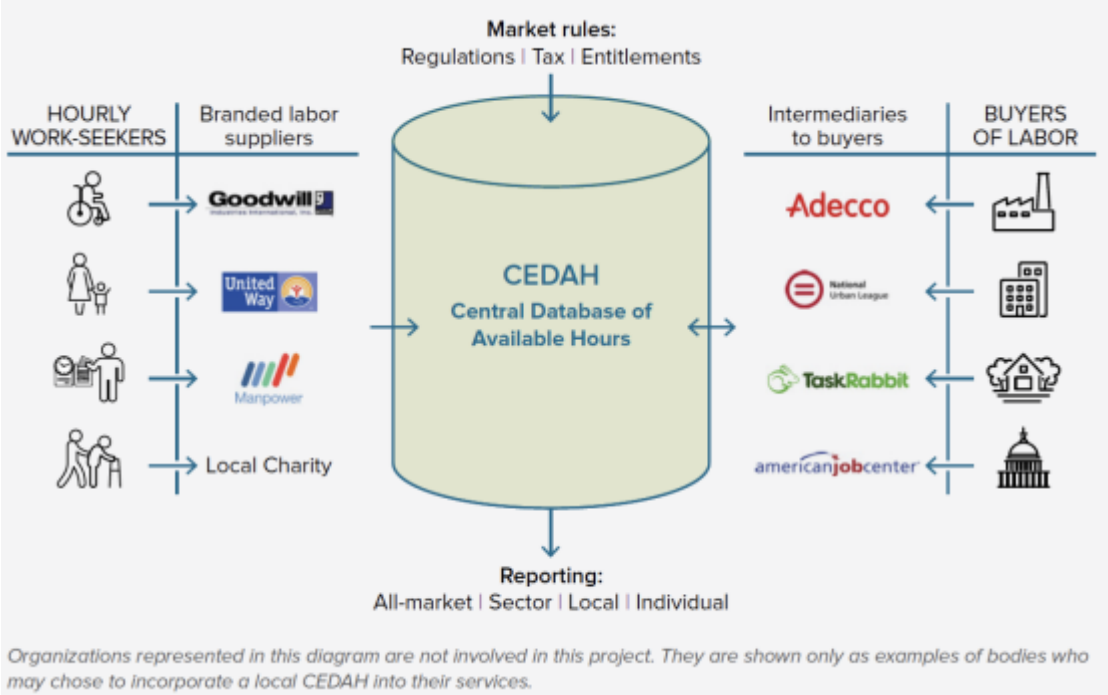

Figure 2

1.1.5 An example transaction in a CEDAH

A manager at ACME market research needs 100 interviews with office workers. She determines a metro station as a likely source and, noting today's fine weather, determines this afternoon's rush hour as optimal timing. She turns to a recruitment agency app on her phone.

The app backends to a regional CEDAH. As soon as she specifies "ACME: Consumer interviewers" and the location, her phone shows a grid of how many of those approved workers are available and willing to work at that metro station each hour for the next 10 weeks. She selects today: 5:00-7:00PM when 9 are available. She immediately sees details of those 9.

Each of these people has been inducted and approved by her company and agreed to its hourly fixed rate. To keep widening the market, the system also shows further available workers vetted and willing to do market research interviews but who have not yet been offered, or not yet accepted, a specific role with her organization. They are priced dynamically based on personal rules around travel distance from home to this station, period of notice, length of booking and so on. She selects 8 workers from her pool and four who are will be new to her company.

It takes a few swipes to select and book the 12 for rendezvous outside the station. Confirmation, post-transaction changes, timesheeting, and payroll/invoicing data is managed in the CEDAH. It ensures confirmation with plenty of time to rearrange workers if needed.

This takes perhaps 30 seconds. She is aware workers from this app seem motivated and reliable, just as Travelocity users take for granted their bookings get honoured, probably without understanding the way a deep underlying market drives hotelier responsiveness and reliability.

She does not need to know that each interviewer is working hours of their choosing, on their own terms, across possibly dozens of roles in many sectors and this afternoon's assignment is helping each towards diverse personal goals with reliably completed bookings earning inductions, higher pay and tracks to a job if that is the person's aim.

Nor need she grasp the way philanthropies and public bodies might be very efficiently intervening in the market; training disadvantaged people for high utilization roles in their travel area, funding subsidized childcare workers to enable pivotal work activity, creating pools of inducted Peer Navigators who support unconfident individuals into their first few bookings.

She may however notice numbers of workers in her company's approved interviewers pool depleting. That tells her the fixed payrate is too low. Every worker has a personal portfolio of roles each of which can be switched to inactive at any time. As workers build a track record they may opt out of display for her needs, or at this company's fixed rate. Bosses will have to either push out a higher payrate or induct new workers. If induction is their route, the CEDAH will help replenish their pool; identifying upcoming individuals with the right experience and proximity then booking them for a group training session to learn how this company conducts its clipboard encounters.

1.1.6 Direction of travel

uFlexi was funded by UK government bodies. It leads the world in this sort of market. Need for fair, horizontal, platforms for fragmented low-skill work has grown dramatically in recent years. So have the possibilities of those platforms.

The GDS has reshaped travel in countless subtle ways. Data allows planes with the most transfer passengers on any given day to park at adjoining fingers, slashing journey times, for example. We are at the beginning of this kind of sophistication for hourly labour.

Like the GDS, our benefits are reliant on scale across the entire spectrum of work. But our data capture and analytics tools can uniquely inform individual progression to higher paying checks/roles, targeted interventions for those struggling in the workforce, business opportunities, local policymaking and accountable local control.

2 Source Code

The uFlexi system is based around 3 main entities: *Buyer*, *Seller*, and *Agency*. Buyers, and sellers both belong to a single agency. Buyers purchase sellers via their agency, sellers sell their time via their agency. Agencies can partner with other agencies, allowing booking of sellers belong to other agencies.

2.1 Software Design

The system design broadly follows a *domain-driven design* model. Domain model classes are mapped on to relational database tables via the Java Persistence API (JPA). Domain model objects are persisted, and retrieved via *repositories*. Repositories operate on portions of the object graph, based on the current unit of work.

Operations are performed via *services*, which mark the transactional boundary for our ACID transactions. We treat repositories as specialised services, and in the case of CRUD operations access them directly from *actions.*

Actions are user interactions with the web application. They are responsible for calling services, and populating the views.

2.2 Legacy Components

The source code contains some components that follow legacy conventions from the original development of the system. Progress has been made to update, and refactor these artefacts, although some remain.

2.2.1 Beans and Data Access Objects

We still have several classes aren't mapped via JPA, e.g. They are flat mappings of database tables, mapped using Spring RowMappers, and JDBC DAOs. In more complex cases, where units of work span multiple database tables, 'rich' versions of the classes are used, with corresponding DAOs, and custom SQL queries.

These classes are being replaced by JPA entities, and repositories.

2.2.2 Model classes

Model classes are more straightforward; they're simply a legacy naming convention for services. These classes can simply be renamed, and moved to an appropriate package.

2.3 Domain Model

The domain model classes reflect the underlying business concepts.

2.4 Users and Roles

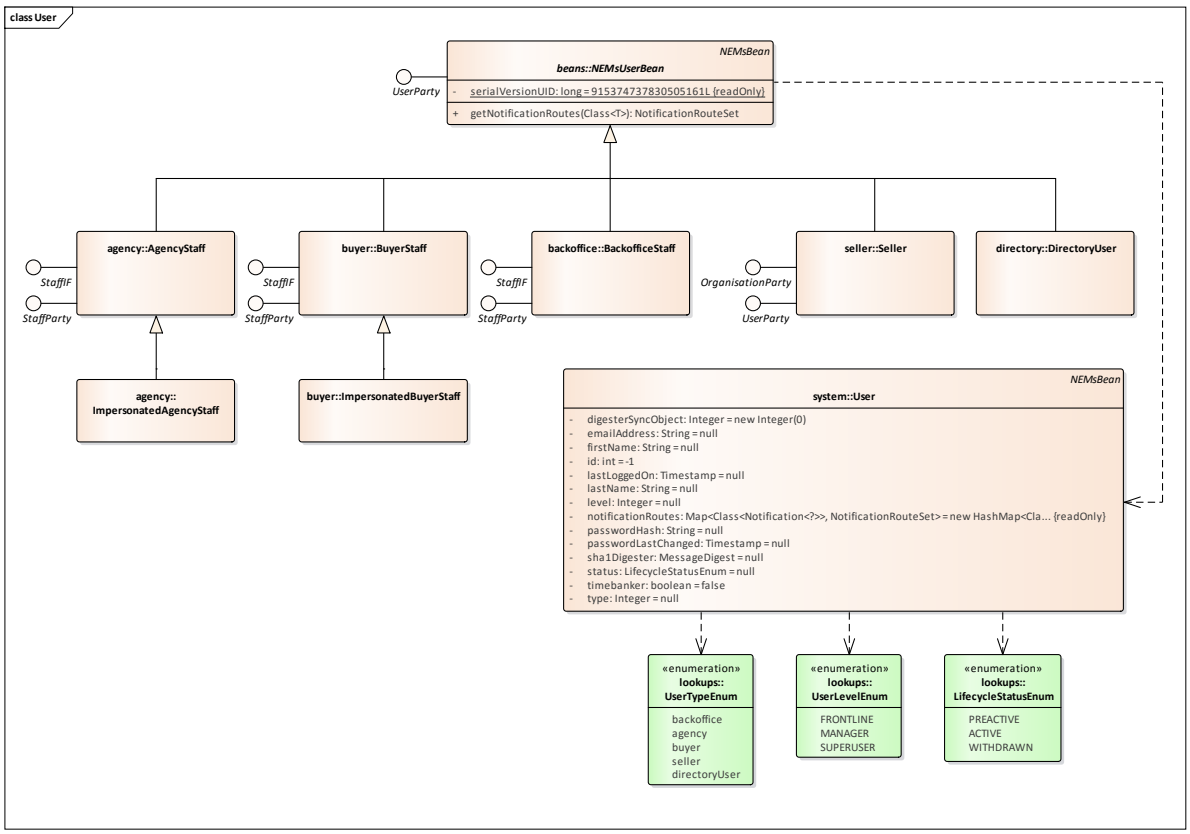

Figure 3 - User classes

2.4.1 User

The `User` class represents the user's security principle. It contains their authentication details.

Users remove through lifecycle states from `PREACTIVE` (registered, but not activated), `ACTIVE`, `WITHDRAWN`.

Once approved, `Users` remain in the system to maintain referential integrity.

2.4.2 NEMsUserBean

`NEMsUserBean` subclasses represent a specific User-role in the system. It is quite possible for a single `User` to hold multiple roles. In the case of impersonated users, we are able to assign a new `NEMsUserBean` to an administrative user, allowing them to act on behalf of that user, without logging in as them.

Current user-roles in the system, represented by `NEMsUserBean` are:

- `AgencyStaff`
- `BuyerStaff`
- `BackofficeStaff`
- `Seller`
- `DirectoryUser`

2.4.3 User Levels

- User Levels are generally inherited, i.e. a Superuser inherits Frontline and Manager Roles.
- Frontline Backoffice Users can impersonate Agency, Buyer, and Seller Users, within their pool.
- Agency Staff can impersonate Buyer and Seller Users within their agency.

Anonymous	Recover Password				
Authenticated					
	Preactive	**Frontline**	**Manager**	**Superuser**	**Withdrawn**
Backoffice	n/a	Act as Agency	Add Backoffice User	Withdraw Backoffice User, Modify System Settings	n/a
Agency	n/a	Act as Buyer/Seller[3]	Manage Agency Staff	Modify Agency Settings, Payroll Export	n/a
Buyer	View basic details	Book Seller	Manage Buyer Staff	Manage Cost Centres	
Seller					View Timesheets, View Jobs
Directory			n/a	n/a	n/a
Timebank		n/a	n/a	n/a	

2.5 Parties

2.5.1 Party

The `Party` abstraction, shown in Figure 4, represents both organisations, and single-user parties that can participate in a `Purchase`. For example, a `Buyer` can be both an organisation, with multiple

[3] At Home Location only when Workforce

users (staff members, including managers, and admin users), or a Buyer can simply be individual user.

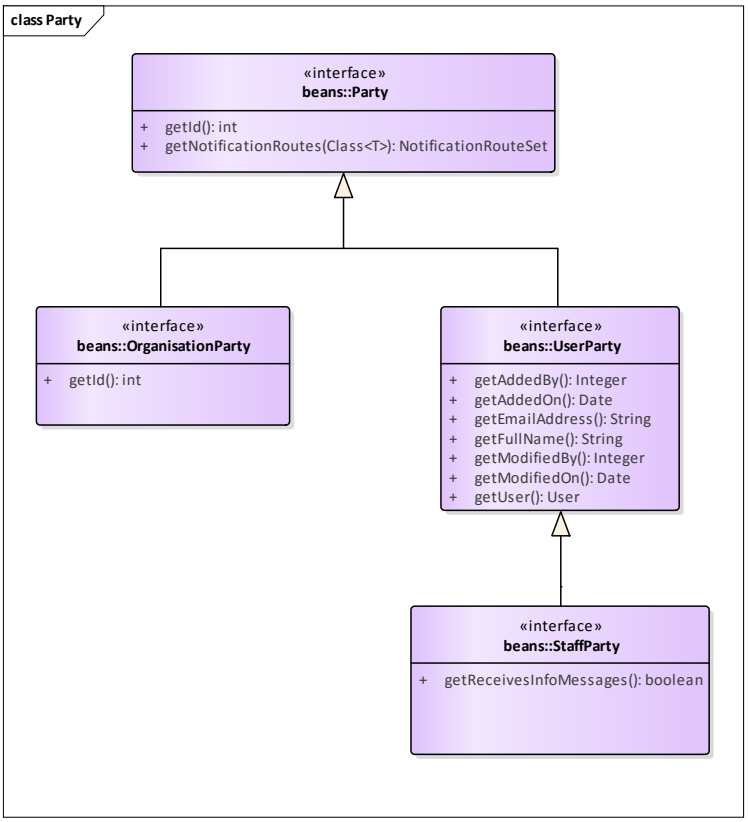

Figure 4 – Party Interfaces

Party types are:

- Seller
- Agency
- Buyer
- Backoffice
- Directory User

2.5.2 Seller

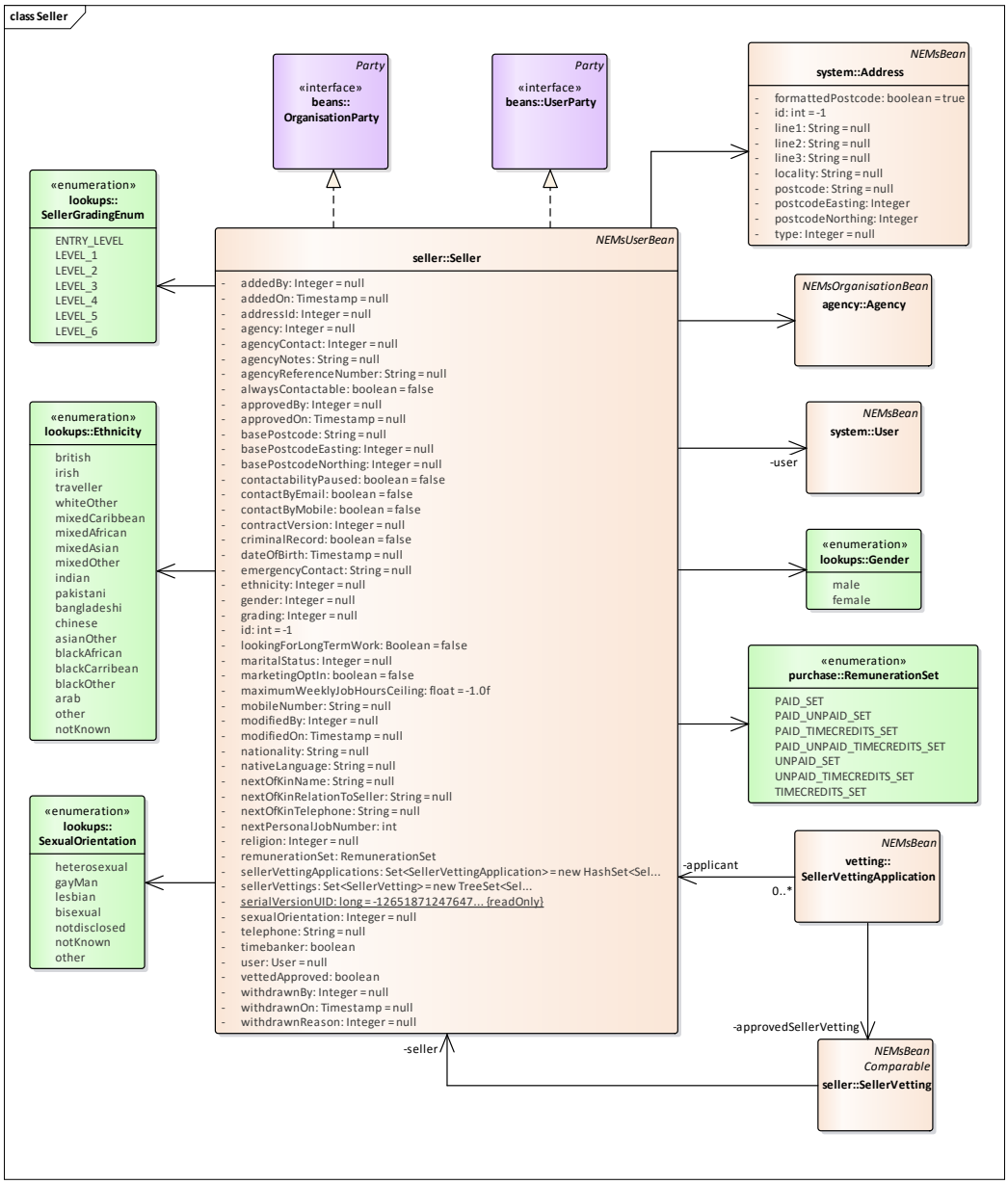

Figure 5 – Seller Class

A `Seller` belongs to an `Agency`, and may have a specific home `AgencyLocation`. They have a location, derived from their address. A seller may have multiple `Vettings`, and can apply for new ones. Has related `SellerAvailabity`, although can be set to be always available. Sellers have a `RemunerationSet` which determines the types of bookings they can be offered.

2.5.3 Buyer

A Buyer belongs to a single `Agency`. It may be an individual, or an organisation. They have at least one `BuyerLocation`, which is used to determine the distance to a seller, when making a booking. They have a `RemunerationSet` containing the remuneration type of bookings they are able to make. Like agencies, buyers can have their own domain name, and branding.

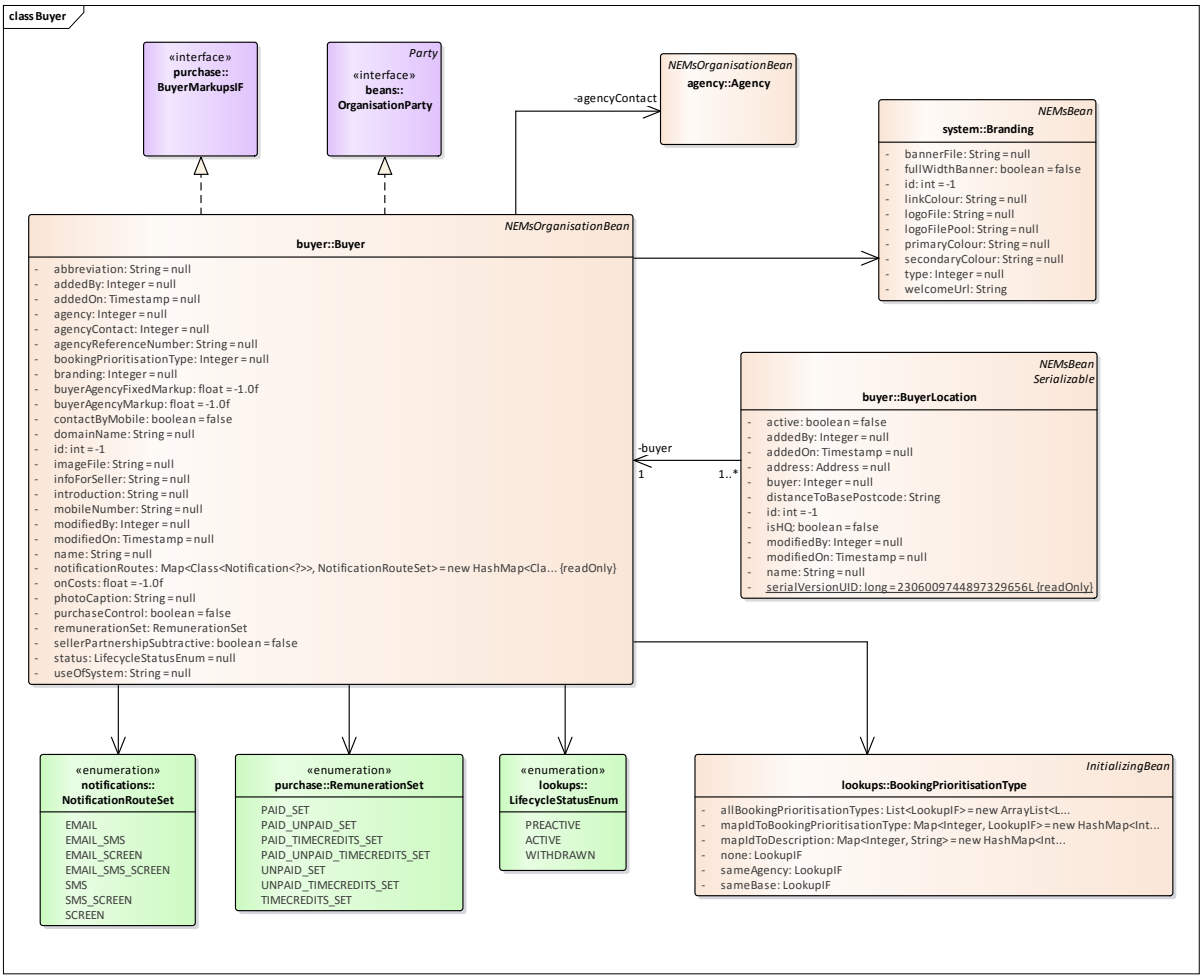

Figure 6 – Buyer class

2.5.4 Agency

An Agency has its own domain name, and branding. They belong to a pool, and can partner with other agencies within that pool.

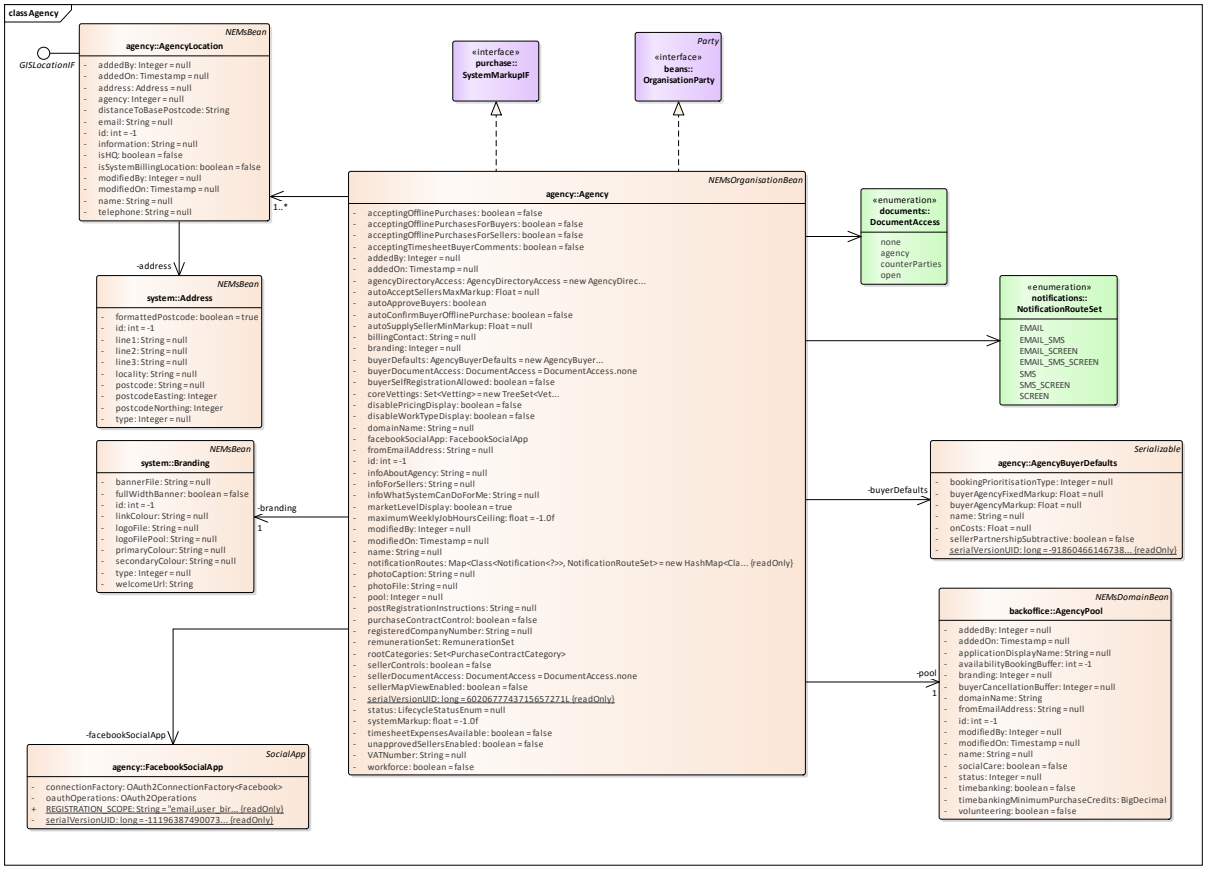

Figure 7 – Agency class

Agencies set rates, and limits that are applicable across the agency:

- The default `Vettings` available at that agency.
- The `RemunerationTypes` (paid, unpaid) available.
- The maximum weekly hours.
- The agency mark-up.

They can also be configured, such that the behaviour of agencies can vary substantially, including:

- Offline bookings.
- Self-registration of buyers.
- The ability for unapproved sellers to be booked.

2.5.5 Pool

`Pools` are groupings of agencies. All agencies belong to a single pool (which may be the default backoffice pool). The pools provide a means for managing sets of agencies, their partnering, and some common settings.

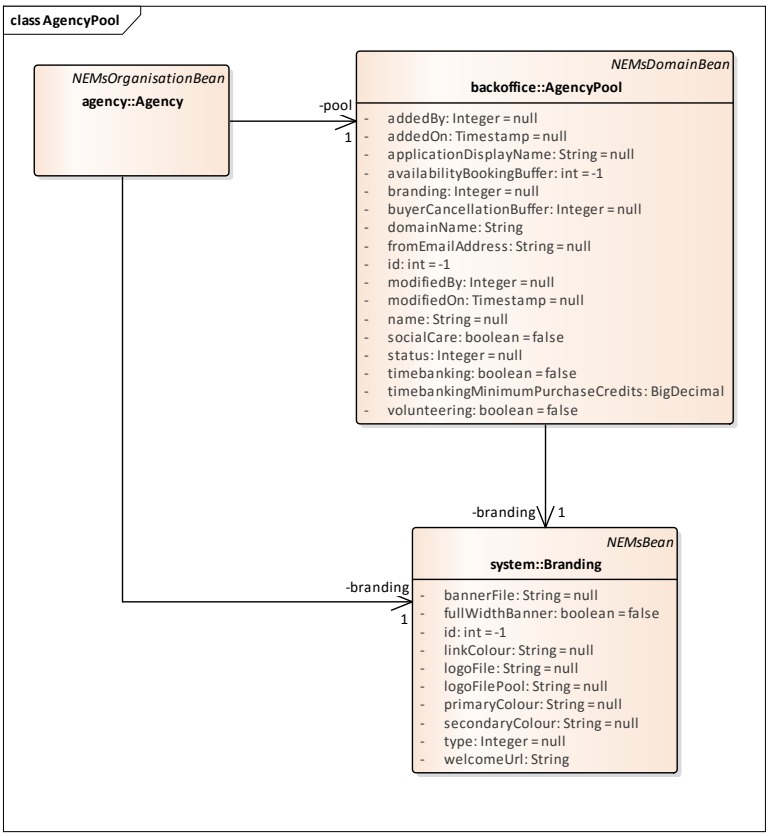

Figure 8 - AgencyPool and Branding classes

2.6 Availability

2.6.1 SellerDiary

A `SellerDiary` entry is a continuous period of, either availability, or a job, for a `Seller`. This sits at the core of the system, determining when the `Seller` is available. The implementation is simply a starting date-time, an end date-time, a type (available, or booked), and a `Job` (if booked).

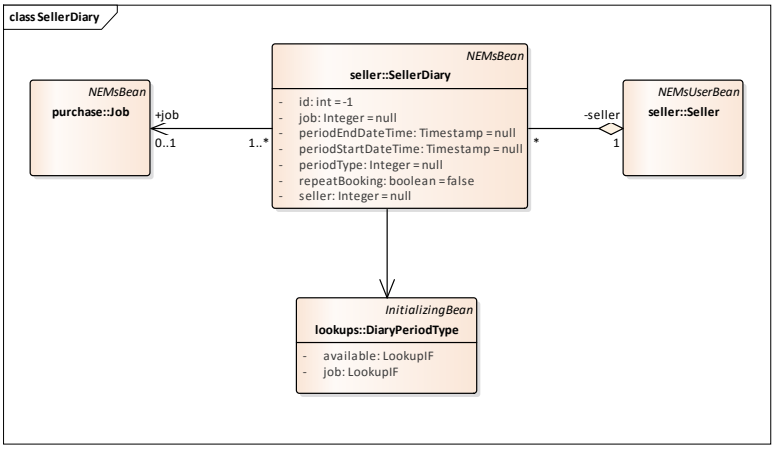

Figure 9 - SellerDiary

2.7 Purchasing and Booking

2.7.1 Purchase

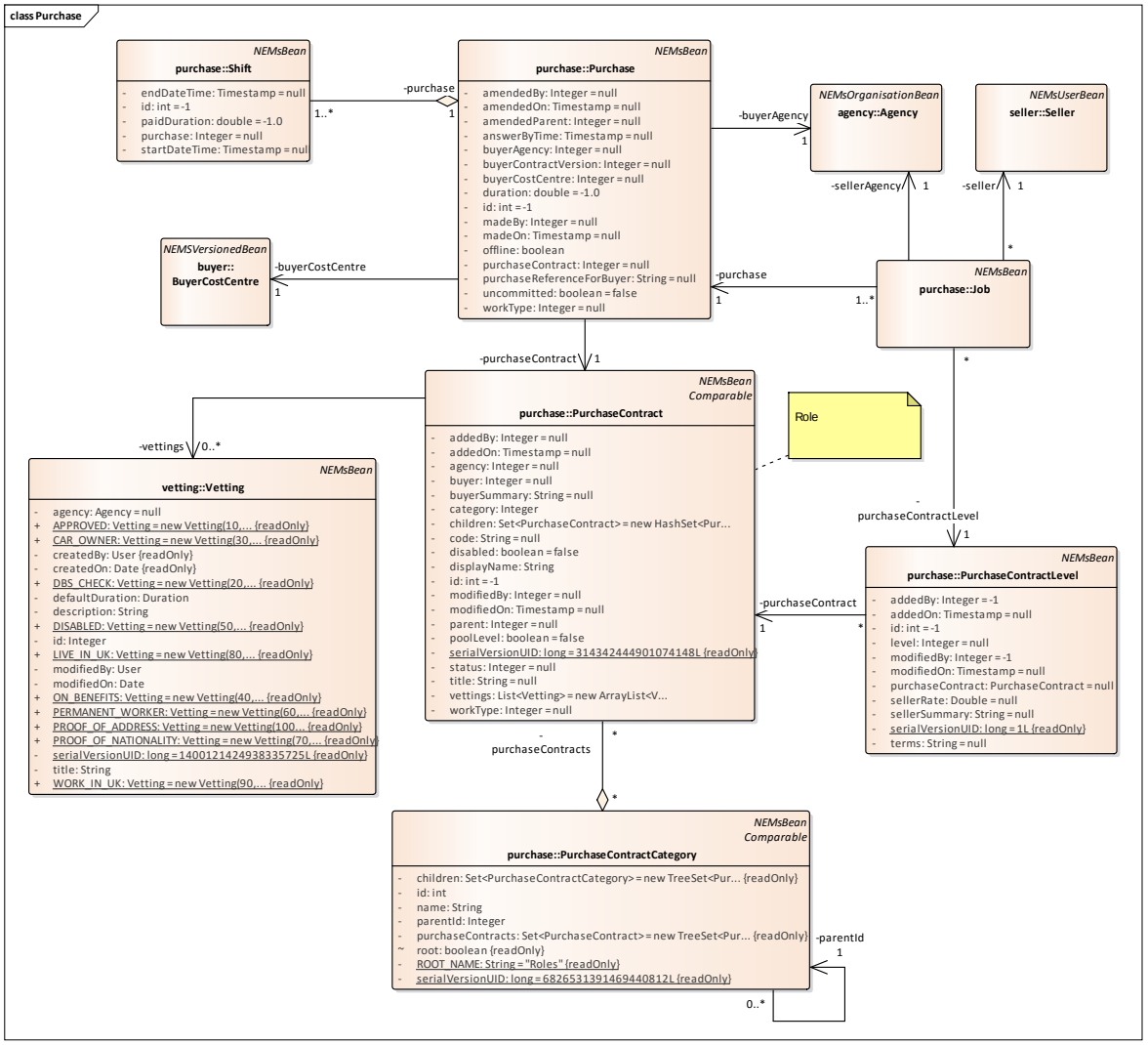

Figure 10 - Purchase Classes

- A `Purchase` is a booking of one or more Sellers.
- `Purchases` are initially in an uncommitted state. A `ScheduledTask` periodically removes uncommitted purchases.
- All purchases are made against a Cost Centre.

2.7.2 Job

A `Job` (Figure 13) is a booking for one `Seller`, for at least one `Shift`.

The state of the `Job` (i.e. `waiting`, `confirmed`, `cancelled`, `rejected`, `responded-late`, `no-response`, `active`, or done) is shown in Figure 12.

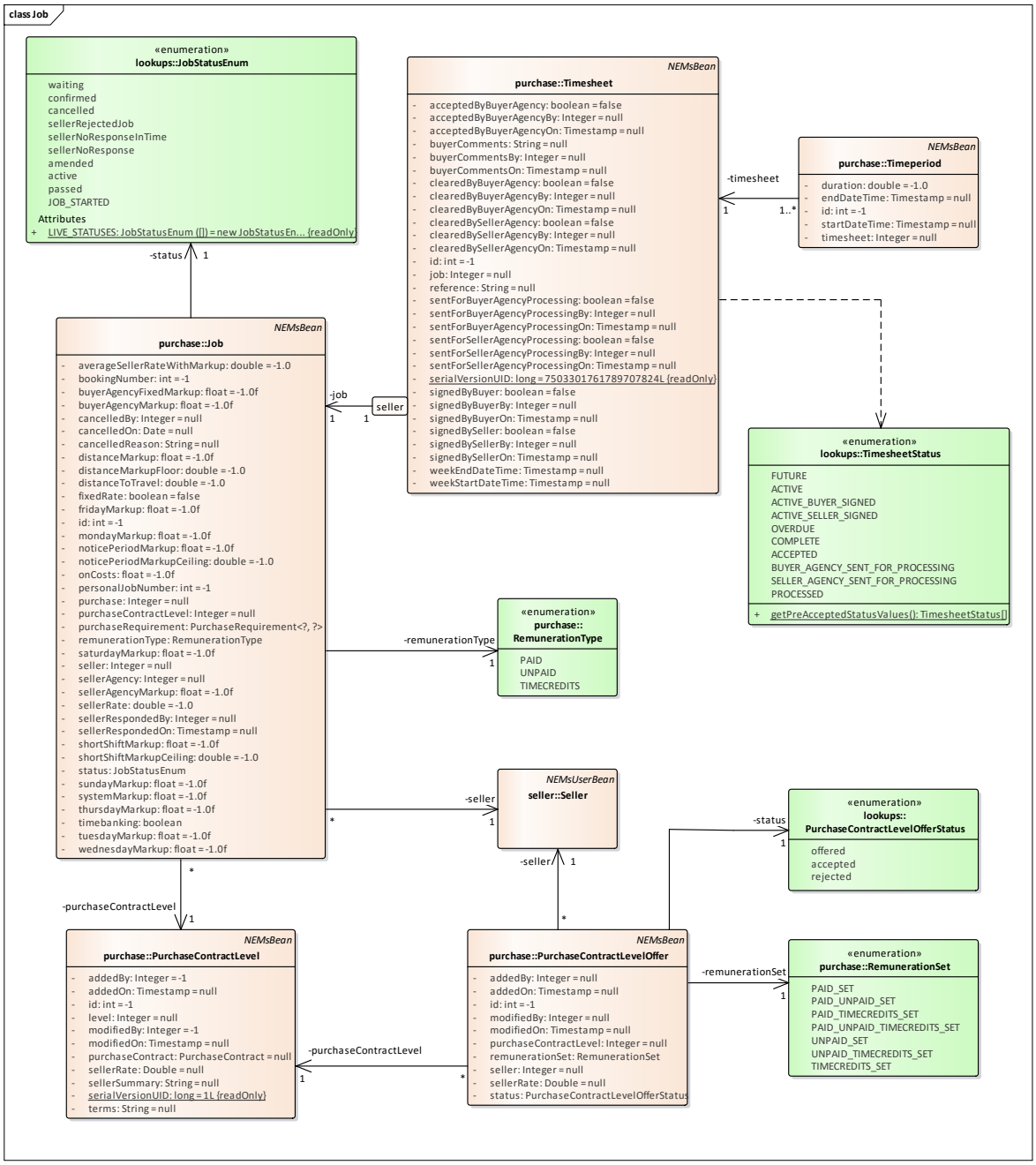

Figure 11 - Job class

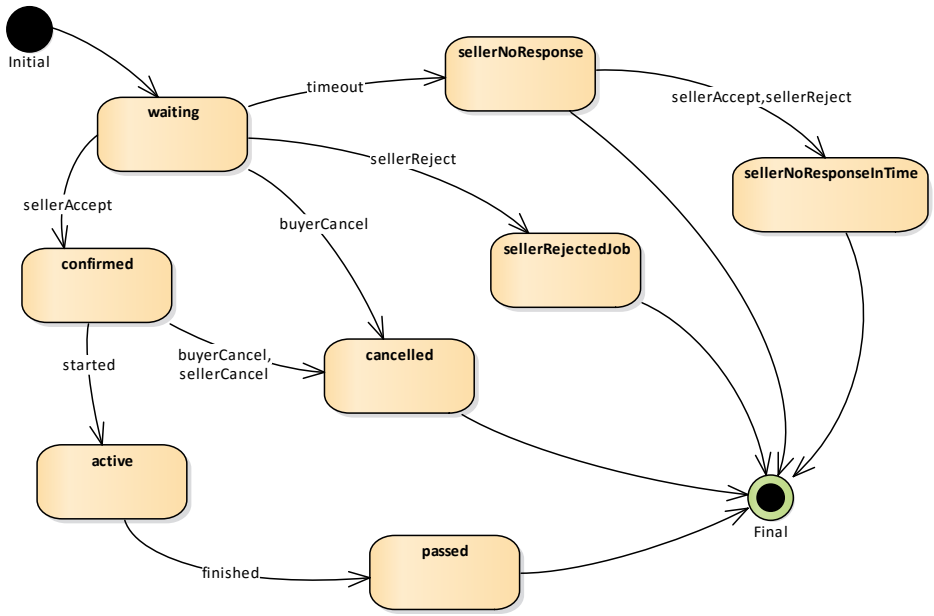

Figure 12 - Job states

2.7.3 Timesheets

Every `Job` has a corresponding `Timesheet` (Figure 11). It provides a record of time worked, plus any additional expenses, where appropriate. Figure 13 shows the timesheets states. Once a job is complete, *and* signed by bother buyer, and seller. Complete timesheets can then be *processed* by the agency (or agencies, in the case of a partnered booking). Typically processing marks the timesheet as having been exported to a payroll system.

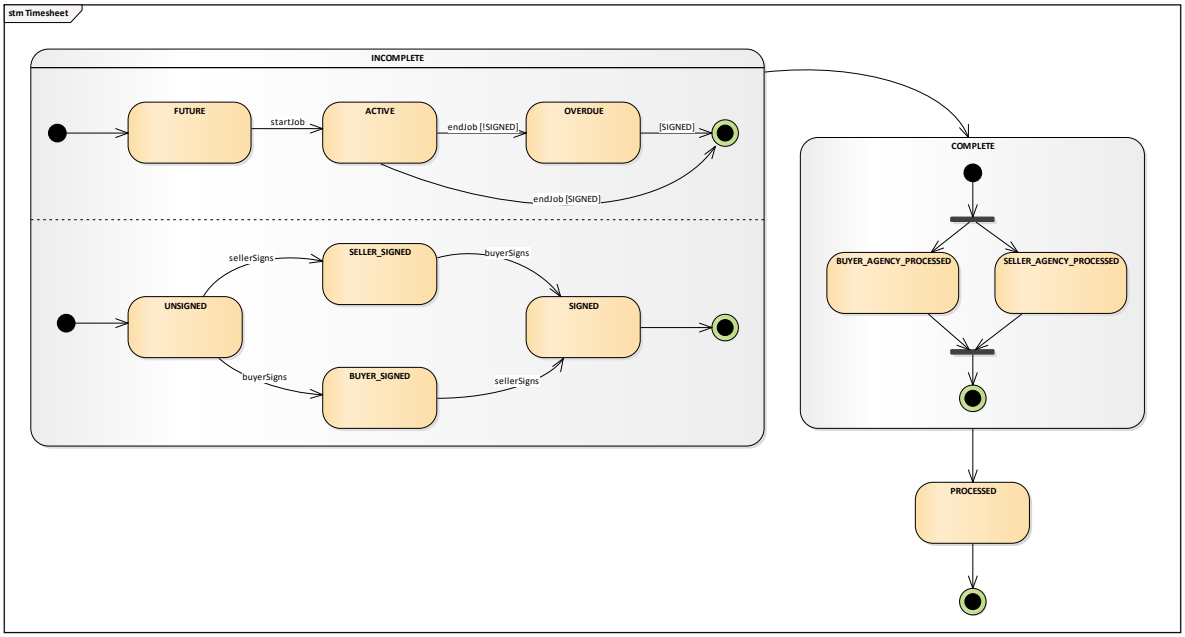

Figure 13 - Timesheet states

2.8 The Booking Process

The booking process is shown in Figure 14. A buyer's requirements are searched using the *Big Calculation*, to match available sellers. Aggregated availability for the coming weeks is returned, allowing the buyer to further refine their requirements to periods when sellers are available.

The buyer is then able to select one or more sellers. At this point the Purchase, with Jobs per seller, is created, in an uncommitted state[4].

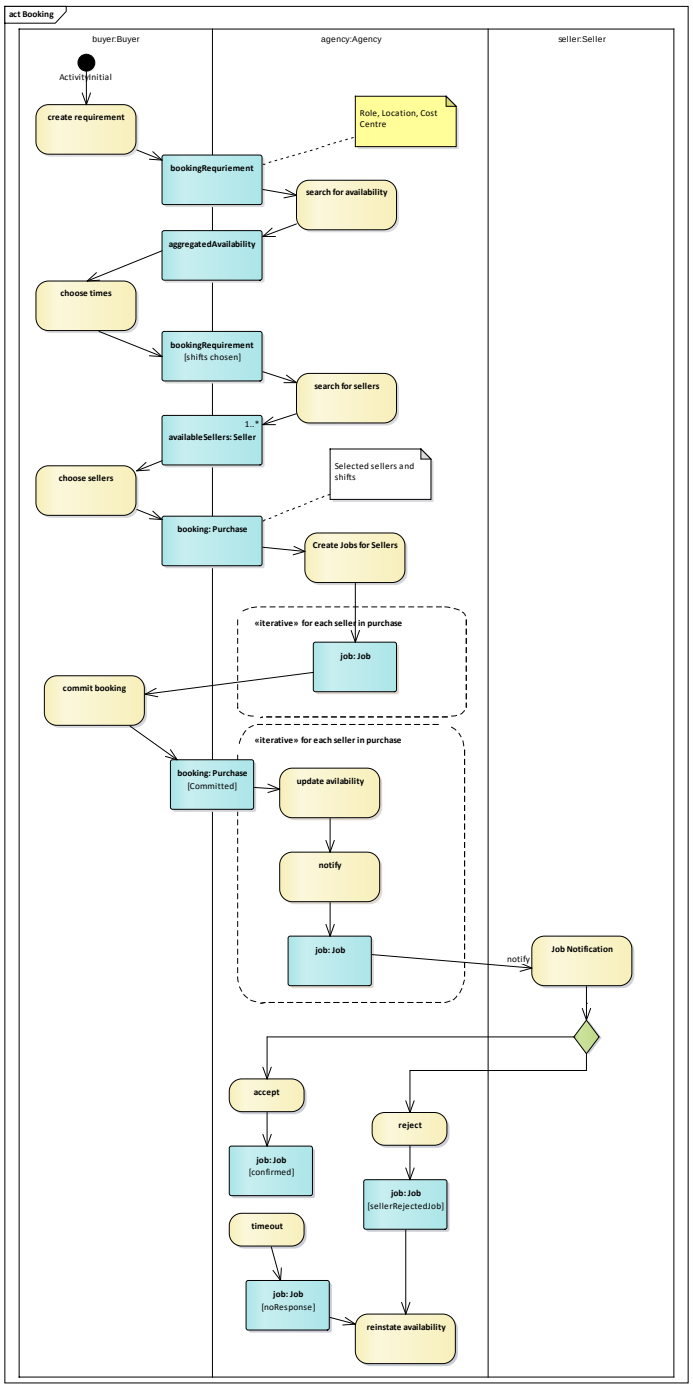

Figure 14 - Booking process

[4] This is *not* a two-phase commit. We don't lock availability, or prices.

The buyer then commits the purchase. At this point a notification is sent to each seller, and their availability is removed. The seller will then either:

- Accept the job.
- Reject, or ignore the job. In which case availability is reinstated, and timesheets removed.

2.8.1 PurchaseContract

Roles are represented by the `PurchaseContract` class as shown in Figure 10. They might represent a specific role, a job type, or a department, say.

Roles have at least one `PurchaseContractLevel`, allowing for different grading within roles. Sellers are linked to `PurchaseContractLevels` by `PurchaseContractLevelOffer` which also contains the seller's price.

In addition, roles:

- May be specific to a Buyer, an Agency, or offered across the entire Pool.
- Can belong to one or more `PurchaseContractCategorys`.
    - `PurchaseContractCategorys` are organised into a tree hierarchy.
- Can have vettings, redistricting which sellers can be booked for the role.

2.8.1.1 Offline Booking

An offline booking creates a regular `Purchase`, but bypasses the usual rules, including allowing past bookings to be entered. In this way uFlexi can be used as a system of record, to record bookings that have been made outside of the system.

2.8.2 CostCentre

A `CostCentre` is An account against which `Purchases` are made. They are linked to a buyer location; every buyer has at least one cost centre.

Cost centres can be used to restrict spending, or just track spending. They have a balance in a given `RemunerationType`. If the cost centre is *controlled* then bookings will be prevented if the balance falls below a given threshold.

The `CostCentre` entity uses an optimistic locking strategy (JPA versioning) to safely handle concurrent purchases against the same `CostCentre`.

2.8.3 PurchaseRequirements and Repeat Bookings

`PurchaseRequirements` are requirement for a given number of sellers to fulfil a `RepeatPurchaseRequirement` (Figure 15).

`RepeatPurchaseRequirement` is a weekly repeating `PurchaseRequirement`. It specifies the role, the required number of sellers, and the shifts. There are two concreate subclasses of `RepeaturchaseRequirement`: `AutoRepeatPurchaseRequirement`, and `ManualRepeatPurchaseRequirement`.

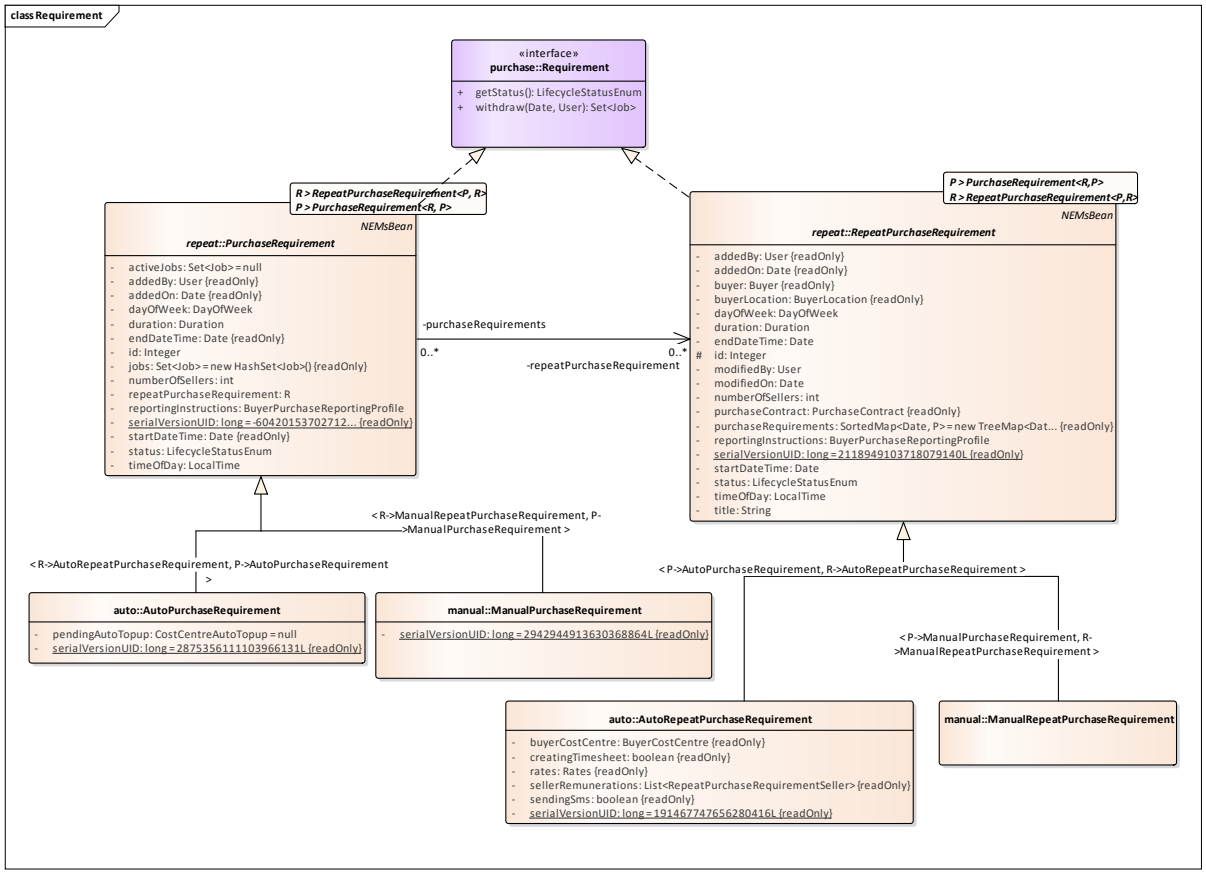

Figure 15 - PurchaseRequirements

For both types of `RepeatPurchaseRequirement`, the `ScheduledTask` `STPopulateRepeatPurchases` creates the appropriate `PurchaseRequirement` for the next 10 weeks (this is configurable).

`ManualPurchaseRequirements` (future requirements) are pending requirements that must be fulfilled manually by the agency.

`AutoPurchaseRequirements` are automatically fulfilled by the `STBookAutoRepeatPurchases` scheduled task. This will attempt to fulfil every unfulfilled `AutoPurchaseRequirement` in the system, until the requirement is either fulfilled, or lapses into the past.

2.8.4 Remuneration

The system provides support for different kinds of remuneration: Paid, unpaid, and, historically, timebanking.

Buyers, sellers, and their roles can be any combination of these, allowing a seller to perform both paid, and volunteering jobs.

2.9 Vettings

`Vettings` provide a mechanism for tracking certifications, qualifications, and other items that may be required before a seller may be booked. Example vettings are:

- Driving licence

- Security clearance
- Work eligibility

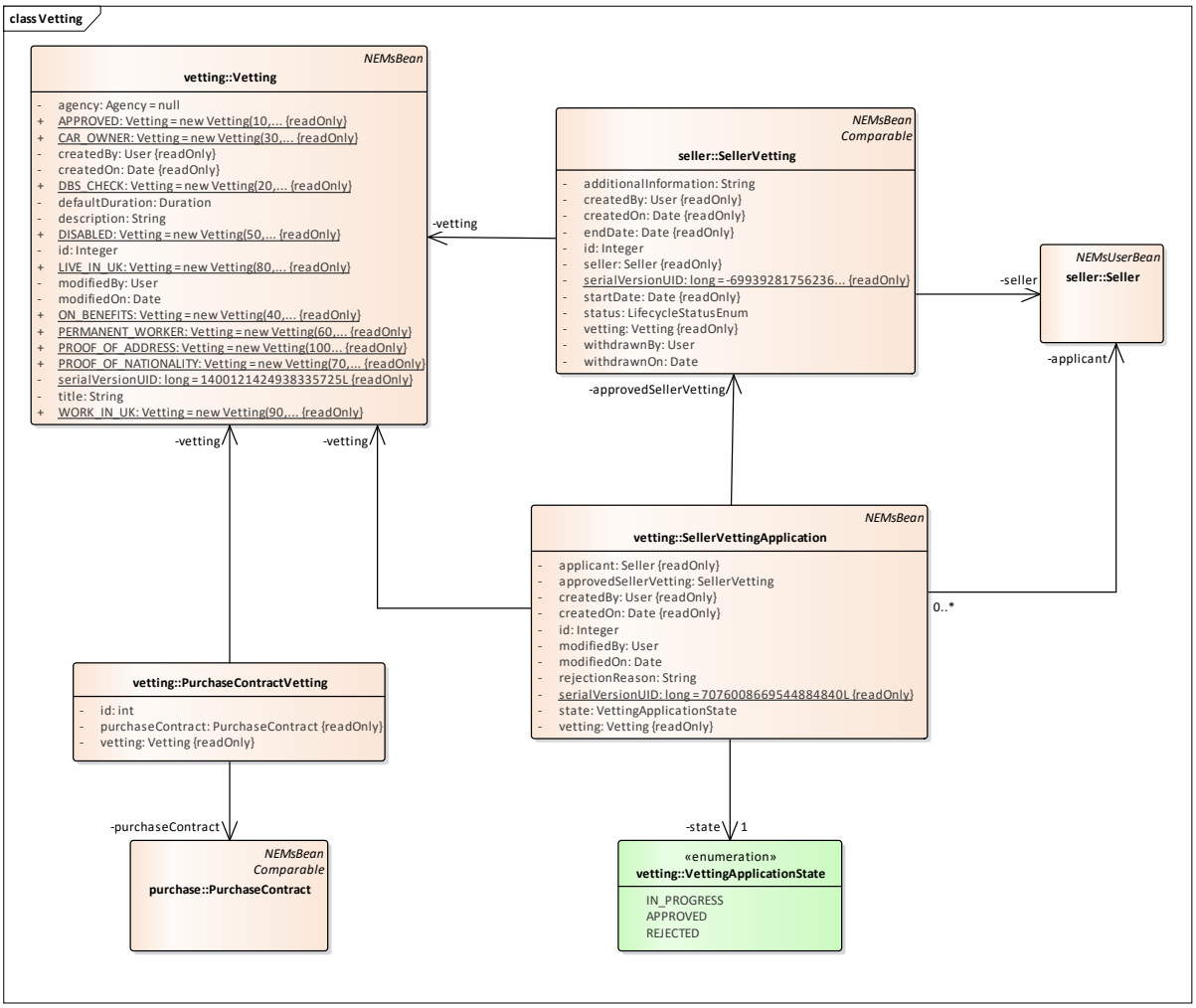

Figure 16 - Vetting class

Sellers can apply for vettings. It is then the responsibility of the agency to confirm any requirements are met. Vettings may have an expiry date.

Roles may have multiple vettings assigned to them. It is only possible to book sellers who hold all of the vettings for the role. Furthermore, the sellers vettings must not expire before the end of the last shift in the booking.

2.10Working Time Restrictions

Working time restrictions allow a set of restrictions to be applied to specific sellers, overriding their default rates and limits. The `WorkingTimeRestriction` class (see Figure 17), defines those rates, and limits. An agency can have multiple working time restrictions, although a seller may have at most one active. It is also possible to define breaks (`SellerWorkingTimeRestrictionBreak`) where the restrictions do not apply, and the system defaults to the sellers own defined rates and limits, or that of the agency.

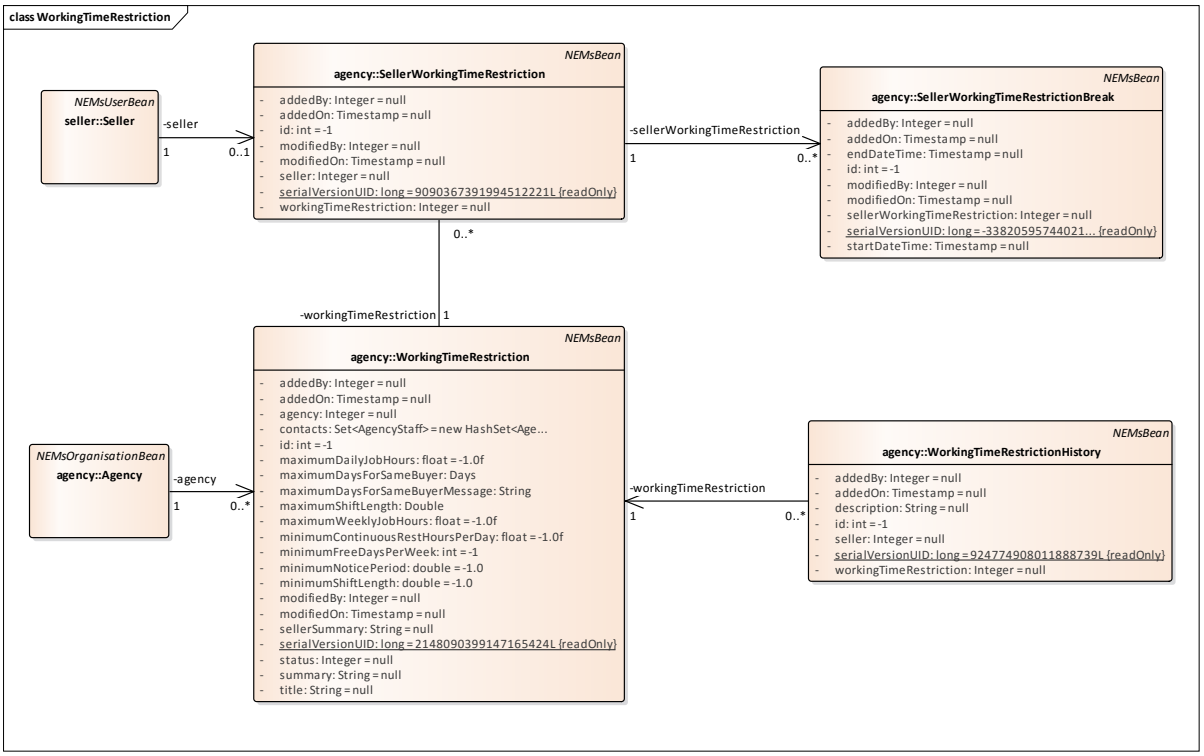

Figure 17 - Working time restriction classes

2.11 Big Calculation

The `PostgreSQLRichSellerManager` DAO (note the legacy naming conventions here), performs what has become known as the *Big Calculation*; the part of the system responsible for matching sellers to buyer requirements.

The code itself is an SQL query, which is built dynamically depending on configuration, and ultimately returning a set of `RichSeller` objects, containing details of matched Sellers, including pricing details.

```
public List<RichSeller> findMatchingSellers(final PurchaseCriteria criteria, final
PurchaseConstraints constraints)
```

The `findMatchingSellers` method takes `PurchaseCriteria` and `PurchaseConstraint` arguments.

2.11.1 Constraints and Criteria

2.11.1.1 PurchaseCriteria

`PurchaseCriteria` Contains details of the buyer, the cost centre, required shifts, and any specific sellers to search on.

2.11.1.2 PurchaseConstraints

`PurchaseConstraints` affect which parts of the Big Calculation are enforced. The following can be enabled, or disabled according to purpose:

- availability
- contactability
- working time restrictions (including seller terms)
- distance
- purchase contract
- minimum period of notice
- remuneration
- vetting
- existing jobs
- cost centre limits

The `PurchaseConstraints` class also contains a number of pre-defined `PurchaseConstraints`.

2.11.2 Aggregated Availability

Aggregated Availability uses the same SQL code as the Big Calculation to count available Sellers, by hour, over a given period of time (e.g. a week).

3 Architecture

This chapter describes the existing uFlexi software architecture, platform, services, and external dependencies. In later chapters enhancements to the system are discussed. Chapters towards the end of the document detail an outline for a complete rebuild.

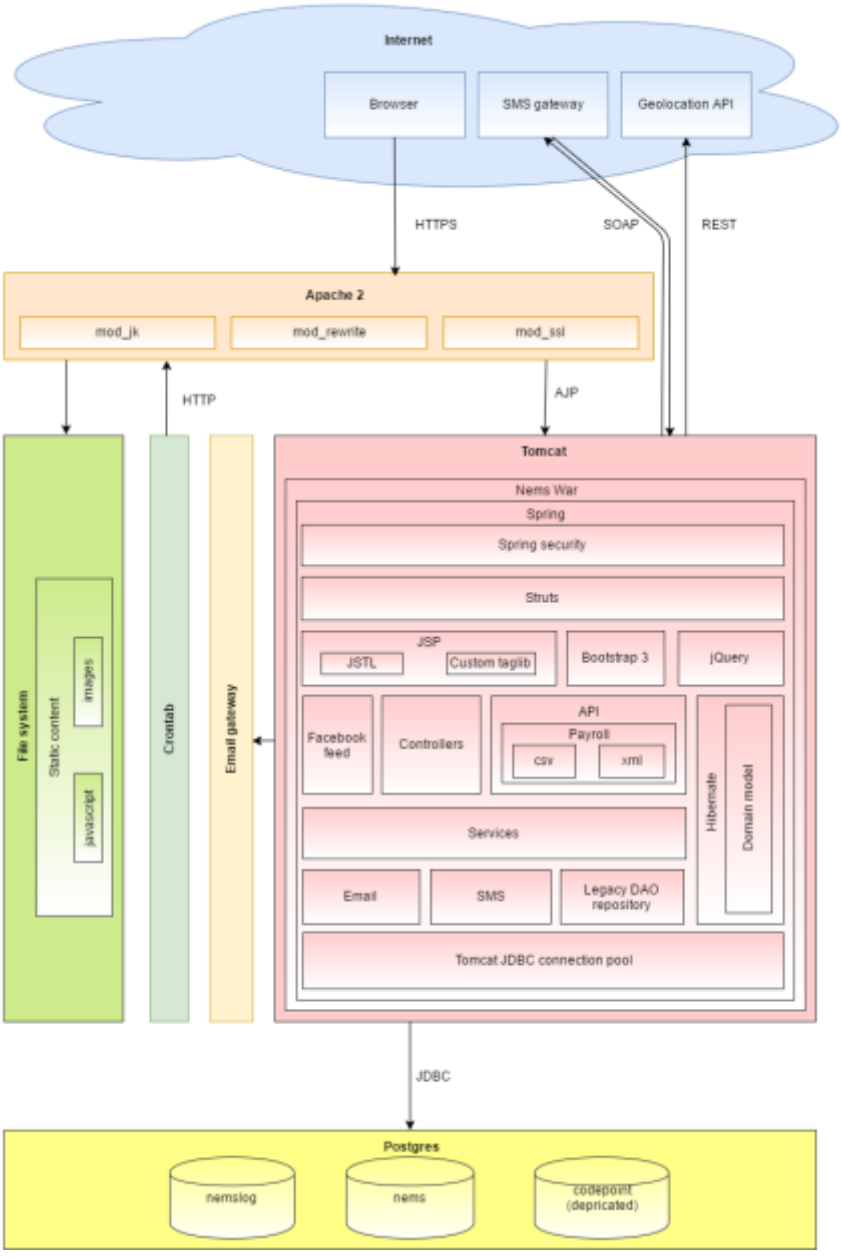

Figure 18 - uFlexi Stack

3.1 Technology Stack

This section discusses the various platforms and technolgies in use and how they relate to each other. The diagram in Figure 18 outlines the components of the system and how they interact with one another and the external environment.

3.1.1 Apache HTTP server

Apache HTTP is used as the internet facing server of the application, and version 2.2.22 is currently installed. While none of the application logic itself resides in Apache, and it forwards most requests to a Tomcat server, it performs the following functions:

- Provides caching of static resources (e.g. images and JavaScript).
- SSL communication, it forwards requests onto Tomcat via the AJP protocol.
- EV certificate covers sub-domains (e.g. myagency.uflexi.com).
- Can provide load balancing, if it becomes a necessity, with horizontal scaling.

3.1.2 Tomcat 7 application server

The main application logic runs in a Tomcat 7 application server which is a standard servlet container. Requests come in from the Apache HTTP server via the AJP protocol and these are converted into requests that the uFlexi application can process.

3.1.3 Java 8

The Tomcat application server, and the uFlexi application it executes, both run in the Java 8 programming language environment (Java virtual machine, or JVM).

3.1.4 Spring

The uFlexi application is built using the Spring framework, and the version in use is 3.2.18. The Spring framework provides many features including:

- Dependency injection
- Integration with various libraries that parts of the application make use of. For example, the object-relational mapping (ORM) tool Hibernate.
- Utility libraries of its own

3.1.5 Spring security

Spring security is used to provide a security abstraction, the version in use is 3.1.3. It provides a mechanism for assigning roles and permissions to users in the application. It ensures that users can only see the parts of the site they are permitted to. The section security has additional information.

3.1.6 Struts 1

The Struts is a model view controller framework. It provides a mechanism for routing incoming requests to *struts actions* which then perform logic and return a view to be rendered. The version in

use is 1.3 which has reached end of life. This would ideally be replaced by a newer technology choice which is still receiving maintenance. The section on Spring Boot contains information on a recommended alternative.

3.1.7 JavaServer Pages

The view technology used to render HTML to be returned to the browser is JavaServer Pages (JSP). Note that while it is possible to have programming logic in JSPs this is generally considered bad practice and the application uses them for rendering HTML only.

3.1.8 jQuery

The JavaScript framework jQuery is used to facilitate application JavaScript, and the version in use is 1.9.1.

3.1.9 Bootstrap

Bootstrap is used to provide a responsive web application, adapting the interface to the device used. It also provides a consistent CSS styling across the application. The version in use it 2.3.2.

3.1.10 Spring JDBC

Spring JDBC is used, though not throughout the entire application, to provide an interface and convenience methods to communicate with the database, and the version in use is 3.2.18. This set of libraries, while functional, has largely been obsoleted by newer frameworks, such as Hibernate.

3.1.11 Hibernate

Hibernate is used in the application, where Spring JDBC is not, for persistence to the database, and the version in use is 4.2.21. Hibernate is much less intrusive, and provides additional security benefits. Ideally the application would use a single technology to communicate with the database and the section on Hibernate in the technology upgrade chapter outlines some more advantages of using this.

3.1.12 Postgres 9.5 DBMS

The persistence layer is a relational database Postgres 9.5. This is a mature open source database solution with powerful features, the PostGIS section details one such extension which the application uses.

3.1.13 Crontab

Crontab is used to run scheduled jobs in the application. It does this by running Wget commands to access HTTP endpoints on the application. Currently the scheduled actions are defined in `NEMsScheduledTaskType` and are:

- Poll for messages. Check if there are SMS messages in the application from the SMS supplier.

- Update seller rate for age. Check that wages are over a specified minimum threshold based on age and fix those that are not.
- Remove uncommitted purchases. Removed uncommitted purchase after a specified timeout.
- Remove expired partnerships. Mark, as removed, partnerships that were scheduled for cancellation.
- Remove expired partnership offers. Expire partnership offers. Remove if expired date more than a specified time in the past.
- Clean-up unattended jobs. Clean up jobs that were not accepted.
- Update seller fixed rates for age. Check that fixed wages are over a specified minimum threshold based on age and fix those that are not.
- Populate repeat availability: Set availability, a specified number of weeks in advance, based on repeat availability.
- Populate repeat purchases.
- Book auto repeat purchases.
- Cost centre auto top up: Apply any automatic top ups to cost centres.

3.1.14 SMS gateway

The uFlexi application sends and receives SMS messages via a third-party API. The later section in this chapter details more information on the SMS gateway.

3.1.15 Apache Ant

The build tool for application is Apache Ant, the section on build process in this chapter has more information.

3.2 Bytemark platform

The current demo and test systems run on the same single virtual machine, hosted by Bytemark, with both applications running in a single Tomcat. Additionally, the Postgres database and the Apache HTTP server also reside on this machine. The machine itself is allocated the following resources:

Memory	16 GB
Processors	5 core 64-bit x86
Storage	SSD
Operating System	Debian Wheezy LTS

See the sections on Tomcat, Apache HTTP server and Postgres for more information on these components.

3.3 Build process

The build process is currently part manual, and part scripted. Apache Ant is used to build, and deploy artefacts. The process is as follows:

1. A developer builds the deployable artefacts locally using Apache Ant. These are targeted to an environment with information on database location, SMS gateways etc. built into them. The section on purging secret information contains some additional information on these properties. The build artefact also has version information stamped into it so that it is possible to tell which version of the code is running.
2. The artefacts are then uploaded to the production server via SCP.
3. The developer then SSHs on to the production server and runs a series of scripts which:
    a. Stop the server
    b. Backup user files. The section on file storage outlines why this is required
    c. Deploy the new artefact and restart the server

While this process works, it is prone to error and is not streamlined. The continuous integration section outlines a preferable approach to building and deployment.

3.4 Security

- HTTPS is used for login and other sensitive actions. Site-wide HTTPS is recommended
- Passwords are stored as un-salted SHA/1 hashes. The section on hashing has more information on how this could be improved
- The uFlexi application provides role-based security via Spring Security and this works by:
    - Restricting URLs based on assigned roles
    - Using security tags in the JSPs determine what is permitted to be shown by role
    - Role-based security in the Struts actions restricted behaviour by role
- Additionally, code-based custom checks are present for when role does not provide enough granularity. e.g. the `NEMsAgencyConsistencyFilter` which checks that a user can access a given agency. This cannot be ascertained using global roles.

3.4.1 Impersonating users

While Spring Security provides the framework for security, it has had its functionality increased by adding code at extension points in its design. This allows impersonation of users by back office and agency staff. Agency staff can impersonate sellers and buyers. Back office staff can impersonate sellers, buyers, and agency users.

Once a user has logged in they can request impersonate another user, assuming they are allowed. When this happens the Spring security framework is called to repeat the authentication mechanism, which results in new roles being assigned. This is done in code and does not require action from the user. It allows the impersonating user to view URLs they wouldn't otherwise been able to.

Additionally, the application keeps details, in the session data, of who is being impersonated and by whom. The application then takes its cue from this session data as to how it should behave.

Outlined below are the steps that are taken to impersonate another user:

1. Initial login
    a. `NemsAuthenticationFilter` called as part of authentication.
    b. Details on whether the user is impersonating anyone is set on a `NEMsWebAuthenticationDetails` and attached to the Authentication token. As this is the initial login there is no information set.
    c. Authentication token is checked and found to be valid.
    d. Roles are assigned. Additional roles can be given based on the content of `NEMsWebAuthenticationDetails`. However, as this is effectively empty only the standard set of roles for the user in question are granted.
    e. `NemsAuthenticationFilter` adds a `NEMsSessionStorage` to the session. This is the object that contains information which the application reads to determine which user is logged in and how it should act.
2. Impersonate another user
    a. A user is already logged in.
    b. Request is sent to a Struts action to act as another user e.g. `BackofficeBecomeAgency`.
    c. This Struts action sets up the `NEMsSessionStorage` to contain information about the user whom is being impersonated. The application will now act as if the current user is the impersonated.
    d. The Struts action then, in code, forces re-authentication.
    e. A `NEMsWebAuthenticationDetails` is filled in with information about impersonation from the `NEMsSessionStorage` and is attached to the Authentication token.
    f. Roles are assigned. Additional roles are now set based on the `NEMsWebAuthenticationDetails`.
    g. Impersonation is complete.

3.4.2 IP Filters

The application has a custom filter to ensure the IP address that the user authenticated remains the same throughout their session. This is to help protect against hijacking of sessions, however, the live site is now completely behind HTTPS, which prevents this form of attack. Thus, the filter is no longer necessary. For completeness, the filter is `NEMsCheckRemoteAddrFilter`.

3.5 File storage

Files are stored as part of the application, for example profile pictures. The files themselves are stored in the Tomcat `webapps/` directory along with the deployed application. The means there is a requirement for specific deploy scripts to ensure that the document store is not overwritten. This also has implications for scaling the application horizontally; see the section on files in a clustered

environment for additional details. This setup works, but is not ideal, and the section on shared file system has information on an alternative approach.

3.6 Session state

The application is not stateless; it requires that user information is saved in the `HttpSession` object between requests. The application will not function without this data, however current design has minimised the amount of information that is required to be stored. For reference the `ScopeStorageKeys` class contains a list of the lookup keys for objects saved in the session. The most important of these is `NEMsSessionStorage` which contains information about the current user. The section on impersonating users has additional information on this process.

In addition to minimizing the data stored in the session all objects stored in it are marked `Serializable`. This means that they can, in principle, be converted into a binary format and transferred to another machine, or written to disk. Where a property of an object stored in the session cannot be serialized it has been marked as `Transient`. An example of this sort of property would be a reference to a database connection. Properties marked as Transient are then ignored by the serialization process. Thus, when these objects are de-serialized they will be incomplete. To address this issue the class `NEMsSessionStorageInjectionFilter` patches any objects which are de-serialized on a different machine. The section on scaling horizontally and more specifically session state has details on why serialization is important.

3.7 Caching

The application uses caching, where it can, to limit the load on the database. While libraries are used to do the actual caching, determining what should be cached is done by custom application code. It makes use of aspect oriented programming (AOP) to intercept method calls and return previously computed results where appropriate. The underlying cache implementation is *OSCache*. Note that this was part of *OpenSymphony* open source group which has ceased to exist. Thus, while the application continues to run with no ill effect, any future changes to caching should consider a different implementation, for example Ehcache.

Alternatives to this approach outlined in the caching section under the scaling horizontally section. This is particularly relevant as the current caching functionality is not distributed, and this makes it unsuitable for a clustered environment, as detailed in the aforementioned sections.

3.7.1 Custom caching rules

The rules on when to save and invalidate cached database entities are complex and this section discusses the code that performs these actions.

All Java classes annotated with `NEMsCacheGroup` become candidates to have the results of their methods cached. The results are stored under an unique `String` key but also with against a `String Array` representing groupings of results. Either the key or one of the groups can be used to invalidate cached results.

The key is a `String` representation of the signature of method being called with parameters passed to it. The following is an example key `String`:

```
public com.torchbox.nems.beans.system.User
com.torchbox.nems.dao.system.PostgreSQLUserManager.getByEmailAndAgency(java.lang.String,int,boolean
)_tom@poly.io_-1_false
```

The groupings have three types:

1. A default group. All cached entities have this entry, providing a mechanism for the entire cache can be invalidated.
2. A `String` representation of the class that the method is being called on, with the id of the returned result.
3. The name specified in the `NEMsCacheGroup` annotation and the id of the returned result.

The following is an example of the String Array group that an entity might get cached against:

```
[DEFAULT_GROUP, com.torchbox.nems.dao.system.PostgreSQLUserManager_1, user_1]
```

With both the key and the groupings the application has enough information to invalidate part of the cache when required e.g. after an update.

3.8 Exception handling

Under certain circumstances the application will throw an `Exception` which can bubble up and out to Tomcat. These fall into two categories, those that we might expect, and those that represent a failure in the system.

In the first instance, where under certain conditions the `Exception` is expected, the Tomcat `web.xml` specifies the page a particular to be shown to the user. This will have more specific information to show the user. An example would be if a buyer withdraws from uFlexi while a second user is attempting to access them. In this case, a `BuyerWithdrawnException` would be thrown and the second user would see a tailored error message.

For errors that were never expected the web.xml specifies a generic an 500 Internal Server Error page. Note that this page only shows on production distribution builds. In the developer environment stack traces are displayed directly to aid with development process.

3.9 Logging

The application uses Commons Logging which delegates to Log4j. This is output in two places: Firstly to the application directory which gets overwritten on redeploy, but more importantly to the Tomcat log `catalina.out` which is preserved. Messages logged out are from the application code and third-party libraries. A configuration file, `log4j.properties` specifies the level of logging output by the application. It is highly configurable, but cannot modified while the application is running. Note that this properties file is part of the production distribution.

In addition to standard to disk logging, important actions are logged to an audit database. There are currently 11 audit message types, and these all relate to scheduled jobs that the application runs. See the section on Crontab for these scheduled audited tasks.

The application also has standard logging from the component frameworks, i.e. Tomcat, Apache, and Postgres.

3.10 Code-Point

The application relies on geographical data to ensure buyers are matched to appropriate sellers. Originally Code-Point provided this data as a CSV file which can then imported into our `codepoint` database, which is separate from the main application database. The data is simply a list of UK postcodes with their associated position in Eastings and Northings. However, this system will not work well in the US where the equivalent postal codes can cover large geographic areas e.g. 89049 which is over 10000 square miles. For this reason the application is being transitioned over to use the Google geocoding API to determine locations, and PostGIS to translate between coordinate systems.

3.11 Google geocoding API

The Google geocoding API provides a REST service to obtain location data for addresses. This provides a globally agnostic way of determining locations from addresses and will work equally as well in the UK as the US.

3.12 SMS gateway

The application both sends and receives text messages from a SMS gateway. The provider is Esendex, and communication is via SOAP over HTTP. The client API was built using Apache Axis. Messages are sent to sellers to inform them of pending work. The sellers can then respond via text to the message. The application polls the gateway for responses at scheduled intervals, see the Crontab section, and updates itself accordingly.

4 Scalability plan

4.1 Overview of current application performance

While there is currently no functioning performance test suite for the uFlexi application, it has in the past, been subjected to load testing and evaluation. There have been changes in functionality since then, but the metrics from the last run should be indicative of the amount of load that the application can handle. Note the aggregated availability grid (AAG) which was deemed to be the bottleneck in the last evaluation has been subsequently optimised.

The previous load tests were designed to measure performance of several user journeys but the bottleneck was found to be in the check availability functionality. A series of tests were performed from 100 sellers up to 10,000.

With this user journey and 2,500 sellers in the database the system could handle 4 user journeys per a minute. Performance degraded at 10,000 with 0.2 user journeys per minute. However, much work has been done subsequently to optimise this section of the code. Finally, as this was a load test, user journeys were being run as quickly as possible. Real world scenarios may differ from this and historically the application has run successfully with approximately 60,000 users, but with the AAG disabled. Again, this was before the AAG optimisation work.

Attached are the full findings of the last performance test:

SciVisum Load Test Report for Slivers of 1

4.2 Determine current capacity

The only way to be sure of how much load the application can currently handle is to retest. This would find any bottlenecks, which can be optimised, and determine how many users the system can cope with.

Ideally the load tests would then be kept up-to-date with the code and run against the application regularly. This would mean future optimisations could be assessed quantitatively and ensure no degradation of performance from new features. Keeping the load tests in synchronisation with the application will require an ongoing effort, but if they are run as part of the build then this should be mitigated to an extent as incompatibilities should be immediately flagged. See the section on continuous integration for more information on builds.

To perform load tests that will yield useful data, the following would be implemented:

- Insert data into the database: As outlined in the section on generating test data for the integration tests the services could be used to insert data. A quicker, though less maintainable, approach maybe to insert rows directly into the database. It might also end up being more complex though. Finally, the test and demo servers both have large data sets

and this may suffice, though it would be less than optimal, not particularly maintainable or configurable.

- Determine representative user journeys: To help with this it should be possible to use the data from Google Analytics and there should be a reasonable history of journeys to explore. However, this is optional and there is anecdotal evidence which paths through the application are likely to see a lot of use or be computationally expensive.
- Generate the load tests: The JMeter load testing tool is an effective tool and would be recommended. JMeter scripts have the advantage that they can be run in services such as BlazeMeter which visualises the results well. There are of course many other testing tools, but irrespective of technology choice, it would be good to run:
    - Soak tests, to iron out e.g. memory leaks and see how the application holds up over an extended period.
    - Stress testing, with rapid spikes in numbers to simulate times in which traffic is high.
- Run the tests against an environment: Preferably this would not be a developer machine, but one which can be standardised so that we can easily repeat test in future. The ideal scenario would be a live-like environment. Should this be prohibitively expensive then something with less resource would still give useful data. If a cloud-based service is used for hosting, it should be possible to spin up a server just for the time it is needed for testing.
- Instrumentation: If the machine being tested against can be instrumented, see the section on Performance Monitoring for details, then more information can be gleaned about where most of the time is spent in the code, which would then drive future optimisation work.

After the tests have been run the results will need to be analysed. Once analysed, work to improve capacity may be required. The next section discusses some general approaches to this.

4.3 Improve capacity

There are several ways to improve the capacity of the system.

4.3.1 Optimising code

Optimising any poorly performing, or frequently executed code will improve capacity. Load tests as outlined above will help in identify where it is. Where code has already been written with efficiency in mind it may be possible to make parts of it run in parallel, this does however require the computation in question to be amenable and some problems can only ever be run in series.

4.3.2 Tuning

Much of the technology stack that application runs in can be tuned, but the Tomcat server, Apache HTTP and the database (see section database tuning) would be good candidates if load testing found them to be bottle necks. For example, the following are a few of the things that maybe tweaked to obtain greater performance from Tomcat:

- Specify how much memory it can use.
- Differing numbers of worker threads.
- Specify a given garbage collection strategy.

4.3.3 More powerful hardware

Running the application on more powerful hardware will improve performance. There is a limit to this as hardware performance has bounds. It can also be expensive, though often less than the development effort required to achieve the same ends. As an approach, this it is probably best suited to parts of the application which cannot be horizontally scaled, see the section on scaling horizontally. An example of where more powerful hardware might be appropriate is the database server. Of course, if the part of the application stack that is being given more resources is not the bottle neck then improvement is unlikely to be seen. This underlines the importance of load tests and determining where the actual issues lie.

Note that, as detailed in the section Bytemark platform, the full application stack of the demo and test site is all running on the same virtual machine. Here at least there is plenty of scope for more powerful hardware and running distinct parts of the application on their own machines.

If the servers are being migrated to more powerful hardware this would also be an appropriate time to automate the building of the environment.

There are several tools to script the building of environments and the section on environmental scripting discusses one of them.

4.3.4 Scaling horizontally

The part of the application stack that runs in the Tomcat application server can be scaled out horizontally by adding more Tomcat instances. A load balancer would sit in front of these and divvy out the requests to them. The section on the Apache HTTP server discusses this as one of its uses. More Tomcat servers equate to the ability to handle more load, if the servers are the rate limiting factor. There are however a few changes that would need to be made to the application to enable this.

4.3.4.1 Session state

Currently the application is working on a single Tomcat server and so session state held local to that machine does not represent a problem. See the section on session state for more details. When scaling out horizontally session state becomes a consideration, as if it is not available to all then a given user's requests must always be directed to the same server.

The following sections elaborate on the different approaches to handling session data in a horizontally scaled (clustered) environment.

4.3.4.1.1 Sticky sessions

With sticky sessions, for example Apache sticky Apache sticky sessions, the load balancer in front of the Tomcat servers is setup so that a particular user is always sent to the same machine. With this approach, there is no need to change the code base and the application works as it always has. The disadvantage is that if a server is taken down, for example for maintenance, the users it was serving cannot be redirected to another. They will have to log in again and will be directed to their home page, losing any work that had not been saved to the database. In practice, this is not that much data but is a jarring experience for the user.

4.3.4.1.2 Session Replication

With session replication, the `HttpSession` state is copied to all the Tomcat servers in the cluster. Thus, it doesn't matter which server in the cluster the user is directed to on the next request. This approach requires that the sessions be `Serializable` to work, but this is currently the case with the application. See section on session state for more details.

The disadvantages of replication are that it introduces development overhead to ensure the session remains `Serializable` and in configuring the Tomcat. Another issue is that it can introduce a lot of network traffic. This can be reduced by strategies such as each machine only replicating to a single backup. Tomcat 8 supports both modes of operation, though the single backup option is relatively new with less time being used in production.

4.3.4.1.3 Offload session data

Offloading the `HttpSession` state from Tomcat, and storing it elsewhere. Initially this might appear to be transferring the problem to another part of the infrastructure. However, it has the advantage that it will be infrastructure specifically designed with this role in mind. For example, a Redis cluster would be a good option and this has been designed with replication in mind. Another place that storage can be offloaded to is the user's browser. So, in this case, cookies with session data are encrypted with a shared server secret, and sent to the browser at the end of the request. They are then returned with the next request, and unencrypted by the server. Care must be taken with this method as potentially sensitive data is being sent to the browser. It is imperative that the method of encryption ensures that it cannot be decrypted at the browser, and that there is sufficient security around the shared server key. Finally, the session must be kept small.

With offloading, it no longer matters which server the user is directed to and additional network traffic is relatively light. This approach takes more development time though.

4.3.4.2 Caching

The application has caching to help reduce load on the database. See the caching section for a more detailed explanation. The caching is not however distributed and, like the session state, this presents issues with clustering. There are two solutions with the simplest being to disable the cache completely. The effect of the cache has not been quantified and it is possible that disabling it will have negligible effect on performance, particularly if the database is not the rate limiting factor.

The second solution is to distribute or replicate the cache. This would require a change of caching library as the current caching library has reached end of life. It would not be expected that replacement would be problematic and caching implementations such as Ehcache have quite similar interfaces.

Finally, it is worth noting that ideally the custom caching be removed and Hibernate second level caching used in a distributed or replicated manner see the section on Hibernate for more information.

4.3.4.3 Files

As mentioned in files storage section uploaded files are kept in the Tomcat deploy directory. This presents an issue in a clustered environment in that one Tomcat would not be able to read the files

on another machine. The simplest solution to this is to have a shared file system which all the Tomcats in the can access. See the section on shared files system in the rebuild section for more information.

4.3.5 Database scaling

There are several ways in which the database can be scaled to increase load, with differing amounts of development effort and implications for each.

4.3.5.1 Optimising queries

Optimising queries run against the database will yield improvements in database performance. Postgres has many build in tools to help with the identification of problem queries such as explain plans and logging of long running queries. Note that significant work has already been taken to optimise some of the queries, particularly those that populate the AAG.

4.3.5.2 Database tuning

Postgres has a many configuration options and tuning these to the specific requirements of the application will improve performance.

4.3.5.3 Read only databases

There are many different solutions under the heading read only databases. As see different replication solutions (not all are read only) for some examples. The central premise is:

- There is a single master database which read / write queries can be sent to
- There are one or more slave databases which are kept in line with the master. Depending on the setup this maybe:
    - Synchronous. Less performant but the data is always up to date
    - Asynchronous. More performant but the data may lag the master
- The slaves can accept read, but not write, queries

Therefore, all the read queries can be distributed to the slaves taking load of the master. This solution works best when the application in general performs few writes, and many reads. While uFlexi does perform writes, it is expected that the majority of database activity would still be reads, and so this solution would work well.

With either a synchronous, or asynchronous approach, some development effort would be required to split out the read queries from the write queries, and ensure that they are sent to the correct database. Also, with both approaches the database should not get into an inconsistent state, as there is only one where writes are allowed. As such this is conceptually significantly easier than a clustered setup where writing is permitted to any node.

If an asynchronous approach is taken to replication then the application will have to be changed to deal with situations where the slave does not have the most up-to-date information. For example, if a seller is already booked, but this data is not yet in a slave database, and another booker tries to book them. While this type of problem is possible, and will need to be coded for, it is unlikely it will

be a common occurrence, as even though the communication is asynchronous it should still be quick.

4.3.5.4 Separating out databases

If there are any logical divisions of groups of tables in the database which have minimal interactions it should be possible to split these out into their own databases. This would require development effort and the difficulty would be determined by how tightly bound the entities are in the code. It would however provide an increase in performance and an ability to scale bits of the application independently. Note that it is envisaged that the Agency functionality would be a suitable candidate to hive out into a separate database.

4.3.5.5 Clustering and sharding

There are several products, commercially and open source, that offer the ability to cluster databases instances by sharding the data. Examples are Postgres-XL and Citus. However, these solutions introduce complexity and would require significant development work to alter the application so that it would work well with sharded data.

5 Accessibility

The application was audited for accessibility several years ago. At that time, it achieved Compliance to WCAG 2.0 Level A. The application has evolved since then, however good principled have been maintained e.g. providing alternative text etc. However, without an audit of the site it is not possible to ascertain what the current standard of the site would be.

Accessibility work will be determined by frontend changes provided by *Torchbox*.

6 Testing plan

6.1 Performance Testing

There is no current performance test suite for the application. Performance testing has been undertaken in the past. Discussion on future performance testing is can be found under the scalability plan part of this document in the section determine current capacity.

6.2 Functional Testing

6.2.1 Current state of testing

There are currently both unit tests and integration tests, with the integration tests connecting to a test database. The test data for the integration tests is generated using DbUnit to read a series of XML files containing test data into the database.

All the tests can be run as a suite from the command line, but the output produced is not in convenient human readable summaries. As such the command line interface is best suited to automated builds. Current development practice is to run the tests in the developer's IDE which provides well formatted output.

In addition to the tests, metrics are available using EMMA to determine how much of the codebase the tests exercise. These metrics give an indication of where further testing effort would be beneficial. Taken over a period they can be used to monitor coverage levels, and ensure that testing is not forgotten as new functionality is developed.

There are approximately 1400 unit and integration tests, and they cover just under half of the classes in the application. However, there has been considerable thought in which areas of the application are tested, with the more complex areas given preferential treatment. Much of the code not tested is boiler-plate, where development of new tests would not be especially productive. Attached is a recent code coverage report, please note that the this is just the front page of the report, links to Java classes will not work as they reference the codebase which is not attached.

coverage.html

6.2.2 Unit test improvements

As stated above, the application has a reasonable number of unit tests. However, it would be beneficial to add more where code coverage is limited. Additionally, there are a few other actions which would improve the value of the tests. It would be beneficial to have them run as part of the build cycle, both for developers, and build agents. See section on continuous integration for additional information.

Currently running the test suite is not an enforced part of the build cycle, though there is a build agent, Jenkins, that does run the tests routinely and reports of failure. However, having the tests baked into the build process would be a significant improvement. Part of achieving this is to move away from the Ant custom build tool and move to a lifecycle-focused build automation tool, such as Maven or Gradle. More details can be found on these in the section on build tools. These tools have conventions for building which included failing builds if the tests fail.

Additionally, it would be useful to have human readable test results artefacts from the command line. This would mean build agents could display the current state of test results in a browser for all to see. Thus, any failing test will be picked up immediately.

6.2.3 Integration test improvements

In addition to the unit test improvements which are also true for the integration tests there is merit in changing the approach to how test data is generated.

6.2.3.1 Generating test data

The current the method of using DbUnit and a series for scripts works, but does present some issues. Perhaps the most notable of these is that when the database schema changes all the XML files must be updated. This can be less than straightforward when the changes are large, for example if a table were to be split up. Moreover, because the files are all xml there is no immediate feedback to the developer that the tests will break, i.e. there is no help from the compiler. Another issue is that because the data is being pushed directly into the database and bypassing the application, then inconsistent data can be inserted as business rules in the application are bypassed.

A better approach to generating test data would be to use the application itself to insert the data. This requires some initial effort to generate a framework to communicate directly with the application services, and in some instances, may require changes to application signatures. For example, it may be necessary to add a time field to a service method call to allow the simulating of historic data.

This method of inserting data has the advantages that it results in consistent records, because it has been through the application validation rules. As it has no direct interface with the database the database can be refactored with impunity, and when refactoring the services, the developer's IDE will be able to immediately identify where changes are required. The disadvantages are that it can take longer to generate the data, though this can be mitigated by using the mechanism to generate database dumps which can then be reused. It is also sometimes not possible to get all the data in via the services. This problem however can be circumvented by calling the repositories directly in these few cases.

6.2.4 Acceptance tests

There are currently no acceptance tests for the uFlexi application but there would be significant value on implementing a suite of them.

Acceptance tests are generally quite high-level and written in a human readable form, so reading them will make sense to business users, though they require technical knowledge to implement. Often, they will follow user stories and business requirements. The following is an example test:

```
As a seller I can log into the application
```

Which is further broken down into:

```
Given a user navigates to the url http://example.com/login
When the user enters user name test@example.com and password password
Then the user will see the text Welcome Mr Test
```

Acceptance tests are generally run against an instance of the running application. With a web application, the test tool will take control of a web browser, navigate around the application as a user would, and check that the behaviour is as expected.

This type of test is useful for a several reasons. They are a useful source of both documentation, and demonstrate that the application conforms to specifications. They show that the application is working when all its components are combined. Perhaps more than any other tests they provide security when refactoring that the code has not caused regression issues as they interact with the user interface only. Once the initial framework is in place they can be written by people with limited technical knowledge. Finally, with web applications they can be used to test the application with different browsers.

The main disadvantage of acceptance tests is that they can take a significant amount of time to initialise and run. As a result, they will often not be run as part of the normal development cycle. This can be mitigated by having a build agent that runs them periodically, and so won't slow development. Additionally, having a method of initial base data inserting into the database, such as that discussed in the generating test data section, will speed them up.

There are several frameworks designed with this type of testing in mind, but with web testing, the final interface with the browser is usually via WebDriver. There are a multitude of tools for the writing of the tests themselves, choice of which depends to an extent on the expertise of the development and test team. These include Substeps, Robot, Cucumber, or simply writing Java code that directly calls the WebDriver.

6.2.5 Manual testing

While automated tests are both extremely useful and efficient, they have not obsoleted the requirement for manual testing. There are issues that automation will not pick up, for example styling issues where only a human eye will see error. Thus, it is useful to have scripts that outline a journey through the site. However, reliance on too much manual testing is not a good thing as it will always be time-consuming and expensive, and so should be kept to an acceptable minimum.

6.3 Security Tests

There are currently no specific security tests suites. However, it should be noted that common security threats such as SQL injection attacks are, to an extent mitigated, by choice of frameworks. For example, the much of the access to the database is via Hibernate and this framework will always escape input before calling the database as a *prepared statement*, thus prohibiting SQL injection attacks. Elsewhere in the code, prepared statements are used directly, and where parameters are provided these will be escaped. As another example, the dangers of cross site scripting are reduced by using the c:out JSTL tags in JSPs which escape the output. This notwithstanding there is merit in explicitly testing for security issues. This can be manual testing, by developers, testers or a dedicated third party. It can be automated such as writing unit tests to check for exploits or by using tools such as OWASP ZAP.

6.3.1 OWASP ZAP

OWASP ZAP is an automated tool to help find security issues in web applications. It works by proxying traffic through a browser so that it can get the layout of the application. It then analyses this for security vulnerabilities and performs attacks to see if there are any security holes. OWASP ZAP can get an idea of the layout of the application in several ways. Firstly, it can be pointed at a starting URL and it can spider from there. Doing this however will most likely miss pages behind logins. Secondly a user with it installed can manually navigate around the site, this will open more of the application to testing and OWASP ZAP be able to view pages past login. Finally, the best solution is to have it proxy acceptance tests, see the section on acceptance tests for more information on these. This way the entire process can be automated and part of the build cycle and help to identify any new security holes. It can be time intensive to run and so it is the sort of task it makes sense to run on a build agent, as an overnight task for example.

6.3.2 OWASP dependency checker

The OWASP dependency checker is a project that is used to determine whether any of the libraries that a web site is built upon have security vulnerabilities and need to be upgraded. This will be easier to implement when the code base has been standardised. See the section on source code layout for additional details. As with OWASP ZAP this would ideally be part of the build cycle so that it is immediately apparent if a new dependency is introduced with a known vulnerability, or a new vulnerability is found in an existing dependency.

6.3.3 Code audit

Code audit for security holes, by an external agency, could be beneficial. Particularly if the plan to open source the code comes to fruition. See the open sourcing plan part of this document. However, the timing to gain most benefit is important. There would be no point in auditing code that is about to be refactored. Thus audit, after most of changes have been made would make sense. Time would of course, still be required to correct any issues that are uncovered.

7 API

7.1 Current APIs

Currently some of the functionality is exposed via an API for the consumption of application JavaScript. I.e. AJAX to transfer data to pages that use JavaScript to dynamically load content. This is not for most of the application, but for places where the use cases have given the greatest benefits. An example of this is when a booking is made. The aggregated availability grid (AAG) is loaded using AJAX to return a JSON representation of the availability. Taking this as an example:

The browser does a POST request to:

`/agencyPurchaseStart.do`

with POST data:

```
abuyerId: 1
location: 2
etc.
```

The server then returns JSON representing the availability:

```
"message":"Success",
   "body":{
     "1503702000000":0,
      ...
     "1504303200000":0
   },
   "status":"ok"
}
```

Note that these have been made available on an ad hoc basis where the frontend JavaScript required it. Additionally, there is no login API exposed. As such they will only work in the browser environment where the user has already logged in and so has a session. Without this they will not work as a API to third parties. Finally, they do not conform to the REST standard, and while this does not impact on their ability to function, it is something that would ideally be changed. See the section on REST for more information.

7.2 Consuming or providing for third-party APIs

The system may have to interface with third party systems with APIs yet unknown. They maybe legacy and that their messaging structure immediately compatible with the uFlexi systems. Nevertheless, it should be possible to integrate these, though each case will have to be taken on its own merits and the following should be considered:

7.2.1 Standard, language agnostic, format

This would be the easiest case. For example, if the API were REST with JSON over HTTP this would simple to integrate with and would be directly implementable in the code base itself. Adding endpoints in the application would be straightforward, though it would require some development effort. Another example would be SOAP messaging. There are examples of both in the application as it currently stands, see sections on SMS gateway and Current available APIs.

7.2.2 Standard format but not agnostic to language

This may be harder. For example, RMI where there is direct language feature messaging. Thus, it may not be advisable to hook directly into the application. In this scenario, there is a case for writing a module that translates these messages to and from agnostic formats. Using a tool like to sit in front of the application to provide this translation layer would be a reasonable design. See the section on adapter interface to see why this is the preferred approach.

7.2.3 Proprietary format

This would not be possible without help from those responsible for the systems to integrate with. They would need to provide a standard format for us to work with or in the case where it is over a standard network protocol giving us details of the syntax of messages.

7.3 Adapter interface

For any of the above scenarios it is suggested that a translation layer between the application and the third-party would be beneficial. The advantages this gives are:

- Independently deployable. Useful in the case where adding a new endpoint and not wanting to take the application down.
- Independent development. This is only true if there are no changes required of the underlying application. For example, if a new format of message needs to be supported but the underlying calls remain the same.
- No need to pollute the existing code base with different messaging formats. We could then stick with one format e.g. HTTP and JSON within the main application.

7.4 Work to be done when integrating with a third party

The following tasks would need to be completed when integrating with a third party:

- Liaising with third party suppliers.
- Setup a translation layer for the various environments. This is onetime only task.
- Develop code to run in the translation layer.
- Develop code to perform the relevant actions, required by the API with associated REST endpoints, in the main application. This may exist already if it is a call another third-party previously needed.
- Deployment process for the new code to be run in the translation layer. It is envisioned that this would become quick once it has been done a few times.

- Integration testing and stubbing out of endpoints for development environments.

7.5 REST

The proceeding sections discusses issues relating to interfacing third party APIs. It is worth discussing the approach the application should take when designing calls to be consumed when there are no constraints on format or protocol. I.e. the preferred design for future communication.

The section on the adapter interface describes away of exposing endpoints. The preferred architecture of these is Representational state transfer (REST) with JSON over HTTP as the message format. REST does not have a formal definition, though there are some rules to abide by, but provides a series of constraints, such as being stateless.

When applied to web applications gives:

- Resources exposed via URLs. For example, `http://example.com/users/1` would reference to user with id `1` in the system
- HTTP protocol indicates intent
    - If the user example above was sent to the server with the HTTP method DELETE, then the message is to delete user with id `1`
- Data is transferred as a representation
    - In the user example if the HTTP method was GET then the server could return a JSON object representation of the user, for example:
    - `{“id”: 1, “name”: “richard”, “email”: “richard@example.com”}`
- Stateless. There is no state persisted between requests and so each request is completely descriptive

The advantages with this design include:

- REST calls are self-descriptive. This makes them easier to rationalise about and navigate. Indeed, if the HATEOAS constraint is used then all further actions a client takes are linked to from the original returned result.
- It is popular and most developers will have had exposure.
- It results in a clean design that is easy to maintain.
- There exist tools to help document the resultant API for third party consumption. The recommended method here is to incorporate the documentation generation as part of the application testing, and have the output as part of the build pipeline. See the section on continuous integration for more information. This ensures that the documentation really does conform to the actual endpoints. The documentation never gets stale and the endpoints can be visualised and, depending on tooling, interacted with directly.

REST with JSON over HTTP would be the preferred method of communicating with other applications. This would include mobile phone apps, as well as JavaScript for the application running in the browser.

7.5.1 Ascii docs

It is worth covering the documenting aspect of API calls in more detail. This is the document that third-parties will be relying on and so it is important that it is accurate and up-to-date. Historically it is one of the parts of application development that becomes obsoleted quickly. This is because if the documentation is standalone there is nothing to force changes in it when the codebase is altered.

The solution to this is to have the generation of the documentation tied to the build of the application itself. Thus, if the code is changed, but the documentation is not the build will fail and the developer will be informed.

The following is recommended:

- Write unit tests which use Spring REST Docs
    - Tests are written which fire requests at the mocked-out application.
    - While mocked out it still has the endpoint interface of the real application.
    - The mock implementation is provided as part of the Spring framework.
    - If the mock interface is different from the expectations in the tests then the tests will fail, which ensures documentation does not get out of date.
    - The tests write out documentation snippets when they run.
- The snippets generated by running the tests are aggregated into an overarching document of all the REST end points
    - The default output from Spring REST Docs is the format used by Asciidoctor.
    - This is processed to generate HTML which can be viewed directly in a browser
    - This HMTL artefact is the documentation which can then be distributed to third-parties.

While the above approach prevents documentation getting stale, at an interface level, it does not prevent the documentation not being written in the first place. To enforce this business process is perhaps the only way, with lack of documentation being picked up in pull requests.

7.6 OAuth 2.0

OAuth is a mechanism by which users can agree to let one website use their data from another's.

Work has already been done on the application which uses the OAuth to obtain, with user permission, information from Facebook. In principle, it should be relatively straight forward to extend this to other providers.

As well as user data OAuth can be used as the basis of login, so that for example, a user could authenticate against Facebook to gain access to the uFlexi system.

8 Open sourcing plan

This part of the document has much in common with the rebuild section. To save duplication, future improvements to the application have only been listed in the section where they are most relevant. That said many of the suggestions for one section are equally valid in the other and there is a reasonable amount of cross reference.

8.1 Open sourcing goals

To be successful, as an open source project, developers should to be able to setup a productive environment as quickly as possible, with a minimum of effort. Ideally, they would be able to clone the code repository, and then stand up a working environment with a single command.

The application should be independent of:

- Operating system. It should work on Linux, Mac OS, or Windows.
- Development environment. For example, it should work equally as well in Eclipse as in IntelliJ IDEA.

It should have minimal installation requirements, for example:

- Java 8
- Bash
- Libraries downloaded from external repositories
- Docker

It should be productive, allowing developers to make hot code changes, and debug within the IDE of their choice.

The current source code includes all required external libraries. Whilst this is initially convenient is does mean that:

- The repository is large.
- The repository contains third party code under various opensource licences.
- It is not easy to identify what the dependencies are between components.
- It is difficult to upgrade to new versions of libraries where there are multiple dependencies.

There are several tools for Java which support dependency management, based around the same concept of a repository: Apache Maven, Apache Ivy, Gradle.

In the past year, tool support for Gradle has reached a point where it can be recommended over Maven.

8.2 Source code layout

The application build tool is currently Ant. This has the advantage of being flexible, and in the past, it allowed the file structure of projects to be highly configurable. However, as time has moved on a

standard project layout has been settled on. Using this standard structure provides many tooling benefits and the application should be migrated to.

8.3 Build tool

The current build tool is Ant. This is flexible, but verbose, and provides limited functionality without developer effort. More modern build tools such as Maven and Gradle value *convention over configuration*. If you conform to the defined convention for build structure there is a lot of functionality for free. The section source code layout adds a few more details while the section on unit test improvements give an example of free functionality.

Another big advantage of Maven and Gradle is that, unlike Ant, they manage library dependencies. The current codebase has dependencies committed to version control and this sub optimal and less than ideal. See the section on open source goals for the reasoning. With Gradle and Maven you simply specify the dependencies in a configuration file and they are pulled down from an external repository, and cached locally, when required.

8.4 Source control

The application is currently in a SVN repository. However, it would be better to migrate this to Git. The section on version control, in the rebuild part of this document, covers this in more detail. However, in brief, Git has become the de facto standard in recent years with high uptake and a multitude of tools. Additionally, there are many services, such as GitHub, which would be the ideal location to host the open sourced code base.

8.5 Standard data set

There is no standard set of data run the application against. Currently developers have their own data sets, which they have maintained over time. If the code is open sourced there will be a requirement to have a standard set of data, so that any developer can spin up the application. The section on generating test data gives a proposed solution to generate this data.

8.6 Purging secret information and configuration

The codebase currently contains privileged information, for example the username and password to connect the external SMS gateway. See the section on current build process for more information on why this is the case.

This would need to be removed to open source the project. To remove it would be necessary to change the way the application builds and runs. Build artefacts would need to be environment independent, and all the environment specific properties be read from the environment itself. One way to achieve this is to have a properties file, external to the application, which contains all secret information, and other specific configuration. This has two advantages. Firstly, the build artefacts are identical across all environments, so testing on the QA environment is testing the same binaries that will be deployed to production. Secondly it means that secrets are not stored in the code base which is not a good practice, and essential if the code is open sourced.

See section on container configuration for more information on how environment agnostic artefacts can be achieved in a containerised environment.

8.7 Hashing

The application does not save passwords in the database but instead stores hashes. See the section on security for more details. This is significantly better, from a security standpoint, than storing the passwords in plaintext however the hashing does not use a salt and more secure functions now exist. For this reason, it is recommended to change to using the BCrypt. This has the following advantages:

- Uses an automatically generated random salt by default, which renders reverse look up of passwords effectively impossible.
- It is part of Spring Security and is the default password hashing mechanism in Spring Boot.
- BCrypt has configurable computational complexity i.e. the number of calculations required to verify a hash can be configured, and by extension the amount of time required. This protects against brute force attacks. As hardware becomes faster it the number of calculations can be increased to take this into account.
- The hash, salt, and complexity are encoded as a single string, making password storage straightforward.

8.8 Stubbed external services

Both email and SMS service endpoint would need to be stubbed out so that developers can spin up the application independently of third party test services, which they will not have access too.

8.9 Documentation

Well there is an internal wiki with information on how to setup a local environment it is not open to the internet and needs updating. Once updated this documentation would need to be available to any developer wishing to work on the project. Assuming GitHub, then the built-in wiki functionality could be used for this. Note that it would be hoped, because of the other changes in this section, that the setup documentation be short.

9 Mobile

9.1 Current Mobile Experience

The uFlexi application is standard web application, leveraging the Bootstrap 2 user interface framework. It provides a responsive experience when phones and tablets. Attention has been paid to key user journeys, and screens to make them more mobile friendly. Notwithstanding, the application was originally designed for desktop, and as such there are there are known usability issues:

- Grids require a tap per cell, where touch and dragging functionality would be better.
- Help text pop-ups require a mouse hover over.
- Some fields break their margins of the viewing area.
- When using a date picker, the keyboard pops up obscuring it.
- Phone numbers are not clickable to call.
- Some bugs – e.g. Profiles can fail to load.

While none of the issues above would prevent a user from using the application on a phone, the experience is not optimal, and changes to address this are to be proposed.

9.2 Web vs Native

Broadly speaking there are two solutions to making functionality available to users on mobile devices: Firstly, make the website mobile responsive, and secondly write a native app. These two approaches are not mutually exclusive, and making the main website more mobile friendly, as first step, may pay dividends in flushing out some of the design choices for the native app later.

9.2.1 Web

Broadly speaking to make the existing website responsive, and not assuming a complete rewrite, then an incremental process of working on the existing screens, with a design team, to make them work well with mobile devices would be required. Pages which are deemed most critical would be dealt with first, though it should be noted that it would be worth doing whole journeys, to avoid a jarring user experience.

It is recommended that the HTML for each page is not specifically for mobile or desktop, i.e. that there is no switching based on device. This way the user experience will remain consistent and the code itself will be easier to maintain.

9.2.2 Native

If the website is accessible then it may seem like the is less of an incentive to have a native app, and this can sometimes be the case; many users now just assume that websites will work seamlessly on mobiles. However, there are still compelling reasons to have a native application:

- Where the application is going to be used where there is limited, or no internet access. This may well be the case here and sellers may well have limited access when wanting to access uFlexi.
- If internet connection bandwidth is limited. As with the previous point on limited access, but a native app can be much more reactive with a given bandwidth, as all that is being sent is pure data. There is no need to send HTML or JavaScript. Again, it is envisaged that users may well need to use the uFlexi in places with poor mobile coverage.
- If specific functions of the device are used. It is possible to interface with some of a devices functions directly from a web application for example geolocation. However, if tighter integration is required or other features of the phone needed then a native app is required.
- If it's the sort of application with users will use regularly, which uFlexi is, then having an easily accessible app from the home screen is convenient.
- Use of background processing (e.g. for geofencing, and real-time geolocation). For example we could track a Sellers location to match against jobs, and automatically clock-in/out.
- Use of platform security features, e.g. Apple's Touch ID for authentication, and payment
- Integration with other devices, such as smartwatches.

9.2.2.1 Cross platform or targeted

Native apps can be specifically targeted at a device e.g. iOS, or they can be written in a framework that enables them to work cross platform. Both approaches have advantages and disadvantages. Targeting a specific platform allows for the maximum integration with devices, and will most likely run faster and more efficiently. For example, with iOS devices the de-facto language is Swift, while for Android it's Java, though alternatives exist with both ecosystems.

The main disadvantage with the approach of an app per-platform is that it requires much more development effort to write initially, but also to maintain. While there will be conceptual designs that can be used across the applications, and server backend communications will be largely the same, it is nevertheless required to write and maintain a codebase per-platform. Also, programmers will often not have skills in all the required languages and frameworks.

The other approach is to use a framework which allows code to be written in a single language, and then run on multiple platforms. The technical mechanisms to enable this used by the frameworks differ, which does have implications, but the end goal is to have a single codebase. Most of these frameworks use JavaScript as the common programming language which has wide developer uptake.

There are a few disadvantages with these frameworks, firstly they relatively new and so there are likely still issues to be ironed out. There has not been, and maybe never will be, an outright winner declared, so deciding which to use isn't entirely straight forward. When new APIs are introduced then they will take a little time to be incorporated, though in practice this is unlikely to be an issue. They tend to only officially support Android and iOS, so targeting other platforms may not be possible or might be experimental. The reverse of this is that, if they do start to support another platform then the application will automatically be ready to run on it with no, or little effort. Finally, while the business logic of the app will be the same it is sometimes necessary to have some UI logic per-platform.

With these caveats, a framework such as React Native or NativeScript is recommended over distinct codebases. Currently NativeScript has the edge in maturity but, as stated before, this is a relatively new arena, and a review at the time of development would be wise.

9.2.2.2 Other considerations

Like the main code base there are number of practices that should be adopted to ensure quality and maintainability. These include:

- Tests: Unit tests as a minimum, but also acceptance tests. These have been historically harder to do than acceptance tests for web-based applications as there has been less means and a wider range of devices. That notwithstanding there are tools, such as NativeScript's Functional Tests, which now do this. Functionality for testing should be taken into consideration when choosing a platform.
- Build pipeline: Again, it has not been historically as easy to incorporate mobile app development into a build pipeline. See the continuous integration section for more information on build tools. There are some plugins to help with this, for example NativeScript's Jenkins plugin. While these are not fully featured yet, complete automation will still be possible by simply getting Jenkins to run Bash scripts where necessary.

9.2.2.3 Changes required of the application

Any native apps will require changes to the main server-based application. This is because, whatever its guise, it will still need to access this source of data. Currently the application does not expose any endpoints. The section on API discusses some of the issues and technicalities of implementing endpoints for the consumption of other applications. However, it is recommended that the output from the server application is REST with JSON over HTTP.

Irrespective of the scope of the native app, at the very least it will be necessary to add endpoints for logging in an obtaining a session token. After this point, the number of endpoints is dictated by the required level of functionality.

9.2.3 Scope of mobile

The scope of work to be done for making the site more mobile accessible has been split into levels of refinement.

9.2.3.1 Fix issues with the application on an ad hoc basis as required

Here the application would fix issues from the list in the current section as required. This would make the website friendlier to mobile users, but does not include any work for a specific native application.

9.2.3.2 Key journeys only

This is the first part where a native application is recommended. The functionality to be included are the user journeys that are most useful and frequently used. These are:

User	Journeys to be fully Mobile enhanced
Agency (Labour market intermediary)	a) View latest bookings (Bookings accordion) b) Make booking (agencyPurchaseStart, includes the AAG and manual entry of booking dates/times) c) View overdue timesheets (Timesheets accordion)
Buyer (Employer)	d) Make booking (buyerPurchaseStart, includes AAG and manual entry of booking dates/times) e) View progress of bookings (Bookings accordion then buyerPurchaseDetails) f) Approve timesheet (Timesheets accordion then buyerInvoicingViewTimesheet)
Seller (Worker)	g) Change Availability (grid in Availability accordion) h) Accept booking (Booking accordion then sellerJobView)

Before starting this work, there would be merit in having user experience wireframe the various journeys through the application to determine the best experience. That notwithstanding there is not a vast number of journeys in this scope and logic will to an extent dictate design.

9.2.3.3 Key and secondary journeys

In this scope, all the work from the key journeys is added to by adding some secondary functionality:

User	Further journeys to be fully Mobile enhanced
Agency (Labour market intermediary)	a) Manage/Approve Buyer (Buyer accordion then agencyBuyerCoreDetailsEdit) b) Manage/Approve Seller (Sellers accordion then agencySellerDetailsEdit) c) Cancel booking (pop up on agencyHome) – NB: currently there are problems with this journey d) View profiles for Buyer/Seller/Role/Check
Buyer (Employer)	e) Registration f) In timesheet change hours and rating plus give feedback (where enabled) g) Rate workers (the current page – buyerStarRatingCostCentre – will be substantially amended to create an *"Our Workers"* section in autumn) h) View profiles for Seller/Role/Check
Seller (Worker)	i) Registration j) Manage Roles (sellerBuyersRoles) k) Manage checks (sellerVettings) l) Manage Buyers (sellerBuyers) m) Change Terms (sellerRatesAndLimitsEdit)

	n) Show bookings history (sellerJobHistoryView) eg at a job interview o) View profiles for Buyer/Role/Check

There is a wider range of journeys here and user experience input would be very valuable in determining the best design from a frontend perspective.

9.2.3.4 All application functionality

The most comprehensive mobile application would allow all the functionality of the website to be accessed via a native app. Again, user experience input would be invaluable, it is likely that we would be turning back office functionality into mobile, which will be tricky to get right from a user interaction point of view.

9.2.3.5 Specific to mobile functionality

Given mobile devices have more information available than desktop machines or laptops it should be possible to enhance the native app over that of the web site by utilising these. For example, mobile devices have location data and contacts list which could enable functionality like:

- Auto clocking in/out based on location
- Move my base postcode
- Travel alerts
- Flash branches
- Location streaming

10 Technology upgrade

Note that this section has much in common with the open source section. However, there are many additional changes that would be made in addition to the open source strategy.

10.1 Coding framework changes

10.1.1 Spring Boot

As outlined above the application is uses Spring as the container to run in and the Struts framework to do request routing and rendering of pages. Struts is now an old framework and is no longer maintained, and the version of Spring is a few behind the current release. Perhaps the best upgrade path would be to Spring Boot. This effectively amalgamates many useful parts of the Spring libraries and would bring the application up-to-date. This would however not be an insignificant undertaking, with much of the work required being replacing the Struts actions with the Spring MVC equivalents. That notwithstanding it should not require any functional changes to the application.

Spring Boot provides another advantage in that it can run independently of an external Tomcat server which makes it lightweight and a reasonable starting point for microservices if this is a direction taken in the future.

To elaborate on the technical changes that would result or be part of moving to Spring Boot:

10.1.2 Struts 1 to Spring MVC

As discussed earlier the Struts 1 is a deprecated technology and Spring MVC would is a good replacement. It has high uptake and most java web developers will be familiar. It has a clean separation of concerns, is easy to work with and provides much functionality for free. As well as generating HTML it is a valuable framework for writing REST API endpoints.

10.1.3 JSP to Thymeleaf

The current templating engine is JSP and this can work with Spring Boot. A better alternative though is Thymeleaf. The advantage of Thymeleaf is that the templates are valid HTML and do not need a server to render them. Without the server, the templates can still be viewed in a browser, with default values for dynamic content. This means that designers can edit them directly without the need to spin up an environment. There is also more of a one to one correspondence with the final output so it is easier for developers to convert vanilla HTML provide by designers into working templates. It is widely used and many developers have had exposure.

10.1.4 XML to annotations

The application currently uses XML for configuration. Moving over to annotation-based configuration would reduce the amount of metadata, add an element of type safety and help with refactoring.

10.1.5 Hibernate

The application originally communicated with the database using Spring templates and prepared statements. New code has used Hibernate to persist to data. Going forward the legacy prepared statement code should be replaced and the application should use Hibernate across the board. This will have several advantages:

1. The code will be consistent
2. The code will be easier to maintain
3. The possibility of security issues such as SQL injection will be removed
4. Developers in general are more familiar with Hibernate and so more productive.
5. Better, generic caching.

It is worth discussing the caching in a bit more detail, as the application has a custom caching mechanism now. While this works care must be taken to ensure that caches are invalidated when data becomes stale, which is complex and prone to error. If Hibernate was used throughout, then second level caching could be introduced. This would be a preferable approach and not require any manual coding from the developers and so less prone to error. With a second level cache reads from the database are maintained, in the form of hydrated java entities, over multiple requests. This is unlike first level which is much more transitory and will generally last only for part of a single request. Depending on the underlying implementation, the cache can be distributed, which is required in environments with multiple servers. See the section on horizontal scaling for more information on multiple server environments.

To migrate a phased approach might be taken:

- Make sure the entities in the system have hibernate annotation, irrespective of whether Hibernate is yet responsible for their persistence.
- Create SpringData repositories for all the CRUD operations and where possible more intricate service specific queries. Note reporting might not be able to be moved seamlessly to SpringData interface naming convention. Specific JPQL may be required in places.
- It may still necessary to use native SQL through the Hibernate interface for some of the more complex reporting queries. Ideally the report would be hived off in its own discrete piece of functionality.
- Purge all the JDBC Spring template functionality.

10.1.6 Tomcat 8.0

Currently only Tomcat 7 is supported by the application. However, the latest incarnation is Tomcat 8 and it would be advisable to move to this[5][6].

10.2 Technical debt

While the application is having its dependencies updated and changes made to accommodate these, it is appropriate time to clean the code base. Overtime, code aggregates and needs to be rationalised, new features and better ways of programming are added to a language and code becomes obsolete etc. As examples of areas of the code that could be cleaned:

- Package rename, to remove to references and *torchbox* in the names and replace with uflexi
- Removal of dead code (code which is no longer used)
- Use new language features where available e.g.
    - Using the Lambdas in Java 8 for a more functional programming approach
    - Using Enums for constants instead of Integers
- Having a view models per a view, instead of adding arbitrary data to the HttpServletRequest object.
- General refactor

10.3 Shared file system

Files are currently stored in the tomcat deploy directory with the running application. The section file storage details this and reasons why this is less than optimal. There are several solutions to this issue but the easiest would simply be to have a network storage file system. The location of this would then be set in environmental properties files. See purging secret information and configuration for details on configuration.

10.4 Continuous integration

Whatever the end design of the application there will be a large benefit from having a continuous integration build environment. In simple terms, these are automated machines which are connected to the code repository, used to build, test and deploy the code to various environments. The exact process of what happens varies depending on exactly what is required but the following steps are usually taken:

1. Work is committed to the code repository.
2. The build server sees that there has been a change and checks out the new code.
3. The code is built, producing a build artefact.
4. Tests are run.
5. The build artefact is stored in a repository.

[5] If the application were to be moved to Spring Boot it may not be necessary to have a standalone web server. This is because Spring Boot applications can run independently without the need for an external server[6].
[6] The server is, in fact, Tomcat, but this dependency is managed by the framework itself.

6. Later a user will access the build server from the web front end and can deploy the artefacts to a given environment. This step can be automatic or scheduled depending on the environment and the appetite for continuous deployment, see the section on build pipeline. For example, it may be convenient to deploy the new code to the QA server every evening after work.

Build tools have many advantages over ad hoc deployment:

1. There is an immediate overview of the status of the project. Any failing tests will be highlighted and can be dealt with immediately.
2. The tools can be a much more secure way of doing deploys, individual developers don't have to be allowed onto production servers, permissions can be restricted to only what is required.
3. The risk of deploy steps being missed, or the knowledge of some archaic process lost to time. The build tool, by its very nature, is the documentation as well as the machinery to perform deploys. Because it is automated it should also be repeatable.
4. Deploys can be done out-of-hours.

Build tools can perform many more tasks as part of the pipeline and while many of these require initial development effort they provide large savings even in the short term. Common pipeline tasks can include tagging and versioning releases, security checks, generating documentation, running acceptance tests, and more.

10.4.1 Suggested build pipeline

The following is a suggested outline of the set of continuous integration jobs that will work effectively with the branching strategy outlined in branching section. It should be noted that there is no definitive solution and different approaches can be taken, and additional builds added to support other environments.

What is outlined here is continuous integration, and could be described as continuous delivery. It is not continuous deployment where every change is automatically pushed out to production. This would require changes in the current architecture, and it is possible continuous deployment would not be appropriate for the business, irrespective of technical issues.

The processes implemented by the continuous integration build servers are split by branch:

Feature branch:

- Developer pushes to feature branch.
- *Feature* branch built.
- Unit and integration tests run.
- (Optional) Runs other tooling as required, EMMA code coverage, SonarQube. Can fail the build depending on outcomes.
- (Optional) Spins up an instance of the application and runs (a subset) of the acceptance tests against it.

Master branch:

- Merged to the master from a feature branch.

    - (Optional) Via a pull request, which will be denied if there isn't a clean build from the corresponding feature branch.
- *Master* branch built.
- Unit and integration tests run.
- (Optional) Spins up an instance of the application and runs (a subset) of acceptance tests against it.
- Stores the build artefacts in a repository for later deployment to other servers.
- (Optional) deploy to the QA servers.

Release branch build:

- Merge in manually from master when a new release is required, or a pull request is raised by a developer.
- *Release* branch built.
- Unit and integration tests run.
- (Optional) Spins up an instance of the application and runs (a subset) of acceptance tests against it.
- Stores the build artefacts for later deploy to production servers.

Release branch release:

- Tags and version the latest successful artefacts from the release branch.
- Deploy to the production environment.

Scheduled jobs:

- Deploys build artefacts of master to QA servers.
- (Optional) Run the full set of acceptance tests.

Optional stages depend on requirements and limitations of the tools available. Ideally all optional steps would be done, but they require development effort.

10.4.2 Continuous integration tools

In terms of the tools to use there are plenty of options, and it further analysis maybe worth doing. However, the following have a proven track record:

- Continuous integration server: Jenkins
- Git repository: GitHub
- Artefact repository: Artifactory
- Task, bug tracking: JIRA

Artifactory would not be the tool if the build artefacts are container images. In this case Docker Hub would be more appropriate.

10.5 Version control and branching strategy

There are numerous version control programs, but Git has become the de-facto standard in recent years. Reasons for its appeal include:

1. Distributed, development and branching can continue when not connected to the internet.
2. Fully featured.
3. Fast.
4. Free and open source.
5. Huge amount of tooling available.
6. Wide uptake; most developers will already have had exposure.

For these reasons Git is the logical choice of version control for our source control.

We recommend the following straightforward branching approach:

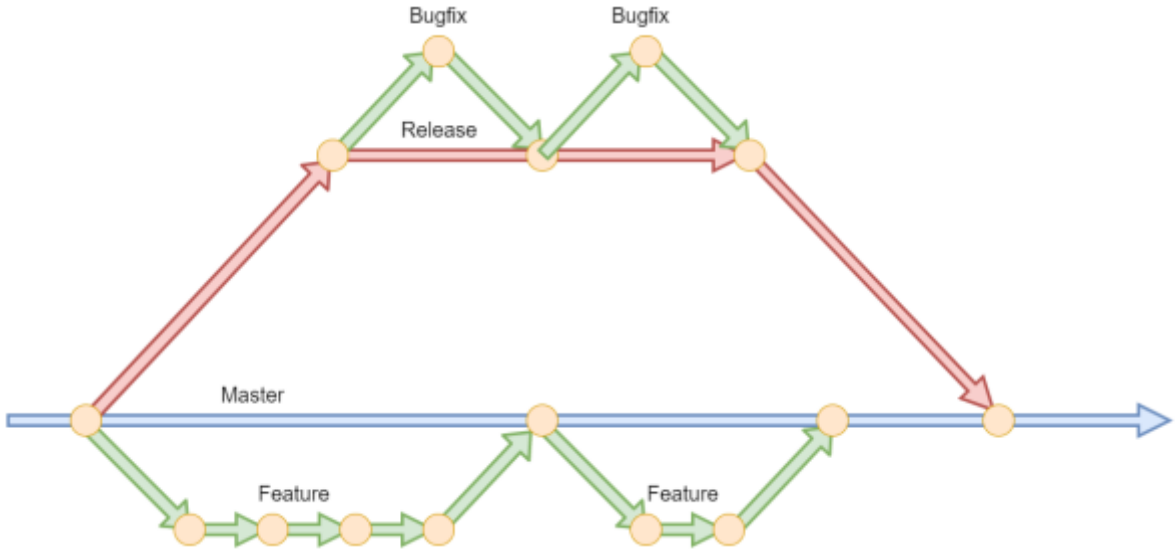

Figure 19 - Braching

1. Developers work on *Feature* branches from *Master*. Their work is merged back into Master when the feature is complete. Note that merging back into *Master* could be predicated on a pull request and a series of conditions being met to ensure that master can always build.
2. When a release is required a *Release* branch is taken from *Master.*
3. If work is need on the *Release* branch then developers can create *Bugfix* branches from here. In general, these would be minor changes.
4. When the *Release* branch has gone live it is tagged and merged back into *Master.*

10.6 Environment scripting

Historically the infrastructure for applications was setup manually e.g. by simply SSHing onto servers and executing Bash commands. While this approach is straight forward it has several disadvantages:

- When needing to generate another environment the entire process must be repeated.
- Documentation is often not kept up-to-date with the current state of the system.
- Knowledge about the system is lost over time as team members come and go.
- Random changes can be made which are undocumented.

The solution to this is to automate the building and running of environments. There are several tools which perform these functions including Chef, Puppet, and Ansible. Ansible is recommended in this instance as it is very straightforward to use and has wide adoption.

At its heart Ansible has playbooks which dictate how an environment should be setup and maintained. For example, it may say that a Tomcat instance should be installed with a given amount of memory, and environmental variables on a particular machine. On another it might dictate an instance of Postgres is installed with the PostGIS extensions. These scripts are then pointed at actual servers, or virtual machines, run and the environment is setup.

All that is required to run Ansible on the client machines is that they support SSH or WinRM protocols.

The Ansible language of playbooks is relatively straightforward and human readable.

Scripting the environmental with, for example Ansible, means that:

- The documentation does not go out of date. The documentation is the playbooks which set the environment.
- Generating another environment is much easier. Machines need to be provisioned and added to the playbook configuration, but then it is simply a question of running the playbook again.
- Arbitrary changes are not made and forgotten about. Changes are added to the playbooks and maintained on the servers by Ansible.

10.7 Anti-virus

There is currently no anti-virus scanning of user uploaded files. While this is not going to cause an issue on the servers themselves, it would nevertheless be better practice to scan uploaded files and quarantine any threats so that they cannot be disseminated to others. There are a few products available but we suggest clamav.

10.8Logging

The most useful logging for the application is output via Tomcat directly on the server. The only way to view it is to SSH to the relevant machine and look at it directly. This is less than ideal firstly it means that access to the production environment is required simply to look at logs. Second if running in a clustered environment then there will be multiple logs on multiple machines and finding relevant information becomes progressively harder. A tool such as Graylog which aggregates all the logs and makes them available via a web front end would be useful.

10.9Performance Monitoring

New Relic is a useful monitoring tool that provides and keeps logs of performance of a running application. It can be used with the live environment which is particularly useful as it gives shows how load is handed in the real world and can show which parts of the code should be optimised to gain the most benefit. At its heart, it records request through the application from the frontend web

server to the database and each of these calls can be broken down, showing for example which service methods are called the most, which take the longest to complete, when the server was under most load, how much memory it is using, which database queries take the longest to complete etc.

11 Internationalisation

11.1 Coordinate system

The current system uses a Cartesian grid of Eastings and Northings for localisation of addresses and performing distance calculations. This system works very well for relatively small geographical areas. However, because the world is spherical and not a flat plain, as the distances increase errors in calculations also increase. For this reason, a coordinate system for Eastings and Northings is often split into multiple different zones, each of which is considered to have an acceptable level of error. The projection used in the application for Eastings and Northings is Universal Transverse Mercator. As stated above this splits the world into multiple zones, and the for example the United States is split into 10 vertical strips. Within a strip, distance calculations will be out by no more than one percent.

The application assumes that all locations are represented in a single zones coordinates. This does not mean that addresses outside the zone cannot be entered. However, it does mean that as locations increase in distance from the designated reference coordinate zone, errors will become significant. Thus, if the application is just covering California then the current approach would work well. However, if the application was covering both Los Angeles and New York then this system of coordinates would not be appropriate.

If the second case is to be considered then probably the best solution would be to move over to Latitude and Longitude.

11.1.1 Latitude and longitude

Converting the application to represent locations as latitude and longitude would result in coordinates that are universal, as there is no need to compartmentalize the world into zones with this system. Additionally, calculating distances using latitude and longitude can give exact results. Moreover, many third-party APIs deal directly with latitude and longitude, and so this data could be used directly without transformation. For example, Google's geocoding API which the application uses to find the location for addresses returns latitude and longitude which then are converted.

There are Postgres modules which can handle latitude and longitude calculations directly.

11.1.1.1 Earthdistance

Earthdistance assumes a perfectly spherical world representation, which would be more than adequate in this scenario. It has two mechanisms for determining distance, Cube and Point. While Point is conceptually easier, dealing directly in longitude and latitude, it has two drawbacks. It's not possible to index and there are accuracy artefacts near the poles and the international date line. Cube does not suffer from these issues and so, while more complex, would be considered preferable. Below is a code snippet to give an example:

```
=# CREATE EXTENSION cube;
=# CREATE EXTENSION earthdistance;
=# CREATE TABLE locations (
      id         SERIAL NOT NULL PRIMARY KEY,
      name       VARCHAR(255) NOT NULL,
      lat        DECIMAL(11,8) NOT NULL,
      lng        DECIMAL(11,8) NOT NULL
    );
=# INSERT INTO locations (name, lat, lng)
          VALUES ('el capitan', 37.7339, 119.6377);
=# INSERT INTO locations (name, lat, lng)
          VALUES ('joshua tree', 34.1347, 116.3131);
-- Distances from San Francisco
=# SELECT name,
          earth_distance(ll_to_earth(37.7749, 122.4194),
                         ll_to_earth(lat, lng)) AS from_sf
     FROM locations;

    name     |     from_sf
-------------+-----------------
 el capitan  | 244863.040225844
 joshua tree | 683095.513495275
```

This is a simplistic example but demonstrates the earthdistance extensions. This approach may incur a performance penalty over using Cartesian coordinates. However, in the current application the SQL query that causes most performance issues has been analysed. It is used to populate the aggregated availability grid (AAG) and, while it involves distance calculations, has a data set that is significantly cut down first by other clauses first.

Note that when indexing longitude and latitude with earthdistance it is the boxing function, earth_box, that gains the benefits. This is a function that restricts results based on a distance from a specified point, a great circle. I.e. indexing works on query like "get all points less than 5km from San Francisco". Thus, it might be that it is not possible to harness the efficiency of the index if it is not possible to construct the required queries in this manner. Alternatively, it may be necessary to cut down results for the AAG query based on distance first to ensure that the index is being used.

As an indicator of performance on a local development machine with default Postgres database and the simple table outlined and the following query to order locations on distance from San Francisco:

```
SELECT id,
       earth_distance(ll_to_earth(37.7749, 122.4194), ll_to_earth(lat, lng)) AS from_sf
  FROM locations
 ORDER BY earth_distance(ll_to_earth(31.7749, 123.4194), ll_to_earth(lat, lng))
 LIMIT 10;
```

1,000 records ~ 40ms (without index) ~ 40ms (with index)

10,000 records ~ 270ms (without index) ~ 180ms (with index)

100,000 records ~ 2.5s (without index) ~ 1.5s (with index)

1,000,000 records ~ 24s (without index) ~ 12s (with index)

Number of records	Time without index	Time with index
1,000	40ms	40ms
10,000	270ms	180ms
100,000	2500ms	1,500ms
1,000,000	24,000ms	12,000ms

And with an index and boxing so any records more than a specified distance are excluded.

```
SELECT id,
       earth_distance(ll_to_earth(37.7749, 122.4194), ll_to_earth(lat, lng)) AS from_sf
  FROM locations
  WHERE earth_box(ll_to_earth(37.7749, 122.4194), 5000000) @> ll_to_earth(lat, lng)
 ORDER BY earth_distance(ll_to_earth(37.7749, 122.4194), ll_to_earth(lat, lng))
 LIMIT 10
```

Number of records	Time
1,000	40ms
10,000	75ms
100,000	400ms
1,000,000	3,500ms

Note that with boxing the times are significantly faster. However, this is only half of the story as the results have been cut down by the boxing so the ordering is on a smaller set of results. That notwithstanding this would provide a performance improvement if limiting on distance is acceptable.

11.1.1.2 PostGIS

PostGIS is much more fully featured Postgres extension which provides spatially accurate distances based on longitude and latitude. Because of being more featured PostGIS is more complex to use than earthdistance. However, it is currently being used for converting between coordinate systems in the current application. Also in the simple test, ordering locations by distance from a given point, it appears to be more performant than earthdistance. For these reasons, this would be the current preference for the geospatial Postgres extension. Below is a code snippet to give an example:

```
=# CREATE EXTENSION postgis
=# CREATE TABLE locations (
                id SERIAL PRIMARY KEY,
                name VARCHAR(64),
                location GEOGRAPHY(POINT,4326)
   );
=# INSERT INTO locations (name, location)
        VALUES ('el capitan', ST_GeographyFromText('SRID=4326;POINT(119.6377 37.7339)'));
=# INSERT INTO locations (name, location)
        VALUES ('joshua tree', ST_GeographyFromText('SRID=4326;POINT(116.3131 34.1347)'));
=# SELECT name,
          ST_Distance(ST_GeographyFromText('SRID=4326;POINT(122.4194 37.7749)'), location)
     FROM locations

    name     |   st_distance
-------------+-----------------
 el capitan  | 245169.32790355
 joshua tree | 682825.23498901
```

Note that SRID=4326 is indicating that the coordinates we are using are latitude and longitude. PostGIS can support a large variety of other coordinate systems as well.

As an indicator of performance repeating the location test performed with earthdistance but with using PostGIS this time gives:

```
SELECT id,
       ST_Distance(ST_GeographyFromText('SRID=4326;POINT(122.4194 37.7749)'), location)
  FROM locations
  ORDER BY ST_Distance(ST_GeographyFromText('SRID=4326;POINT(122.4194 37.7749)'), location)
  LIMIT 10
```

1,000 records ~ 15 ms (with index)

10,000 records ~ 60ms (with index)

100,000 records ~ 280ms (with index)

1,000,000 records ~ 2.5s (with index)

As can be seen in this test, the performance is better than that of earthdistance.

11.2 Address location lookup

The application originally used a postcode to location database, Code-Point, to determine the location of addresses. However, as outlined in the Code-Point section this will not work well in the US and as discussed in the Google geocoding API section the system is being transitioned. Note that the coordinate system used still expects Eastings and Northings and so PostGIS is being used to translate the locations obtained from Google. For a more discussion see the chapter on transforming location data sets from the UK to America.

11.3 SMS gateway

The company that currently provides the SMS gateway for the application, Esendex, also operate in the US. Therefore, it should be straightforward to port the existing code to work in the US. In a worst-case scenario, it may be necessary to regenerate the client code from the WSDLs however this should not present a major issue. See the section on SMS gateway in the platform and infrastructure part of the document for additional details.

11.4 Language localisation

11.4.1 Static Text

The application has text, such as terms and conditions, which have been developed for the UK market, but which will not be appropriate for the US. There are two solutions to this issue. The most straightforward is simple to find those parts and change them so that they are appropriate. The second is to localise the application so that, depending on your locale, you will see different content. This approach is more generic and will be the one to go with if there is a case for having different languages supported, e.g. English and Spanish. All the text for the site is put in a series of properties files. Then the locale and language the user has selected in the browser determines which of these files is used.

In either case this is work for a developer but also a content manager with experience drawing up terms and conditions etc.

There are a few areas of the application that might be harder to tackle than others, anywhere text is directly embedded into JavaScript for example. It will be necessary in these cases to ensure that the text is read correctly depending on language.

11.4.2 Dynamic Text

There are sections of the application where the end users enter content, such as reporting instructions, i.e. where and when to turn up. In this case, it is not possible to work out ahead of time what the translation should be. In this case a third-party API such as Google Translate could be used. This service provides a simple to use REST interface to translate, and moreover it has a detection service as well. This may be preferable to assuming that the text input is in English, or based on the browser local type. If this approach is to be taken it makes sense to store the translations in the database when they are entered, or via a scheduled job, and not simply look them up on the fly. The reasons being that this will be faster and these services have a cost associated.

Note that there are a few places where this will not be quite so straightforward:

- Where users can enter formatted text e.g. html. In these instances, it would be necessary to parse out the text from the formatting and then reinsert.
- Single words, such as tags. While translating is not perfect, usually text in a phrase can be understandable. However single words do not have context and can be esoteric, e.g. acronyms or ambiguous. As an example, *nail* in English has two meanings but in French they

are two different words: metal nail *Clous* and finger nail *ongles*. Therefore, for areas on the application like tags automating the translation may not be appropriate. If translation in these areas becomes a necessity then functionality could be developed to allow a human to patch up the data.

Given the above comments it would seem like a sensible approach to initially tackle the areas of the application that would benefit from it most.

11.5 Time zones

The application currently works in the UK time zone i.e. GMT and BST. If the application is ported over to only one area in the states, e.g. Los Angeles, then there shouldn't be much code that needs changing, though there are a few pieces of JavaScript that will need attention. However, changing the servers to work in the new time zone should fix most of the issue.

The above solution however will not work if the application is to span multiple time zones, which may well be the case in the US if states are to be spanned. Additionally, a more robust approach to handling time is recommend. The comprehensive solution is to store times internally in a time zone agnostic format. E.g. GMT. However, when these times are output to or input by the user they are converted using the user's locale to obtain the time zone. It will probably be necessary to allow a user to set their time zone too to override the default provided by the browser.

Note that while this approach is conceptually straightforward time zone functionality is often fraught with difficulties, and extensive testing of any changes would be advised. This notwithstanding this change would make the application significantly more flexible.

11.6 Business Rules

Without knowing the exact specification, it is hard to determine what changes will need to be made to the application, but it is likely that some of the business rules will need to be changed. However, as one example, the taxes that are added have been flagged as something which will likely need to much more customisable. Currently, in the UK it is a flat rate, but in the US it is likely to be much more variable and take into account factors such as which state is being worked in. In principle, there is no logical reason why this should present an issue from a coding point of view, if all the data is available to make the calculation. That said the use case is different from the current design and so the following should be considered:

- The ability to upload the *rules* and change them as required to a running server.
- Some form of resolution for when multiple *rules* are encounter. This may be as simple as ANDing all the rules together.
- If possible with the data available validate new rules determine whether there are any conflicts.
- Have date ranges for when the rules come into effect, which would be triggered by when the work was being performed.
- Depending on business needs, apply these changes retrospectively and inform users of any changes. This would require significantly more development effort.

In any event the central premise remains the same which is the ability to upload some *rules* in some guise and have them interpreted when calculation is required. The exact format that these *rules* take would have would be determined by the likely complexity of them, but maybe as simple as some xml, or at the other end of the spectrum running java code, though with this approach security would need to be considered. In all cases though, while the rules do not need to be known ahead of time, the data that the rules are based on is.

12 Transforming location data sets

Historically the application was developed for the UK market, and as a result some of the technical decisions were predicated on this. For example, using UK postcode as a proxy for location data which is reasonable as postcodes are tightly bound to geographical locations. This is not the case for US and so an alternative approach is required.

In addition, the dataset on the demo server has grown over the years that it has been in operation. This data is based in the UK with UK addresses. For demonstration purposes, it would be useful to be able to transform this data so that it is representative of the area where the demonstration is required e.g. Los Angeles.

The solution to the first problem has been to change the application so that it uses whole addresses for location and not just postcodes and this has required a change in the way we look up locations to use the Google geocoding API. Note that this work is not complete and some parts of the application still use postcode. While this functionality will continue to work, as the geocoding API will give a central location for a postcode, it will be less useful as US zipcodes cover large areas. For example asking "What is near 89049" does not yield as useful results as "What is near S11 8TY" and it may be necessary to change the application in places so a user inputs a whole address or picks a location on a map, or uses the location reported by their phone.

The solution to the second problem has been to lift all the UK addresses on the demo server and to move them over to the US. The approach taken requires some manual steps but in principle should work, with some caveats, for moving the dataset anywhere. Outlined below is the steps taken:

12.1 Translation

12.1.1 Cleaning of data.

The addresses in the database were inspected and where they were missing location data, this was patched up where possible by querying the location to postcode database.

12.1.2 Linear transformation of coordinates.

The original data was in Eastings and Northings for the UTM 30U zone. The data needed to be lifted to the UTM 11 zone. Most of the original data was centred around London. The translated data needed to be centred around Los Angeles. A point was found at the centre of each of these cities in their respective coordinate zones. The difference in absolute terms, not considering zone, between these points was determined. A linear transform of this difference was applied to all the coordinates in the system. Thus, an address which was located at the centre of London would now have coordinates in the centre of Los Angeles.

12.1.3 Finding new addresses

The new coordinates we then used to find new addresses. Firstly, PostGIS was used to convert the coordinates from UTM 11 zone coordinates to a latitude and longitude. These coordinates were

then feed to the Google geocoding API to get a new address. The initial coordinates did not always yield an address and where this was the case the net was cast a bit wider. This was done by querying the API with locations gradually increasing from the initial position. Most of addresses, approximately 80 percent, found a match in this manner. However, there were about 20 percent that did not match, these replaced with a dummy address.

The reason for many of these failed matches is because Los Angeles is on a coast whereas London isn't. By applying a linear transformation some of the outlying address locations ended up of the coast. However, by selecting a centre of LA in land this problem was minimised and for the demonstration datasets should not be an issue.

12.1.4 Applying to other cities

This technique should be applicable to moving the demo data to another city if required, with a couple of caveats. If it is another costal city it will be necessary to choose a translation point reasonable far in land. Also, if the city is not big enough then a reasonable number of addresses may fall outside the city limits and so might not be able to find a suitable new address. Finally, if this is to be attempted reasonably often then it would be worth automating the process completely.

13 Cloud-based platform

Having uFlexi hosted by cloud-based providers has many advantages but not having to maintain the infrastructure and leave it to companies specialising in this is a major one as is benefiting from scales of economy. Currently uFlexi is hosted in the cloud see the section on Bytemark. It is a relatively simple setup however with just a single Virtual Machine provisioned and acting to all intent and purpose as if there was a real uFlexi server sitting in an office somewhere. There are many other cloud providers offering differing products and features, not just the ability to spin up virtual machines. This chapter details mostly the services that AWS provides as an example of the features that are available and would benefit the uFlexi. This is just an example, as AWS is the platform the uFlexi team has most experience with. There are many other providers and Google Cloud would be another appropriate alternative. If the container orchestration tool Kubernetes is used it may be a better solution. See the section on Orchestration for more information on this topic. The reason Google Cloud maybe be a better solution, in this scenario, is that its Container Engine tool is a hosted Kubernetes environment which would take much of the effort out of the initial setup.

Whichever cloud-based provider is chosen the following would features would be desirable

- US based data centres. Multiple zones ensure data is transmitted and stored within a single jurisdiction.
- Wide devops adoption.
- Managed infrastructure for common platforms. E.g. Tomcat and Postgres.
- Cost effective solution.

13.1 Simple VM migration

Migrating the application to a new virtual machine with another cloud provider would gain a US-based server. If this approach is taken it is recommended that a fresh VM is created and the application reinstalled. This would mean that all the latest patches and security updates would be present, which would not be the case if the VM images was ported over.

There would, depending on provider be additional advantages for example:

- Identity and access management e.g. Google Cloud or AWS.
- Simple resizing the VM if more capacity is required. I.e. increasing the power of the machine.
- DNS services e.g. Google Cloud or AWS .
- Content Delivery Network e.g. Google Cloud or AWS.
- Managed relational databases.

The following section describe some of these features in more detail using AWS as the example, but as stated before other companies provide similar functionality.

13.2 Managed Infrastructure

This is likely to be the most scalable and cost-effective approach. Instead of having a series of VMs that require maintaining, the hosting company provides for example a managed Tomcat server. This means that it is not necessary to worry about e.g. patching the underlying VM, this is all taken care of by the provider.

AWS provide command line utilities and APIs which allow applications to be defined in scripts. New environments, for production or testing, can be deployed easily. Integration and continuous deployment is straightforward.

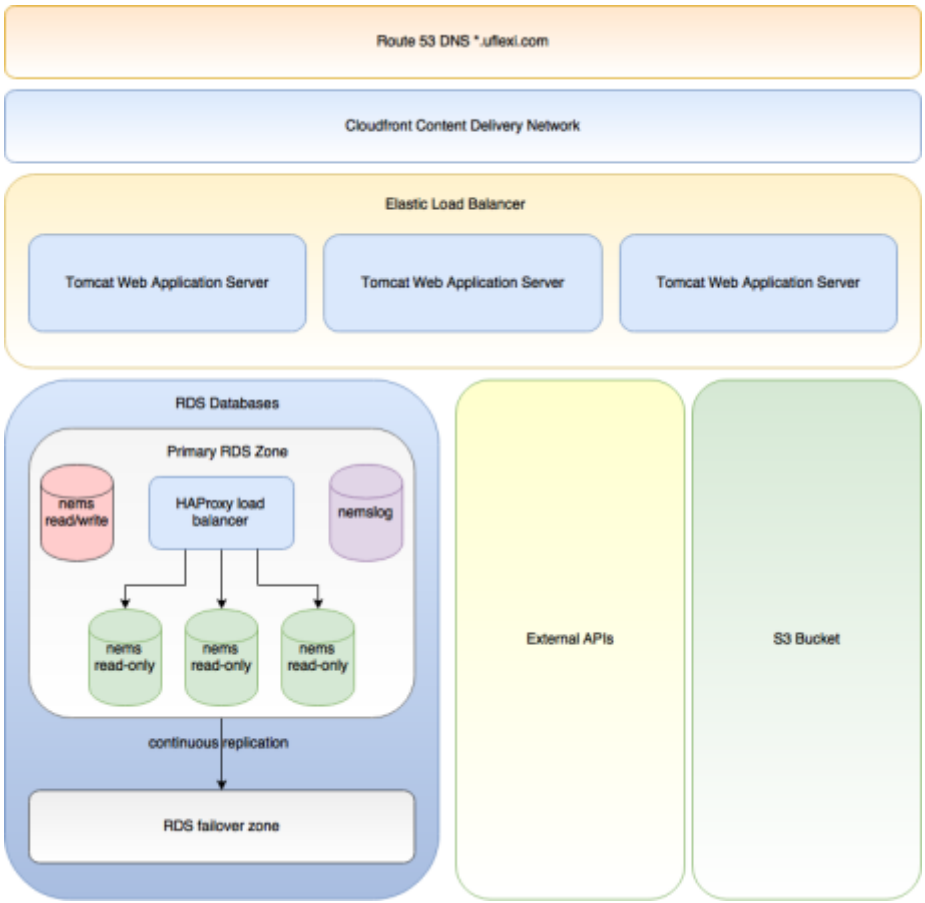

Figure 20 - AWS Managed Architecture

13.2.1 Route 53

DNS Services would be provided by Route 53 which is highly available and scalable.

Additionally, SSL certificates, excluding EV, are free of charge for hosted environments.

13.2.2 CloudFront

CloudFront is the AWS content delivery network. This can be used to cache resources, e.g. images, from the uFlexi, taking the load of the application itself. Additionally, it will optimise the serving of these resources.

13.2.3 Elastic Beanstalk

Elastic Beanstalk is the AWS managed service for deploying web applications. uFlexi, in its current incarnation, would use a managed Tomcat service. Using this service has the following advantages:

- Load balanced application servers provide horizontal scaling. The application would however have to be changed to enable this to work. See section on horizontal scaling for more information on what would be required.
- Load-balancer automatically launches additional instances.
- Automatic updates with the latest OS and Tomcat patches.
- Managed deployments with rolling updates. New servers are brought online once healthy and old servers removed again this requires changes outlined in the horizontal scaling section.

13.2.4 S3

S3 provides fast a distributed file storage. Currently we files are stored in the deploy directory of the Tomcat server. See the section on file storage for more information. This should be changed and S3 file storage would provide a good shared storage for this purpose.

13.2.5 RDS

RDS is the AWS managed relational database service and uFlexi would use the Postgres flavour.

It provides:

- Multi-zone failover. Provides transparent failover in the event of failure, and gives us a continuous backup in a different geographic location.
- Multiple read-only replicated databases allow horizontal scaling for read-only queries. A load balancer such as HA Proxy can be used to distribute load. This isn't currently standard. Work would need to be done to the application to enable scaling in this manner, see the section on database scaling for more information.

14 Containerised Platform

This chapter describes a containerised approach to infrastructure.

14.1 Containers

Containers are a way to package up software that can run on a shared operating system host. In this way, they are like virtual machines, but unlike these they are light weight and use much of the hosts functionality. In this way as well as being light weight they are also relatively small, and only contain the minimum of libraries required to function. Additionally, they spin up quickly as they are not having to boot strap an entire operating system. The canonical example of container technology is Docker.

A Docker container might be a web application, and so the libraries bundled in it would include Java, Tomcat, and the actual application as a deployed WAR file, but nothing else unnecessary. Thus, is results in a relatively small package.

With this infrastructure, the proposed unit of build artefact is a Docker image. I.e. the output from the build pipeline would be a Docker images. These would, for example, contain the application microservices with any libraries need to run them. Equally it could contain a larger web application with a Tomcat server, such as the application in its current state.

These docker images would then be deploy to a given environment, the orchestration of which can be handled by a tool for example Kubernetes. See the section on Kubernetes for more information.

This approach has several advantages over the traditional approach of producing artefacts like WAR files which are then deployed to servers. These include:

- Developers can get started quickly and have minimum install requirements. The dependencies are contained with the images themselves so no need to have to install e.g. a Tomcat server.
- The containers are the same from development as for production, which helps to eliminate environmental issues.
- Moving from one supplier of infrastructure to another is less painful. The images should be relatively easy to deploy to another environment particularly.
- Decouples application from infrastructure.

There are also some disadvantages, which include:

- Difficultly with hot reloads for developers.
- Significant initial infrastructure work and investment.
- More technologies in the stack adding to the complexity, though these should be mitigated as the tooling is designed to make things easier in the long run.
- Can be developer pain especially if they have not used technologies like Docker before.

There is an alternative to Docker, Rkt, however, Docker currently has a greater uptake, and is the de-facto standard for containers.

14.2 Orchestration

Containers provide the units of functionality. However, these still need to be configured to talk to each other deployed to machines, orchestrated. This could be done as a manual job with shell scripts but a better approach is to have a tool to perform these actions. There are a few tools that can do this with Kubernetes and Docker Swarm being popular. Kubernetes would currently be our preferred choice.

14.2.1 Kubernetes

Kubernetes is a tool to deploy and manage container images and will work with both Docker and Rkt.

Briefly to cover some of the terminology, the unit of overall deployment is a Kubernetes cluster. This consists of a Master and a series of Nodes, effectively virtual machines. Collections of containers are run in a Pod, which runs on a Node. A Service represents a collection of Pods, which may not be running on the same Node, and manages network travel into them and exposes them to the external environment. The Master oversees coordination of the cluster such as deploying containers, scaling and rolling out updates. The Masters behaviour is dictated by a YAML configuration file.

By using this abstraction, the underlying infrastructure is divorced from the application. Thus it is possible to migrate from one infrastructure supplier to another, all that is required is a Kubernetes cluster, and there are tools to minimise the effort in doing this for many cloud-based providers. It is also possible to run on a Bare Metal environment.

Other benefits of Kubernetes include:

- Scalability. For example, if a service finds it is under heavy load it can auto scale and add another Pod on another Node.
- Rolling deployments. Updating a Service, a Pod at a time.
- Minikube, a tool to setup a Kubernetes cluster on a local development machine and deploy to it which can take the production YAML configuration, within limits.

14.3 Statelessness

Note that the solutions above, both Docker and Kubernetes, assume a statelessness to the applications that they contain or manage. Without this it would not be possible to just add another container when wanting to scale or take a container down and replace it when upgrading. Writing most of the application so that it does not contain state is possible, see the section on horizontal scaling for some strategies. However, there will always be need for some persistent state such as a database or file storage.

The persistent storage parts of the infrastructure may need to be out of the direct control of Kubernetes, and this may entail a standard database for example.

As well as persistent data storage, there may be a requirement to have more transient state, for example a cache for the database. If this is required then it should still be possible. If the cache is in a

Redis container, for example, then as when one container is deployed it can replicate its cache from another. Additionally, with this caching example, losing the cache is not a big issue as the cache will fill up again as the database is read. Session state is the other transient data discussed in this document. While what is said about Redis, and replication remains true, if a microservices design is implemented session state will not be present. See the section on microservices authentication for additional details.

14.4 Configuration

Configuration is another area worth consideration. Section outlined why it is better to produce build artefacts which are environment agnostic. With traditional Java web applications, this was done by having configuration read from a properties file that lived with the server. The production environment had a different properties file to the QA one. Now that the unit of build artefact is a container, this is not possible. One solutions to this are ConfigMaps, and Secrets, which allow the injection of configuration into containers. This allows the application to be configured differently per an environment, so, for example, the API key for google maps could be different on production to QA.

15 New Architecture

This selection discusses, at a high level, some conceptual ideas behind a complete rebuild for the system, and how it would be made both flexible and scalable. The topics discussed here are the preserve of reasonable sized projects. The designs put forward here would require extensive upfront development effort, but the payoff in the long run is greater.

15.1 Infrastructure

As discussed in the containerised section, there are many benefits to having an orchestrated virtual environment. These include portability, scaling, ease of generating new environments, consistency between them to mention a few. For these reasons from an infrastructure point of view the build artefacts would be Docker images. Rkt is an alternative to Docker, however, does not have the wide uptake of the later, though would be worth considering when this work is to take place. These can then be orchestrated and configured with a technology such as Kubernetes.

15.2 Microservices

Microservices are the independently deployable services organised around business capabilities. These microservices, communicating with each other's APIs, together form the complete application. In the right context, when designed correctly, microservices have in many advantages over a traditional monolithic architecture. These include:

- Each microservice has a small code base, which is easier to rationalise about, test and refactor.
- Independently deployable. The whole application does not be taken down to deploy a new piece of functionality.
- Cross functional teams (the unit of division is the business capability), which means that each team has a deeper understanding of what it is they are working on.
- Encapsulation in each microservice means it is quite possible to have different programming languages and technology stacks for each one, so enabling which is ever most appropriate.
- Ability to scale only parts of the application that require it.
- Makes continuous deployment more practical.
- Makes zero downtime deployments more practical.

It should be noted that a microservice design does come with some disadvantages, namely that it introduces additional complexity. As each part is independent it is possible that some of the services are down, e.g. because of a deploy, and others might still attempt to communicate with it. This means that each microservice needs to be able to deal with failure and deal with it gracefully. Additionally, while the microservices are independent of each other in development terms, they are dependent on each other's API. Thus, breaking changes cannot be introduced into APIs, or versioning needs to be maintained.

With a rebuild of the uFlexi application a microservice design is proposed. The first issue to resolve is determining where the demarcation boundaries, along business capabilities, are. Currently there is a relatively clean separation between Buyer, Seller and Agency and so these would be the initial logical candidates. It may be possible to subdivide these further, but at a minimum, a service for each of these separate areas is advised. On top of that it is likely that there are small, none application specific, areas which can be hived off into microservices with ease. These include notifications for sending emails and SMS, location lookups for addresses, distance and journey time calculations, reporting and many more.

15.2.1 Asynchronous communications

One way to make the application more resilient to failure, and to increase encapsulation further is to have the microservices communicate, with each other, via asynchronous communication. So instead of having then linked together by, for example direct HTTP calls which are synchronous, they might use an asynchronous message system such as Kafka. The advantages of asynchronous messaging include:

- Better encapsulation, the microservice does not expect a reply from any other service so becomes independent of its functioning.
- Services can be taken down independently, allowing for parts of the system to be upgraded independently.
- A failure in one service doesn't break the rest of the application.
- Enables separated development.

As well as the advantages, there are also disadvantages which include:

- It is not always possible, at the very least communication with the database will have to be synchronous.
- As synchronous reads between services are to be avoided, each service must replicate the data that it needs from other services. This data can be updated by message passing from the other service and will be eventually consistent.
- More development effort is required to deal with the asynchronous nature of the application, having to deal with more error cases, replication, eventually consistent etc.

Note that communication with the outside world is still synchronous e.g. the requests from the browser.

15.2.2 Authentication

Authentication are worth discussion for microservices as the design is different from that of a traditional monolithic application. Usually in a monolith when a user logs in they are given a token, this token is also stored in the application with details of the user. The next time the user requests a page they present this token. The application looks it up in its internal store to check its valid and who the user is.

In a microservice environment this isn't possible because the user will be sending requests to multiple different services which do not share an internal datastore. The solution to this is token-based but works slightly differently. When a user logs in they are directed to an authentication

microservice. This can be a custom microservice or alternatively a generic third-party application such as Autho. This authentication service will validate login details and will then provide the user with a token with their relevant information. Part of the token is signed with a secret key. The user would be unable to forge a token without this.

The technique of signing is important and there are two distinct ways of doing it. These are symmetric, which has a single secret key to generate the signature and validate it, and asymmetric which has a secret private key to generate the signature, but a public key to validate it. In this section, the asymmetric approach as assumed as it is more flexible.

Once the user has logged in and received the token from the authentication service then this is passed this with all future requests. When another microservice receives a request, it reads this token and determines if it is valid. To do this it uses the public key of the authentication service and check that the signature is valid. If it is, then the only way it could have been generated is if the authentication service had signed it with the secret private key. Thus, the information in the token can be trusted. Note that if the service checking the token already has the public key it does not need to communicate with the authentication service at all.

15.3 Overview of redesigned architecture

Considering the preceding sections and the part of the document on containerised platform this section outlines a proposed solution a redesign in general terms.

It is necessary to identify business capabilities so that the application can be split along these lines. With uFlexi the current major unit of business is the Agency with work being done in one Agency being mostly isolated from another. This can be subdivided into Agency, Seller and Buyer as discrete business functionalities see section on microservices. Thus, these are the logical starting positions for the dividing lines between microservices. It is likely that these could be subdivided later and ancillary service such as email and SMS also split out.

These microservices would communicate asynchronously with each other using messages and a service such as Kafka would be ideal. They would have a stateless design allowing scaling and rolling deploys. Each of these microservices would have its own persistent data storage and would be independently deployable. In addition, there would be a universal data store that would be written too, which will be eventually consistent.

Authentication and authorisation to and from the outside world would be handled using JWTs.

The build artefact for each microservice would be a container image, most likely Docker. These would then be configured and run in a Kubernetes cluster. See Figure 21, below:

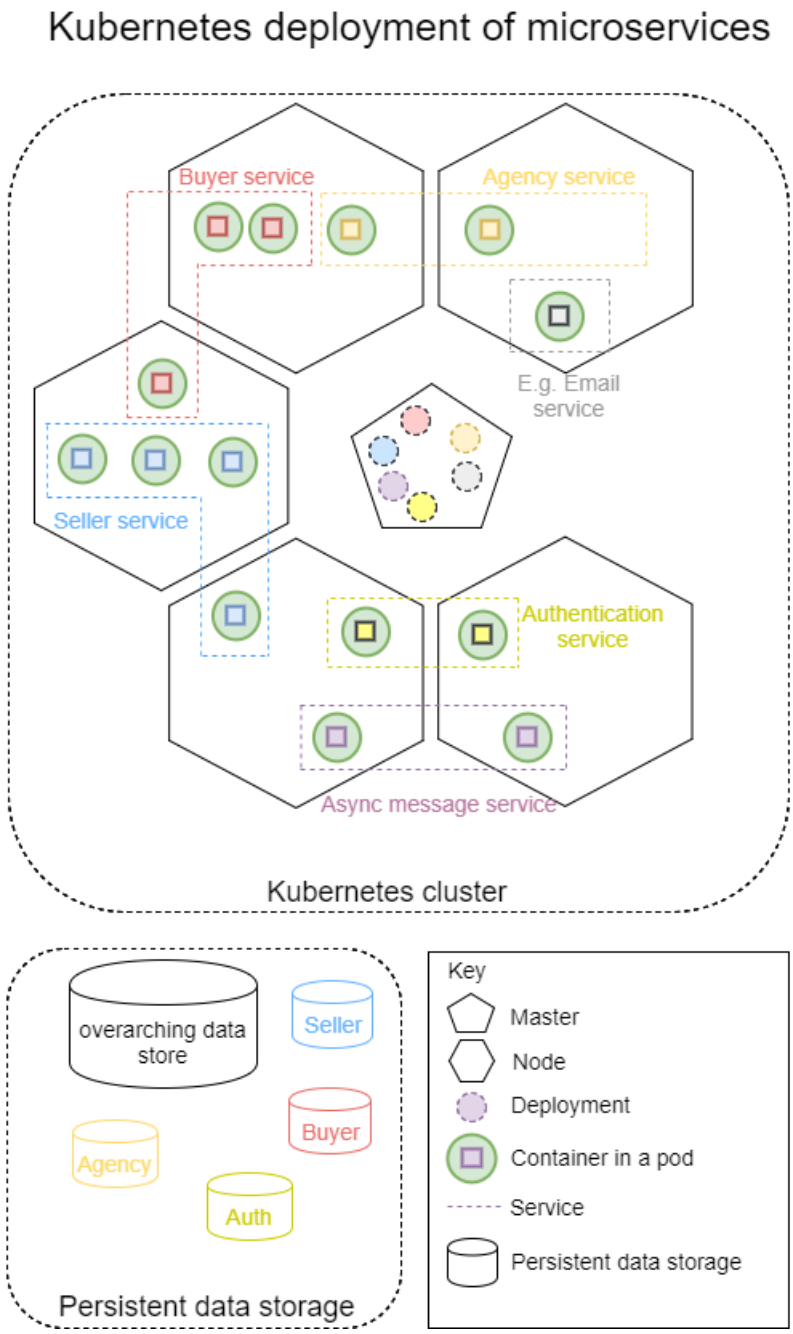

Figure 21 - Kubernetes and microservices

The web application could either be completely JavaScript-based communicating with the microservice endpoints directly, via synchronous REST with JSON over HTTP. Alternatively, there could be a thin layer of web tier application that would sit between the user's browser and microservices aggregating the results and generating html. Each approach has its merits, the first giving a much richer user experience, the latter allowing the website to function without JavaScript. It should be noted that a JavaScript only website does not by definition mean accessibility will be an issue.

For other functionality that needs to access the services, for example a native mobile phone app, would access them directly. The diagram in Figure 22 shows some of these features.

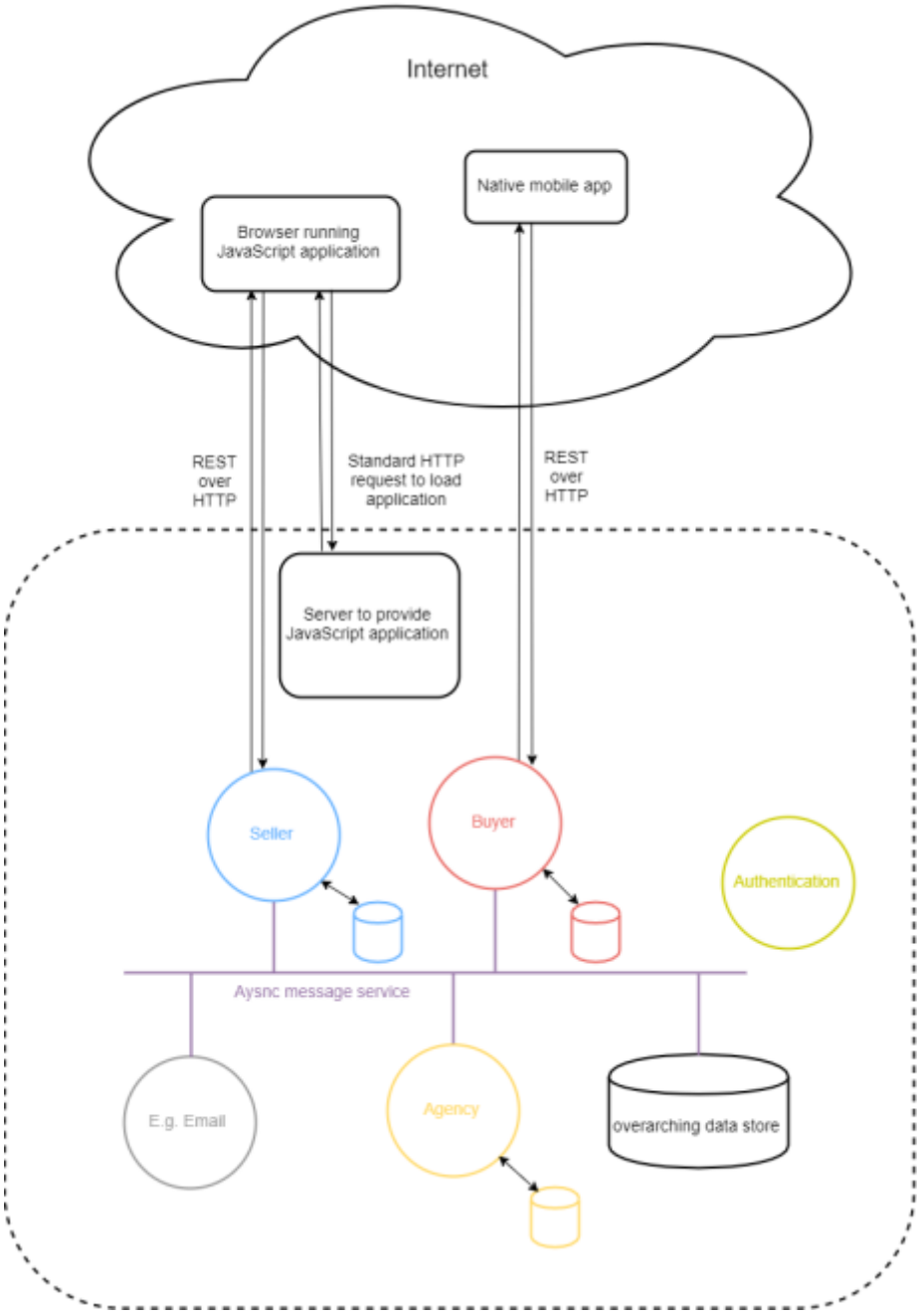

Figure 22 - New architecture

15.4 Revaluating business requirements

If uFlexi is to be rebuilt completely then it would be advisable to re-evaluate the business requirement, as there is a clean slate to work with. These requirements are likely to have an impact of the technical design of the application. While it is not possible to know exactly what these would be it is a reasonable likelihood that there might be more physical separation between Agencies. So that, unlike the preceding section, Agencies might have their own dedicated Kubernetes cluster or at the very least be namespaced in the same cluster. Some of the pros and cons of this *cluster per an Agency* approach are outlined below.

15.4.1 Cluster per Agency

With this approach Agencies (the Agency microservice and its corresponding Buyer and Seller microservice) would live on separate Kubernetes clusters or be isolated from each other by use of namespaces. In either case, they become much more decoupled from each other, which has the following implications:

- Independently deployable. Each Agency can have a different deployment schedule.
- Agencies can choose their own infrastructure provider.
- Agencies can run code specific to them, however this would result in an ongoing maintenance cost.
- Agencies can scale independently. If one Agency has a huge amount of traffic they can provision their own machines.
- Various kinds of business entity. There is no longer a restriction to a series of functionally identical Agencies. Anything that can consume the standard endpoints can live in the ecosystem. For example:
    - Broker Agency. Buy and sell from the wider market.
    - Dynamic auction-based market price setting.
    - Pure buyer.
    - Pure Seller.
    - Machine Learning Smart Agency. Use historical trends to predict and pre-book based on current market conditions.

To obtain these advantages there are certain design implications which include:

- It is unlikely that new Agencies can be generated on the fly. This would require intervention from the devops team to provision more hardware, or virtual machines. However, this limitation could be viewed as a positive:
    - Being able to allocate more machines and infrastructure is the sort of action that normally requires oversite.
    - While intervention would be required setting up new environments should be relatively painless as it should be mostly scripted.
- To enable Agencies to interact with each other it would be necessary to have
    - Point to point communication between the different clusters.
    - Or messages routed through a central broker.
    - In either case there would be a requirement for a discovery service where Agencies can register to work together.
    - Agencies can be physically separated and so communication will be via the internet, not just the internal network on which they are deployed.
- A centralised CEHAD database could still be maintained, where Agencies insert availability data they wish to share. Again, this would need to have internet communication.

- While internally the different Agencies could have their own differing code, they would nevertheless need to be able to communicate other over a common API:
    - This API would have to either support versioning, not allow breaking changes or change very infrequently. For if breaking changes are introduced all the Agencies would need to upgrade.
    - This is not a trivial task and it is recommended that the API is well specified ahead of time to ensure changes required are minimised.