Software Architecture Course Notes

Semester 1, 2022

Brae Webb & Richard Thomas

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Contents

Software Architecture

February 21, 2022

Brae Webb & Richard Thomas

1 Introduction

An introduction to Software Architecture would be incomplete without the requisite exploration into the term 'software architecture'. The term is often overloaded to describe a number of completely detached concepts. The overloaded nature of the term makes an introduction quite challenging. Martin Fowler wrestles with this difficultly in his talk on "Making Architecture Matter"[1]. In the talk Fowler settles on the slightly vague definition from Ralph Johnson [**?**]:

> **Definition 1. Software Architecture**
> The important stuff; whatever that is.

In this course, we will try to narrow the scope slightly. We need a definition which encompasses the numerous practical strategies which you need to survive and thrive in industry life. This definition should not be attributed to the term 'Software Architecture'; that term is much too broad to define succinctly. This is purely the definition used to provide an appropriate scope for the course.

> **Definition 2. Software Architecture: The Course**
> The set of tools, processes, and design patterns which enable me to deliver high quality software.

2 High Quality Software

We assume that as software engineers you wish to deliver high quality software systems. What makes life interesting[2] is that *quality*, like *beauty*, is in the eye of the beholder. As a diligent and enthusiastic software engineer, you may think that *high quality* means well designed software with few defects. On the other hand, your users may think that *high quality* means an engaging user experience, with no defects. While your project sponsor, who is funding the project, may think that *high quality* means the software includes all the features they requested and was delivered on-time and on-budget. Rarely is there enough time and money available to deliver everything to the highest standard. The development team has to balance competing expectations and priorities to develop a software system that is a good compromise and meets its key goals.

From the perspective of designing a software architecture, competing expectations provides what are sometimes called *architectural drivers*.

2.1 Functional Requirements

A seemingly obvious driver is the functional requirements for the software system, i.e. what the software should do. If you do not know what the software is meant to do, how can you design it? You do not need an extensive and in-depth description of every small feature of the software, but you do need to know

[1] https://www.youtube.com/watch?v=DngAZyWMGR0

[2] As in the apocryphal Chinese curse "May you live in interesting times."

what problem the software is meant to solve, who are the key users, and what are the key features of the software. Without this information, you do not know what style of architecture is going to be appropriate for the software system.

For example, consider an application that allows users to write and save notes with embedded images and videos. Say the decision was made to implement it as a simple mobile app that saves notes locally. If it is then decided that web and desktop applications are needed, allowing users to sync their notes across applications and share notes with others, the application will need to be redesigned from scratch. Knowing up-front that the software needed to support multiple platforms, and that syncing and sharing notes was important, would have allowed the developers to design a software architecture that would support this from the start.[3]

2.2 Constraints

Constraints are external factors that are imposed on the development project. Commonly, these are imposed by the organisation for whom you are building the software system. The most obvious constraint is time and budget. A sophisticated software architecture will take more time, and consume more of the total budget, than a simple but less flexible architecture.

Other common constraints are technology, people, and the organisation's environment. Technology constraints are one of the most common set of constraints that affect the design of the architecture. Even if it is a "greenfields" project[4] there will usually be restrictions on choices of technology that may or may not be used. For example, if all of the organisation's existing applications are running on the Google cloud platform, there will probably be a restriction requiring all new applications to be built on the same platform to avoid the overheads of working with different cloud providers.

People constraints takes into consideration the skills that are available within the organisation and the availability of developers for the project. Designing an architecture that requires skills that are not available, will add an overhead to the project's development cost. If contractors can be hired, or training is available, these may reduce the overhead but the decision needs to be made based on the risks and benefits.

The organisation's environment may influence other constraints, or add new constraints. An organisation that encourages innovation may be flexible in allowing some technology constraints to be broken. If the project is of strategic value to the business, there may be options to negotiate for a larger budget or to adopt a new technology[5], if they could lead to a more robust solution with a longer lifespan. Politics can introduce constraints on architectural choices. If there is an influential group who promote a particular architectural style, it may be difficult or impossible to make different choices.

2.3 Principles

Principles are self-imposed approaches to designing the software. Typically these are standards that all developers are expected to follow to try to ensure the overall software design is consistent. From a programming perspective, coding standards and test coverage goals are examples of principles that all developers are expected to follow. Architectural principles typically relate to how the software should be structured and how design decisions should be made to work well with the software architecture. Consequently, principles usually do not influence the architecture[6], rather the architecture will influence which principles should be prioritised during software design.

As an example, if the software architecture is designed to be scalable to accommodate peaks in load, then an architectural principle might be that software components should be stateless to allow them to

[3] This is different to building a quick-and-dirty prototype to explore options. That is a valid design process, and in that case minimal effort will be put into creating a well-designed app.

[4] This refers to the idea that it is a new project and is not limited by needing to conform to existing system's or expectations.

[5] I have consulted with organsiations who have adopted new, and risky at the time, technologies to potentially gain business benefits like easier extensibility.

[6] The exception to this is if some principles are constraints enforced by the organisation.

be easily replicated to share the load. Another principle might be that an architecture that relies on an underlying object model, will adopt the SOLID design principles [**?**].

2.4 Quality Attributes

While the functional requirements specify what the software should do, non-functional requirements specify properties required for the project to be successful. These non-functional requirements are also termed *quality attributes*.

Often quality attributes are specified by phrases ending in -ility. Medical software needs *reliability*. Social media needs *availability*. Census software needs *scalability*.

Below is a collection of non-exhaustive quality attributes to help give you an idea of what we will be looking at in this course.

Modularity Components of the software are separated into discrete modules.

Availability The software is available to access by end users, either at any time or on any platform, or both.

Scalability The software is simultaneously usable by a large amount of end users.

Extensibility Features or extensions can be easily added to the base software.

Testibility The software is designed so that automated tests can be easily deployed.

Quality attributes are one of the main focuses of a software architect. Quality attributes are achieved through architecture designed to support them. Likewise, software architecture quality and consistency is achieved by principles put in place by a software architect and the development team.

Architects are responsible for identifying the important attributes for their project and implementing architecture and principles which satisfies the desired attributes.

2.5 Documentation

The importance of these architectural drivers means they need to be documented[7]. The extent and format of the documentation will depend on the size of the project, team organisation, and the software engineering process being followed. The larger the team, or if the team is distributed between different locations, the more important it is that the documentation is well-defined and easily accessible. The documentation may be stored as notes in a wiki or as a collection of user stories and other notes. Or, it could be an extensive set of requirements and design specifications following a standard like ISO/IEC/IEEE 15289[8].

3 Attributes in Tension

One of the defining characteristics of quality attributes is that they are often in conflict with each other. It is a valiant yet wholly impractical pursuit to construct software which meets all quality attributes.

The role of a software architect is to identify which quality attributes are crucial to the success of their project, and to design an architecture and implement principles which ensure the quality attributes are achieved.

[7] Documentation is the castor oil of programming, managers know it must be good because programmers hate it so much [**?**].

[8] https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:15289

The first law of software architecture, as defined by Richards [**?**], reflects the difficultly in supporting multiple quality attributes.

Definition 3. The First Law of Software Architecture

Everything in software architecture is a trade-off.

Galster and Angelov [**?**] define this as 'wicked architecture'. They identify the 'wicked' nature of architecture as the main difficultly in teaching the subject.

Definition 4. Wicked Architecture

There are often no clear problem descriptions, no clear solutions, good or bad solutions, no clear rules when to "stop" architecting and mostly team rather than individual work.

They stress that "In contrast, other software engineering topics such as programming lead to solutions that can be checked and measured for correctness. Architecture may be the worst-case scenario when it comes to fuzziness in software engineering".

Despite this difficultly, in this course we intend to expose you to a number of case studies, architectures, and tools which aim to give you experience in tackling the trade-offs involved in software architecture.

4 The World Today

Software architecture today is more important than ever. The importance of architecture can be considered a result of *expectations* and *infrastructure*. Today we expect our software to be available 24/7. To exemplify this point, in October last year Facebook went offline for 6-7 hours out of the 8760 hours of the year. Those 6-7 hours of downtime resulted in mass media coverage, $60 million loss of revenue, and a 5% drop in company shares which caused Zuckerberg's wealth alone to drop $6 billion. Interestingly, the outage also caused other sites such as Gmail, TikTok, and Snapchat to slowdown.

This is a massive change in public expectations for software availability. As recently as 2017, human resources at UQ would monopolise the university's computing resources for an evening each fortnight to calculate the payroll. Nowadays that lack of availability from even the university's software systems would be completely unacceptable. The change in expectations has forced developers to adapt by designing architectures capable of supporting this heightened up-time.

In addition to shifting expectations, developers now have access to a range of Infrastructure as a Service (IaaS) platforms. IaaS empowers developers to quickly and programmatically create and manage computing, networking, and storage resources. In part, this enables individual developers to support up-times comparable to tech giants. Of course, to be able to support these up-times software has increased in overall complexity. A website is now commonly spread over multiple servers, marking a change from centralised systems to distributed systems.

5 Conclusion

You should now have some understanding of what software architecture is and that it is, at least in our view, important. Let's return to our definition for the course.

Definition ??. Software Architecture: The Course

The set of tools, processes, and design patterns which enable me to deliver high quality software.

In practical terms, what does this mean you should expect from the course? First, you will learn how to communicate your visions of software architecture through *architectural views*. From there, you will use

quality attributes such as extensibility, scalability, etc. to motivate the introduction of common architectural patterns, processes, and tooling which support those attributes.

For example, you can expect extensibility to motivate an introduction to plugin-based architectures. You can expect scalability to motivate our introduction to load balancers. And testability to motivate a look into A/B testing practices.

You can view the planned outline for the course on the course website[9]. All the course material can be found on GitHub[10]. If you think you can explain a concept more succinctly, or find a typo, you are encouraged to submit a pull request to help improve the course. We really hope that you enjoy the course and, perhaps more importantly, benefit from it in your careers as software development professionals!

[9] https://csse6400.uqcloud.net/

[10] https://github.com/CSSE6400/software-architecture

Layered Architecture

Software Architecture

February 21, 2022

Richard Thomas & Brae Webb

1 Introduction

In the beginning developers created the *big ball of mud*. It was without form and void, and darkness was over the face of the developers[11]. The big ball of mud is an architectural style identified by it's lack of architectural style [**?**]. In a big ball of mud architecture, all components of the system are allowed to communicate. If your GUI code wants to ask the database a question, it will write an SQL query and ask it. Likewise, if the code which primarily talks to the database decides your GUI needs to be updated a particular way, it will do so.

The ball of mud style is a challenging system to work under. Modifications can come from any direction at any time. Akin to a program which primarily uses global variables, it is hard, if not impossible, to understand everything that is happening or could happen.

Aside

Code examples in these notes are works of fiction. Any resemblance to a working code is pure coincidence. Having said that, python-esque syntax is often used for it's brevity. We expect that you can take away the important points from the code examples without getting distracted in the details.

```
import gui
import database

button = gui.make_button("Click me to add to counter")
button.onclick(e =>
    database.query("INSERT INTO clicks (time) VALUES {{e.time}}"))
```

Figure 1: A small example of a *big ball of mud* architecture. This type of software is fine for experimentation but not for any code that has to be maintained. However, it does not work well at scale.

2 Monolith Architecture

And architects said, "let there be structure", and developers saw that structure was good. And architects called the structure *modularity*[12].

The monolithic software architecture is a single deployable application. There is a single code-base for the application and all developers work within that code-base. An example monolith application would

[11] Liberties taken from Genesis 1:1-2.
[12] Liberties taken from Genesis 1:3-5.

be one of the games developed by DECO2800[13] students at UQ[14]. (e.g. Retroactive[15]). A monolith should follow design conventions and be well structured and modular (i.e. it is not a big ball of mud).

Most developers are introduced to the monolith implicitly when they learn to program. They are told to write a program, and it is a single executable application. This approach is fine, even preferred, for small projects. It often becomes a problem for large, complex software systems.

2.1 Advantages

The advantages of a monolith are that it is easy to develop, deploy and test. A single code-base means that all developers know where to find all the source code for the project. They can use any IDE for development and simple development tools can work with the code-base. There is no extra overhead that developers need to learn to work on the system.

Being a single executable component, deployment is as simple as copying the executable on to a computer or server.

System and integration testing tends to be easier with a monolith, as end-to-end tests are executing on a single application. This often leads to easier debugging once errors are found in the software. All dependencies and logic are within the application.

There are also fewer issues to do with logging, exception handling, monitoring, and even scalability if it is running on a server.

2.2 Disadvantages

The drawbacks of a monolith are complexity, coupling and scalability. Being a single application, as it gets larger and more complex, there is more to understand. It becomes harder to know how to change existing functionality or add new functionality without creating unexpected side effects. A catch phrase in software design and architecture is to build complex systems, but not complicated systems. Monoliths usually become complicated as they grow to deliver complex behaviour.

Related to complexity is coupling, with all behaviour implemented in one system there tends to be greater dependency between different parts of the system. The more dependencies that exist, the more difficult it is to understand any one part of the system. This means it is more difficult to make changes to the system or to identify the cause of defects in the system.

A monolith running on a server can be scaled by running it on multiple servers. Because it is a monolith, without dependencies on other systems, it is easy to scale and replicate the system. The drawback is that you have to replicate the entire system on another server. You cannot scale components of the system independently of each other. If the persistence logic is creating a bottleneck, you have to copy the entire application on to another server to scale the application. You cannot user servers that are optimised to perform specialised tasks.

3 Layered Architecture

And architects said, "let there be an API between the components, and let it separate component from component[16]".

The first architectural style we will investigate is a layered architecture. Layered architecture (also called multi-tier or tiered architecture) partitions software into specialised clusters of components (i.e. *layers*)

[13] https://my.uq.edu.au/programs-courses/course.html?course_code=DEC02800
[14] https://www.uq.edu.au/
[15] https://github.com/UQdeco2800/2021-studio-7
[16] Liberties taken from Genesis 1:6-8.

and restricts how components in one layer can communicate with components in another layer. A layered architecture creates superficial boundaries between the layers. Often component boundaries are not enforced by the implementation technology but by architectural policy.

The creation of these boundaries provides the beginnings of some control over what your software is allowed to do. Communication between the component boundaries is done via well-specified *contracts*. The use of contracts results in each layer knowing precisely how it can be interacted with. Furthermore, when a layer needs to be replaced or rewritten, it can be safely substituted with another layer fulfilling the contract.

3.1 Standard Form

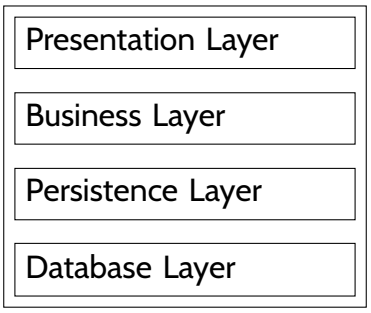


Figure 2: The traditional specialised components of a layered architecture.

The traditional components of a layered architecture are seen in Figure **??**. This style of layered architecture is the four-tier architecture. Here, our system is composed of a presentation layer, business layer, persistence layer, and database layer.

The presentation layer takes data and formats it in a way that is sensible for humans. For command line applications, the presentation layer would accept user input and print formatted messages for the user. For traditional GUI applications, the presentation layer would use a GUI library to communicate with the user.

The business layer is the logic central to the application. The interface to the business layer is events or queries triggered by the presentation layer. It's the responsibility of the business layer to determine the data updates or queries required to fulfil the event or query.

The persistence layer is essentially a wrapper over the database, allowing more abstract data updates or queries to be made by the business layer. One advantage of the persistence layer is it enables the database to be swapped out easily.

Finally, the database layer is normally a commercial database application like MySQL, Postgres, etc. which is populated with data specific to the software. Figure **??** is an over-engineered example of a layered architecture.

```
» cat presentation.code

import gui
import business

button = gui.make_button("Click me to add to counter")
button.onclick(business.click)
```

Figure 3: An unnecessarily complicated example of software components separated into the standard layered form.

```
» cat business.code
import persistence

def click():
    persistence.click_counts.add(1)
```

```
» cat persistence.code
import db

class ClickCounter:
    clicks = 0

    def constructor():
        clicks = db.query("SELECT COUNT(*) FROM clicks")

    def get_click():
        return clicks

    def add(amount):
        db.query("INSERT INTO clicks (time) VALUES {{time.now}}")

click_counts = ClickCounter()
```

Figure 3: An unnecessarily complicated example of software components separated into the standard layered form.

One of the key benefits afforded by a well designed layered architecture is each layer should be interchangeable. A typical example is an application which starts as a command line application but can later be adapted to a GUI application by just replacing the presentation layer.

3.2 Deployment Variations

While the layered architecture is popular with software deployed on one machine (a non-distributed system), layered architectures are also often deployed to separate machines.

Each layer can be deployed as separate binaries on separate machines. A simple, common variant of distributed deployment is separating the database layer, as shown in figure **??**. Since databases have well defined contracts and are language independent, the database layer is a natural first choice for physical separation.

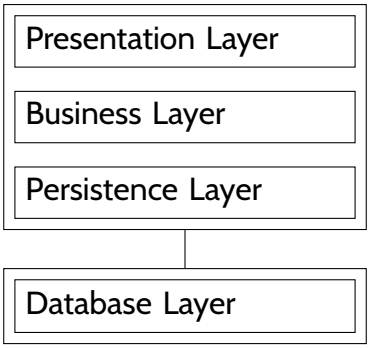


Figure 4: Traditional layered architecture with a separately deployed database.

In a well designed system, any layer of the system could be physically separated with minimal difficulty. The presentation layer is another common target, as shown in figure **??**. Physically separating the presentation layer gives users the ability to only install the presentation layer and allow communication to other software components to occur via network communication.

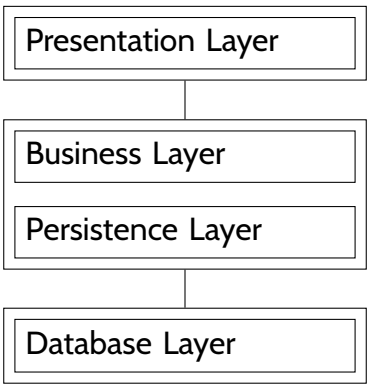


Figure 5: Traditional layered architecture with a separately deployed database and presentation layer.

This deployment form is very typical of web applications. The presentation layer is deployed as a HTML/JavaScript application which makes network requests to the remote business layer. The business layer then validates requests and makes any appropriate data updates.

Some database driven appliation generators will embedded the application logic in the database code so that all logic runs on the database server. The presentation layer is then separated from the application logic, as shown in figure **??**.

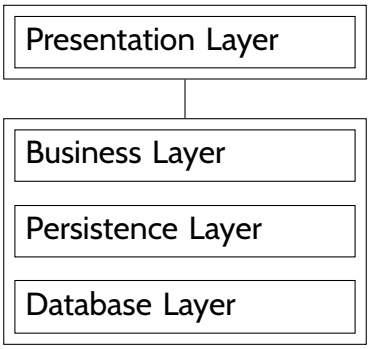


Figure 6: Traditional layered architecture with a separately deployed presentation layer.

An uncommon deployment variation (figure **??**) separates the presentation and business layers from the persistence and database layers. An updated version of our running example is given in figure **??**, the

presentation layer remains the same but the communication between the business and persistence layers is now via REST.[17]

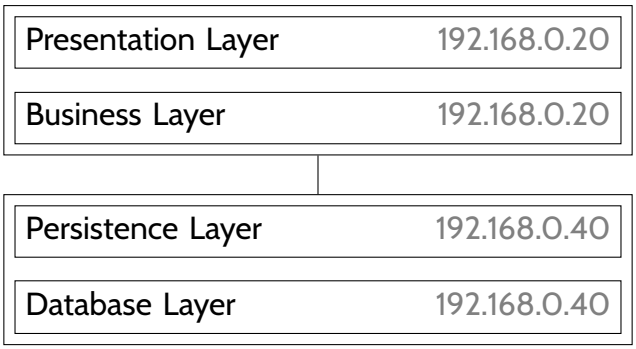


Figure 7: A contrived example of a deployment variation.

```
» cat business.code
```

```
import http

def click():
    http.post(
        address="192.168.0.40",
        endpoint="/click/add",
        payload=1
    )
```

```
» cat persistence.code
```

```
import db
import http

class ClickCounter:
    ... # as above

click_counts = ClickCounter()

http.on(
    method="post",
    endpoint="/click/add",
    action=(payload => click_counts.add(payload))
)
```

Figure 8: Code adapted for the contrived example of a deployment variation.

[17] https://restfulapi.net/

3.3 Layered Principles

Separating software into layers is intended to increase the modularity and isolation of the components within each layer. Isolation is provided by defining a public interface through which all communication with the layer is to be performed.

Definition 5. Layer Isolation Principle

Layers should not depend on implementation details of another layer. Layers should only communicate through well defined interfaces (*contracts*).

Layering should be enforced. One layer should not "reach across" another layer to access behaviour implemented in some other layer. For example, in our standard form of the layered architecture, if the presentation layer uses a component from the presistence layer, it defeats the intent of having a business layer in the architecture.

A consequence of this is chains of message passing. An extreme example would be if the presentation layer needed to display some information from the database, the presentation layer would send a message to the business layer to get the object to be displayed. The business layer would send a message to the persistence layer to retrieve the object. The persistence layer would then send a message to the database layer to load the object.

Typically, there would not be a need to send messages from the highest to lowest layer. If the business layer knew it had an up-to-date copy of the object, it would return it to the presentation layer without messaging the persistence layer. If the persistence layer had already retrieved the object from the database, it would return it to the business layer without messaging the database layer.

Definition 6. Neighbour Communication Principle

Components can communicate across layers only through directly neighbouring layers.

Layers should be hierarchical. Higher layers depend on services provided by lower layers but not vice versa. This dependency is only through a public interface, so that components in the lower layer may be replaced by another component implementing the same interface. Components in a lower layer should not use components from a higher layer, even if the layers are neighbours.

Definition 7. Downward Dependency Principle

Higher-level layers depend on lower layers, but lower-level layers do not depend on higher layers.

Downward dependency does not mean that data is not passed to higher layers. It does not even mean that control cannot flow from a lower level to a higher level. The restriction is on dependencies or usage, not on data or control flow. A lower layer should not use components from a higher layer, even through the higher layer's interface. Breaking this increases the overall coupling of the system and means it is no longer possible to replace a lower layer with another layer.

Lower layers need a mechanism to be able to notify a higher layer that something has happened, of which the higher layer needs to be aware. A common example of this is the presentation layer wants to be notified if data that it is displaying has been updated in a lower layer. The observer design pattern[18] is a common solution to this notification issue. The component responsible for displaying the data in the presentation layer implements the *Observer* interface. The object containing data that may be updated implements the *Subject* interface. The subject and observer interfaces are general purpose interfaces that do not belong to the presentation layer. The lower layer uses the observer interface to notify the presentation layer that data has changed and the presentation layer can decide whether to retrieve the new data

[18] https://refactoring.guru/design-patterns/observer

and display it. This allows the higher layer to be notified of events, without the lower layer using anything from the higher layer.

The same issue occurs with error handling and asynchronous messaging. If a component in a higher layer sends a message, through an interface, to a component in a lower layer, the component in the lower layer needs a mechanism to report errors. A simple boolean or error code return may work in some situations, but often that is not appropriate. If the message is meant to return a value, in most languages it cannot also return an error result. There may also be different types of errors that need to be communicated to the higher layer. (e.g. The call from the higher layer broke the contract specified in the interface. Or, the lower layer is encountering a transient fault and the higher layer should try again later.) Exception handling would work, if all layers are within one executable environment, but a key purpose of a layered architecture is to allow separation of the layers, so throwing an exception is not appropriate.

Callbacks[19] are used to deal with this issue for both error handling and asynchronous messaging. A component from a higher layer in the architecture passes a function as a parameter when it sends a message to a component in a lower layer. This function is called by the component in the lower layer of the architecture to report an error or to indicate that an asynchronous call has completed.

Definition 8. Upward Notification Principle

Lower layers communicate with higher layers using general interfaces, callbacks and/or events. Dependencies are minimised by not relying on specific details published in a higher layer's interface.

The subject and observer interfaces are examples of supporting logical infrastructure. Logging frameworks are another example of supporting infrastructure. Commonly, all layers will need to use the logging framework. These are typically defined in separate "layers" that can be used by any of the other layers. These are sometimes called *sidecar* or acquaintance layers, as visually they are often drawn on the side of the layered diagram.

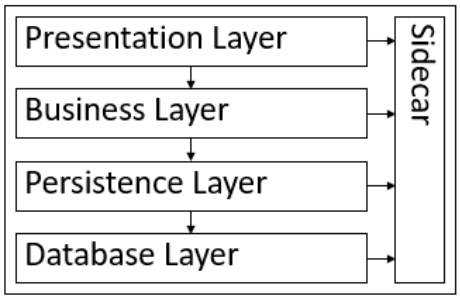


Figure 9: Layered architecture with sidecar.

Definition 9. Sidecar Spanning Principle

A sidecar layer contains interfaces that support complex communication between layers (e.g. design patterns like the observer pattern) or external services (e.g. a logging framework).

A purist design approach says that a sidecar layer may only contain interfaces. In some environments, an architecture may decide that multiple sidecars are beneficial, and may even use these for reusable components from broader organisational systems or for objects that hold data passed to higher layers. Figure **??** is an example of using sidecars for both of these purposes in a J2EE[20] application.

In the example shown in figure **??**, the servlets and action classes layer is equivalent to the presentation layer. The controller and service classes layers are a further partitioning of the business layer. The DAO

[19] https://www.codefellows.org/blog/what-is-a-callback-anyway/
[20] https://www.oracle.com/java/technologies/appmodel.html

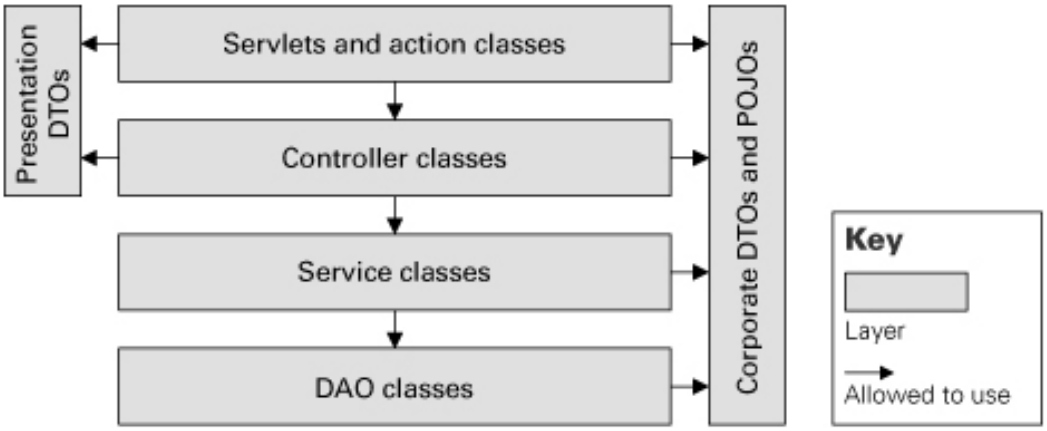


Figure 10: Layered architecture with sidecars delivering implementation (figure 2.27 in Clements et al, 2010) [**?**].

(Data Access Objects) classes layer is equivalent to the persistence layer.

The Presentation DTOs (Data Transfer Objects) sidecar contains simple JavaBeans[21] that contain data that is to be displayed. This approach takes advantage of J2EE's mechanism that automatically populates and updates data in the presentation layer.

The Corporate DTOs and POJOs (Plain Old Java Objects) sidecar contains classes implemented by corporate-wide systems, and which are shared between systems. These provide common data and behaviour that spans multiple layers in many systems.

3.3.1 Closed/Open Layers

Some textbooks discuss the concept of closed and open layers. This is a way to descrive how communication flows between layers. Layers are categorised as either *open* or *closed*. By default layers are *closed*. Closed layers prevent direct communication between their adjacent layers, i.e. they enforce the neighbour communication principle. Figure **??** shows the communication channels (as arrows) in a completely closed architecture.

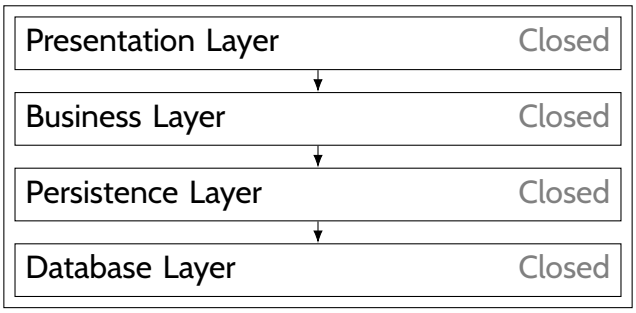


Figure 11: All layers closed requiring communication to pass through every layer.

An architecture where all layers are closed provides maximum isolation. A change to the communication contracts of any layer will require changes to at most one other layer.

Some architects will advocate that there are some situations where an *open* layer may be useful. Open layers do not require communication to pass through the layer, other layers can "reach across" the layer. The preferred approach is to use general interfaces, callbacks and/or events, as discussed in the sections describing the downward dependency, upward notification, and sidecar spanning principles. This provides mechanisms that allow data and control to flow both up and down in a layered architecture, without

[21] https://www.educative.io/edpresso/why-use-javabean

breaking the isolation principle that was the original intent of using a layered architecture. Open layers in architecture design should be avoided.

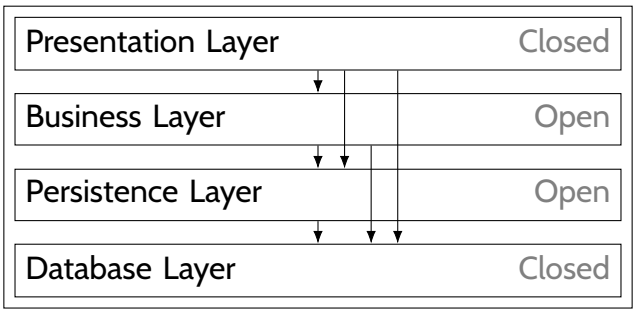


Figure 12: A wolf in layer's clothing [**?**].

3.4 Advantages

The layer isolation principle means that the implementation of a layer can be changed without affecting any other layer, as long as the interface does not change.

The layer isolation principle also means that a developer only needs to understand the public interface to use a layer, and not its implementation details.

The neighbour communication and downward dependency principles mean that if a layer changes its public interface, at most one other layer needs to change.

The upward notification and sidecar spanning principles mean that complex systems, with sophisticated flows of control and data, can be implemented while maintaining the other layered architecture design principles.

Lower layers in the architecture can be designed to deliver common services that may be reused across multiple applications. (e.g. The persistence layer can be designed to allow general purpose access to the database layer, allowing any type of database to be substituted into the system.)

Layers may be deployed on different computing infrastructure. This enables the hardware to be optimised for the types of services provided by just one layer. It also enables scaling and replication by allowing layers to be duplicated across multiple servers.

3.5 Disadvantages

Poorly designed layers will encourage developers to break the layered architecture design principles in order to get the system to work. This can lead to a system that in detail more closely resembles a big ball of mud, than a layered design.

Layering often introduces performance penalties. Requiring a chain of message passing to obtain a service from a much lower layer in the architecture adds to the cost of delivering the behaviour.

Security Principles

Software Architecture

March 7, 2022

Brae Webb

1 Introduction

One quality attribute which developers often overlook is security. Security can be the cause of a great deal of frustration for developers; there are no comfortable architectures, nor command-line tools to magically make an application secure. While the world depends on technology more than ever, and, at the same time the impacts of cyber attacks become more devastating, it has become crystal clear that security is everyone's responsibility. As users of technology, as developers, and as architects, we all need to ensure we take security seriously.

Learning, and for that matter, teaching, how to make software secure is no easy task. Every application has different flaws and risks, every attack vector and exploit is unique; managing to keep up with it all is daunting. None of us will ever be able to build a completely secure system but that is no reason to stop trying. As developers and architects, security should be an on-going process in the back of your minds. A nagging voice which constantly asks 'what if?'.

We introduce security first to set the example. As we go through this course, the principle of security will recur again and again. With each architecture introduced, we will stop and ask ourselves 'what if?'. In your future careers, you should endeavour to continue this same practice. Each feature, pipeline, access control change, or code review, ask yourself, 'what are the security implications?'.

With that said, there are some useful principles, and a handful of best practices which we will explore. But again, even if you follow these practices and embody the principles, your applications will still be hopelessly insecure, unless, you constantly reflect on the security implications of your each and every decision.

2 You

Before we even fantasise about keeping our applications secure, let's review if you are secure right now. As developers we often have heightened privileges and access, at times above that of even company CEOs. If you are not secure, then nor is anything on which you work. Let's review some of the basics.

Keep your software up to date. Are you running the latest version of your operating system? The latest Chrome, or Firefox, or god-forbid, Edge? If not, then there is a good chance you are currently at risk. Software updates, while annoying, provide vital patches to discovered exploits. You must keep your software up to date.

Use multi-factor authentication. This may be hard to explain to your grandmother but this should be obvious to software developers. One million passwords are stolen every week [**?**]. If you do not have some sort of multi-factor authentication enabled, hackers can access your account immediately after stealing your password.

Use a password manager. Following from the startling statistic of a million stolen passwords per week, we must seriously consider how we use passwords. Our practices should be guided by the fact that at least one service we use likely stores our password insecurely. We should assume that our password will be compromised. What can we do about this? The first thing is to avoid password reuse, one password per service. Of course, humans have very limited memory for remembering good passwords. So go through and update your passwords with randomly generated secure passwords, and then store them in a password manager.

3 Principles of Securing Software

Okay, now that we are not a security risk ourselves, we can start considering how to secure the software we develop. Before looking at pragmatic practices, we will develop a set of guiding principles. These principles are far from comprehensive but they provide a useful foundation to enable discussion of our security practices. The principles presented in this course are derived from Saltzer and Schroeder [**?**], Gasser [**?**], and Viega and McGraw [**?**]. Some principles have been renamed for consistency and clarity. Refer to Appendix **??** for the comprehensive list of principles from these sources.

3.1 Principle of Least Privilege

> Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.
>
> — Jerry Saltzer [**?**]

The principle of least privilege was identified in 1974 by Jerry Saltzer [**?**]. This principle is mostly common sense and the benefits should be apparent. If you maintain the principle of least privilege then you minimise your attack surface by limiting the amount of damage any one user can do. This protects software from intentionally malicious actors while also minimising the damage of bugs which occur unintentionally in the software.

Example Consider a web application which lists COVID close contact locations for a state. We will assume that the locations are maintained within an SQL database. Assume that each time the tracing page is loaded, an SQL query is sent to the database to find all current close contact locations. If the developers follow the principle of least privilege, then the account used to query that data would only be able to list the current locations.

For this example, the tracing website had to be developed and rolled out quickly, as such, the developers created only one SQL user which they used for both the tracing website and the portal where the government can log new locations. This user account would have the ability to create new close contact locations, and if done poorly enough, the account might even have access to delete existing locations.

Since the developers have violated the principle of least privilege, their software is now at risk. If a malicious actor is able to gain database access via SQL injection, or, just as likely, if the software has a typo in an SQL query, the integrity of the tracing data could be jeopardised. This could be mitigated by using the principle of least privilege and creating separate user accounts for modifying and subsequently viewing the data.

Exemplar One of the primary examples of a good application of this principle is within Unix operating systems. In the Unix operating system, a sudoer (a user account which can use the `sudo` command) has a lot of destructive power. Commands running at the sudo level can do essentially anything they wish, including wiping the computer. However, a sudoer is not typically using these privileges. The user has to specify that they intend to run a command with elevated privileges, which helps avoid accidental destruction of the computer.

Fail-safe defaults The principle of fail-safe defaults is often presented on its own. Fail-safe defaults means that the default for access to a resource should be denied until explicit access is provided. We include fail-safe defaults as a property of the principle of least privilege given the strong connection.

3.2 Principle of Failing Securely

> Death, taxes, and computer system failure are all inevitable to some degree. Plan for the event.
>
> — Howard and LeBlanc [**?**]

Computer systems fail. As we will see in this course, the more complicated your software, the more often and dramatically it can be expected to fail. The principle of failing securely asks us to stash away our optimism and become a realist for a moment. When designing an architecture or a piece of software, plan for the ways your software will fail. And when your software does fail, it should not create a security vulnerability [**?**].

Example An interesting example of failing securely comes from Facebook's October 2021 outage which we discussed previously. As you may be aware, one cause of the outage was a DNS resolution issue triggered by Facebook's data centres going offline [**?**]. The DNS resolution issue meant that the internal Facebook development tools and communication channels went down as well. As you might expect, losing your tools and access to your team members makes resolving an outage far more difficult.

Early reports of the incident indicated that the outage of Facebook's internal infrastructure also meant employees were locked out of the data centres. While it seems that developers were locked out of their buildings, the data centres were not affected. Nevertheless, it is interesting to consider whether an internal outage should, somewhat understandably, lock access to the data centres.

This example highlights the key difference between a system which *fails safely*[22] and a system which *fails securely*. In a fail *safe* system, an internal outage would allow physical access to the data centre to enable maintenance to fix the problem. Whereas in a fail *secure* system, the outage would cause the data centre to lock and prevent access to maintain the security of the data within. There isn't one correct way to deal with failure. While in this case it would have helped Facebook resolve the issue quicker, if a data breach occurred through an intentional outage there would be equal criticism.

Regardless of the security policy you choose, it is always important to prepare for failure and weigh the pros and cons of each policy.

3.3 Principle of KISS

> Simplicity is the ultimate sophistication
>
> — Leonardo Da Vinci[23]

We will keep this principle simple. The principle of Keep it Simple Stupid (KISS) is needed as complicated software or processes are, more often than not, insecure. Simple means less can go wrong.

3.4 Principle of Open Design

> One ought to design systems under the assumption that the enemy will immediately gain full familiarity with them.
>
> — C. E. Shannon [**?**]

The principle of open design, also known as Kerckhoffs' principle, stresses that security through obscurity, or security through secrecy, does not work. If the security of your software relies on keeping certain implementation details secret then your system is not secure. Of course, there are some acceptable secrets such as private keys, however, these should still be considered a vulnerability. Always assume that if an implementation detail can be discovered, it will be. There is software constantly scanning the internet for open ports, unpublished IP addresses, and routers secured by the default username and password.

[22] No relation to fail-safe defaults.
[23] maybe

Example An example which immediately springs to mind is our first and second year assignment marking tools. To date, I am not aware of the tools being exploited, however they are currently vulnerable. The tools currently rely on students not knowing how they work. There are ways to create 'assignment' submissions which pass all the functionality tests for any given semester. Fortunately, the threat of academic misconduct is enough of a deterrent that it has yet to be a problem.

The example does illustrate why the principle of open design is so frequently violated. In the short-term security through obscurity can work, and work well, but it is a long-term disaster waiting to happen. It is also common place to violate the principle slightly by trying to build systems which do not rely on secrecy but keeping the systems secret 'just in case'. In many situations this is appropriate, however, a truly secure system should be open for community scrutiny.

3.5 Principle of Usability

The final principle we should explore is the principle of usability, also known as 'psychological acceptability'. This principle asks that we have realistic expectations of our users. If the user is made to jump through numerous hoops to securely use your application, they will find a way around it. The idea is that the security systems put in place should, as much as possible, avoid making it more difficult to access resources.

Example The example for this principle includes a confession. The university has implemented a multi-factor authentication mechanism for staff.[24] Unfortunately, there is a bug in the single sign-on which means that MFA is not remembered causing the system to re-prompt me at every *single* log in. A direct consequence of this inconvenience is that I have installed software on all my devices which automatically authenticates the MFA, bypassing, in part, the intended additional security.

The university through violating the principle of usability has made it more difficult for users to be secure than insecure. As predicted by the principle, the inconvenience leads straight to bypassing. Violation of this principle often has relatively minimal *visible* impacts, which results in the principle not being considered as often. The long-term implications are that what was an annoyance circumvented by your users, may become the cause of a major security breach long after the security feature was implemented.

4 Practices of Secure Software

With some of the guiding principles of secure software now covered, there are a few useful practices which can be employed to help secure our systems.

4.1 Encryption

In the arms race between hackers and security practitioners *encryption* was one of the first weapons wielded. Encryption is the act of encoding data in a way which is difficult to decode. The most notable early example of encryption was the Enigma Machine in the 1930s, if you trace the history of cryptography back far enough you will eventually end up in World War II Germany [**?**].

While encryption is outside of the scope of this course, it is well worth having knowledge of available encryption techniques. You should not attempt to design your own encryption algorithms, unless you are an expert in security and encryption. Rather, you should use existing well-known and well-tested encryption algorithms. In the design of a secured software system encryption can play many roles.

[24] coming soon to students

4.2 Sanitisation

Another practice which you should endeavour to utilise is sanitisation. If we assume that you cannot tell the difference between your user and a potential attacker, then you cannot trust your user. If you cannot trust your user, then you should not trust the data they give you. One of the oldest and most common forms of attacks is user input injection. In such an injection attack, a user intentionally or unintentionally provides input which damages your system.

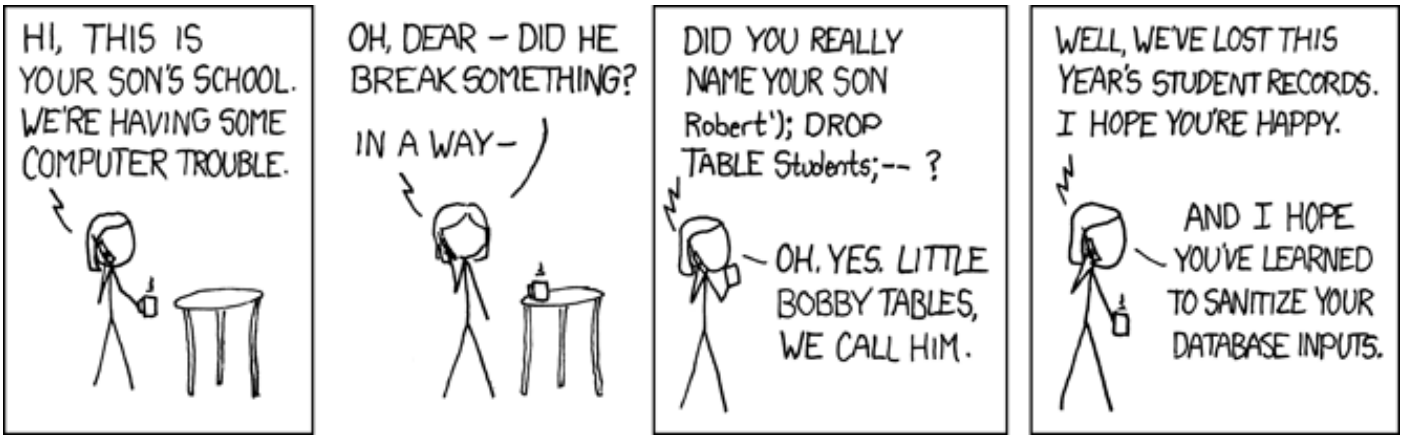


Figure 13: https://xkcd.com/327/

When done maliciously, attackers enter characters or sequences of characters which are commonly used to escape a string. The input which follows would then be run as code on the victim's system. The method for preventing against injection attacks is called *sanitisation*, which is simply a process of removing dangerous or unnecessary characters from a user's input.

4.3 Firewalls

A firewall is a piece of networking software which is designed to protect a computer system against unwanted traffic. Firewalls scan incoming and outgoing network packets and are able to drop the packet if it is deemed unwanted. Firewalls can be configured by a set of rules defining which traffic is unwanted. A firewall rule could specify that a computer system drop all packets destined for port 80 or all packets not destined for port 80. They may also be configured to support more advanced packet filtering.

From our perspective the primary advantage of a firewall is to prevent traffic which we did not intend. If we are hosting a web server, only allowing traffic on port 80 and port 443 is desirable, and prevents users accessing a database on the server which we forgot to password protect. The principle for firewalls is to block by default and allow when required.

4.4 Dependency Management

Modern software has a lot of dependencies. Each new dependency a software adopts is a new potential attack surface. Relying on software which is automatically updated is dangerous, the authors can at any point inject malicious code into your system. There is a particularly interesting recent example of dependency injection which is worth reading about [**?**].

One simple practice to help prevent against dependency injection is by using lock files. Most package managers generate lock files when installing software. Lock files keep track of the exact version of a library that is installed. If you ask a package manager to install version >=2.4.0 of a library and it actually downloads version 2.5.6, this will be tracked in the lock file. Additionally, lock files track the hash of the dependency so that if someone attempts a dependency injection, the code change will be noticed by the package

manager. Lock files should be tracked in version control and updated periodically when a safe and tested version of a dependency is released.

5 Conclusion

We have looked at a few guiding principles for designing and developing secure software. This list of principles is incomplete, security requires constant consideration. Software security is an ongoing arms race for which we are all currently unqualified to enlist. Secure yourself as best you can and, when you are in doubt, consult with or hire an expert.

A Original Security Design Principles

Saltzer and Schroeder

1. Economy of mechanism ..... Principle of KISS
2. Fail-safe defaults ..... Principle of Least Privilege
3. Complete mediation ..... Not covered
   Access to resources must *always* be authorised.
4. Open design ..... Principle of Open Design
5. Separation of privilege ..... Not covered
   No one user account or role should hold too much power.
   Consider multi-role authentication where appropriate.
6. Least privilege ..... Principle of Least Privilege
7. Least common mechanism ..... Not covered
   Minimise the amount of resources shared between users.
8. Psychological acceptability ..... Principle of Usability

Gasser

1. Consider Security from the Start ..... Implicit
2. Anticipate Future Security Requirements
3. Minimise and Isolate Security Controls
4. Enforce Least Privilege ..... Principle of Least Privilege
5. Structure the Security-Relevant Functions
6. Make Security Friendly ..... Principle of Usability
7. Do Not Depend on Secrecy for Security ..... Principle of Open Design

Viega and McGeaw

1. Secure the Weakest Link
2. Practice Defense in Depth
3. Fail Securely . . . . . Principle of Failing Securely
4. Follow the Principle of Least Privilege . . . . . Principle of Least Privilege
5. Compartmentalise
6. Keep It Simple . . . . . Principle of KISS
7. Promote Privacy
8. Remember That Hiding Secrets is Hard
9. Be Reluctant to Trust
10. Use Your Community Resources

Architectural Views

Software Architecture

March 7, 2022

Richard Thomas & Brae Webb

1 Introduction

Understanding software is hard. It is often claimed that reading code is harder than writing code[25]. This principle is used to explain a programmers' innate desire to constantly rewrite their code from scratch. If software is hard to understand, then software architecture is near impossible. Fortunately, architects have developed a number of techniques to manage this complexity.

A software architecture consists of many dimensions. Programming languages, communication protocols, the operating systems and hardware used, virtualisation used, and the code itself are a subset of the many dimensions which comprise a software architecture. Asking a programmer's monkey brain to understand, communicate, or document every dimension at once is needlessly cruel. This is where architectural views come in.

Architectural views, or architectural projections, are a representation of one or more related aspects of a software architecture. Views allow us to focus on a particular slice of our multi-dimensional software architecture, ignoring other irrelevant slices. For example, if we are interested in applying a security patch to our software then we are only interested in the view which tells us which software packages are used on each host machine.

The successful implementation of any architecture relies on the ability for the architectural views to be disseminated, understood, and implemented. For some organisations, the software is simple enough, or the team small enough, that the design can be communicated through word of mouth. As software becomes increasingly complex and developers number in the thousands, it is critical for design to be communicated as effectively as possible. In addition to facilitating communication, architectural views also enable architectural policies to be designed and implemented.

2 4+1 Views

Philippe Kruchten was one of the earliest to advocate the idea of using views to design and document software architectures. In "4+1 View Model of Software Architecture" [**?**] he describes five different views. These are logical, process, development, physical, and scenario views, which are summarised below.

Logical How functionality is implemented, using class and state diagrams.

Process Runtime behaviour, including concurrency, distribution, performance and scalability. Sequence, communication and activity diagrams are used to describe this view.

Development The software structure from a developer's perspective, using package and component diagrams. This is also known as the implementation view.

Physical The hardware environment and deployment of software components. This is also known as the deployment view.

Scenario The key usage scenarios that demonstrate how the architecture delivers the functional requirements. This is the '+1' view as it is used to validate the software architecture. This is also known as the use case view, as high-level use case diagrams are used to outline the key use cases and actors.

[25] Though evidence suggests that an ability to read and reason about code is necessary to learn how to program well [**?**] [**?**].

The experience which led to the development the 4+1 View Model was developing the air traffic control management system for Canada. The system provides an integrated air traffic control system for the entire Canadian airspace. This airspace is about double the size of the Australian airspace and borders the two busiest airspaces in the world. The project's architecture was designed by a team of three people led by Philippe. Development was done by a team of 2500 developers from two large consulting companies. The project was delivered on-time and on-budget, with three incremental releases in less than three years[26]. This project also developed many of the ideas that led to the Rational Unified Process [**?**].

3 *Software Architecture in Practice* Views

The seminal architecture book, *Software Architecture in Practice* [**?**], categorises architectural views into three groups. These three groups each answer different questions about the architecture, specifically:

Module Views How implementation components of a system are structured and depended upon.

Component-and-connector Views How individual components communicate with each other.

Allocation Views How the components are allocated to personnel, file stores, hardware, etc.

3.1 Module Views

Module views are composed of modules, which are static units of functionality such as classes, functions, packages, or whole programs. The defining characteristic of a module is that it represents software responsible for some well-defined functionality. For example, a class which converts JSON to XML would be considered a module, as would a function which performs the same task.

The primary function of module views is to communicate the dependencies of a module. Rarely does software work completely in isolation, often it is constructed with implicit or explicit dependencies. A module which converts JSON to XML might depend upon a module which parses JSON and a module which can format XML. Module views make these dependencies explicit.

Module views focus on the developer's perspective of how the software is implemented, rather than how it manifests itself when deployed in a computing environment.

3.2 Component-and-Connector Views

Component-and-connector views focus on the structures that deliver the runtime, or dynamic behaviour of a system. Components are units which perform some computation or operation at runtime. These components could overlap with the modules of a module view but are often at a higher level of abstraction. The focus of component-and-connector views is how these components communicate at runtime. Runtime communication is the connector of components. For example, a service which registers users to a website might have new registrations communicated via a REST[27] request. The service may then communicate the new user information to a database via SQL queries.

When we look at software architecture, component-and-connector views are the most commonly used views. They are common because they contain runtime information which is not easily automatically extracted. Module views can be generated after the fact, i.e. it is easy enough for a project to generate a UML class diagram. (Simple tools will create an unreadably complex class diagram. Tagging important information in the source code, or manually removing small details is required to end up with readable diagrams.) Component-and-connector views are often maintained manually by architects and developers.

[26] Contrast this to the United States Federal Aviation Administration's Advanced Automation System project from a similar era. The original estimate was $2.5 billion and fourteen years to implement. The project was effectively cancelled after twelve years. By then the estimate had almost tripled and the project was at least a decade behind schedule [**?**].

[27] https://www.ibm.com/cloud/learn/rest-apis

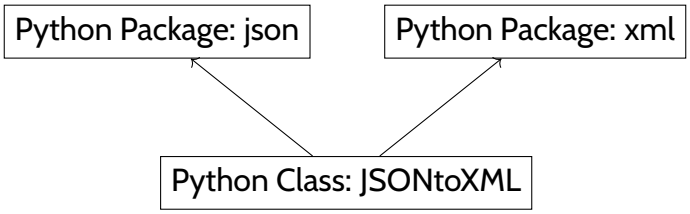
```
import json
import xml

class JSONtoXML:
    def load(self, json_file):
        with open(json_file) as f:
            data = json.load(f)
        self.data = self.convert(data)

    def export(self, xml_file):
        xml.write(xml_file, data)

    def convert(self, data: JSON) -> XML:
        ...
```

(a) Pseudo-code to convert JSON to XML



(b) An example of a module view which illustrates the dependencies of the `JSONtoXML` class

Figure 14: A simple module view of a JSON to XML program.

3.3 Allocation Views

According to Bass et al, allocation views map the software's structures to the system's non-software structures [**?**]. They include concepts such as who is developing which software elements, where are source files stored for different activities such as development and testing, and where are software elements executed. The first two points are important for project management and build management. The last point of how the software is executed on different processing nodes is important for architectural design. This is sometimes called the *deployment structure* or the software system's *physical architecture*.

Understanding the physical architecture (simplistically the hardware[28] on which the software is executed) is important when designing the software's *logical architecture*. Component-and-connector views describe the software's logical architecture. This design of the logical architecture must contain components that can be allocated appropriately to processing nodes, and these nodes must have communication links that enable the components to interact.

4 Sahara eCommerce Example

Sahara[29] eCommerce is an ambitious company who's prime business unit is an on-line store selling a wide range of products. They provide both web and mobile applications to deliver the shopping experience to customers.

[28] Whether it is virtualised or physical hardware

[29] Yes, that is intentionally a *dry* joke.

4.1 Architecturally Significant Requirements

Architecturally significant requirements (ASR) are functional or non-functional requirements, or constraints or principles, which influence the design of the system architecture. The structure of a software architecture has to be designed to ensure that the ASRs can be delivered.

Not all requirements for a system will be architecturally significant, but those that are need to be identified. Once ASRs are identified, an architecture needs to be designed to deliver them. This may require some research, and experimentation with prototypes, to determine which options are appropriate. Tests should be designed to verify that the architecture is delivering the ASRs. Ideally, these should be part of an automated test suite. This may not be possible for all tests. Regardless, the ASR tests should be executed frequently during development to provide assurance that the system will deliver the ASRs.

Inevitably, some ASRs will be discovered later in the project. The existing architecture will need to be evaluated to determine if it can deliver the new ASRs. If it can, new tests need to be added to the ASR test suite to verify delivery of the new ASRs. If the architecture is no longer suitable due to the new ASRs, a major redesign needs to be done to create a new, more suitable, architecture.

The architecturally significant requirements for the Sahara eCommerce system are:

- Customers can start shopping on one device and continue on another device. (e.g. Add a product to their shopping cart while browsing on their computer when they are bored at school. Checkout their shopping cart from their mobile phone on their way home on the bus.)
- The system must be scalable. It must cater for peaks in demand (e.g. Cyber Monday and Singles Day). It must cater for an unknown distribution of customers accessing the on-line store through web or mobile applications.
- The system must be robust. The system must continue to operate if a server fails. It must be able to recover quickly from sub-system failures.
- The system must have high availability. Target availability is "four nines"[30] up time.

The following sections will describe the physical and software architecture for this system, and demonstrate how it delivers these ASRs.

4.2 Allocation View

Figure **??** uses a UML deployment diagram as a visual representation of the physical architecture of the system as part of the allocation view. The diagram also shows some of the important software components that are deployed onto parts of the physical architecture. For simplicity, details such as load balancing and failover are not shown in this example.

There are both web and mobile applications that customers use to shop at the store. A J2EE server (e.g. TomEE[31]), running on a web server hardware platform, handles browser requests from customers, using the HTTPS protocol over the Internet. A JavaScript module called `ProductAnimator` is downloaded to the customer's browser to allow them to see interactive 3d views of products. The `ProductBrowsing` and `ShoppingCartView` components run on the J2EE server, providing those aspects of the user interaction. The J2EE server uses RMI[32] over a network connection to an application server.

The application server provides the shared logic of the on-line store. This supports implementing the functional requirement that a customer can start shopping on one device and continue on another. Running the applicaiton server on its own device allows easier scalability of the system. The application server

[30] A number of nines (e.g. four nines) is a common way to measure availability. It represents the percentage of time the system is "up". Four nines means the system is available 99.99% of the time, or it is not available for less than one hour per year.

[31] `https://tomee.apache.org/`

[32] Remote Method Invocation

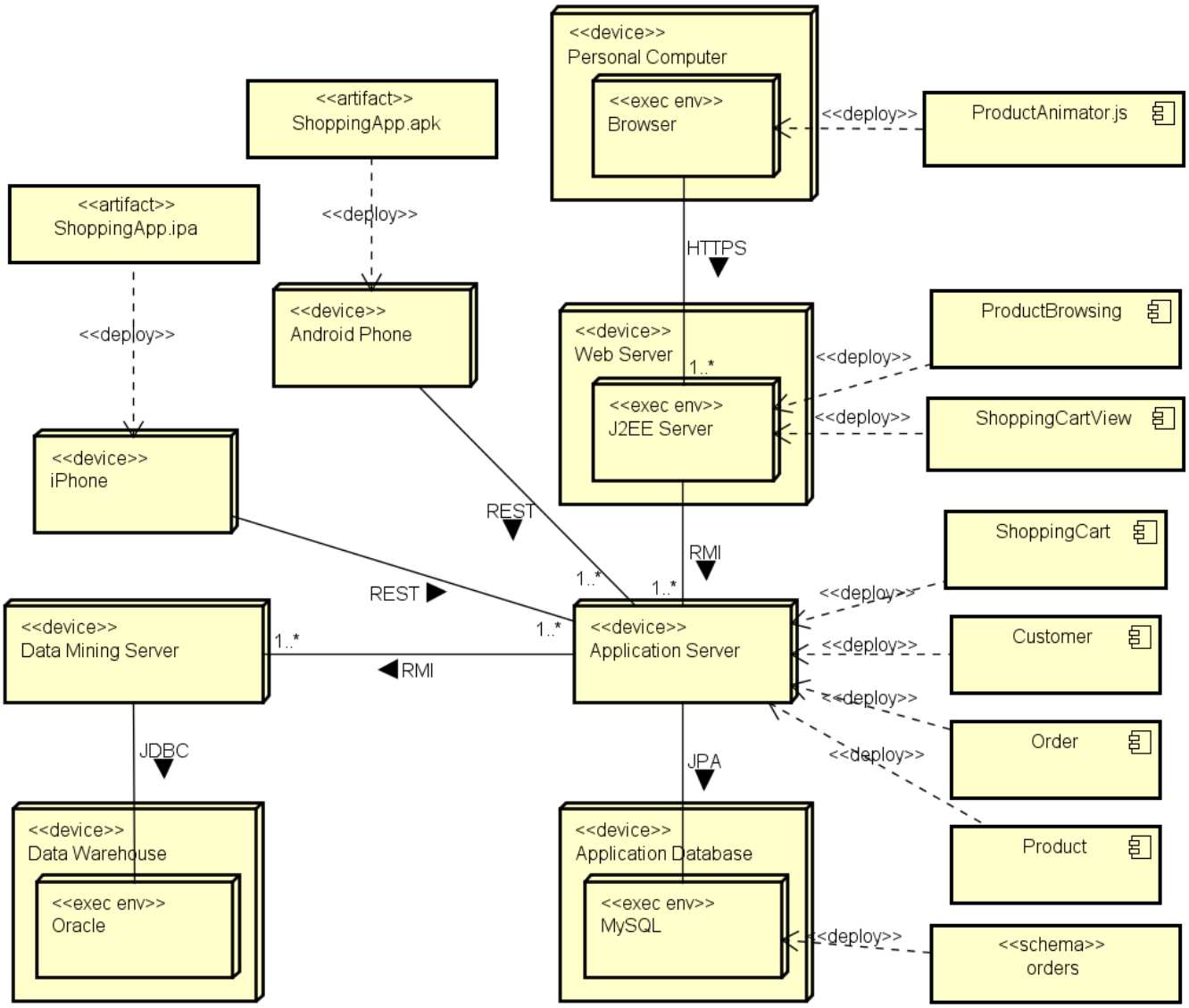


Figure 15: Example physical architecture for the Sahara eCommerce system.

could be replicated on multiple devices to handle requests from different sources, without duplicating unneeded web or database logic. Examples of components that would run on the application server are `Customer`, `ShoppingCart`, `Order` and `Product`.

The application server uses JPA[33] over a network connection to communicate with the application database running on a separate server. The `orders` schema is an example of one of the tables that would be created in the application database. The application server also uses RMI over a network connection to communicate with a data mining server.

The data mining server uses JDBC[34] over a network connection to a server running the data warehouse. The mobile applications run on their respective phone environments and use REST API calls over the Internet to interact with application server.

Note, for web applications the customer's computer and browser would only be shown in the deployment diagram if the software system downloads application logic that executes in the browser (e.g. JavaScript). In this example the `ProductAnimator.js` module is an important part of the application's functional requirements.

[33] Java Persistence API
[34] Java DataBase Connectivity

4.2.1 Deployment Diagram Notation

In the diagram, cube icons represent *nodes*. Nodes are computational resources that can execute software artifacts. Nodes may be «device»'s, representing hardware. They may also be «executionEnvironment»'s, representing software that provides an environment in which other software artifacts can be executed. (The keyword has been shortened to «exec env» in this example.) Execution environments need to be allocated to devices. In figure **??**, `Browser` and `J2EE Server` are software environments running, respectively, on the hardware devices `Personal Computer` and `Web Server`.

The solid lines between nodes are *associations*, which represent communication paths. A communication path can represent a *physical connection* or *protocol*. Formally, a stereotype (e.g. «protocol») is used to distinguish the type of communication path. In figure **??**, all communication paths are protocols so the stereotype is not included. The protocol name is used to indicate which one is being used on the communication path. The end of an association can indicate *multiplicity*. In a deployment diagram this is used to indicate that some nodes may be replicated (e.g. for performance or robustness). The '*' symbol is used to indicate many instances may be involved.

The rectangles with a 'plug' icon in their top-right corner are *components*. Components are executable software which need to be deployed to a node on which they will run. The dashed dependency arrow, with the stereotype «deploy», indicates the node on which the component will be deployed for execution.

Note, this approach of showing components being deployed to nodes is an older style of UML. The current version of UML has the concept of *artifacts* being deployed to nodes. Artifacts can implement (manifest in UML terminology) components, which provides an additional layer of abstraction. Formally, components would be manifested by an artifact that is then deployed to a node. In figure **??**, components are deployed directly to nodes to keep the diagram simple. UML provides multiple ways to indicate which artifacts are deployed on a node, e.g. a textual list of artifacts inside the node icon. The approach taken for a diagram should be chosen to aid readability and reduce clutter.

Artifacts are used in this diagram to represent software that has to be packaged (i.e. deployed through a manifest), which corresponds to the idea of an artifact in the note above. An artifact is represented by a rectangle with the «artifact» keyword and the name of the artifact that is created for deployment.

The «schema» stereotype indicates that the artifact is a database schema describing tables to be created in the database.

4.3 Component-and-Connector View

Figure **??** uses a UML component diagram as a visual representation of the logical components that deliver system behaviour as part of the component-and-connector view. It models the logical architecture of the components that allow customers to browse for products, add them to their shopping cart, and purchase them. To keep the example manageable, this is the only part of the system that is shown in this view.

As was shown in figure **??**, the `ProductBrowsing` and `ShoppingCartView` components are deployed on the J2EE server. These two component provide the user interaction behaviour in the web application of browsing for products, adding them to the shopping cart, and purchasing the products.

The `Product`, `ShoppingCart`, `Order` and `Customer` components are deployed on the application server. These components deliver the logical behaviour of providing information about products, tracking what is in the shopping cart, and placing orders.

The `ProductBrowsing` component uses the *`ProductInformation`* and *`ManageCart`* interfaces. These two interfaces are realised (or implemented) by the `Product` and `ShoppingCart` components respectively. The `ShoppingCartView` component uses the *`CartCheckout`* interface, which is realised by the `ShoppingCart` component. These interfaces describe the communication pathways between the components on the different nodes of the physical architecture.

The `ShoppingCart` component uses the `Product`, `Customer` and `Order` components. At the programming level, there could be interfaces between these components which are not shown in this diagram.

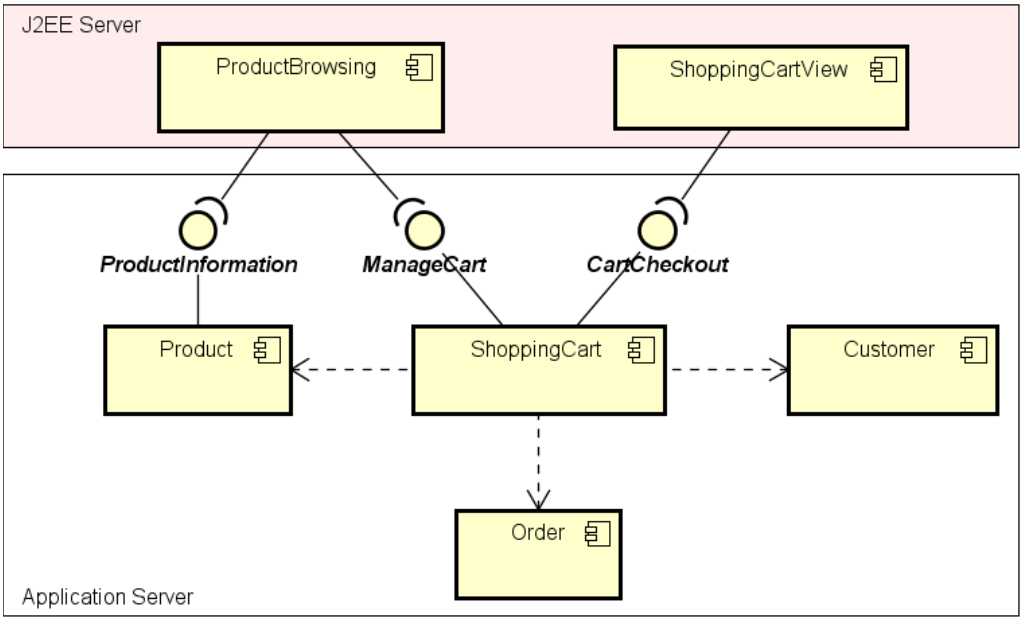


Figure 16: Example logical architecture – browsing for products and purchasing them.

The application database is not shown in this component diagram as all of its behaviour is defined within the application server's logic. JPA automates the process of saving and retrieving objects from a relational database. Consequently, the database is just a storage mechanism and does not implement any application logic. If the database implemented logic (e.g. through stored procedures and constraints), then components representing that logical behaviour would be included in this diagram.

4.3.1 Component Diagram Notation

As indicated in **??**, rectangles with the 'plug' icon represent *components* in UML.

Circles represent *interfaces*, and are labelled with the interface's name. A line from a component to an interface circle indicates that the component provides (or realises) the interface.

Cups represent a *required interface*, and a line from a component to a cup indicates that the component depends on (or uses) the interface. This notation visualises the connection between components.

Dependency arrows point from a component whose runtime behaviour depends on behaviour provided by the target component.

Boxes around groups of components are *subjects*. They represent a logical grouping of elements in a UML diagram and can be given a name to describe the grouping. Colours and shading can be used to help distinguish between different logical groups. In this example, the J2EE Server and Application Server subjects represent system boundaries for the two respective deployment environments.

4.3.2 Behaviour Structure

For complex systems, describing how important behaviour is implemented can aid in understanding the intent of the architecture's design. UML sequence and communication diagrams can describe this behavioural structure. Figure **??** is a high-level sequence diagram describing how the `ProductBrowsing` component in the J2EE server collaborates with the `ShoppingCart` component on the application server to enable a customer to add a product to their shopping cart. It also shows the `ShoppingCart` component communicating with the application database.

Figure **??** does not provide much additional information that is not already shown or implied by figures **??** and **??**. Normally sequence or communication diagrams are used to describe behaviour that is not clear from other diagrams and descriptions. This can include complex interactions between modules, complex concurrency, real-time constraints, or latency constraints.

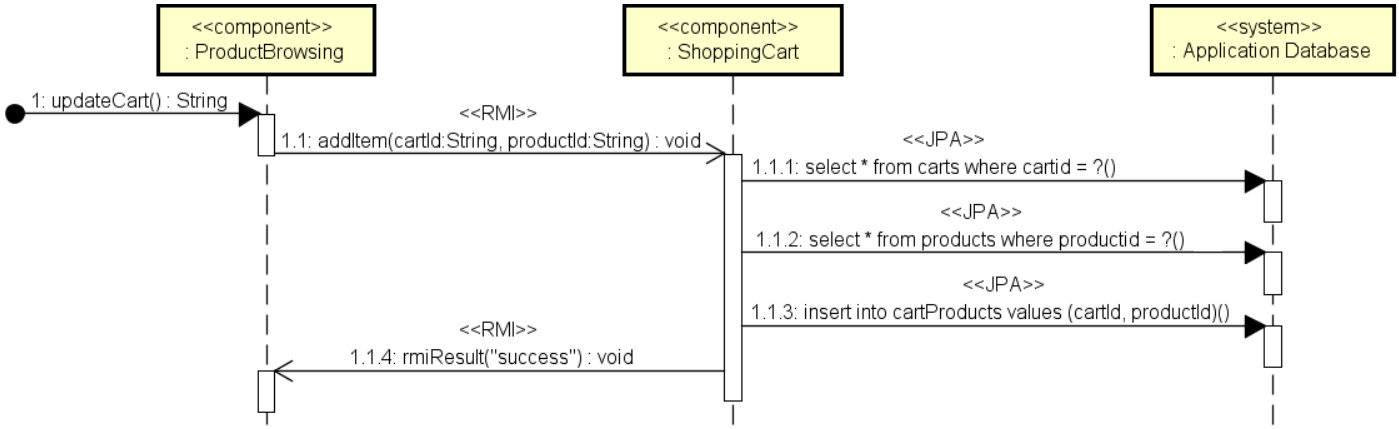


Figure 17: Example behavioural description – customer adding a product to their shopping cart.

For example, when Boeing was upgrading the combat control system of the F-111[35] for the Australian Airforce, they designed a software architecture that used CORBA[36] as middleware. The implementation of the middleware caused a fixed delay in sending messages between components. From an architectural design perspective, it was important to document this delay and enforce a maximum delay on the time taken to complete any process. A sequence diagram can use constraints to indicate these types of real-time restrictions in your design.

4.3.3 Sequence Diagram Notation

Sequence diagrams are read from the top down. The top of the diagram represents the start of the scenario, and execution time progresses down the diagram. The bottom of the diagram is the end of the scenario. Duration constraints can be placed between messages indicating information like the maximum allowed time between the start and end of a message.

Rectangles with lines descending from them are *lifelines*. They represent an instance of a participant in the scenario being described. The name at the top of a lifeline describes the participant. In figure **??**, these are components or the database system.

The horizontal lines are *messages* sent between participants. Messages use hierarchical numbers to indicate both nesting and sequence of messages. Message 1.1 is sent by message 1. Message 1.1.1 comes before message 1.1.2. Message 1 in figure **??** is a *found message*, meaning that the sender of the message is not shown in the diagram.

A closed arrowhead on a message (e.g. message 1) indicates that it is a synchronous message. An open arrowhead on a message (e.g. message 1.1) indicates that it is an asynchronous message. In figure **??**, stereotypes have been placed on most messages to indicate the protocol used to send the message.

The vertical rectangles sitting on top of lifelines are *execution specifications*. They indicate when an instance is executing logic. For example, after the asynchronous message 1.1 is sent to the `ShoppingCart` component, message 1 finishes executing. When the synchronous message 1.1.1 is sent to the database, message 1.1 is still active as it is waiting for message 1.1.1 to finish before message 1.1 can continue to the next part of the logic.

4.4 Module View

Figure **??** uses a UML class diagram as a visual representation of the static structure of the classes that implement the `ShoppingCart` component as part of the module view. Usually only architecturally significant operations and attributes are shown. (e.g. Operations and attributes needed to understand relationships

[35] https://www.youtube.com/watch?v=xUcpZJE050s
[36] https://www.ibm.com/docs/en/integration-bus/9.0.0?topic=corba-common-object-request-broker-architecture

and behaviour in the component-and-connector view.) And for simplicity in this diagram, only the classes and interfaces related to adding items to a shopping cart and checking out are shown.

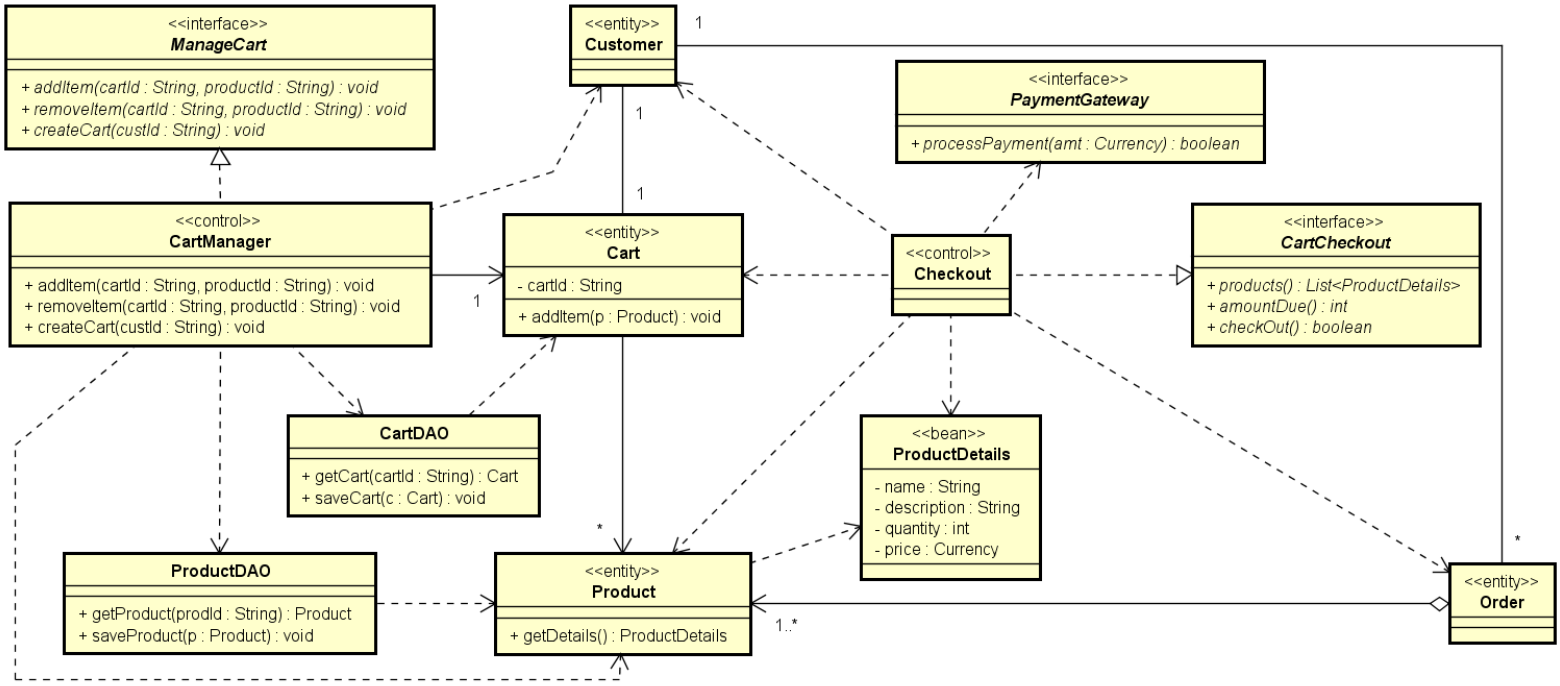


Figure 18: Example static structure for part of the shopping cart package in the class model.

The `CartManager` and `Checkout` control classes implement, respectively, the *`ManageCart`* and *`CartCheckout`* interfaces. These two classes implement the Façade design pattern and manage how adding products to a shopping cart and checking out are delivered by the classes in this package. Going back to the component and connector view (figure **??**), when a customer, via their web browser, selects to add a product to their shopping cart, the `ProductBrowsing` component's logic uses the *`ManageCart`* interface's `additem` operation to send a message to the `ShoppingCart` component.

In the implementation of the `ShoppingCart` component, the `CartManager` class uses the cart and product JPA data access objects (DAOs) to load the product details and the customer's shopping cart from the application database. The DAOs create `Product` and `Cart` entity objects, and `CartManager` adds the product to the cart. Once this is done the `CartDAO` is used to save the updated cart data into the database.

When a customer wants to checkout the products in their shopping cart, the `ShoppingCartView` component uses the *`Checkout`* interface's `products` operation to get a list of the product details to be displayed in the shopping cart. The `ProductDetails` class is a Java bean that is used to pass the data about each product to the `ShoppingCartView`. Once a customer decides to buy the products in their shopping cart, the `ShoppingCartView` sends the `checkOut` message to the `ShoppingCart`. `Checkout` uses the *`PaymentGateway`* interface to process the payment.

4.4.1 Class Diagram Notation

Formally in UML, rectangles represent *classifiers*. A *class* is one type of classifier. In a class diagram, a rectangle represents a class, unless a keyword is used to indicate that it is a different type of classifier. Classifier rectangles have three compartments. The top compartment contains its name and optionally includes a keyword, stereotypes and properties for the classifier. The middle compartment contains *attributes*. The bottom compartment contains *operations*.

Solid lines represent *associations*, which may optionally have an arrow indicating the direction of the relationship. An association indicates a structural relationship between classes. Typically this means that the target of an association will be an implicit attribute of the class. The end of an association can use *multiplicity* to indicate the number of objects of the class that may take part in the relationship.

A diamond on the end of an association indicates *aggregate* relationship. The diamond is on the end that is the aggregate, and the other end is the part. The diamond may be filled or not. A filled diamond represents *composition* in UML. This indicates 'ownership', where the aggregate controls the lifespan of the part. A hollow diamond, as in the relationship between `Order` and `Product`, indicates *aggregation* in UML. This is a weaker relationship than composition, as the aggregate does not control the lifespan of the part, but it still indicates a strong relationship between the classes.

A dashed line with an open arrowhead (e.g. from `CartManager` to `Product`) indicates that one classifier *depends* on (or uses) another. This is meant to indicate a transient relationship.

A dashed lines with a closed and hollow arrowhead (e.g. from `Checkout` to *`CartCheckout`*) indicates that the class is *realising* (or implementing) that interface.

Italicised names indicate an abstract classifier. Keywords are used to indicate the type of a classifier. In this example, the keyword «interface» indicates that the classifier is an interface. Stereotypes use the same notation as keywords. Three standard stereotypes for classes in UML are:

«entity» Represents a concept (*entity*) from the problem domain.

«control» Provides logical behaviour from the solution domain.

«boundary» Communicates with something outside of the system. (Not shown in diagram.)

An additional stereotype «bean» is used to indicate that the class is a Java bean.

4.4.2 Detailed Behaviour Structure

Figure **??** is a detailed sequence showing how the class model in figure **??** implements the behaviour of a customer adding a product to their shopping cart. Like with the high-level sequence diagram, you would only provide detailed sequence diagrams to describe architecturally important details of the design.

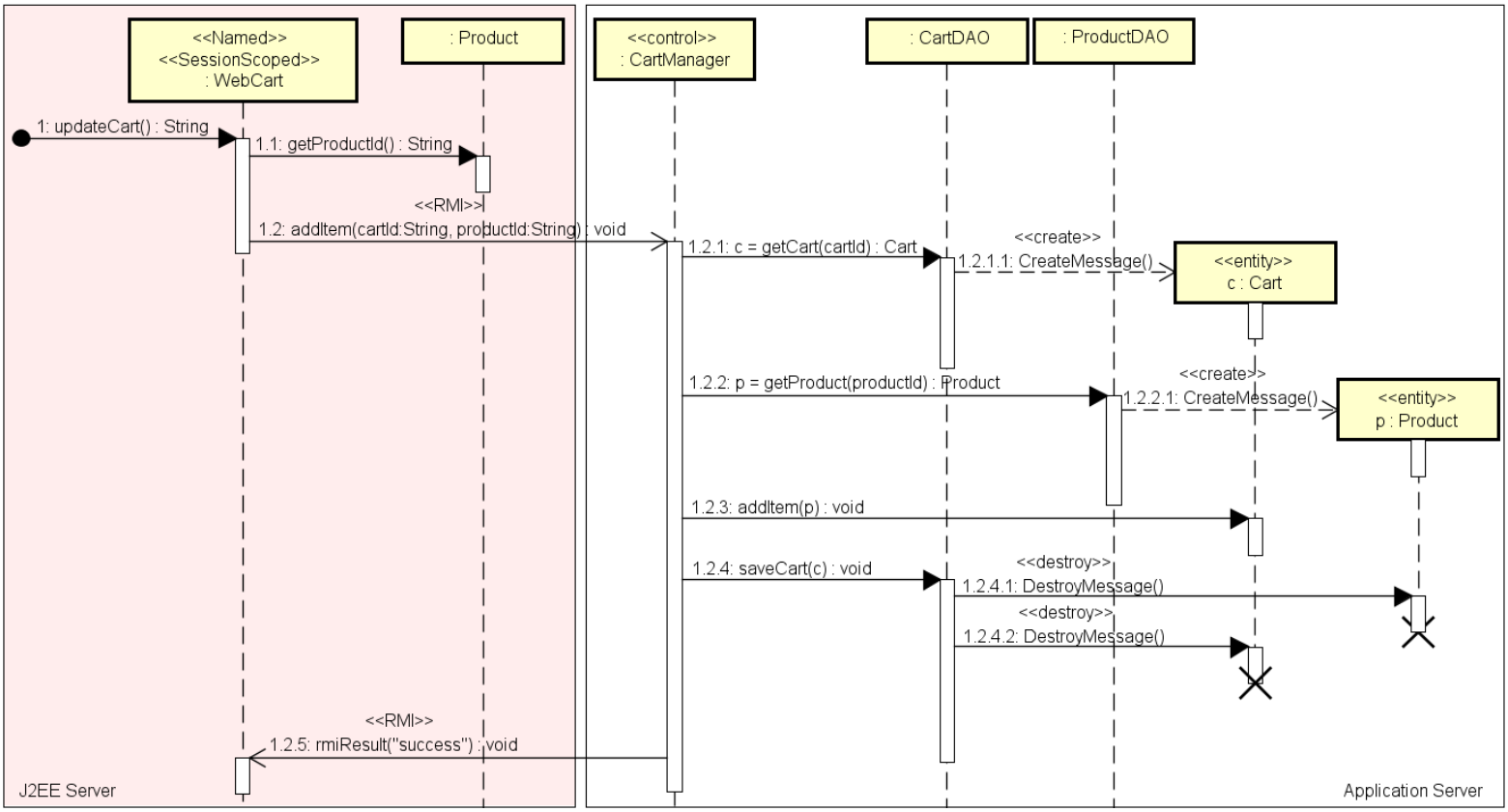


Figure 19: Example detailed sequence diagram showing the implementation of customer adding a product to their shopping cart.

The scenario starts with the JSF session-scoped bean `WebCart` receiving the `updateCart` message from the browser. `WebCart` retrieves the product id and uses the `ManageCart` interface to send it, along with the cart

id, to the ShoppingCart component on the application server. The ManageCart interface is implemented by the CartManager class in the ShoppingCart component.

The CartManager uses two JPA data access objects (DAOs) to retrieve the cart and product entities from the application database. Once the product is added to the cart, the CartDAO saves the updated cart details to the database. Upon successfully saving the updated cart, the CartManager notifies the WebCart object in the ProductBrowsing component in the J2EE server.

4.4.3 Sequence Diagram Notation

The «create» and «destroy» stereotypes indicate when instances are created or destroyed. When an instance is created, its lifeline starts at the level of the sequence diagram that indicates the time when it is created. When an instance is destroyed, its lifeline finishes, with a large X. Lifelines that are at the top of the diagram indicate instances that existed before the start of the scenario. Lifelines that reach the bottom of the diagram indicate instances that still exist after the end of the scenario.

In figure **??**, system boundary boxes are used to indicate which objects execute on which nodes from the deployment diagram.

4.5 Delivering Architecturally Significant Requirements

In section **??**, four ASRs were identified for the Sahara eCommerce system. These were the abililty to continue shopping on different devices, scalability, robustness and availability.

Implementing shared logic on an application server, as shown in figure **??**, enables the web and mobile applications to share common logic and state. This delivers the functionality of allowing a customer to start shopping on one device and to continue on another device. It also minimises duplication of logic, as it is shared by the frontend applications.

Using separate servers for the web server, application server, application database, and data mining server, as shown in figure **??**, provides more options to deliver scalability, robustness and availability. For scalability and performance, each computing environment can be optimised for the services it delivers. It also means that new servers can be deployed to target specific bottlenecks.

The system follows the *stateless architecture* pattern[37]. The web and mobile applications do not store any application state (e.g. products currently stored in the shopping cart). Every time the customer performs a transaction (e.g. viewing product details or adding a product to their shopping cart), the web or mobile application sends a message to the application server to perform the action. The application server then saves or loads data to or from the application database. This means that web and mobile applications can send messages to a different application server for each request. This facilitates scalability, robustness and availability. A new application server can be started to cater for increasing system load, or to replace a failed server. The stateless nature of the application server logic means that no data will be lost if a server fails, or if a frontend application accesses a different application server in the middle of a customer's shopping experience.

Having multiple application servers, and multiple application databases and data mining servers, means that if one server fails its load can be picked up by other servers. Automating the process of starting or restarting servers improves robustness and availability.

One challenge of a stateless architecture is providing a replicated database that contains up-to-date copies of the system's state. We will look at this issue later in the course.

By designing the architecture as a set of components running on different servers, it is also easier to migrate the application to cloud-based infrastructure. Figure **??** does not constrain the system to run on physical hardware hosted by Sahara eCommerce. Any of the nodes could be provisioned by a service offered by a cloud provider. Figure **??** in section **??** provides an example of a hybrid cloud solution. In that

[37] https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless

example, the on-line store components of the architecture run on hardware hosted by Sahara eCommerce. The data mining components run on Oracle's cloud infrastructure.

5 C4 Model

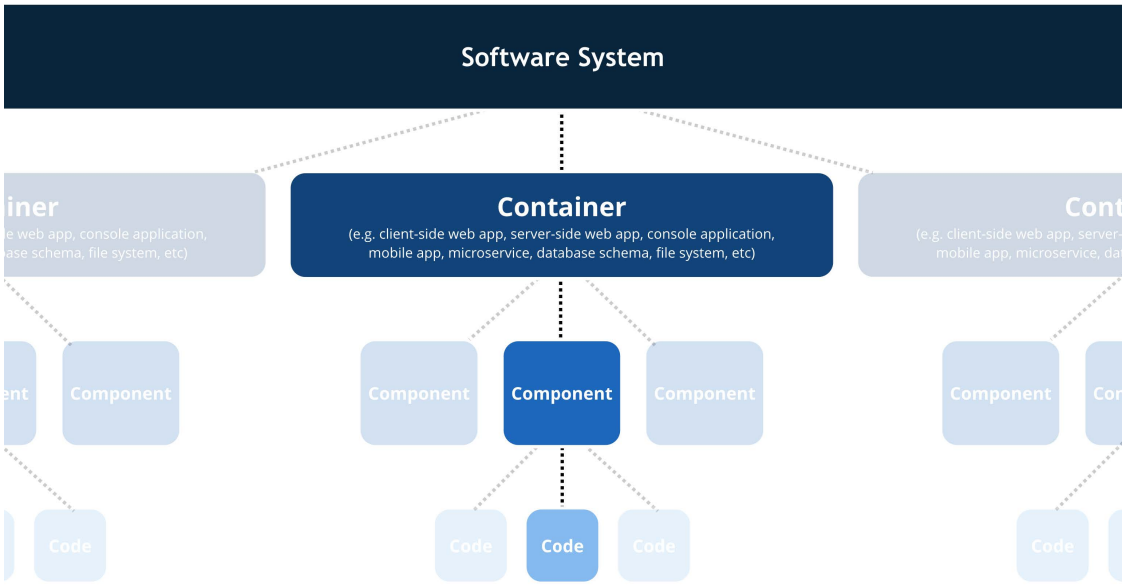
Simon Brown's C4 model provides a set of abstractions that describe the static structure of the software architecture [**?**]. The C4 model uses these abstractions in a hierarchical set of diagrams, each leading to finer levels of detail. The hierarchical structure is based on the idea that a software system is composed of containers, which are implemented by components, that are built using code.

Software System Something that delivers functional value to its users (human or other systems).

Containers Deployable 'block' of code or data that provides behaviour as part of the software system.

Components Encapsulate a group of related functionality, usually hidden behind a published interface.

Code Elements built from programming language constructs, e.g. classes, interfaces, functions, ...



A **software system** is made up of one or more **containers** (web applications, mobile apps, desktop applications, databases, file systems, etc), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (e.g. classes, interfaces, objects, functions, etc).

Figure 20: Levels within the C4 model (figure 2.1 from [**?**]).

This leads to describing the static structure of software architecture through four levels of abstraction. Each level providing detail about parts of the previous level.

Context How the software system fits into the broader context around it.

Containers How the containers are connected to deliver system functionality.

Components How the components are structured to implement a container's behaviour.

Code How the code is structured to implement a component.

5.1 System Context

The system context provides the 'big picture' perspective of the software system. It describes the key purpose of the system, who uses it, and with which other systems it interacts. The context diagram is usually a simple block diagram. The software system being designed typically sits in the centre of the diagram surrounded by uers and other systems. The intent is to set the context for thinking about the software system's architecture. It can also be used to communicate basic structural ideas to non-technical stakeholders. Figure **??** is a context diagram for the Sahara eCommerce example from section **??**.

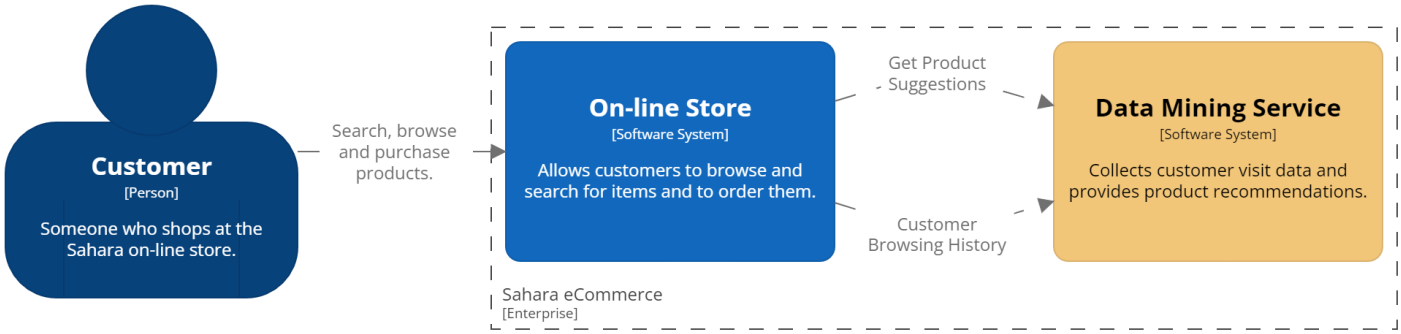


Figure 21: Context diagram for the Saraha eCommerce on-line store.

Figure **??** is the key to help interpret the context diagram. A key is important for C4 diagrams, as they do not have a formal syntax and specification like UML diagrams.

Figure 22: Context diagram key.

The context diagram situates the on-line store software system in the environment in which it will be used. There are customers who shop at the on-line store, which is part of Sahara eCommerce's software ecosystem. The on-line store uses a data mining service that is also implemented by the company. The two key relationships between the on-line store and the data mining service are that the on-line store sends customer browsing data to the service, and that the on-line store requests the data mining service to recommend products for a customer.

In C4, arrows are used to indicate the main direction of the relationship, not the flow of data. So, in this example, the arrow points from the on-line store to the data mining service as it is the store that manages the communication.

In UML, a high-level use case diagram can be used to convey similar information to the C4 context diagram. Kruchten's "4+1 View Model of Software Architecture" [**?**] uses this approach.

5.2 Containers

Container diagrams provide an overview of the software architecture. They describe the main structure of the software system and the technologies selected to implement these aspects of the system. Containers are 'blocks' of code that can be independently deployed and executed. Examples of containers are web or mobile applications, databases, message buses, ... It is important to note that this is not a type of

deployment diagram. Containers may be deployed on the same computing infrastructure or on different devices.

While the container diagram does not explicitly show computing infrastructure, some of the infrastructure may be implied by the types of containers. Decisions about how containers are connected and communicate have major implications for how the components and code will be designed and deployed. Figure **??** is a container diagram for the on-line store.

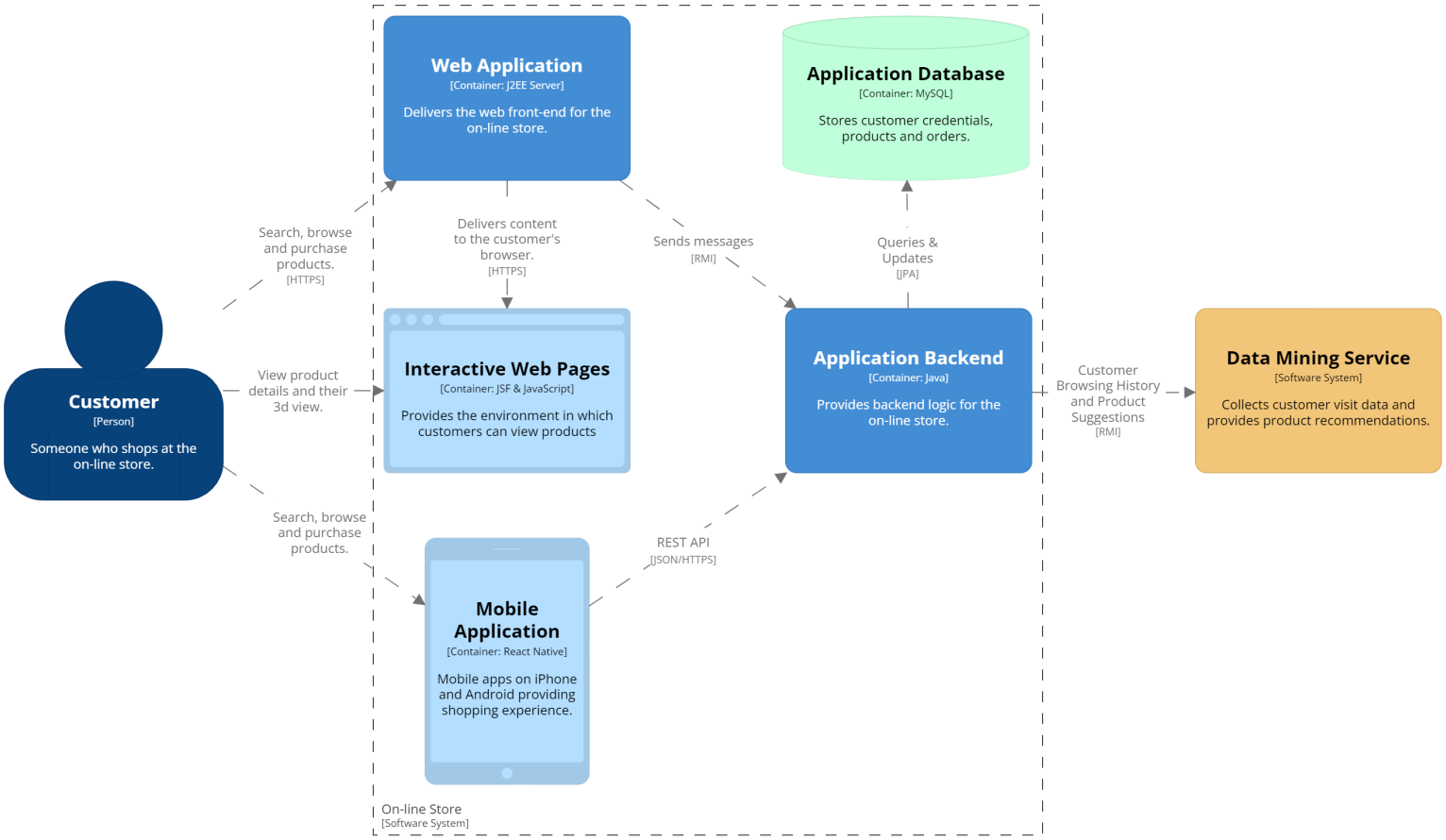


Figure 23: Container diagram for the on-line store software system.

Customers can access the on-line store through either web or mobile applications. The `Interactive Web Pages` container indicates that some of the web application's behaviour is delivered in a separate container. This indicates similar information as in figure **??**, where the customer's computer and browser were included in the deployment diagram to show that a JavaScript component was to be deployed to run in the browser.

The web and mobile applications both communicate with the application backend, via different protocols, to provide the logical behaviour of the on-line store. The backend uses JPA to perform database operations on the application's database.

To provide a link to the context diagram, a container diagram usually shows which containers communicate with which external elements. The text inside the square brackets in a container, and on a relationship, indicates the technology used to implement that container or relationship.

While a container diagram does not explicitly show computing infrastructure, some of it can be implied by the types of containers in the diagram. Clearly, the mobile app and the code running in the interactive web pages have to be separate computing platforms to the rest of the on-line store's software system.

Colours and icons can be used to provide further information in the diagrams. The diagram key in figure **??** explains the purpose of each icon and colour. UML also allows you to use icons and colours to add further information to a model, it is just difficult to do in many UML modelling tools.

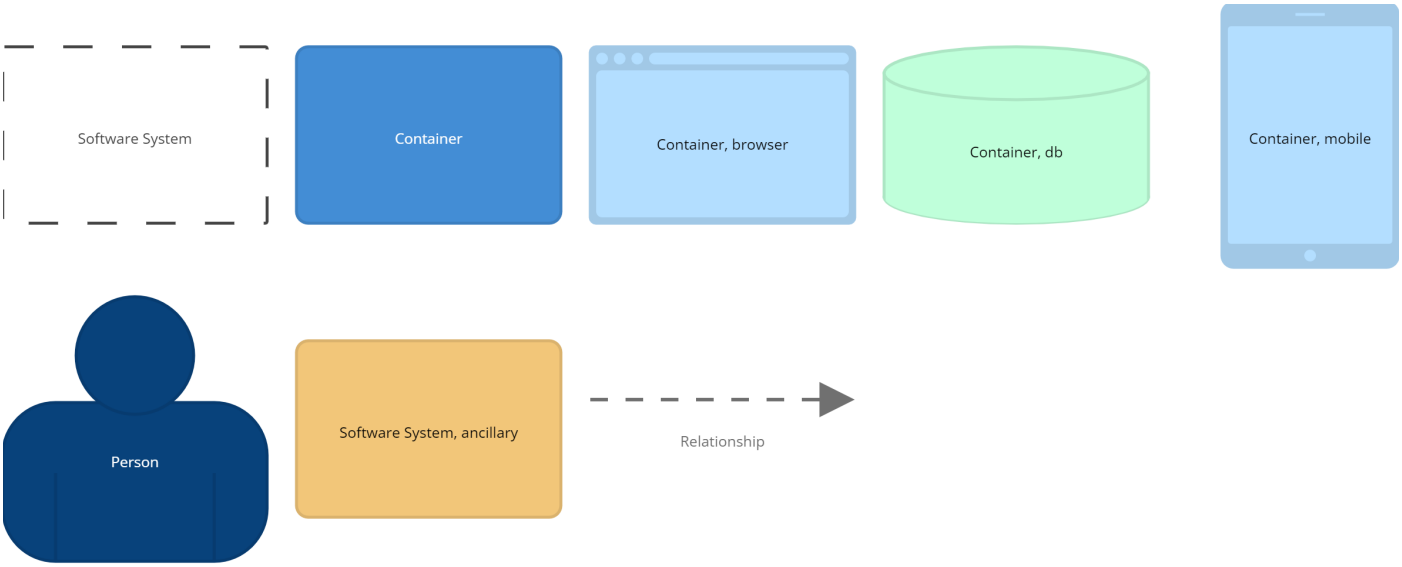


Figure 24: Container diagram key.

The data mining service software system (figure **??**) has three main containers.

`Data Mining Interface` provides the interface used to interact with the data mining service. It accepts data to store for future data mining. It returns suggestions based on requests from external systems, such as the on-line store.

`Data Mining Process` performs the data mining logic.

`Data Warehouse` stores the data and provides an SQL-based query system to manipulate the data.

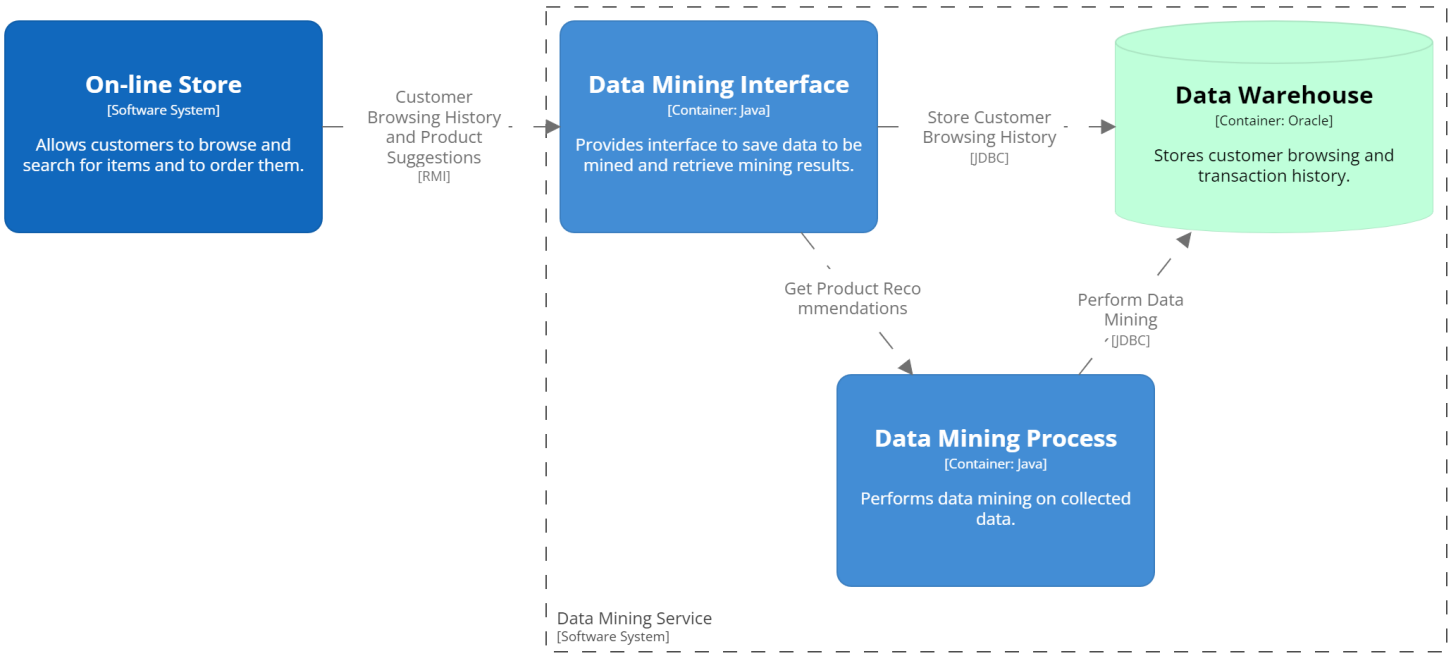


Figure 25: Container diagram for the data mining service software system.

5.3 Components

Component diagrams describe the major parts of containers and how they are connected. Components should describe their important responsibilities and the technology used to implement them (e.g. using React to implement a web frontend component). Like a container diagram, a component diagram can include elements from higher level diagrams to provide context of how the components interact with elements outside of the container.

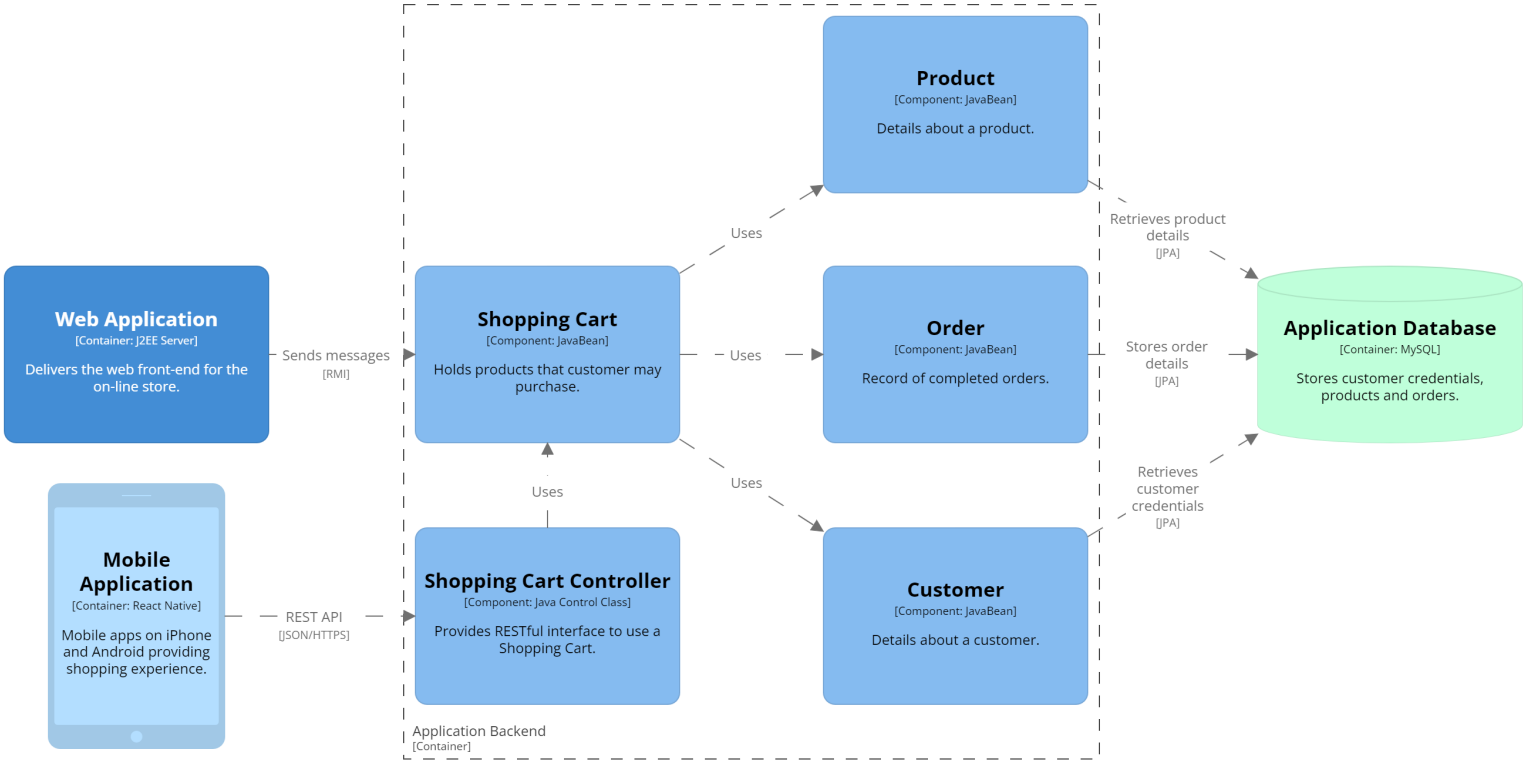


Figure 26: Component diagram for the application backend container.

In figure **??**, the application backend is divided into five main components. The `Shopping Cart` component provides the backend logic of implementing a shopping cart. The web application interacts with the `Shopping Cart` component via RMI. The `Shopping Cart` component would provide interfaces for this interaction. The mobile applications use a REST API provided by the `Shopping Cart Controller` component to interact with the `Shopping Cart`. `Shopping Cart` uses the `Product`, `Order` and `Customer` components to deliver its behaviour. These are all implemented as Java beans.

The `Product`, `Order` and `Customer` components use JPA to retrieve and store data in the application database. (As indicated in section **??**, only the components related to managing the shopping cart are shown in this example.)

Figure **??**, shows the icons and colours used to represent different elements in the component diagrams.

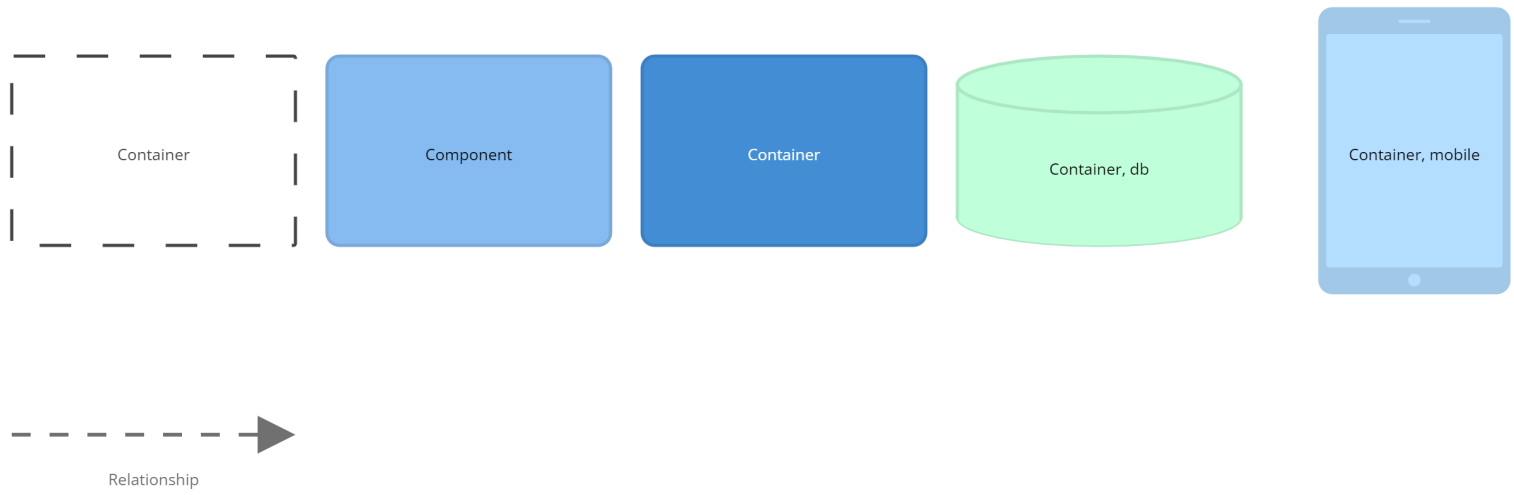


Figure 27: Component diagram key.

Figure **??** shows the components that provide the frontend behaviour of browsing for products, adding them to the shopping cart, and purchasing them.

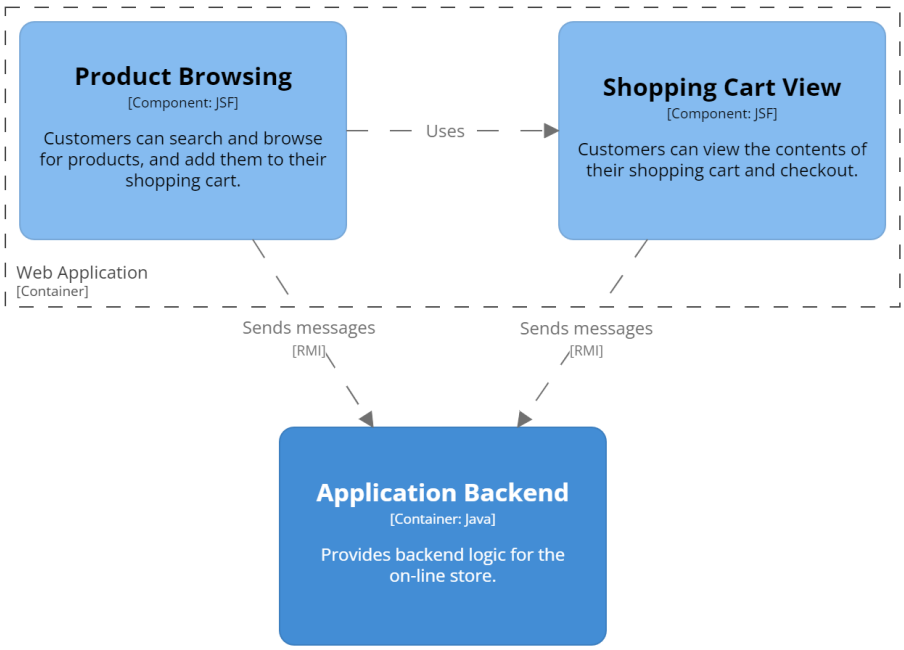


Figure 28: Component diagram for the web application container.

Figure **??**, shows that the `Product Animator` component is downloaded from the web application to the customer's browser and that it is implemented in JavaScript. This provides similar information about the `Product Animator` component, as described in section **??**.

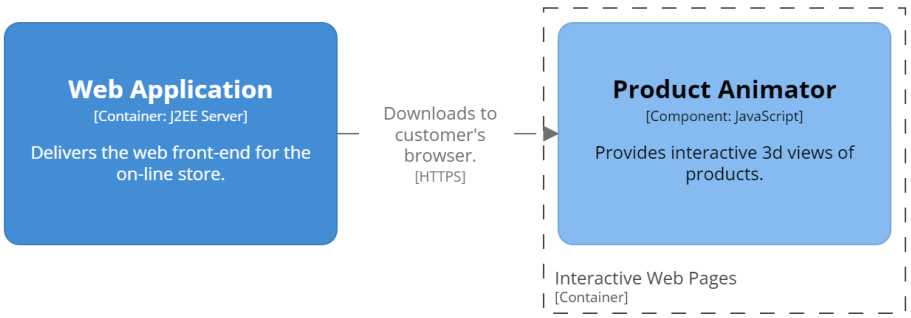


Figure 29: Component diagram for the interactive web pages container.

5.3.1 Component Diagram Detail

There may be some components that are important parts of the software design, but which may not necessarily be included in component diagrams. For example, a logging component is an important part of many software systems. But, due to the nature of a logging component, most other components will use it. Adding it to component diagrams will often clutter the diagrams without adding much useful information. Usually it is better to add a note indicating which logging component is used in the system. If it is helpful to indicate which components use the logging component, it may be better to colour code these components or use an icon to represent that they use the logging component.

5.4 Code

The code-level diagrams describe the structure of the code that implements a component. The intent is to provide a visual representation of the important aspects of the code's structure. Rarely do you need to

provide all the detail that replicates the source code. (The source code could be considered a fifth level to the C4 model.)

The C4 model suggests using diagrams appropriate to your programming paradigm. Assuming the implementation is in an object-oriented language, a UML class diagram would be an appropriate way to model the design of the code. Figure **??**, from section **??**, would be an example class diagram of how the `Shopping Cart` component is implemented.

5.5 Dynamic

Dynamic diagrams in C4 are similar to UML communication diagrams. (Communication and sequence diagrams are types of interaction diagrams. They show the same information but with different visual emphasis. Sequence diagrams focus on the time or ordered sequence of events that occur in a scenario. Communication diagrams focus on the links and extent of communication between objects.)

As was mentioned about behavioural structures in sections **??** and **??**, dynamic diagrams are usually only provided to explain how the architecture delivers complex behaviour.

Figure **??** provides an overview of how the `Product Browsing` component in the `Web Application` container collaborates with the `Shopping Cart` component in the `Application Backend` container to deliver the behaviour of a customer adding a product to their shopping cart. It also shows the communication between the `Shopping Cart` component and the application database.

Figure 30: Dynamic diagram for adding a product to the customer's shopping cart.

If the information is useful, a more detailed dynamic diagram, like the detailed sequence diagram in figure **??**, can be provided. Sequence diagrams can be used instead of C4 dynamic diagrams, if the order of events is important.

5.6 Deployment

While not one of the "four C's", deployment diagrams are important for most systems. They describe the physical architecture or infrastructure on which the system will be deployed. It shows which containers will run on which computing platforms (*deployment nodes*). Deployment nodes can be nested, as shown in figure **??**. They may also contain infrastructure nodes or software system instances[38].

Figure **??** is an example C4 deployment diagram for the Sahara eCommerce system. It takes a slightly different approach to the physical architecture than was shown in figure **??**. It shows that the on-line store software system runs in Sahara's data centre. The data mining service runs on Oracle's cloud infrastructure. This approach of a system that uses cloud services for some of its implementation is called a *hybrid cloud* application. There are still the apps running on mobile devices and the code running in the customer's browser.

Like UML, a software environment is embedded in the hardware environment on which it runs. The web application runs in an Apache TomEE J2EE server, which is running on a Ubuntu server. The "`x4`" inside

[38] https://github.com/structurizr/dsl/blob/master/docs/language-reference.md#deploymentNode

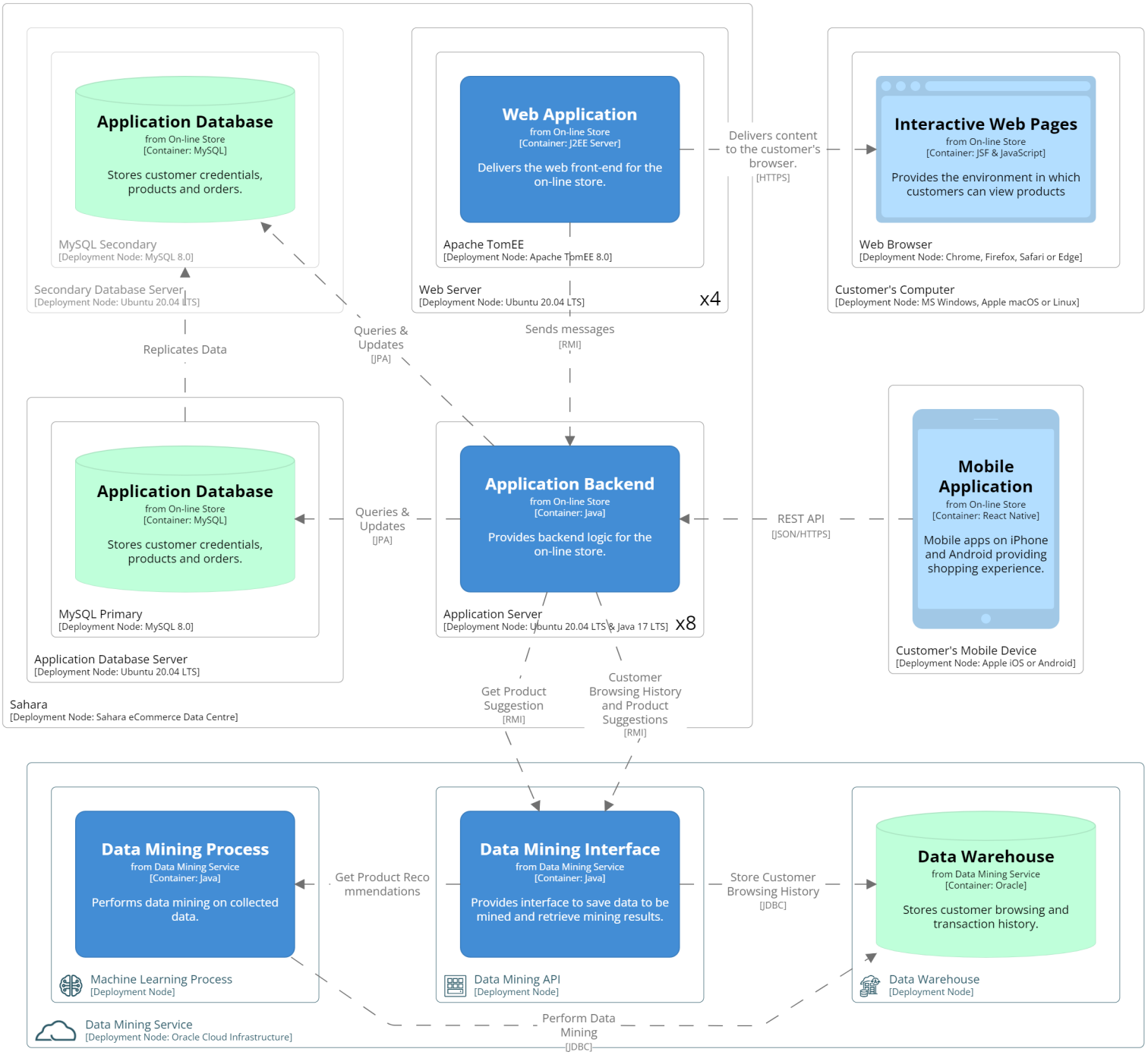


Figure 31: Deployment diagram for the Sahara eCommerce System.

the web server deployment node indicates that there will be four of these servers to share the load. The application backend runs on eight Ubuntu servers, providing the core business logic shared by the web and mobile applications.

The application database runs in MySQL on its own Ubuntu server. Replication of the application database is explicitly shown in this example, whereas it was not shown in figure **??**. Replicating the database running on another server allows for failover. The application backend can continue to operate if the primary application database fails.

In this version of the example, the application backend communicates with the data mining service through an API published by the data mining interface running in a virtual machine on Oracle's cloud infrastructure. The data mining service uses Oracle's machine learning services to perform the data mining. Oracle's cloud-based data warehouse infrastructure is used to hold all the data.

Figure **??** is the key describing the icons and colours used in the deployment diagram.

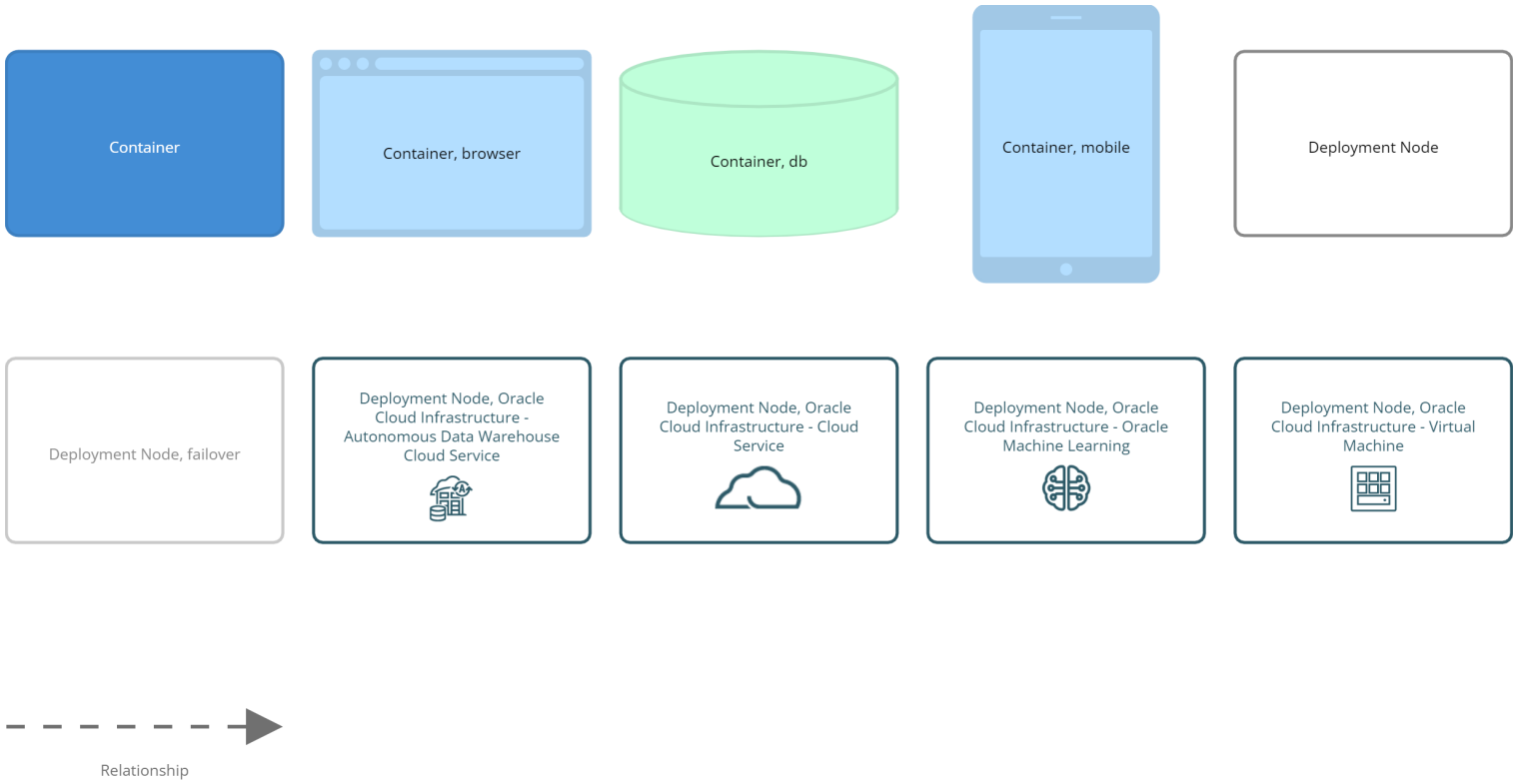


Figure 32: Deployment diagram key.

6 Tools

We do not advocate a specific notation or support tools. Both UML and C4 have been used so that you are aware of some options. UML is a standardised notation, with formal syntax and semantics. There will be situations where the formality will be useful. C4 is popular because it has a basic structure, but the rules are intentionally loose to make it easy to adopt. Regardless of whether you use UML, C4, or another notation, you should use tools to aid the creation of your diagrams and documentation.

The important thing is that you should use a modelling tool, not a drawing tool. Many drawing tools provide UML templates, and some also support C4. The issue with drawing tools is that they do not know what the elements of the diagram mean. If the name of an operation in a class is changed in a drawing tool, you will need to manually change it wherever it is referenced in other diagrams (e.g. in sequence diagrams). A modelling tool will track the information that describes the model, so that a change to a model element in one place, will be replicated wherever that element appears in other diagrams.

There are many tools that support UML. In a commercial project using UML on a large system, the cost of professional UML tools is negligible and is quickly recovered by the automation they provide. There are a number of free UML tools. Some to consider are Astah[39], ModelIO[40], or PlantUML[41]. Visual Paradigm[42] is not as recommended, as their free cloud-based tool is only a drawing tool, and not a modelling tool.

Astah is a commercial product that supports visual modelling in many notations. They provide a free UML tool for students.

[39] https://astah.net/products/free-student-license/
[40] https://www.modelio.org/
[41] https://plantuml.com/
[42] https://www.visual-paradigm.com/

ModelIO is an open source visual UML modelling tool.

PlantUML Is an open source text-based descriptive language that generates UML diagrams. PlantText[43] is an online tool supporting it.

Visual Paradigm is a commercial product that supports visual modelling in many notations. They provide a simple free cloud-based drawing tool that supports UML and some limited aspects of C4, but it lacks full modelling support.

There are fewer tools that support C4. Some to consider are Structurizr[44], C4-PlantUML[45], Archi[46], IcePanel[47], or Gaphor[48].

Structurizr was developed by Simon Brown as a tool to support generating C4 diagrams from textual descriptions. UQ students may register for free access to the paid version of the Structurizr Cloud Service[49]. You must use your `student.uq.edu.au` or `uq.net.au` email address when you register to get free access. Structurizr is an open source tool[50]. You can use a domain specific language to describe a C4 model, or you can embed the details in Java or .Net code.

C4-PlantUML which extends PlantUML to support C4.

Archi is an open source visual modelling tool that supports C4[51] and ArchiMate models.

IcePanel is a cloud-based visual modelling tool that supports C4. There is a limited free license for the tool.

Gaphor is an open source visual modelling tool that supports UML and C4.

6.1 Textual vs Visual Modelling

The tools described above include both graphical and textual modelling tools. Graphical tools, such as Astah, ModelIO, Archi and Gaphor, allow you to create models by drawing them. This approach is often preferred by visually oriented learners[52]. Text-based tools, such as PlantUML and Structurizr, allow you to create models by providing a textual description of the model. This approach is often preferred by read/write oriented learners[53].

Despite preferences, there are situations where there are advantages of using a text-based modelling tool. Being text, the model can be stored and versioned in a version control system (e.g. git). For team projects, it is much easier for everyone to edit the model and ensure that you do not destory other team members' work. It is also possible to build a tool pipeline that will generate diagrams and embed them into the project documentation.

Text-based modelling tools, such as Structurizr or PlantUML, use a domain specific language[54] (DSL) to describe the model. These tools require that you learn the syntax and semantics of the DSL. The following sources of information will help you learn the C4 DSL:

[43] https://www.planttext.com/
[44] https://www.structurizr.com/
[45] https://github.com/plantuml-stdlib/C4-PlantUML
[46] https://www.archimatetool.com/
[47] https://icepanel.io/
[48] https://gaphor.org/
[49] https://structurizr.com/help/academic
[50] https://github.com/structurizr/
[51] https://www.archimatetool.com/blog/2020/04/18/c4-model-architecture-viewpoint-and-archi-4-7/
[52] https://vark-learn.com/strategies/visual-strategies/
[53] https://vark-learn.com/strategies/readwrite-strategies/
[54] https://opensource.com/article/20/2/domain-specific-languages

- language reference manual[55],
- language examples[56],
- on-line editable examples[57], and
- off-line tool[58].

You may find that the Sahara eCommerce C4 model is useful as an example of a number of features of the DSL.

The following sources of information will help you learn how to use the PlantUML DSL:

- language reference manual[59],
- language overview with examples[60], and
- on-line editable examples[61].

6.2 Example Diagrams

You are able to download the UML and C4 models of the Sahara eCommerce example, from the course website. The UML model[62] was created using Astah[63]. You may use this as an example of creating a model using a visual modelling tool. There is a little more detail in the Astah model than what is shown in these notes.

The C4 model[64] was created using the Structurizr[65] DSL (Domain Specific Language[66]). You may use the C4 model as an example of creating a model using a textual description of the model (the DSL).

7 Conclusion

Architectural views help developers understand different dimensions and details of a complex software architecture. They are useful both during design and as documentation. During design, views help you to focus on a particular aspect of the software architecture and ensure that it will allow the system to deliver all of its requirements. As documentation, views help developers to understand how different aspects of the architecture are intended to behave.

We have intentionally looked at a few different approaches to helping you describe a software architecture. You should be conversant with multiple different approaches. The hallmark of a professional is to know when to select a particular approach.

If you compare the "4+1 View Model" [**?**] with the views described in *Software Architecture in Practice* (SAP) [**?**], you will see that there are obvious similarities but also some differences. The logical, development and process views from the 4+1 view model map closely to the module and component-and-connector (C&C) views from SAP. The physical view corresponds to the allocation view. The 4+1 view model also includes the scenario view, which does not correspond directly to the SAP views.

The C4 model does not explicitly include the concept of views. Like SAP, it emphasises the structure of the software architecture.

[55] https://github.com/structurizr/dsl/blob/master/docs/language-reference.md
[56] https://github.com/structurizr/dsl/tree/master/docs/cookbook
[57] https://structurizr.com/dsl
[58] https://github.com/structurizr/cli/blob/master/docs/getting-started.md
[59] https://plantuml.com/guide
[60] https://plantuml.com/
[61] https://www.planttext.com/
[62] https://csse6400.uqcloud.net/resources/Sahara_eCommerce.asta
[63] https://astah.net/products/free-student-license/
[64] https://csse6400.uqcloud.net/resources/c4_model.zip
[65] https://www.structurizr.com/
[66] https://opensource.com/article/20/2/domain-specific-languages

The scenario view is used to demonstrate how the architecture described in the other views delivers the core functional requirements. It is used while designing a software architecture to validate that it is suitable for the system.

Kruchten intentionally separated the process view from the logical and development views, rather than bundling them together like the C&C view in SAP. This is because for some systems the dynamic details, which are described by the process view, can be complex and important. Dealing with issues such as complex concurrency, real-time interactions or latency, can often be more easily considered by having a separate view for them.

Kruchten's experience with Canada's integrated, nation-wide, air traffic control system was such a case. Data from radar systems and aircraft transponders have to be processed and reported to air traffic controllers in near real-time. With thousands of input sources and hundreds of controller stations, understanding concurrency issues is critical. Tracking aircraft from one control space to another means that communication latency is important. Each control space has its own hardware and is separated from neighbouring spaces by hundreds or thousands of kilometres.

As a software architect you need to choose which views provide meaningful information about your software system. The graphical notation used to describe a view is only one part of the view (though an important part). Ensure you provide enough supporting information so others will know how to work with your architecture and why you made the choices that you did.

Architectural Decision Records

Software Architecture

March 7, 2022

Richard Thomas

1 Introduction

Documenting reasons why a decision was made about some aspect of the software architecture is important for the long-term maintainability of the system. Architecture decision records (ADR) are a simple but informative approach to documenting these reasons.

Michael Nygard was one of the early proponents of ADRs. His argument is that no one reads large documents but not knowing the rationale behind architecturally important decisions can lead to disastrous consequences, when later decisions are made that defeat the earlier decisions [**?**]. Nygard created ADRs as a light-weight approach to documenting important architecture decisions, to suit agile development processes. The ideas were based on Philippe Kruchten's discussion of the importance of architecture decisions in his article "The Decision View's Role in Software Architecture Practice" [**?**]. In this article Kruchten discusses extending his "4+1 View Model of Software Architecture" [**?**] to capture the rationale for important decisions while designing each view of the architecture.

Architecture decision records should capture important decisions that are made about the design of the architecture. These include decisions that influence the

- structure of the architecture,
- delivery of quality attributes,
- dependencies between key parts of the architecture,
- interfaces between key parts of the architecture or external interfaces, and
- implementation techniques or platforms.

These decisions provide important information for developers who have to work with the architecture, so that they know why decisions have been made and know how to design their code to fit into the architecture. They also provide important information for those who have to maintain the software in the future. ADRs help maintainers to know why decisions were made, so that they do not inadvertently make decisions that result in breaking expectations about the software.

ADRs are short notes about a single decision. The intent is to make it easy to record the information about the decision, so that it is more likely to be documented. ADRs are usually kept with the key project documentation. The argument is often made that this should be within the version control system (e.g. git) holding the project's source code. For example, a directory `doc/architecture/adr` in the project repository to contain the ADRs. The C4 model recommends you create a directory in the project repository to hold the C4 diagrams with a subdirectory for documentation, and another subdirectory for ADRs (e.g. `c4-model`, `c4-model/docs` and `c4-model/adrs`). Note that the C4 modelling tools do not like having the subdirectory containing the ADRs within the documentation subdirectory.

Each ADR is written in a separate file, using a simple notation like markdown. The file names are numbered so the history of decisions can be viewed. It is recommended that the file name describe the decision, to make it easier to scan for information about specific types of architectural decisions. Examples of meaningful file names include:

- `0001-independent-business-logic.md`
- `0002-implement-JSF-webapp.md`
- `0003-choose-database.md`

The directory containing these ADR files is the history of all architectural decisions made for the project.

2 Template

There are a few templates available to help provide consistent formatting of an ADR. The recommended template format contains the following sections.

Title Short phrase that describes the key decision.

Date When the decision was made.

Status Current status of the decision (i.e. proposed, accepted, deprecated, superseded or rejected).

Summary Summarise the decision and its rationale.

Context Describe the facts that influence the decision. State these in value-neutral language.

Decision Explain how the decision will solve the problem, in light of the facts presented in the context.

Consequences Describe the impact of the decision. What becomes easier and harder to do? There will be positive, neutral and negative consequences, identify all of them.

3 ADR Example

The following is an example ADR from the Sahara eCommerce application from section 4 of the *Architectural Views* notes.

1. Independent Business Logic
Date: 2022-01-06
Status: Accepted
Summary
In the context of delivering an application with multiple platform interfaces, *facing* budget constraints on development costs, *we decided* to implement all business logic in an independent tier of the software architecture, *to achieve* consistent logical behaviour across platforms, *accepting* potential complexity of interfaces to different platforms.
Context

- The system is to have both mobile and web application frontends.
- Marketing department wants a similar user experience across platforms.
- Delivering functional requirements requires complex processing and database transactions.
  - Product recommendations based on both a customer's history and on purchasing behaviour of similar customers.
  - Recording all customer interactions in the application.
- Sales department wants customers to be able to change between using mobile and web applications without interrupting their sales experience.
- Development team has experience using Java.

Decision
All business logic will be implemented in its own tier of the software architecture. Web and mobile applications will implement the interaction tier. They will communicate with the backend to perform all logic processing. This provides clear separation of concerns and ensures consistency of business logic across frontend applications. It means the business logic only needs to be implemented once. This follows good design practices and common user interface design patterns.

The business logic will be implemented in Java. This suits the current development team's experience and is a common environment. Java has good performance characteristices. Java has good support for interacting with databases, to deliver the data storage and transaction processing requirements.

Consequences
Advantages
- Separation of concerns, keeping application logic and interface logic separate.
- Ensures consistency, if business logic is only implemented in one place.
- Business logic can execute in a computing environment optimised for processing and transactions.
  - Also makes load balancing easier to implement.

Neutral
- Multiple interfaces are required for different frontend applications. These can be delivered through different Java libraries.

Disadvantages
- Additional complexity for the overall architecture of the system.

4 Quality

A well written ADR explains the reasons for making the decision, while discussing pros and cons of the decision. In some cases it is helpful to explain reasons for not selecting other seemingly good options. The ADR should be specific about a single decision, not a group of decisions.

The context should provide all the relevant facts that influence the decision. The decisions section should explain how business priorities or the organisation's strategic goals influenced the decision. Where the team's membership or skills has influenced the decision, these should be acknowledged. All consequences should be described. These may include decisions that need to be resolved later or links to other ADRs that record decisions that were made as a result of this decision.

The summary follows the format of a Y-statement [**?**]. It includes key information about the decision for quick consumption, leaving the details for the rest of the ADR. The template for a Y-statement is:

In the context of *functional requirement or architecture characteristic*,
facing *non-functional requirement or quality attribute*,
we decided *selected option*,
to achieve *benefits*,
accepting *drawbacks*.

The Y-statement can be expanded to include **neglected** *alternative options* after the **we decided** clause. A **because** clause providing *additional rationale* can be added to the end of the statement. Some teams use this expanded form of the Y-statement as the complete ADR for simple decisions.

ADRs should be reviewed after about one month. Consider how the consequences have played out in reality. Revise bad decisions before too much has been built based on the decision. The ADR may need to be updated to include any consequences that have been uncovered in practice.

ADRs should be immutable. Do not change existing information in an ADR. An ADR can have new information added to it, like adding new consequences. If the decision needs to be changed, create a new ADR. The original ADR should have its status changed to *deprecated*, and then *superseded* once the new decision is accepted and put into practice. Add a link in the original ADR to the new ADR, and the new ADR should link back to the original.

Sometimes an ADR is not superseded by a new ADR, instead it is extended or amended by another ADR. In these cases, the original ADR still has an *accepted* status but a link is added indicating the other ADR that amends or extends the original ADR. The new ADR then includes a link back to the original ADR, indicating that the new ADR extends or amends the original.

5 What to Put in an ADR?

Only document important decisions that affect the architecture's structure, non-functional characteristics, dependencies, interfaces, or construction techniques. Michael Nygard calls these "architecturally significant" decisions [**?**]. The types of decisions that usually should be documented are:

- Critical or important to delivering system functionality.
- Helps deliver important quality attributes.
- Unconventional or risky approach to a solution.
- Has expensive consequences.
- Has long lasting effects or will affect a large part of the design.
- Will affect a large number of stakeholders or an important stakeholder.
- Took a long time or significant effort to decide.
- Unexpected decisions that were required late in the project. These may also be important learning experiences.

6 Conclusion

Decisions that affect the software architecture need to be recorded for reference in the future. In a large project it is not possible for everyone to be aware of every decision that is made. ADRs provide a mechanism for team members to find reasons for decisions to help them understand how to work within its constraints. ADRs give new team members a starting point to learn about the architecture and understand how it evolved into its current state. They also provide a reminder for yourself when you are wondering why you did something a particular way, or when you need to explain your decision to someone else, months after you made the decision.

The takeaway is, write an ADR whenever you make an important decision. You, or a colleague, will need to refer to it at some point in the future. Eli Perkins has a good summary of why you should write ADRs at the GitHub Blog[67] [**?**].

[67] https://github.blog/2020-08-13-why-write-adrs/

Pipeline Architecture

Software Architecture

March 7, 2022

Brae Webb & Richard Thomas

1 Introduction

Pipeline architectures take the attribute of modularity of a system to the extreme. Pipeline architectures are composed of small well-designed components which can ideally be combined interchangeably. In a pipeline architecture, input is passed through a sequence of components until the desired output is reached. Almost every developer will have been exposed to software which implements this architecture. Some notable examples are bash, hadoop, some older compilers, and most functional programming languages.

Definition 10. Pipeline Architecture
Components connected in such a way that the output of one component is the input of another.

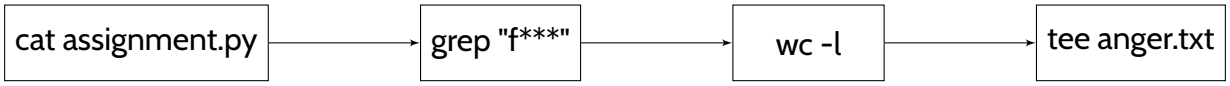


Figure 33: An example of using bash's pipeline architecture to perform statistical analysis.

The de-facto example of a well-implemented pipeline architecture is bash, we will explore the philosophy that supports the architecture shortly. The above diagram represents the bash command,

```
$ cat assignment.py | grep "f***" | wc -l | tee anger.txt
```

.

If you are unfamiliar with Unix processes (start learning quick!).

cat Send the contents of a file to the output.

grep Send all lines of the input matching a pattern to the output.

wc -l Send the number of lines in the input to the output.

tee Send the input to stdout and a file.

2 Terminology

As illustrated by Figure **??**, a pipeline architecture consists of just two elements;

Filters modular software components, and

Pipes the transmission of data between filters.

Filters themselves are composed of four major types:

Figure 34: A generic pipeline architecture.

Producers Filters where data originates from are called producers, or source filters.

Transformers Filters which manipulate input data and output to the outgoing pipe are called transformers.

Testers Filters which apply selection to input data, allowing only a subset of input data to progress to the outgoing pipe are called testers.

Consumers The final state of a pipeline architecture is a consumer filter, where data is eventually used.

The example in Figure **??** shows how bash's pipeline architecture can be used to manipulate data in Unix files. Figure **??** labels the bash command using the terminology of pipeline architectures.

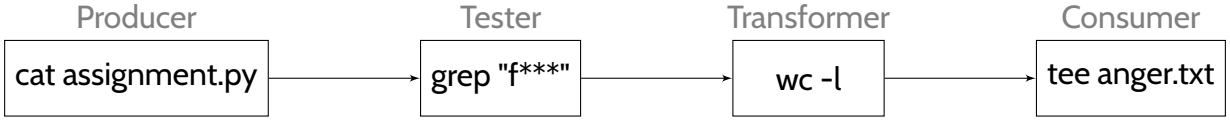


Figure 35: Figure **??** with labelled filter types.

3 Pipeline Principles

While the concept of a pipeline architecture is straightforward, there are some principles which should be maintained to produce a well-designed and re-usable architecture.

Definition 11. One Direction Principle

Data should flow in one direction, this is the *downstream*.

The data in a pipeline architecture should all flow in the same direction. Pipelines should not have loops nor should filters pass data back to their *upstream* or input filter. The data flow *is* allowed to split into multiple paths. For example, figure **??** demonstrates a potential architecture of a software which processes the stream of user activity on a website. The pipeline is split into a pipeline which aggregates activity on the current page and a pipeline which records the activity of this specific user.

The One Direction Principle makes the pipeline architecture a poor choice for applications which require interactivity, as the results are not propagated back to the input source. However, it is a good choice when you have data which needs processing with no need for interactive feedback.

Definition 12. Independent Filter Principle

Filters should not rely on specific upstream or downstream components.

In order to maintain the reusability offered by the pipeline architecture, it is important to remove dependencies between individual filters. Where possible, filters should be able to be moved freely. In the example architecture in figure **??**, the EventCache component should be able to work fine without the TagTime component. Likewise, EventCache should be able to process data if the Anonymize filter is placed before it.

Producers and consumers may assume that they have no upstream or downstream filters respectively. However, a producer should be indifferent to which downstream filter it feeds into. Likewise a consumer should not depend on the upstream filter.

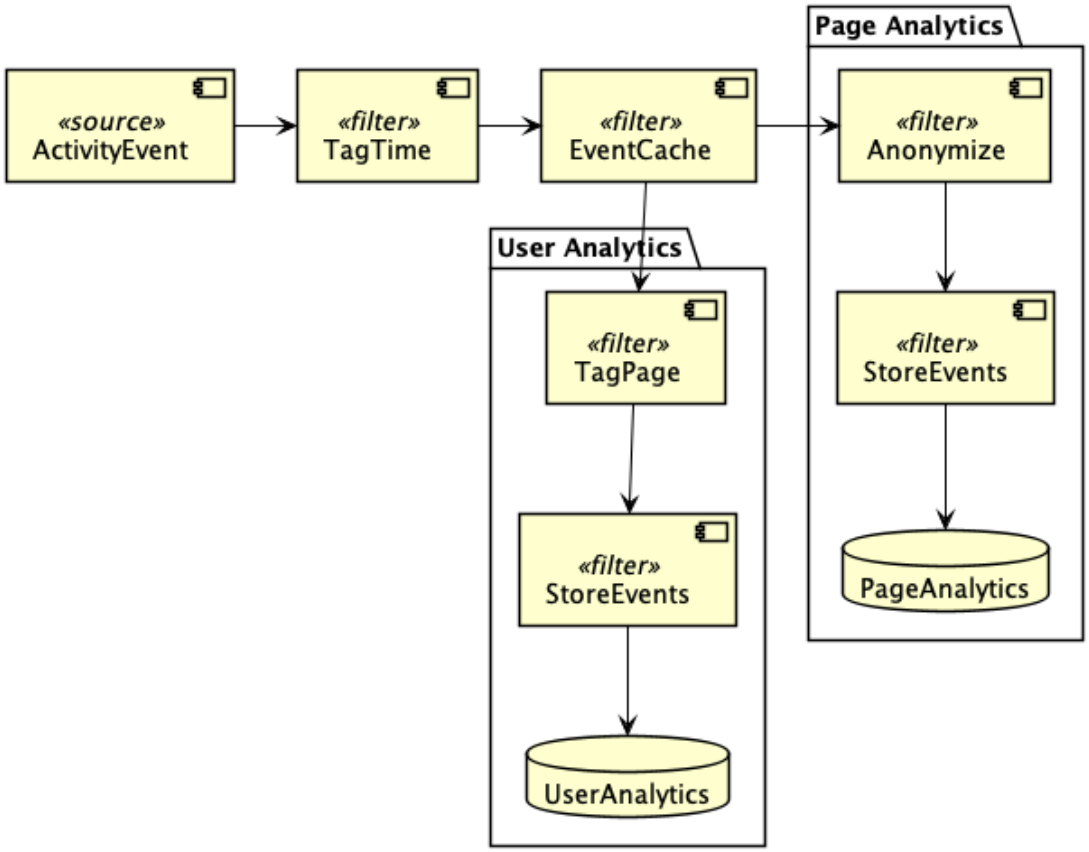


Figure 36: Pipeline architecture for processing activity on a website for later analytics.

Corollary 1. Generic Interface

The interface between filters should be generic.

Corollary **??** follows from definition **??**. In order to reduce the dependence on specific filters, all filters of a system should implement a generic interface. For example, in bash, filters interface through the piping of raw text data. All Unix processes should support raw text input and output.

Corollary 2. Composable Filters

Filters (i.e. Transformers & Testers) can be applied in any order.

Corollary **??** also follows from definition **??**. If filters are independent they can be linked together in any order. This means that a software designer can deliver different behaviour based on the order in which filters are linked. For example, in bash, applying a tester before or after a transformer can lead to different results.

4 Conclusion

A pipeline architecture is a good choice for data processing when interactivity is not a concern. Conceptually pipelines are very simple. Following the principles of a pipeline architecture will deliver a modular system which supports high reuse.

Containers

Software Architecture

March 14, 2022

Brae Webb

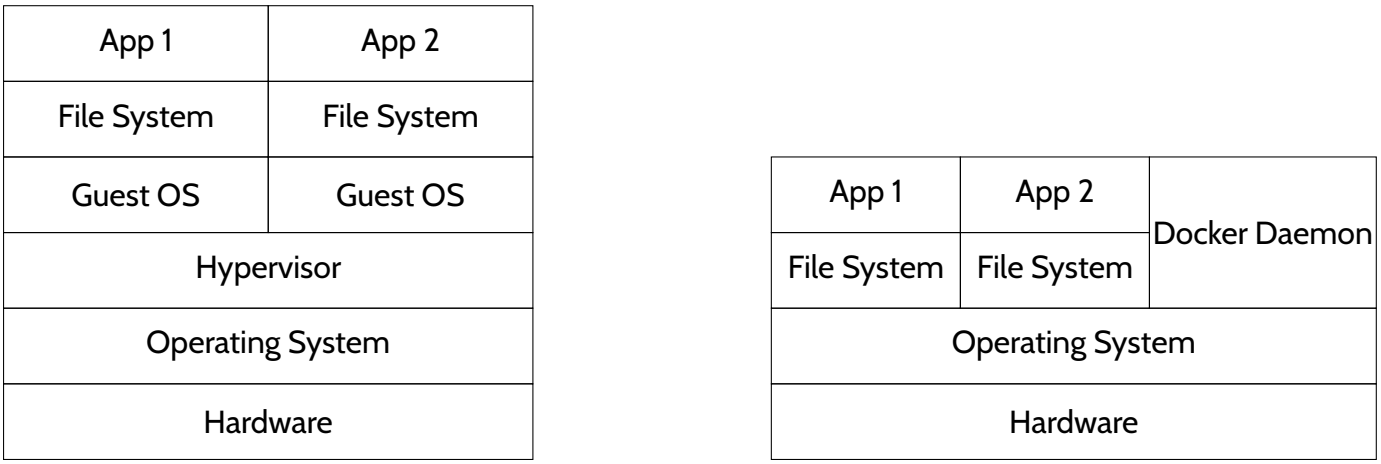
> I don't care if it works on your machine! We are not shipping your machine!
>
> — Vidiu Platon

1 Introduction

As developers, we often find ourselves relying on some magical tools and technologies. Version control, operating systems, databases, and containers, to name a few. Containers, and specifically, Docker, are magical tools which have gained wide-spread industry adoption. Over the past decade Docker has enabled some fanciful architectures and developer workflows. Docker is the proposed solution to the age-old programmer proverb, "it works on my machine!". However, before we subscribe to that belief, let's see how Docker actually works to learn what it is, and what it is not.

2 History and Fundamentals

Relative to other tools in the magical suite, Docker is new on the scene. Docker was first made available publicly in 2013 at PyCon.[68] Pitched to deliver the isolation benefits of Virtual Machines (VMs) while maintaining efficient execution. Virtual machines themselves are a much older invention, dating back to the 1960s, around the time that UQ was having it's first computer installed.[69] The concept of a virtual machine, unlike its implementation, is straight-forward — use software to simulate hardware. From there, one can install an operating system on the simulated hardware and start using a completely isolated, completely new computer without any additional hardware. Of course, this process puts great strain on the real hardware and as such, VMs are deployed sparingly.



(a) Two virtual machines running on a host

(b) Two containers running on a host

Figure 37: Comparison of virtual machines and containers

[68] https://www.youtube.com/watch?v=wW9CAH9nSLs
[69] https://www.youtube.com/watch?v=DB1Y4GrfrZk

Unlike virtual machines, containers do not run on virtual hardware, instead, containers run on the operating system of the host machine. The obvious advantage of this is containers run much more efficiently than virtual machines. Containers however, manage to remain isolated and it is at this point that we should explain how Docker actually works. Docker is built upon two individually fascinating technologies; namespaces, and layered filesystems.

2.1 Namespaces

The first technology, namespaces, is built into the Linux kernel. Linux namespaces were first introduced into the kernel in 2002, inspired by the concept introduced by the Plan 9[70] operating system from Bell Labs in the 1980s. Namespaces enable the partitioning and thus, isolation, of various concepts managed and maintained by an operating system. Initially namespaces were implemented to allow isolated filesystems (the so-called 'mount' namespace). Eventually, as namespaces were expanded to include process isolation, network isolation, and user isolation, the ability to mimic an entirely new isolated machine was realised; containers were born.[71]

Namespaces provide a mechanism to create an isolated environment within your current operating system. The creation of such an isolated environment with namespaces has been made quite easy — you can create an isolated namespace with just a series of bash commands. Niklas Dzösch's talk 'Docker without Docker', uses just 84 lines of Go code (which Docker itself is written in), to create, well, Docker without Docker.[72] But namespaces are just the first technology which enables Docker. How do you prepopulate these isolated environments with everything you need to run a program? To see what is in the isolated environment, we would run `ls` which, oh, requires the `ls` binary. Furthermore, to even consider running a command we need a shell to type it in, so, oh, we should also have a `bash` binary. And so on until, oh, finally, we have a Linux kernel at least![73]

2.2 Layered Filesystem

A core principle of Unix operating systems is that everything is a file. Naturally then, if we want to start using an isolated environment, all we need to do is copy in all the files which together form an operating system, right? Well, yes, kind of. In principle this is all you need do but this would hardly enable the popularity Docker enjoys today.

Say that you want to send your coworker a Docker container which has nginx (a tool for routing web traffic) setup in just the way you need to pass incoming traffic to various components of your application. Let's assume that you have setup nginx in Ubuntu. All you would need to do is zip up all the files which compose the Ubuntu operating system (an impressively small 188MB), then all the files installed by nginx (about 55MB) and finally all the configuration files which you have modified, somewhere in the order of 1000 bytes or 1 KB. In total you are sending your coworker about 243MBs worth of data, less than a gigabyte, so they are not too upset.

Now once we have finished developing our application and we are ready to package it up and send it to the world. Rather than trying to support every known operating system, we bundle all our services in Docker containers, one for nginx, one for mysql, one for our web application, etc, etc. If your applications containers are as popular as nginx, this means one *billion* downloads of your container. At a size of 243MBs, you have contributed 243 *petabytes* to the world's collective bandwidth usage, and that is just your nginx container.

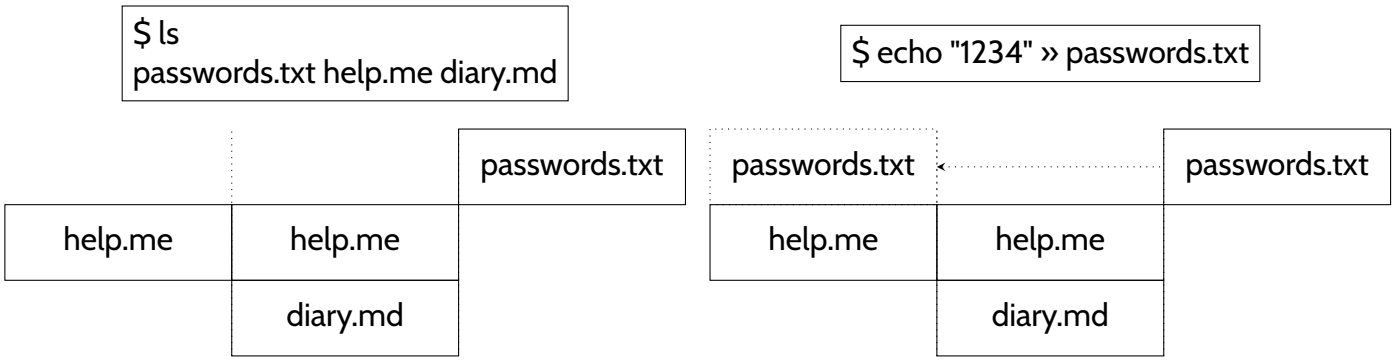
[70] https://p9f.org/

[71] Of course, containers in a rudamentry form existed before introduction to the linux kernel but we have to start somewhere. https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016

[72] https://www.youtube.com/watch?v=7H_eF6hvWg

[73] You might be asking yourself, wait but I have a Windows operating system and I can still run Docker, what gives? The answer, ironically enough, is a virtual machine!

Docker's success over other containerisation applications comes from the way it avoids this looming data disaster. A concept known as the layered, or overlayed, or the stacked filesystem solves the problem. First proposed in the early 1990s, layered filesystems enable layers of immutable files to be stacked below a top-level writable system. This has the effect of producing a seemingly mutable (or writable) filesystem while never actually modifying the collection of immutable files. Whenever a immutable file is 'written to', a copy of the file is made and all modifications are written to this file. When reading or listing files, the filesystem overlays the layers of immutable files together with the layer of mutable files. This gives the appearance of one homogeneous filesystem.



(a) Read operation in a layered filesystem

(b) Write operation in a layered filesystem

Figure 38: Read and write operations in a layered filesystem. The leftmost column in each diagram represents the most recent 'writable' layer.

Docker uses this technique to reduce the amount of duplicated files. If you have Docker containers which run nginx, MySQL, and Python but all containers run on Ubuntu, then your computer will only need to store one copy of the Ybuntu filesystem. Each container will store the changes to the base operating system required to install each application and project that layer over the immutable Ubuntu filesystem.

2.3 Summary

While Docker itself only came out in 2013, the two primary technologies which it is composed of, namespaces and the layered filesystem, were around since the early 1990s. Docker combines these technologies to enable applications to run in an isolated environment which can be efficiently replicated across different host computers. The reproducablity of Docker containers and the fact that they are so light weight makes them an ideal target for two important aspects of modern development; developers simulating production environments locally, and duplicating production machines to scale for large loads of traffic.

3 Docker FROM scratch

Now that we understand the fundamentals of how Docker works, let's start building our very first Docker container. To follow along, you will need to have Docker installed on your computer[74] and have access to basic Unix command line applications.[75]

To start with, we will write a `Dockerfile` which builds a Docker image without an operating system and just prints 'Hello World' [**?**]. The Docker 'code' is written in a `Dockerfile` which is then 'compiled' into a Docker image and finally run as a Docker container. The first command in your `Dockerfile` should always be `FROM`. This command tells Docker what immutable filesystem we want to start from, often, this will be your chosen operating environment. In this exercise, since we do not want an operating system, we start with `FROM scratch`, a no-op instruction that tells Docker to start with an empty filesystem.

[74] https://docs.docker.com/get-docker/

[75] For windows users, we recommend Windows Subsystem for Linux.

Let's get something in this container. For this, we will use the `COPY` command which copies files from the host machine (your computer) into the container. For now, we will write `COPY hello-world /`, which says to copy a file from the current directory named `hello-world` into the root directory (`/`) of the container. We do not yet have a `hello-world` file but we can worry about that later. Finally, we use the `CMD` command to tell the container what to do when it is run. This should be the command which starts your application.

```
FROM scratch
COPY hello-world /
CMD ["/hello-world"]
```

Next, we will need a minimal hello world program. Unfortunately, we will have to use C here as better programming languages have a bit too much overhead. For everyone who has done CSSE2310, this should be painfully familiar, create a `main` function, print hello world, with a new line, and return 0.

```
#include <stdio.h>

int main() {
    printf ("Hello World\n");
    return 0;
}
```

Let's try running this container. Try to guess if this is going to work. Why? Why not?
First, the hello world program needs to be compiled into a binary file.

```
    >> gcc -o hello-world hello-world.c
    >> ls
    Dockerfile hello-world hello-world.c
```

Next we will use the `Dockerfile` to build a Docker image and run that image. Images are stored centrally for a user account so to identify the image, we need to tag it when it is built, we will use 'hello'.

```
    >> docker build --tag hello .
    >> docker run hello
    standard_init_linux.go:228: exec user process caused: no such file or
       directory
```

Unless this is Docker's unique way of saying hello world, something has gone terribly wrong. Here we are illustrating the power of Docker isolation as well as the difficulty of not having an operating system. This very simple hello world program still relies on other libraries in the operating system, libraries which are not available in the empty Docker container. `ldd` tells us exactly what `hello-world` depends on. The hello world program can be statically compiled so that it does not depend on any libraries [**?**].

```
>> ldd hello-world
    linux-vdso.so.1 (0x00007ffc51db1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcff8553000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fcff8759000)
>> gcc -o hello-world hello-world.c -static
>> ldd hello-world
    not a dynamic executable
>> docker build --tag hello .
>> docker run hello
Hello World
```

Now we have a Docker image built from scratch, without an operating system, which can say 'Hello World'!

> If you are interested in exploring in more depth, try using the '`docker image inspect hello`' and '`docker history hello`' commands.

Microkernel Architecture

Software Architecture

March 14, 2022

Richard Thomas

1 Introduction

The microkernel architecture aims to deliver sophisticated software systems while maintaining the quality attributes of simplicity and extensibility. This is achieved by implementing a simple core system that is extended by plug-ins that deliver additional system behaviour. Microkernel is also known as a "plug-in" architecture.

Many common applications use the microkernel architecture. Web browsers, many IDEs (notably Eclipse), Jenkins, and other tools all use this architecture. They deliver their core functionality and provide a plug-in interface that allows it to be extended. Eclipse[76] is famous as a simple text editor that can be extended to be a sophisticated software development tool through its plug-in interface.

Definition 13. Microkernel Architecture

A core system providing interfaces that allow plug-ins to extend its functionality.

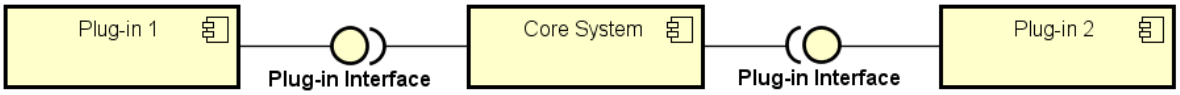


Figure 39: Microkernel architecture – generic structure.

For example, a web browser provides the core behaviour of rendering web pages. Plug-ins extend the browser with specialised behaviour, such as a PDF viewer or dark mode reader.

2 Terminology

The microkernel architecture consists of four elements. The *core system*, *plug-ins* and *plug-in interface* shown in figure **??**, plus a *registry*.

Core system implements the system's base functionality.

Plug-ins extend the system by providing independent, specialised behaviour.

Plug-in interface describes how the core system and plug-ins interact.

Registry tracks which plug-ins are available to the core system and how to access them.

The core system implements the minimal functionality that provides the base, or framework, of the system. This may be the core functionality of the system, like the Eclipse text editor or a web browser's page rendering engine. Alternatively, the core system may implement a general processing path, like a payroll system's payment process.

[76] https://www.eclipse.org/downloads/packages/

The payment process may be simply, identify an employee, calculate their fortnightly pay, and send payment to their bank account. Calculating pay can be a very complex process[77]. There are different pay rates, bonuses, incentives, salaried staff, staff paid commission, staff paid hourly rates, overtime rates, penalty rates, deductions, taxes, and many other pay related calculations to consider for each individual.

Plug-ins are independent components that extend the behaviour of the core system. The simple approach is that the system is delivered as a monolith, including the core system and the plug-in. In this case the core system uses the plug-in via a method invocation.

In the payroll example, plug-ins can remove the complexity of different pay adjustment calculations from the core system, by moving each calculation to a separate component. This also improves extensibility, maintainability and testability of the system. New calculations can be added to the system by implementing a new plug-in. As each plug-in is independent of the others, it is easier to implement a new calculation, or changing an existing one, than trying to do so in a large monolith. New plug-ins can be tested independently and then integrated into the system for system testing.

There is usually a single standard interface between the core system and the plug-ins for a domain. The interface defines the methods that the core system can invoke, data passed to the plug-in, and data returned from the plug-in. A plug-in component implements the interface and delegates responsibilities within the component to deliver its behaviour.

Figure **??** is an example of a possible interface for the pay adjusment calculation plug-ins. Each pay adjustment plug-in returns a `MonetaryAmount` indicating the amount by which the employee's base pay is to be adjusted in the pay period. The amount may be positive or negative. The `employee`, `periodDetails`, `employeeConditions` and `basePay` objects are passed as parameters to the plug-in. The plug-in can extract the data it needs from these objects to perform its calculation. The `periodDetails` object provides data about the work performed by the employee during the pay period (e.g. time worked, overtime, higher duties adjustments, ...). The `employeeConditions` set provides data about each of the employee's pay conditions (e.g. before tax deductions, union membership, ...).

```
1 public interface PayAdjustmentPlugin {
2     public MonetaryAmount adjustPay(Employee employee,
3                                     WorkDetail periodDetails,
4                                     Set<Condition> employeeConditions,
5                                     MonetaryAmount basePay);
6 }
```

Figure 40: Example plug-in interface for payroll system.

The registery records which plug-ins are available to the core system and how they are accessed. For the simple case of plug-ins being used by method invocation, the registry just needs to record the name of the plug-in and a reference to the object that implements the plug-in's interface. For the payroll example, this could be a simple map data structure. The core system could lookup a plug-in by its name and then apply it by invoking the `adjustPay` method on the plug-in object.

3 Microkernel Principles

While the concept of a microkernel architecture is straightforward, there are some principles which should be maintained to produce a maintainable and extendable architecture.

[77] See the Queensland Health payroll disaster https://www.henricodolfing.com/2019/12/project-failure-case-study-queensland-health.html

Definition 14. Independent Plug-in Principle

Plug-ins should be independent, with no dependencies on other plug-ins. The only dependency on the core system is through the plug-in interface.

Plug-ins should be independent of each other. If a plug-in depends on other plug-ins it increases the complexity of the design. This complexity is called *coupling*[78], which is a measure of how dependent different parts of a system are on each other [**?**]. High coupling (many dependencies) makes it difficult to understand the software, which in turn makes it difficult to modify and test the software. Consequently, if a plug-in depends on other plug-ins it requires understanding the dependencies on the other plug-ins to modify or test the plug-in. This can lead to an architecture that resembles a "big ball of mud".

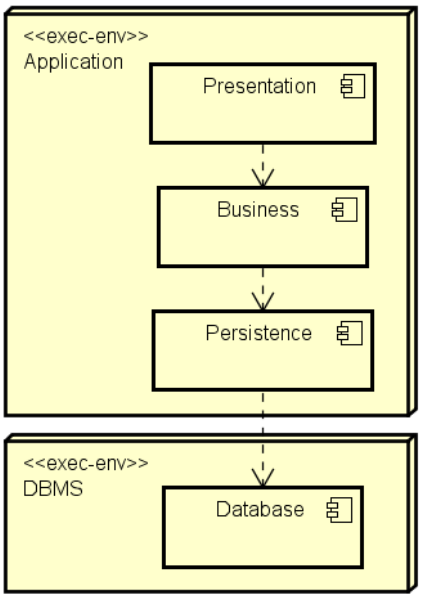
Plug-ins and the core system should be *loosely* coupled. The core system should only depend on the plug-in interface and data returned via that interface, not any implementation details of individual plug-ins. Plug-ins should only depend on the data passed to them via the plug-in interface. Plug-ins should not rely on implementation details of the core system, nor its datastore. If plug-ins and the core system are not isolated from each other by an interface, then any changes to the core system may require changes to some or all of the plug-ins. Similarly, the plug-in interface should mean that any changes to the plug-in will have no impact on the core system.

Definition 15. Standard Interface Principle

There should be a single interface that defines how the core system uses plug-ins.

To provide an extensible design, the core system needs a standard way to use plug-ins. This means that the plug-in interface needs to be the same for all plug-ins. That way the core system can use any plug-in without needing to know details about how the plug-in is implemented. This again is about reducing coupling between the core system and the plug-ins. The standard interface principle means that there is no additional complexity if the core system uses two plug-ins or thousands of plug-ins.

4 Architecture Partitioning



r0.32

[78] https://leanpub.com/isaqbglossary/read#term-coupling

4.1 Technical Partitioning

The course notes about layered architecture [**?**] described the idea of partitioning an architecture into layers, as in figure **??**.

This approach to partitioning the architecture is called *technical partitioning*. Each layer represents a different technical capability. The presentation layer deals with user interaction. The business layer deals with the application's core logic. The persistence layer deals with storing and loading data. The database layer deals with file storage and access details.

An advantage of technical partitioning is that it corresponds to how developers view the code. Another advantage is that all related code resides in a single layer of the architecture, making each layer cohesive[79] from a technical perspective. This makes it easier to work with different parts of the code that deal with the same technical capability. The *layer isolation*, *neighbour communication*, *downward dependency*, and *upward notification* principles help reduce coupling between layers.

[79] https://leanpub.com/isaqbglossary/read#term-cohesion

A disadvantage of technical partitioning is that business processes cut across technical capabilities. Consider when a customer adds a new product to their shopping cart in the Sahara eCommerce example from the architectural views notes [**?**]. In a technically partitioned architecture, code to implement this is required in all layers of the architecture. If the team is structured with specialist developers who work on different layers of the architecture, they all need to collaborate to implement the behaviour, which adds communication overheads to the project.

Another disadvantage is the potential for higher data coupling, if all domain data is stored in a single database. If it is decided that the system needs to be split into distributed software containers for scalability, the database may need to be split to have separate databases for each distributed container. If the database was originally designed to cater for a monolith architecture, large parts of the database may need to be refactored.

Note that in figure **??**, and most of the following figures, nodes from UML deployment diagrams are used to explicitly indicate where components are executing in different environments.

4.2 Domain Partitioning

Domain partitioning is an alternative approach, where the architecture is split into partitions (or major components) corresponding to independent business processes or workflows (the *domains*). The implementation of a domain is responsible for delivering all of that workflow's behaviour. Eric Evans popularised domain partitioning in his book *Domain-Driven Design* [**?**]. Figure **??** is an example of domain partitioning for an on-line store, similar to the Sahara eCommerce example from the architectural views notes [**?**].

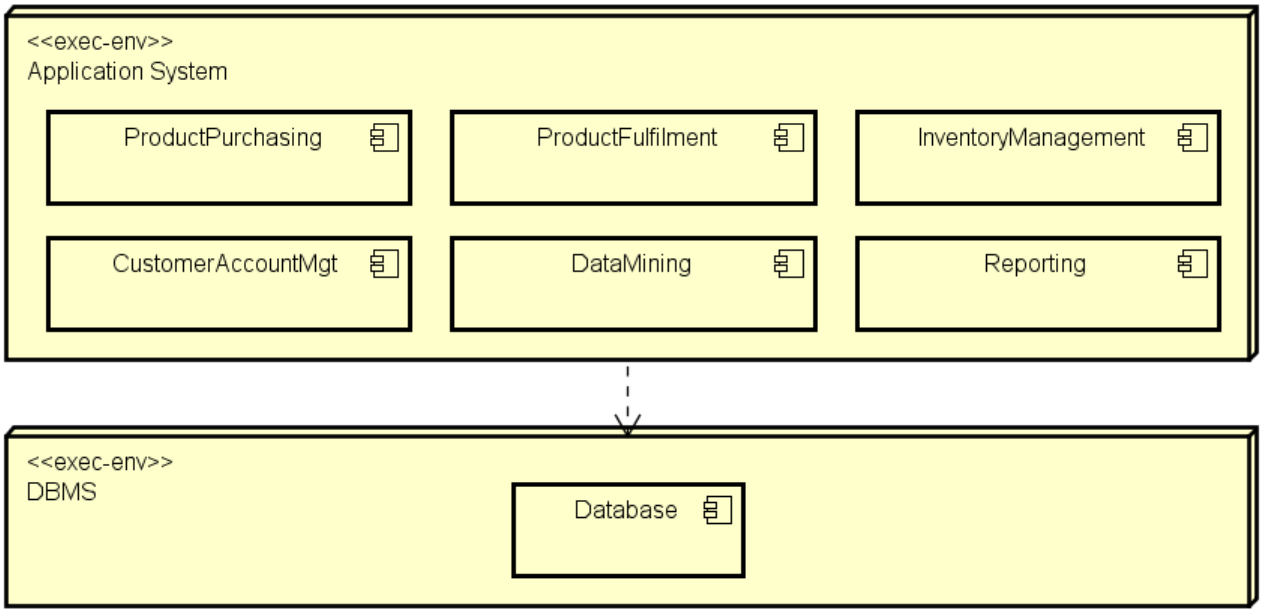


Figure 41: Example of domain partitioning.

An advantage of domain partitioning is that it can model how the business functions, rather than it being structured based on technical decisions. This results in message flows within the software matching how communication happens in the problem domain. This makes it easier to discuss features, and the high-level view of how they are implemented, with business stakeholders who are familiar with how the business operates.

If the domain partitions are independent of each other, it usually makes it easier to split the system into distributed software containers. Even if the system starts with a single database, the independence of domain partitions is likely to lead to a design with lower data coupling, making it easier to split the database.

Another advantage of domain partitioning is that it fits well with an agile or continuous delivery process. A single user story will be implemented entirely within one domain. This also makes it easier to implement multiple user stories concurrently, if they are all from different domains.

A disadvantage of domain partitioning is that *boundary* code is distributed throughout all the partitions. This means that any large changes to a particular boundary (e.g. changing the persistence library) will result in changes to all domain partitions. Boundary code is code that interacts with actors or agents outside of the system. The two common technical partitions of the presentation and persistence layers are examples of boundary code. The presentation layer is the boundary between the system and the users. The persistence layer is the boundary between the system and the storage mechanisms. Other boundary partitions may be for message queues or network communication.

It is possible for each domain partition to use technical partitioning within it, to provide some of the lower coupling benefits of a layered architecture for the domain.

4.3 Achitecture Patterns and Partitioning

Different architecture patterns are biased more towards technical or domain partitioning. Those that focus on technical structure are biased towards technical partitioning (e.g. layered architecture). Those that focus on message passing between components are biased towards domain partitioning (e.g. microservices architecture). The core system of the microkernel architecture can follow either technical or domain partitioning.

A consequence of this leads to a variation of principle **??**.

Definition 16. Domain Standard Interface Principle

Each *domain* should have a single interface that defines how the domain uses plug-ins.

Some domains may share a plug-in interface, but allowing domains to have different plug-in interfaces means that each business process can be customised independently by a set of plug-ins.

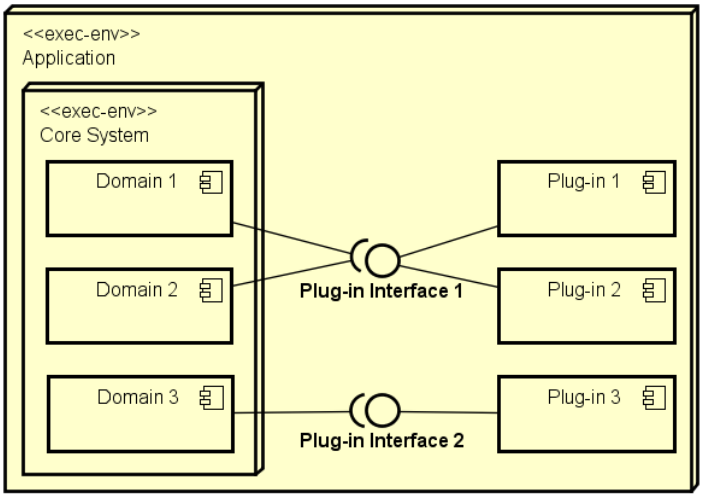


Figure 42: Microkernel architecture with domains.

In figure **??**, nodes are used to explicitly indicate that the core system and plug-ins are conceptually separate from each other. They are all contained within the Application node to indicate that they are still all part of one monolithic application.

5 Extensions

While the microkernel architecture often implements the core system as a monolith and it, along with all its plug-ins, is often deployed as a single monolith (e.g. a web browser), that is not the only approach to using the microkernel architecture.

5.1 Distributed Microkernel

The internal partitions of the core system can be distributed to run on different computing infrastructure. Figure **??** is an example of implementing a distributed microkernel architecture using three computing tiers.

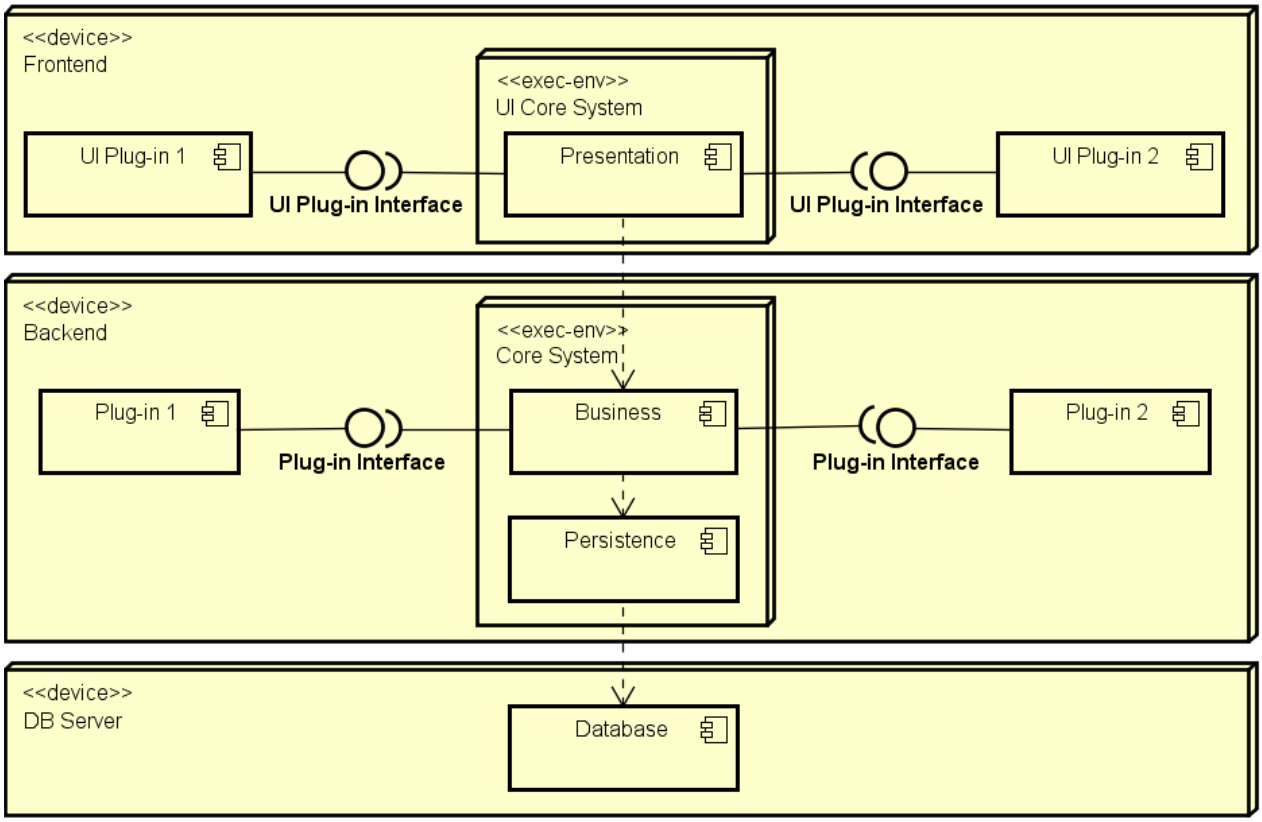


Figure 43: Microkernel architecture distributed by technical partitions.

The presentation layer runs on a frontend such as a web or mobile application. It is possible that the presentation layer could have its own set of plug-ins. The business and persistence logic run on a backend server and have their own set of plug-ins. The database runs on a separate database server, and typically does not have any plug-ins from the application's perspective.

5.2 Alternative Contracts

Figure **??** was an example interface that could be used if plug-ins were invoked directly within the core system. This is possible if the plug-ins are implemented within the application or are loaded as code modules at run time (e.g. via JAR or DLL files).

There may be times when plug-ins are external services used by the core system. In those situations communication is via some communication protocol. Remote method invocation would still allow the core system to use a compiled interface. Registration of components would become a little more complicated, as the core system would need to a communication interface to be notified of the availability of plug-ins.

More commonly, the communication protocol will be some other mechanism (e.g. REST, as shown in figure **??**). In this case, the registry needs to record more information than just the name and interface of a plug-in. The registry will need information about the communication pathway (e.g. URL of the REST endpoint). It will also need details about the data structure passed to the plug-in and returned by the plug-in.

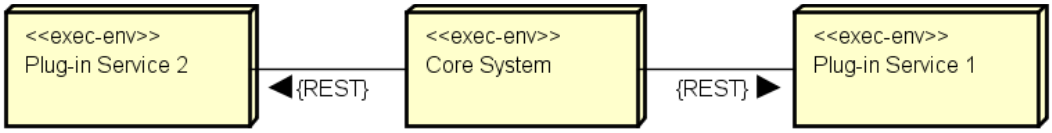


Figure 44: Microkernel architecture – separate plug-in services with REST communication.

5.3 Plug-in Databases

As mentioned in the discussion of principle **??**, plug-ins should not use the core system's data directly. The core system should pass to the plug-in any data that it needs. For sophisticated plug-ins, they may need their own persistent storage. They should not request the core system to manage this. Rather, these plug-ins should maintain their own databases, as shown in figure **??**.

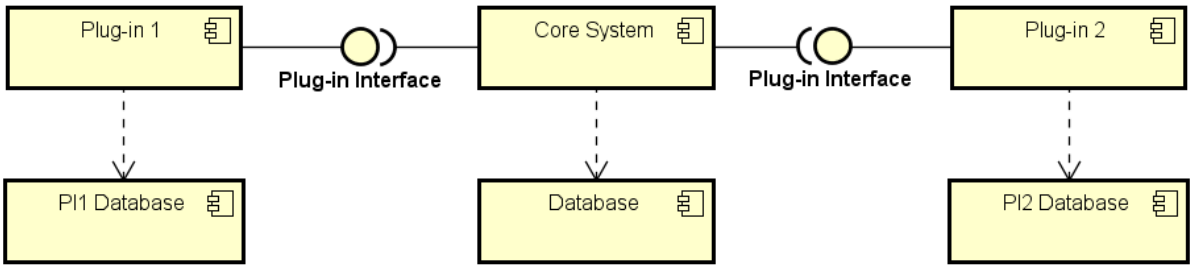


Figure 45: Microkernel architecture with plug-ins maintaining their own databases.

5.4 Non-Standard Interfaces

Principles **??** and **??** indicate that each domain, or the entire core system, should use a single plug-in interface. This is not always possible when the plug-ins are services provided by external agencies. In these situations you would use the adapter design pattern[80] to isolate the external service's interface from the core system.

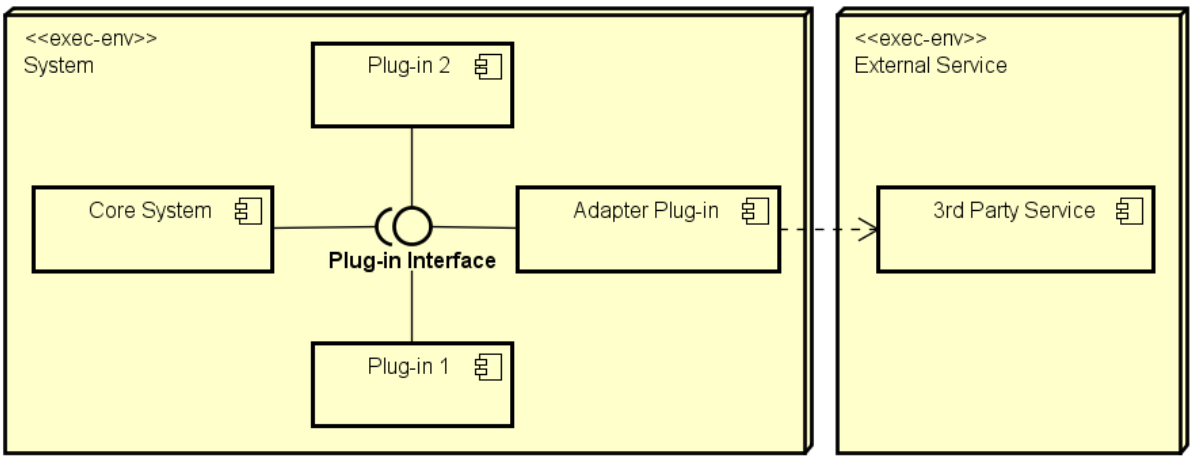


Figure 46: Adapter plug-ins to work with third party services.

As shown in figure **??**, an adapter plug-in is implemented that manages communication with the external service. From the core system's perspective, the external system is just like any other plug-in.

[80] https://refactoring.guru/design-patterns/adapter

6 Conclusion

The microkernel architecture is a suitable option for systems were extensibility is a key quality attribute. Following the microkernel design principles will lead to a design where the core system and plug-ins are not affected by changes to each other. Existing plug-ins can be replaced and new plug-ins added with no impact on the core system.

Service-Based Architecture

Software Architecture

March 21, 2022

Richard Thomas

1 Introduction

Service-based architecture is one of the simpler, but still flexible, distributed architectural styles. It provides good support for delivering quality attributes of modularlity, availability, deployability, and simplicity (in the context of a distributed system). The key characteristic of a service-based architecture is that it uses domain partitioning, and each domain becomes its own distributed service. This partitioning provides high-level modularisation that helps ensure the domain partitions are independent of each other. The distribution of domains means that multiple instances of a service can be made available through a load balancer, providing better availability and some level of scalability.

Many medium-sized bespoke systems[81] are built using a service-based architecture. For example, an on-line store might be partitioned into services such as ProductBrowsing, ProductPurchasing, ProductFulfilment, InventoryManagement, and CustomerAccountManagement. Each of these could be independent distributed services that use a shared database.

Definition 17. Service-based Architecture

The system is partitioned into business domains that are deployed as distributed services. Functionality is delivered through a user interface that interacts with the domain services.

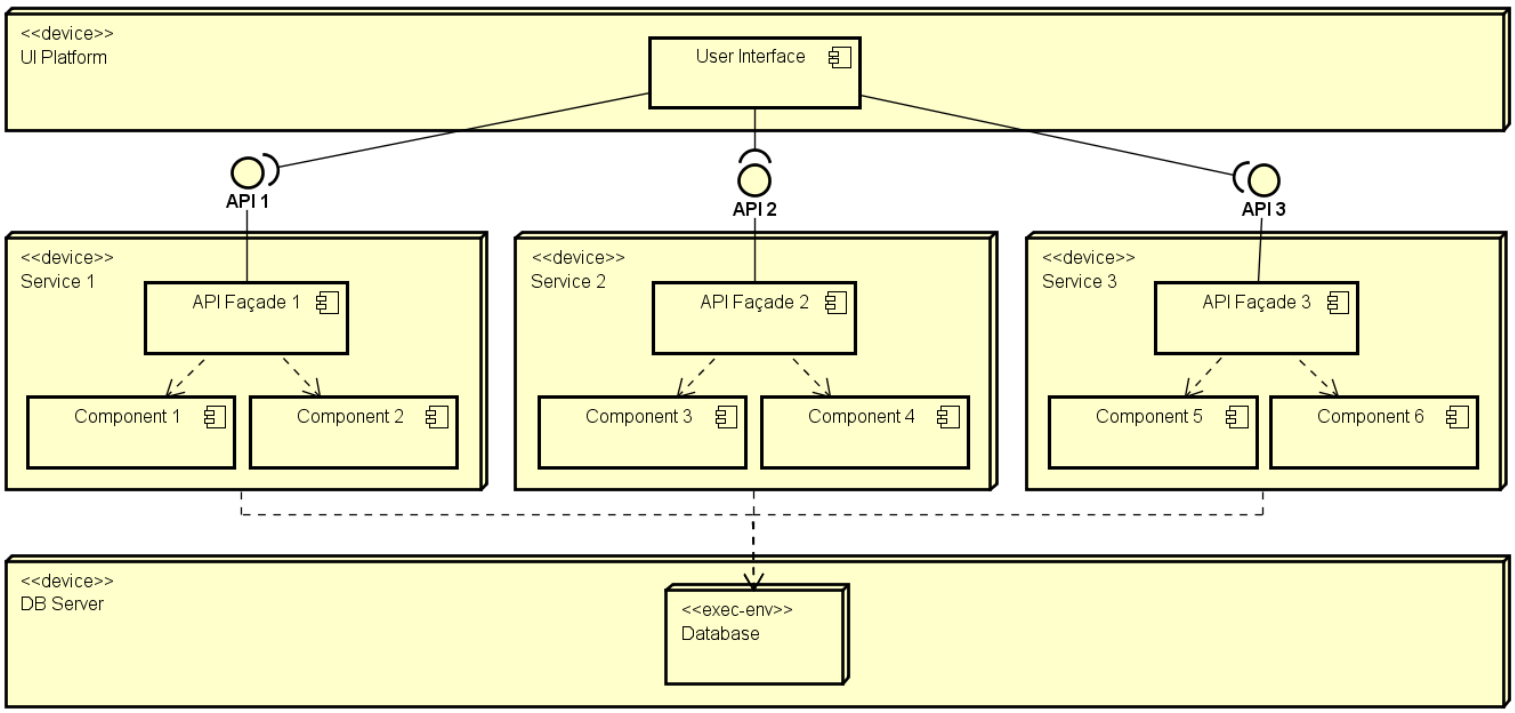


Figure 47: Service-based architecture – general structure.

[81] *Bespoke systems* are custom designed and built for a single organisation.

2 Terminology

The service-based architecture consists of four elements. The *user interface*, *services*, *service APIs*, and *database*, as shown in figure **??**.

User Interface provides users access to the system functionality.

Services implement functionality for a single, independent business process.

Service APIs provide a communication mechanism between the user interface and each service.

Database stores the persistent data for the system.

The user interface runs as a standalone process to manage user interactions. It communicates with the services through their service APIs to invoke system behaviour. This requires a remote access communication protocol, such as REST, a message transport service, remote method invocation, SOAP[82] or some other protocol.

To reduce coupling between the user interface and the service APIs, and to provide easier extensibility, the user interface often uses a service locator design pattern[83] to manage access to the services[84]. This provides a registry of the services and the details of the API of each service. The user interface uses the registry to retrieve an object that communicates with a service through its API. This also makes it easier to add new services to the application, as the details of the new service are encapsulated in the registry.

Services implement the application logic for independent business processes. These are often called "coarse-grained" services, as each one implements a significant part of the system's functionality. Each service is deployed on its own computing infrastructure. Commonly, there is a single instance of each service but it is possible to run multiple instances of services. Multiple instances of services improves availability because if one instance goes down, other instances can handle future requests from the user interface. To provide higher reliability, in the context of running multiple instances of a service, it should implement the stateless service pattern[85]. A system running multiple instances of a service, that does not implement the stateless service pattern, would still have higher availability if a service instance went down, as other instances could handle future requests. But, any user in the middle of performing a business process would need to restart their activity, thus lowering system reliability.

Services implement their own service API using the façade design pattern[86]. This defines the communication protocol used between the user interface and the service. For simplicity, usually all services use the same communication protocol. The façade design pattern reduces coupling between the user interface and the services, as the user interface does not depend on the implementation details of the services.

The service API provides the benefit that different user interfaces can all use the same services. For example, an application with web and mobile interfaces could use the same set of distributed domain services.

The database stores persistent data for the system. Often, a single database is shared by all the services as it is common that some data will be shared between services. A shared database makes it easier to maintain data integrity and consistency. This is because each service implements a single business process and can usually perform all transaction management related to the data involved in the process. For example, the ProductPurchasing service for an on-line store can manage the entire database transaction for making an order. If the product is no longer available or payment fails, the service can rollback the transaction to ensure data integrity.

[82] https://www.w3schools.com/xml/xml_soap.asp

[83] https://www.baeldung.com/java-service-locator-pattern

[84] Martin Fowler provides good commentary about using the service locator pattern at https://martinfowler.com/articles/injection.html#UsingAServiceLocator. He expands further on some tradeoffs of the pattern than other more superficial descriptions of the pattern.

[85] https://www.oreilly.com/library/view/design-patterns-and/9781786463593/f47b37fc-6fc9-4f0b-8cd9-2f41cb36xhtml

[86] https://refactoring.guru/design-patterns/facade

3 Design Considerations

A service-based architecture is typically used for medium-sized systems. This is because the user interface interacts with all services through their APIs. The user interface becomes more complicated when it has to deal with many services. If the services follow the common approach of using a shared database, it means the the greater the number of services, the more complicated the database design becomes. There is also a potential performance bottleneck if many services are using a shared database. Strategies to improve database performance, like replicated databases, defeat some of the benefits of a shared database (e.g. consistency). Typically a service-based architecture will have six to twelve domain services. There is no specific upper or lower limit on the number of services allowed, it is a tradeoff that architects make based on all the requirements for the system.

Coarsed-grained services will usually have some internal complexity that requires some architectural structure. This internal structure may follow either technical or domain partitioning. Technical partitioning will typically consist of three layers, the API façade, business logic and persistence. Domain partitioning will break the service domain into smaller components related to each part of the domain's behaviour. For example, the ProductPurchasing service domain may have components for the internal behaviours of checking out, payment and inventory adjustment. Payment would use an API to process payment through a financial service gateway. Figure **??** provides an example of the structure for both technical and domain partitioning of a service.

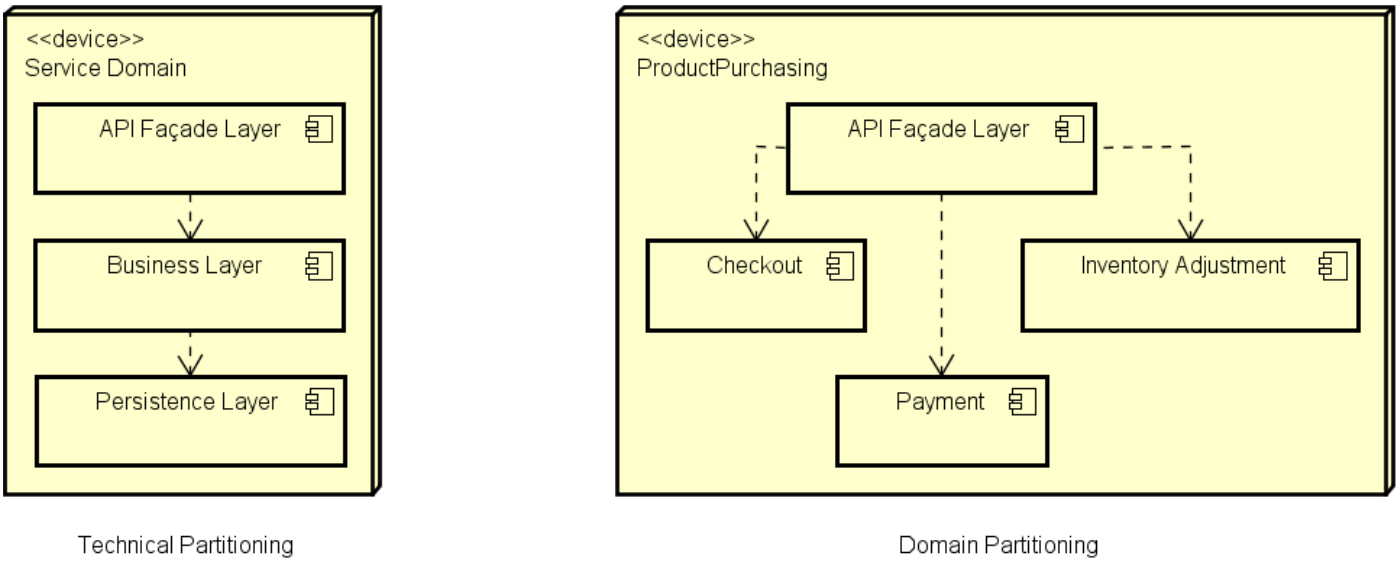


Figure 48: Partitioning options for a service domain.

Consequences of a shared database are increased data coupling between the services and lower testability. Increased data coupling means that if one service changes its persistent data, then all services that share that data need to be updated, as well as the tables storing the data in the database. Lower testability is the consequence of shared data and services implementing complete business processes. A small change to one part of the service requires the entire service to be tested, and all other services that share data with the service also need to be tested.

To mitigate data coupling, design a minimal set of shared (or *common*) persistent objects and their corresponding tables in the database. Implement a library containing the shared persistent classes that is used by all services. Restrict changes to the shared persistent classes and their database tables. Changes may only occur after consideration of the consequences to all services. A variation is to not only have shared persistent objects, but other persistent objects that are only shared with a subset of services.

Each service may have its own set of persistent objects and corresponding database tables. These are independent of other services, so there are no external consequences to changing these within a service. Figure **??** is an example of shared persistent objects and a service with its own persistent objects.

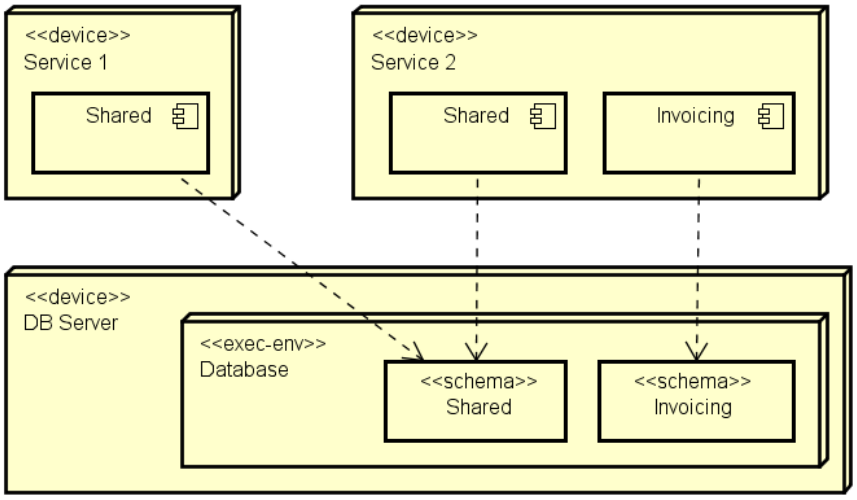


Figure 49: Database logical partitioning example.

4 Service-Based Principles

There are a couple of principles which should be maintained when designing a service-based architecture to produce a simple, maintainable, deployable and modular designs.

Definition 18. Independent Service Principle

Services should be independent, with no dependencies on other services.

Services should be independent of each other. If a service depends on other services they either cannot be deployed separately, or they require communication protocols between services, which increases the coupling and complexity of the system design.

Definition 19. API Abstraction Principle

Services should provide an API that hides implementation details.

The user interface should not depend on implementation details of any services. Each service should publish an API that is a layer of abstraction between the service's implementation and the rest of the system. This provides an interface through which the service can be used and reduces coupling between the service and its users. In a service-based architecture, the user interface is the primary client of service APIs but it is not necessarily the only client. Auditing services may also need to use domain services. In more sophisticated environments, services may be shared across different systems.

5 Extensions

There are a few common variations of the service-based architecture to consider.

5.1 Separate Databases

The first variation we will consider is to have separate databases for each service. This extends the idea of logical partitioning your database, as described in section **??**. Figure **??** shows a few options of how this can be implemented.

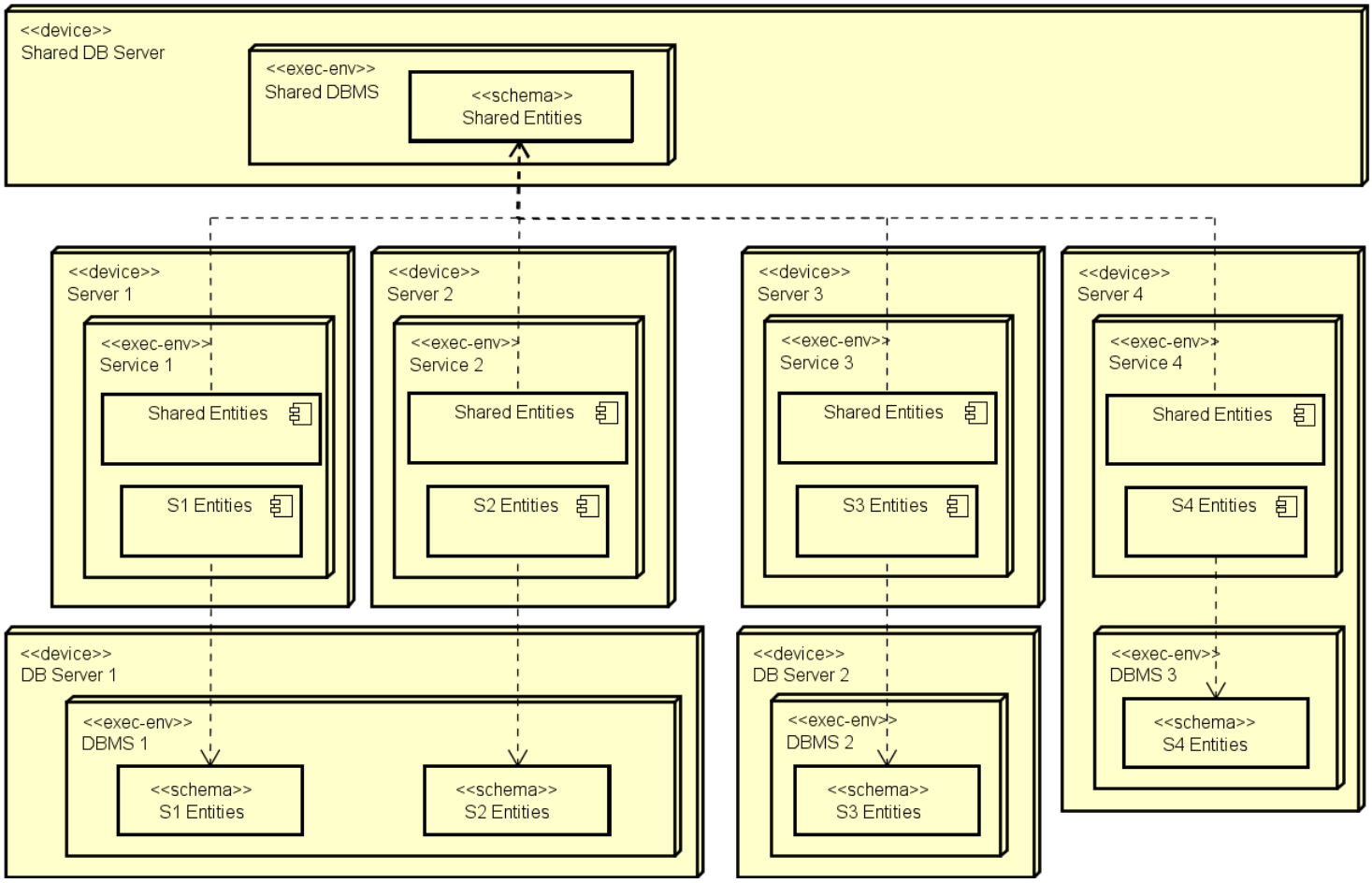


Figure 50: Separate databases example.

In figure **??**, there is a shared database that contains the entity data that is shared across services. Service 1 and 2 are deployed on separate servers but share a single database server that hosts different tables for each service. Service 3 has communicates with its own database server that hosts its tables. Service 4 uses a database that is running on the same server as the service.

Each of these approaches have their own advantages and disadvantages. A key consideration is whether a service has enough unique data to make it worth creating a separate database for just the service. If most data is shared between services, it may be easier to implement the system with just a single shared database. If there is some data that is unique to some services, a single database server with either logical partitioning of the data or even indepedent database services, may provide enough performance for the system. A separate database server for some or all services provides greater flexibility for scaling, if database performance is likely to become a bottleneck for the service. Running an independent database on the same server as the service provides easier communication with the database and may suit cases where an independent database is useful but it is not large enough to warrant to run on its own server.

5.2 Separate User Interfaces

A similar variation can be applied to the user interface, so that there are separate user interfaces for some services. This allows separate interfaces to be implemented for different user interface domains (e.g. standard users, power users, administrators, ...). Figure **??** shows some options of how this can be implemented.

In figure **??**, there is one user interface that interacts with two services, like the single interface in the general structure shown in figure **??**. There are then two interfaces that interact with service 3. A benefit of the service API is that it allows multiple interfaces to be implemented to work with the backend services.

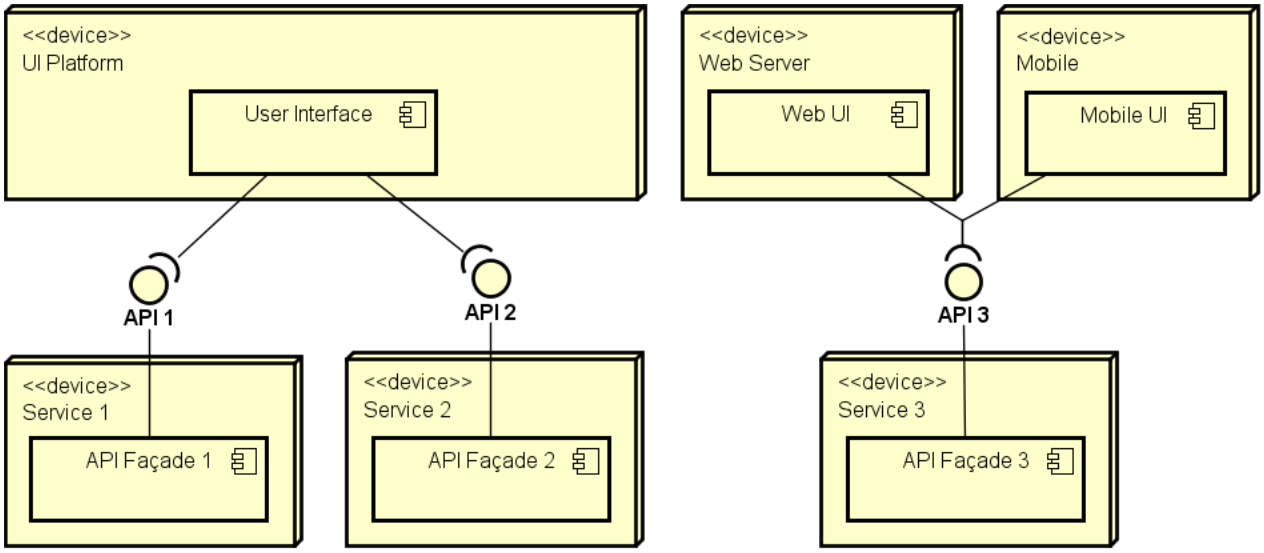


Figure 51: Separate user interfaces example.

5.3 API Layer

It is possible to place an API layer between the user interface and the services, as shown in figure **??**. The API layer is a reverse proxy or gateway that provides access to the services, while hiding details of the services.

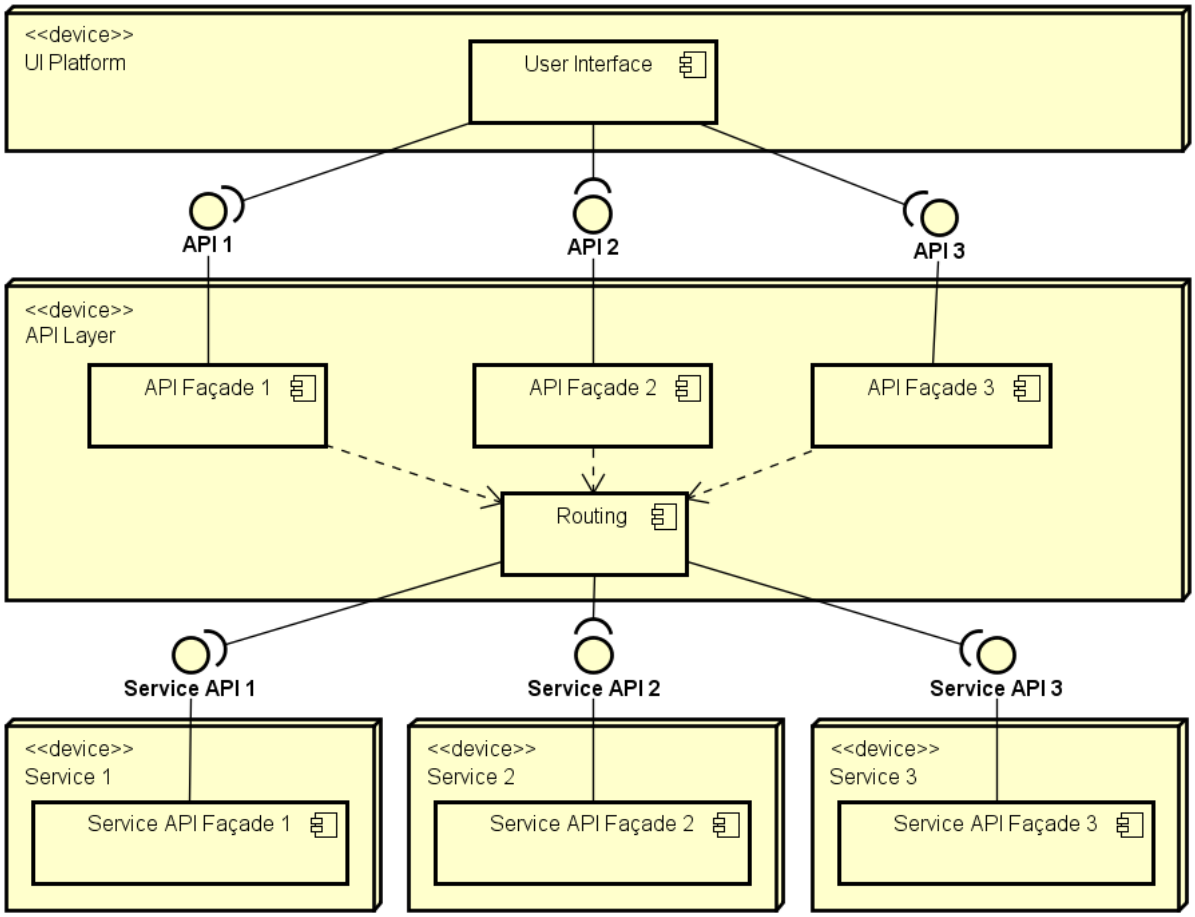


Figure 52: API layer separating the user interface from services.

If any domain services are used by external systems, a reverse proxy hides the internal network struc-

ture of your system's architecture. This extra layer of abstraction means that you can expose a different interface to external systems than to internal systems. This can help deliver the security principle of least privilege.

The API layer allows the implementation of *cross-cutting* concerns to be separated from the user interface. For example security policies, user access logging, or service discovery could be implemented in the API layer.

Service discovery allows new services to be registered and provides a mechanism for clients to "discover" the new services. The API layer would provide an interface that allows services to be registered, including information about their interface. The API layer would also provide an interface to clients that allows them to query what services are available and to retrieve an interface that the client can use to access the service through the API layer.

The API layer can also perform load balancing by delegating service requests to different servers running the same domain service.

6 Service Oriented Architecture

Service Oriented Architecture (SOA) is an extension of service-based architecture. In SOA, each service implements a single business process and provides one or more interfaces that allow access to its functionality. The primary purpose of the API layer is to provide a robust service discovery platform and to implement security policies. Systems are implemented by integrating a set of services to deliver the required functionality. Consequently, the user interface is part of the system being implemented and not part of the service architecture. SOA requires each service to be independent of the others, including very low data coupling. Typically in SOA, each service would have its own database.

SOA sounds like a good idea, but in practice many organisations made compromises that neutered much of the expected benefits. Business processes are often not perfectly independent and there is often data dependencies between processes. If these dependencies are not managed well, the services are not independent. This results in services having to be used together, which reduces their reusability and composability.

Another common issue was a poor service discovery implementation. One problem was that the mechanism to discover new services required too much knowledge about the services. This results in the clients needing to know the services, before they can discover them. Another problem was poor interface design, that caused dependencies on some implementation details of the services.

Microservices were designed to deliver the promised benefits of SOA without some of the implementation issues. In particular, microservices deliver much smaller elements of functionality, reducing the possibility of tight coupling between services. There are challenges in designing and delivering microservices to achieve the benefits. These will be explored later in the course.

7 Conclusion

Service-based architecture is an approach to designing a distributed system that is not too complex. Domain services provide natural modularity and deployability characteristics in the architecture design. Well designed service APIs improve the encapsulation and hide implementation details of the services.

Infrastructure as Code

Software Architecture

March 21, 2022

Brae Webb

1 Introduction

Configuration management can be one of the more challenging aspects of software development. In sophisticated architectures there are roughly two layers of configuration management: machine configuration; and stack configuration.

Machine configuration encompasses software dependencies, operating system configuration, environment variables, and every other aspect of a machine's environment.

Stack configuration encompasses the configuration of an architecture's infrastructure resources. Infrastructure includes compute, storage, and networking resources.

In a sense, machine configuration is a subset of the stack configuration, however, stack configuration tends to be focussed on a higher level of abstraction. In this course, our focus when looking at Infrastructure as Code (IaC) is *stack configuration*. We rely on containerization tools, such as Docker, to fill the hole left by the lack of treatment of machine configuration.

2 Brief History

In the 'Iron Age' of computing, installing a new server was rate limited by how quickly you could shop for and order a new physical machine. Once the machine arrived at your company, some number of weeks after the purchase, someone was tasked with setting it up, configuring it to talk to other machines, and installing all the software it needed. Compared to the weeks it takes to acquire the machine, a day of configuration is not so bad. Furthermore, because so much physical effort was needed to provision a new machine, a single developer would only be responsible for a few machines.

With virtualization one physical machine can be the home of numerous virtual machines. Each one of these machines requires configuration. Now, with new machines available within minutes and no physical labour involved, spending a day of configuring is out of the question — introducing: *infrastructure code*.

3 Infrastructure Code

Infrastructure code arose to ease the burden of the increased complexity where each developer configured and maintained many more machines. Infrastructure code encompasses any code that helps manage our infrastructure. Infrastructure code can often be quite simple. A shell script which installs dependencies is infrastructure code. There's an assortment of infrastructure code out in the world today, ranging from simple shell scripts up to more sophisticated tools such as Ansible and Terraform.

Definition 20. Infrastructure Code

Code that provisions and manages infrastructure resources.

Definition 21. Infrastructure Resources

Compute resources, networking resources, and storage resources.

To explore the range of infrastructure code available, skim the below examples of infrastructure code in BASH, Python, and Terraform. Each snippet performs the same functionality. Infrastructure Code does not have to be strictly tools such as Terraform, although the two are often conflated. One thing which should be noted is that Infrastructure Code specific languages, unlike most traditional software, is tending towards a declarative paradigm.

```bash
#!/bin/bash

SG=$(aws ec2 create-security-group ...)

aws ec2 authorize-security-group-ingress --group-id "$SG"

INST=$(aws ec2 run-instances --security-group-ids "$SG" \
           --instance-type t2.micro)
```

```python
import boto3

def create_instance():
    ec2_client = boto3.client("ec2", region_name="us-east-1")
    response = ec2.create_security_group(...)
    security_group_id = response['GroupId']

    data = ec2.authorize_security_group_ingress(...)

    instance = ec2_client.run_instances(
        SecurityGroups=[security_group_id],
        InstanceType="t2.micro",
        ...
    )
```

```
resource "aws_instance" "hextris-server" {
    instance_type = "t2.micro"
    security_groups = [aws_security_group.hextris-server.name]
    ...
}

resource "aws_security_group" "hextris-server" {
    ingress {
        from_port = 80
        to_port = 80
        ...
    }
    ...
}
```

4 Infrastructure as Code

In this course, we have chosen to make a distinction between *Infrastructure Code* and *Infrastructure as Code*. We define *Infrastructure Code* as above, as code which manages infrastructure. We separately define *Infrastructure as Code* as the practices of treating Infrastructure Code as if it were traditional code. In the real world no distinction is made. We now introduce *Infrastructure as Code* and clarify why this distinction is important.

Definition 22. Infrastructure as Code

Following the same good coding practices to manage Infrastructure Code as standard code.

Given the above definition of IaC, the natural follow-up question is 'what are good coding practices?'. Before reading further, pause for a moment and consider what coding practices you follow when developing software that you would consider good practice. Got some ideas? Good, let's enumerate a few that we think are important.

4.1 Everything as Code

Everything as Code is a good coding practice which seems so obvious in regular programming that it is rarely discussed. The idea is to ensure that your code does not depend on manual tasks being performed. Unfortunately, as developers are already familiar with their cloud platform's UI, it can be easy to take the easy way and to quickly provision a resource in the UI.

Introducing manual changes outside of your Infrastructure Code creates *configuration drift*, infrastructure configuration which has diverged from your Infrastructure Code. Configuration drift can produce *snowflake* machines, unique machines which your project depends on but no one can safely recreate. Once you introduce just one snowflake machine into your infrastructure, the benefit of reproducibility afforded by IaC is lost.

4.2 Version Control

We hope that everyone can agree that version control in regular code is a luxury we would not wish to live without. It should then be common sense to apply the benefits of version control; restorable code, accountable code, and working collaboratively to Infrastructure Code.

There is however, one significant catch with versioning infrastructure code: *state*. Most Infrastructure Code tools utilize some form of state. In Terraform, applying changing updates a file called *terraform.tfstate*. The state is important to map the resources specified in Infrastructure Code to their real-world counterparts. However, the state is updated frequently and needs to be kept in live-sync between all developers. These requirements make version controlling state a generally bad idea. The solution is counter-intuitively a remote state which exists outside of Terraform controlled infrastructure and out of reach of version control. A remote state might exist on an S3 bucket which all developers can access and update in real time. An non version controlled file such as this might seem like a step backwards, however, if you consider it as a cache sitting between your Infrastructure Code and your infrastructure it is a *slightly* nicer pill to swallow.

4.3 Automation

The goal of Infrastructure Code is to help developers manage the complexity of resource management via automation. In order to maximize our confidence that Infrastructure Code is reflected in reality, it is

important that our build pipelines provide an assurance of consistency.

There are two methods for achieving consistency through automation: fail the pipeline if the Infrastructure Code would change the reality; or make changes to reality in our pipeline. Either of these options would ensure that the checked-in Infrastructure Code is an accurate reflection of reality that developers can safely build atop.

4.4 Code Reuse

A good developer should embody the practice of working smarter not harder — it's the core practice of our industry. When developing Infrastructure Code it is important to remember that you are (likely) not the first to solve this problem. You're not the first person to connect a public EC2 instance to an S3 bucket. A good coding practice that happens to be a win-win for everyone is code reuse. You can't copy and paste another developer's mouse clicks to configure their server via a UI, but you can, and in most cases should, import the Terraform module they published to configure servers in just the way you needed.

4.5 Testing

We'll cover testing in the lecture :)

5 Conclusion

We have looked at Infrastructure Code which enables developers to manage the vast amount of infrastructure resources necessitated by the Cloud Age. Then we explored the benefits of applying the good coding practices we know and love from regular programming to Infrastructure Code. We termed this practice Infrastructure as Code despite the tools (Infrastructure Code) and practices (Infrastructure as Code) being combined in the real world as Infrastructure as Code. It is important to note that IaC is still in relatively early stages, there are a number of issues which need to be addressed, such as: importing existing infrastructure; improving state management; and enabling proper refactoring. Despite these issues, it is the best system we have to date and a significant improvement on manual management.

> Infrastructure as Code is the worst form of infrastructure management except for all those other forms that have been tried from time to time.
>
> — Paraphrased Winston Churchill