Architecture-Driven Development

University of St Andrews

170002815 · MATRIC NO · MATRIC NO · MATRIC NO

CS5033 · Practical 2

22 March 2022

1 Overview

This report explores an example architecture for a healthcare system which handles the management of patient information and appointments. The process of creating the architecture is explored in Sections 2 and 3, with the resulting architecture being described in Section 4; Sections 5 and 6 provide an implementation plan for and a critical evaluation of the chosen architecture respectively. The report is completed by a conclusion (§7) and set of references.

2 Architecting Process

2.1 Resources

The primary literature resources used to influence the design, evaluation, and implementation plan of the chosen architecture were Robert C. Martin's Clean Architecture [1], and Mark Richards and Neal Ford's Fundamentals of Software Architecture [2].

Clean Architecture provides information on architecture from a practical standpoint, including fundamental principles to follow when architecting components and code, as well as a critical analysis of modern software-architecture conventions. The principles relating to the architecture of code which explored in the book are the SOLID design principles, which are summarised as follows.

summaris

- **Single-Responsibility Principle (SRP)**:
- **Open-Closed Principle (OCP)**:
- **Liskov Substitution Principle (LSP)**:
- **Interface Segregation Principle (ISP)**:
- **Dependence Inversion Principle (DIP)**:

The principles relating to the architecture of components which are explored Clean Architecture are divided into two groups: those which deal with component cohesion, and those which deal with component coupling. The principles relating to component cohesion are as follows.

- **Reuse/Release Equivalence Principle (REP)**:
- **Common Closure Principle (CCP)**:
- **Common Reuse Principle (CRP)**

The principles relating to component coupling in Clean Architecture are as follows.

- **Acyclic Dependencies Principle (ADP)**:
- **Stable Dependencies Principle (SDP)**:
- **Stable Abstractions Principle (SAP)**:

2.2 Technologies

Throughout the design process, the following technologies were used.

- **Microsoft Teams** [3]: Used for group text and video communication.
- **Google Drive** [4]: Used to store and share images, progress reports, meeting notes, and other documents.
- **LaTeX** [5]: Used for creating this report.
- **GitHub** [6]: Used to store LaTeX code and enable asynchronous work on the report by different team members.
- **Lucidchart** [7]: Used to create architecture diagrams.
- **diagrams.net** [8]: Used to create system functionality flow charts.

3 Requirements Analysis

3.1 Assumptions

4 Architecture

4.1 Modelling Language

A relatively informal modelling language is used to describe the final architecture produced for this report, with the C4 model being used in combination with a notation similar to UML. This combination was chosen as all team members were relatively familiar with the concepts used for both practices, focussing the project's time to designing and evaluating the architecture, rather than learning modelling languages.

C4 Model

The C4 model primarily consists of four different architectural viewpoints, described in the form of diagrams, which represent an architecture as a hierarchical set of abstractions [9]. The viewpoints are designed to reflect to how software architects and developers think about and build software, and are summarised as follows.

- **System Context Diagram**: A viewpoint which describes a system's architecture as a whole, allowing viewers to see the "big picture". The diagram has a focus on the users of the system, rather than specific details, such as which technologies are used. Section 4.2 explores the system context diagram for the chosen architecture in this report.

- **Container Diagram**: A viewpoint which shows the details of a single subsystem within a system context diagram. The diagram is composed of containers, which represent an application or datastore (e.g. a web application, filesystem, database, etc.), and high-level references to other subsystems. Section ??? explores several container diagrams for the chosen architecture in this report.

- **Component Diagram**: A viewpoint which describes a single container within a container diagram, with respect to the major building blocks (i.e. components) that make up the container. Component diagrams contain some specific details, such as the technology used to implement or communicated between services, but still provide a relatively high-level overview of a container. Section ??? explores some component diagrams for the chosen architecture in this report.

- **Code Diagram**: A viewpoint which describes a single component within a component diagram as it is implemented within code, including the classes and interface involved, and their relationships. Code diagrams are not explored in this report, as they reference relatively low-level implementation details, and could be automatically generated from code written during the system's development.

The C4 model also includes three supplementary diagrams: the system landscape diagram, the dynamic diagram, and the deployment diagram. However, none of these are reference within this report, as the four core diagrams are sufficient to capture the chosen architecture [9].

code diagrams no used

Notation

find out from Ben & Jordan what notation is used

4.2 Overview

The architecture derived for the scenario listed in Appendix A is fundamentally a service-oriented architecture (SOA), as the key functionality of the system is implemented by multiple independent, distributed services. The core services used and their roles within the architecture are as follows.

- **Patient Information Service**: Handles create/read/update/delete (CRUD) operations relating to patient information, employing business rules in combination with authentication to authorise information appropriately. . This service is described further in Section ???.

- **Appointment Service**: Handles CRUD operations relating to appointments, employing business rules in combination with authentication to authorise information appropriately. This service is described further in Section 4.3.

- **Ambulance Service**: Affords the logging of calls and the dispatching of ambulances. This service is described further in Section 4.6.

- **GP Notification Service**: Provides the ability to asynchronously pass notifications to GPs. This service is described further in Section 4.5.

which reqs are satisfied by which svc?

In order to enable user interaction with services, a client-server-like architectural style is employed for user-service interactions; Section 4.7 discusses this further.

Figure 6 displays the system context diagram (see §4.1), where the set of core services, users, and interactions can be seen.

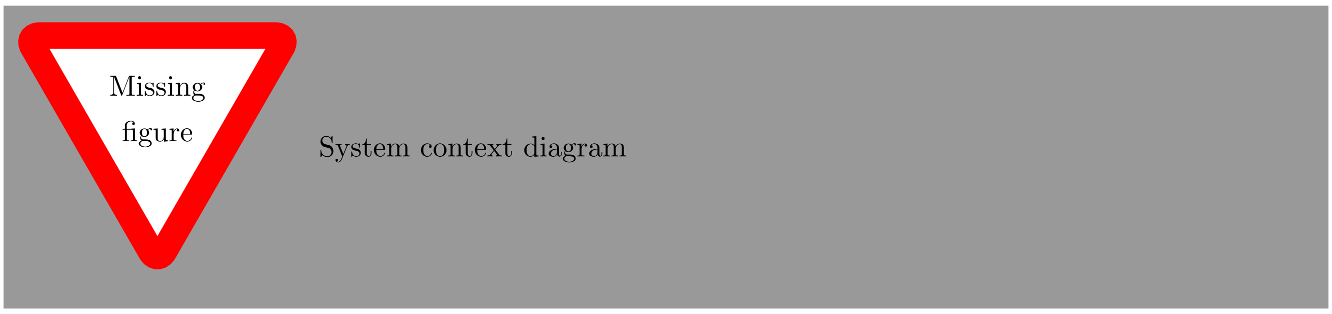

Figure 1: The system context diagram for the chosen architecture, displaying the service-oriented approach used as the fundamental architecture for the system.

Service-Oriented Architecture

As the target system's domains can be partitioned with relative ease, an SOA is a suitable choice for the overarching architectural style [10]. Although there are other applicable architectural styles for the target system, the following benefits of an SOA were the primary driving factors in the decision to prioritise it over other architectural styles.

- Services can be developed independently, reducing time-to-market, as independent teams can develop each service, and improving business agility, as each service's business rules can be adapted independently, provided the boundaries within the architecture are enforced effectively (see §)[11]. This is primarily an advantage over more rigid architectures, such as those using a monolithic style.

ref SOA not an arch in Eval sec

- Services can be deployed independently, facilitating the maintainability, short-term scalability (elasticity), and long-term scalability of the target system. This is primarily an advantage over architectures which are designed to have a one-to-one mapping from system instance to deployment machine, such as an architecture employing a layered monolithic style.
- Services are more independent with respect to failure than other architectural styles. This means that if one service fails, the failure of other services is not guaranteed, which is particularly beneficial for the target system, as the domains are mostly disconnected. For example, if the GP Notification Service fails, the other services in the system are not guaranteed to fail, meaning that the system will still provide the majority of its functionality to users. Overall, this makes the system more resilient to
- Similarly to failure, the services are more independent with respect to performance than other architectural styles. For example, if the GP Notification Service handles requests slowly due to a bug in its implementation, the Ambulance Service would be unaffected in terms of performance, provided it is deployed on a different machine; this is particularly beneficial for the target system, as some services, such as the Patient Information Service, require a certain level of performance to function as required.
- Due to the network-based nature of an SOA, services and clients are typically designed to be more robust to failure, improving the overall reliability of the system.
- Integration with external systems, such as the , is facilitated, as the target system will be deployed on a network due to the nature of an SOA. Similarly, integration with legacy systems which might need to be included in the newly developed system is facilitated [11].

hospital system

- The enforcement of architectural boundaries of the target system's domain is improved, as each service's boundary can be mapped directly to each domain's architectural boundary [10].
- The domain-driven separation of services, in combination with their low levels of interaction, mitigate the negative effects of an SOA, including selecting an appropriate granularity and handling service choreography/orchestration [10].

4.3 Appointments Service

The Appointments Service handles read and write requests relating to patient appointments. The service is designed with a relatively typical architecture for a read-heavy workload (see ???), consisting of a service which contains the business logic for the Appointments Service, a datastore, and a cache. Figure 2 displays the container diagram for the Appointments Service.

assump that appointments ar read mor than written

right diagram?

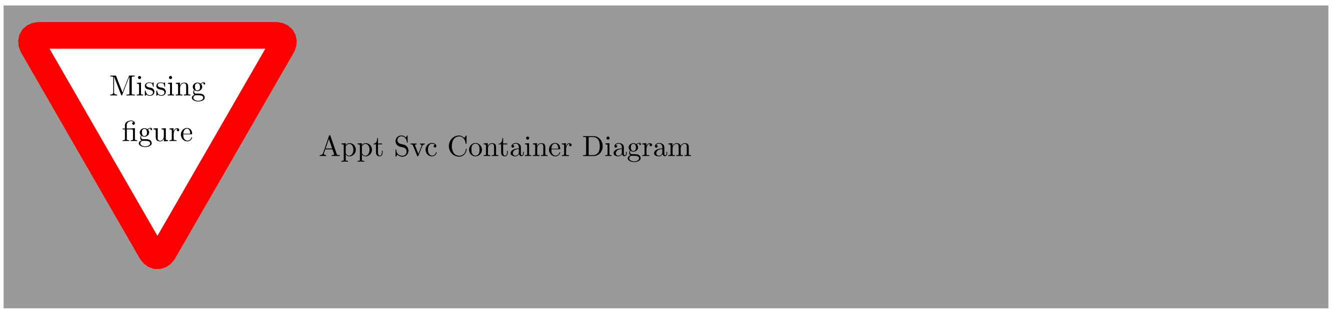

Figure 2: The Appointments Service described using a component diagram.

Datastore

The Appointments Service's datastore is used to persist data relating to appointments, improving the service's reliability. From Figure 2 (CAN IT?), it can be seen that the datastore is integrated within the architecture as a dependent on the service component within the Appointments Service, rather than the other way round. This integration style employs the Stable Abstractions Principle (see §2.1), as the dependency is pointing from the less stable datastore to the more stable business contained within the service component.

can it?

The application of the SAP also follows the advice given in Martin's Clean Code regarding how datastores should be integrated within architectures [12], and also does not confine the choice of data storage technology, allowing for solutions ranging from a simple file system to a complex RDBMS to be used, improving both the maintainability and scalability of the Appointments Service.

Cache

To improve the performance of the Appointments Service with relatively low overhead, an in-memory cache is used to respond to some requests without querying the datastore. In order to do so, the cache would store requests and their respective results as they are returned from the service/datastore after they had been retrieved from the service/datastore for the first time

Two key considerations when introducing a cache into a system are how consistency and coherency will be handled during its operation. Handling these concerns, particularly if the cache or service to which the cache is applied to is replicated, can quickly become a complex problem which introduces a significant amount of overhead [13]. Fortunately, as the Appointments Service's workload is read-heavy (see ???), the overhead introduced is likely worth handling when considering the potential performance gains introduced by the cache. Other overhead introduced by the cache, including storing entries, checking for entries, and handling evictions is considered to be marginal when compared against the potential performance gains provided.

complete ref to assumption

As it is assumed that there are many users , it is likely that evicting items from the cache on a commonly used lease-recently-used (LRU) basis is inefficient. This is because the cache will quickly fill up with requests relating to individual users which cannot be reused for other users (e.g. fetching a user's set of appointments), potentially evicting cached information which can be reused (e.g. the grand total of number of appointments for all users). While an alternative cache eviction policy, such as least-frequently used (LFU) or re-reference interval prediction (RRIP) policies, it considered as an implementation detail to determine this, and is therefore excluded from the scope of the architecture.

cite assumption

4.4 Patient Information Service

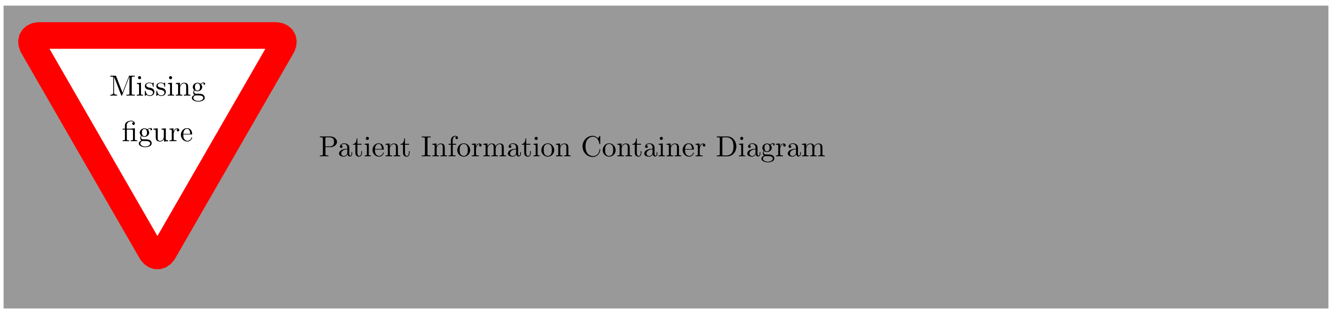

Figure 3: The Appointments Service described using a component diagram.

Datastore & Cache

As the data handled by the Patient Information Service and the Appointments Service (see §4.3) are both related to individual patients and are associated with read-heavy workloads (see ???), the datastore and cache used within the Patient Information Service are similar to those used in the Appointments Service (see §4.3), with respects to both operation and reasons for introducing them as part of the architecture (i.e. reliability and performance).

ref assumption

Multilevel Priority Queue

One of the more interesting architectural features of the Patient Information Service is the application of a multilevel priority queue to handle write requests from

look at
spec /
require-
ments
to find
proper
wording

[14]

4.5 GP Notification Service

The GP Notification Service is a comparatively simple service which provides the ability to asynchronously pass notifications to GPs through the use of a publish-subscribe messaging system. This service was introduced in the architecture for the following primary reasons.

1. The GP Notification Service allows for the GP client device to be decoupled from the services which send notifications to GPs, such as the .

   which ser
   vice?

2. The addition of future features which require GP notifications are greatly facilitated, as the new features would simply need to interact with the GP Notification Service, with no other architectural changes being required.
3. Future scaling of the system is better supported with a publish-subscribe architecture in comparison to direct communication between services sending notifications and GP interface devices.
4. The GP Notification Service can buffer notifications sent to GPs, enabling asynchronous communication, which in turn can improve the reliability of the system. For example, if a GPs device briefly goes offline, notifications are not necessarily lost as the GP Notification Service can buffer them while the device is offline; however, if services were to communicate directly with GP devices, then notifications would have likely been lost in this scenario.

Figure 4 displays a container diagram for the GP Notification Service.

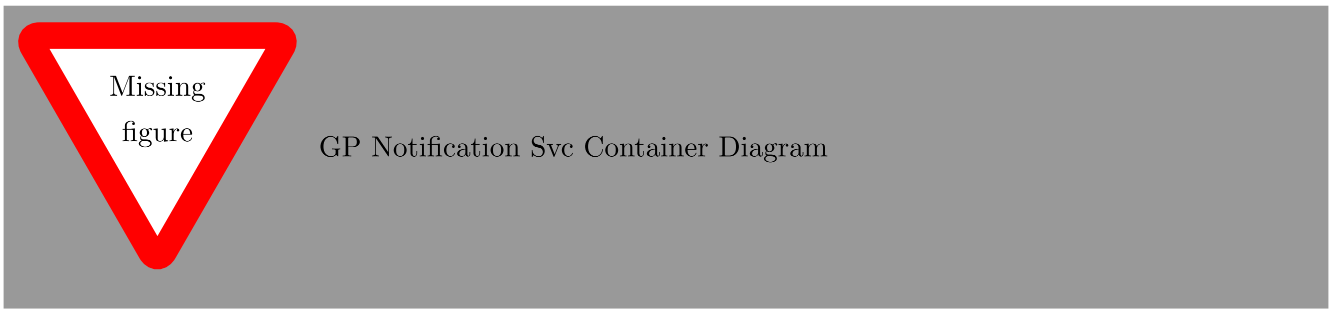

Figure 4: The Appointments Service described using a component diagram.

Publish-Subscribe Pattern

The GP Notification Service applies an event-driven architectural style in the form of a publish-subscribe (pub-sub) ,system which buffers notifications sent from other services, such as the ,

what?

A pull configuration is used, where the GP client device receives notifications by polling the GP Notification Service, rather than a push configuration, where the GP Notification Service would actively communicate with the GP client device to deliver notifications, for the following reasons.

1. The GP devices are external to the system, meaning that initialising communication from the GP Notification Service to each GP device is complicated to achieve in comparison to the other way around [15].
2. The rate of communication (e.g. polling frequency) can be dynamically adjusted based on the GP device's current network connection. This is more beneficial than the GP Notification Service having this control, as the GP device's network connection is likely to fluctuate more than the GP Notification Service's connection.
3. The polling interval can be adjusted for each individual GP by the GP's device, allowing for a trade-off in network usage and the amount of delay caused by the gap between the notification

being received and polled. For example, if the GP's device has a poor internet connection or is battery powered, the rate of polling could be decreased to improve network efficiency or reduce power consumption.

4. The notifications being sent to GPs are assumed to not be time-critical .

ref assumption

As seen in Figure 2 the GP Notification service

4.6 Ambulance Service

The Ambulance Service handles all ambulance-related concerns for the target system, including the logging of calls, the dispatching of ambulances, and the indication of callouts being dealt with. The ambulance-related requirements of the target system's domain appears relatively disconnected from other requirements, providing good reason for this service to be a part of the chosen architecture for this report.

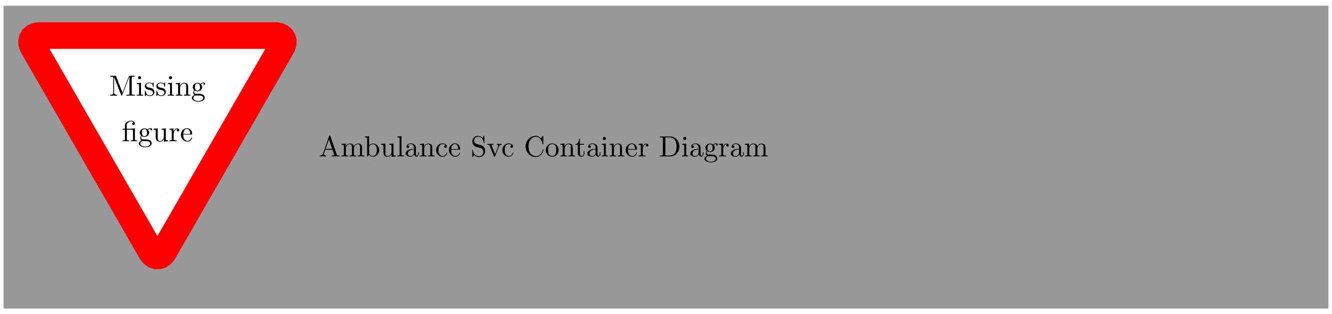

Figure 5: The Ambulance Service described using a component diagram.

Publish-Subscribe Pattern

Similarly to the GP Notification Service (see §4.5), the Ambulance Service employs an event-driven architectural style in the form of a publish-subscribe (pub-sub) messaging system. The pub-sub system is used primarily for performance purposes, and therefore uses a push configuration for message delivery, contrasting to the pull configuration used for the GP Notification Service.

A push configuration is more performant in comparison to a pull configuration with respect to message delivery latency, because the delay between the message arriving at the Ambulance Service and being sent to the client device is marginal, in comparison to a pull configuration where the delay is on average half of the polling interval [15]. Although a push configuration adds complexity from communicating with client devices outside the system's internal network, and reduces flexibility for the power and network consumption for client devices, these trade-offs are accepted in favour of the reduced message delivery delay, as this characteristic is critical to the Ambulance Service's functionality.

4.7 Clients

client-server pattern / style

The clients within the architecture offer the functionality of the services in the form of a user interface (UI). Figure ??? displays WHAT .

container diagrams for clients

what

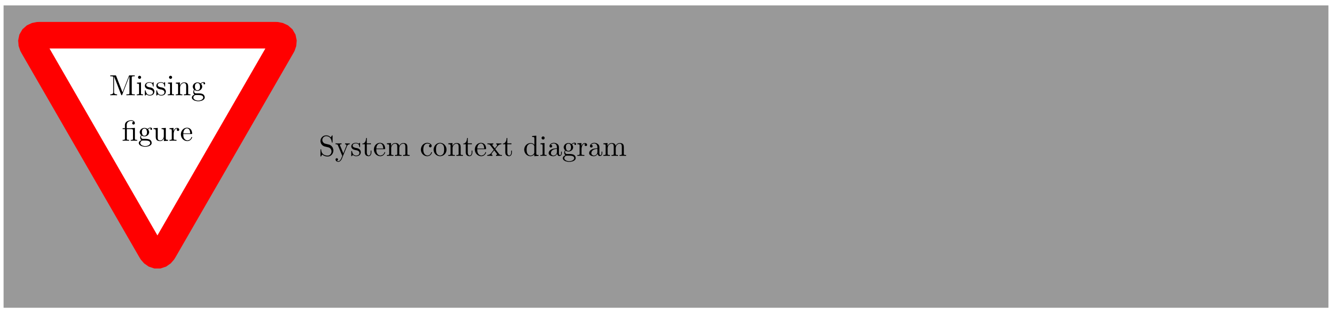

Figure 6: A component diagram for the WHAT? client, displaying the use of a gateway and backend.

caption

In order to provide seamless, flexible, reliable, and secure integration with the services, the clients exhibit two primary architectural patterns: gateways and backends for frontends.

Gateway Pattern

The gateway architectural pattern employs an intermediate service to mediate requests which pass from client devices to the services within a system, rather than having client devices communicate directly with the services. This can be seen in Figure ???, where the .

finish sen

The gateway architectural pattern provides the following advantages.

- Cross-cutting concerns, such as authentication, and data transformations, can be handled in a uniform manner for all client devices, simplifying the implementation of each user interface [16].
- The communication protocol used by requests can be changed before being forwarded to services, allowing for client devices to communicate using web-based protocols (e.g. HTTP/HTTPS) with services that are implemented to communicate using non-web-based protocols, such as QUIC [17][16].
- As all requests are passed through a single location before entering the internal system where the services lie, the gateway pattern facilitates the application of performance and security measures, such as load balancing and request filtering.
- Requests to different services can be coalesced to reduce data ingress and egress for client devices, improving performance. For example, if a patient's mobile device was loading the patient's information and appointments, the two separate requests to the Patient Information Service and Appointments Service could be combined into a single request to the respective gateway node, reducing the required network communication overhead [16].

The primary disadvantages of the gateway architectural pattern is the increased architectural complexity and the potential performance concerns surrounding the gateway services becoming a bottleneck or single point of failure. As the gateways used in the chosen architecture in this report have potential to be stateless, they can be replicated to mitigate the issues of performance and reliability; however, the issue of increased architectural complexity cannot be mitigated. Overall, the advantages of the using gateways for the target system's architecture outweigh the disadvantages.

Backend For Frontend Pattern

The backend for frontend (BFF) architectural pattern is the application of one intermediary service (the backends) per client-side user interface (the frontends).

While similar to the gateway pattern, and often considered synonymous [18], the BFF pattern is applied within the chosen architecture for this report for different reasons to the gateway pattern, which are as follows.

- The development of frontend applications using the humble object pattern is greatly facilitated, as the business logic for the applications are contained within the backend services instead. The humble object pattern is the separation of logic from frontend components and increases the testability of such components; the application of the humble object pattern is generally considered to be good software-engineering practice [19].
- Each BFF backend can be tailored specifically to its respective frontend, allowing for platform-specific intricacies to be handled cleanly, in turn facilitating development, and improving performance and maintainability. This is in contrast to the gateway pattern which uses a single backend for all frontends.

The trade-offs of using both the BFF and gateway pattern in the target system's architecture are discussed in Section ??? of this report.

4.8 Authentication

5 Implementation

deploy or different nodes to ensure failures

6 Evaluation

Data Loss in Event-Driven Styles

- Ambulance & GP service can lose info (responsibility down to devs to req-reply)

- Patient Info service priority queues do not back up info - Tricky to fix, but maybe just better to ignore, as information such as blood pressure can quickly be regathered

we were thinking about tw separate info svcs, so maybe argue the case as to why / why not

BFF & Gateway Patterns

- using both rather than just one might be considered unusual - reuse functionality through gateway (e.g. security filters) - some can't be (e.g. req coalescing) - Combining both is beneficial

SOA is not **that** important in ar arch, but instead we shoul focus on boundaries (see Clean Architectru book ch 27 [20])

7 Conclusion

Overall, this report provides an overview of an architecture for the system described in Appendix A, along with a detailed breakdown of the system's requirements, an implementation plan, and a critical evaluation of the proposed architecture.

A wide variety of both common and unusual architectural features, patterns, and styles have been applied to shape the proposed architecture towards the needs of the target system, including the use of the simple client-server pattern (see §4.7) and a multilevel priority queue (see §4.4).

experienc of group members

References

[1] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson, 2017.

[2] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly, 2020.

[3] "Microsoft Teams," Accessed: 2022-03-25. [Online]. Available: https://www.microsoft.com/en-gb/microsoft-teams/group-chat-software

[4] "Cloud Storage for Works and Home, Microsoft Teams," Accessed: 2022-03-25. [Online]. Available: https://www.google.com/intl/en-GB/drive/

[5] "LaTeX - A document preparation system," Accessed: 2022-03-25. [Online]. Available: https://www.latex-project.org/

[6] "GitHub," Accessed: 2022-03-25. [Online]. Available: https://github.com/

[7] "Lucidchart," Accessed: 2022-03-27. [Online]. Available: https://www.lucidchart.com/

[8] "Diagram Software and Flowchart Maker, diagrams.net," Accessed: 2022-03-27. [Online]. Available: https://www.diagrams.net/

[9] "The C4 model for visualising software architecture," Accessed: 2022-03-23. [Online]. Available: https://c4model.com/

[10] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly, 2020, ch. 13, pp. 163–177.

[11] "SOA (Service-Oriented Architecture), IBM," Accessed: 2022-03-25. [Online]. Available: https://www.ibm.com/uk-en/cloud/learn/soa

[12] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson, 2017, ch. 30, pp. 277 – 283.

[13] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* O'Reilly, 2021, ch. 9, pp. 321–383.

[14] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating Systems Concepts.* Wiley, 2014, ch. 5.3.5, pp. 166 – 167.

[15] "Choose a subscription type, Cloud Pub/Sub, Google Cloud," Accessed: 2022-03-26. [Online]. Available: https://cloud.google.com/pubsub/docs/subscriber

[16] "The API gateway pattern versus the Direct client-to-microservice communication, Microsoft," Accessed: 2022-03-25. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern

[17] "QUIC, a multiplexed transport over UDP," Accessed: 2022-03-25. [Online]. Available: https://www.chromium.org/quic/

[18] "Pattern: API Gateway / Backends for Frontends, Microservices.io," Accessed: 2022-03-25. [Online]. Available: https://microservices.io/patterns/apigateway.html

[19] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson, 2017, ch. 23, pp. 211 – 215.

[20] ——, *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson, 2017, ch. 27, pp. 239 – 247.

A Scenario

You are required to develop an architecture for a patient digital appointments and management system.

This system should act as a central point for creating and delivering patient appointments as well as storing patient data. Digital appointments should be created via this system, allowing a GP to ascertain and monitor vital patient data. The patient's own data should also be stored in the system allowing for a full 360 degree view of patient information. This system is being built in order to alleviate pressure on the health service.

Each person resident in Scotland is registered with a General Practitioner (GP) practice. The health care service will maintain a record of each person's name, a unique health care id, date of birth, contact details, information on next of kin, the GP practice they are registered with and medical history. Patient records contain highly sensitive information. A patient may be seen via digital appointments or at the practice by a nurse (for minor injuries or tests) or a GP. A patient can only access the GP practice if they are referred to after a digital appointment, except in case of emergencies when they can either call an ambulance or go directly to the accident and emergency (A&E) department of a hospital. An ambulance crew will either treat a patient on site or take them to a hospital. A GP will either treat a patient themselves or refer the patient to a hospital run by the local health board, except when the required service is not available locally. Each appointment or treatment will result in an entry in the person's medical history, which should be accessible across different health boards and services. When a patient has been treated at a hospital or by an ambulance crew, a notification should be sent to their GP to flag up the new entry.

The GP should be able to generate digital appointments for patients. These appointments should allow the GP (via the patient's personal device) to obtain vital information about the patient – namely their blood pressure, heart rate, heart rate variability, oxygen saturation and respiration rate. This data should be generated in real-time and stored securely in the patient information system. The system should provide different functionalities depending on the category of user. A patient must be able to use the service to make appointments at their GP practice. They should also be able to see all the vital data generated by all their digital appointments in the system. GPs should be able to access the full record of any patient without delay, initiate appointments and add entries to the patient record. They should also be able to order one or more tests for the patient within the practice or refer the patient to a hospital, either within or outside the local health board. Nurses can see a limited part of the patient record and add entries relating to treatment, tests or test results. Practice administrators can make or cancel appointments for patients and produce statistical reports on the performance of the practice without accessing details of any patient.

An ambulance service administrator must be able to log calls to the service and dispatch ambulances to patients who require them. This aspect of the service is highly time sensitive. They can also update patients' records according to the service delivered. Ambulance crews indicate when each callout has been dealt with. Hospital doctors can view all the details of any patient they see and add entries on diagnoses made and treatment given.

A well-designed and implemented system should also support the following features:

- An intuitive UI appropriate to the user category,
- Access authentication for different categories of users and restriction of available information and functionality accordingly,
- Support for multiple types of devices,
- Concurrent access,
- Support for several different views and analyses over the data,
- Validation of input data where applicable,
- Generation of digital appointments and using patients' devices functionalities to extract the necessary patient data, and
- Deal with potential uncertainties when patients' devices cannot extract this data.