**Software Architecture**

CONTINUOUS ASSESSMENT ACTIVITY – CAA 1

Nicolas D’Alessandro Calderon

Presentation

This first CAA serves as an introduction to software architecture for the development of distributed applications where the fundamentals, techniques, and skills applicable to the modern definition of software architectures are covered, and the selection of the most appropriate architectural style according to the type of application to be developed is explored in depth.

To elaborate on the activity, four exercises are formulated: the first exercise will evaluate some of the fundamental aspects in the definition of an architecture, while in the second and third exercises, you will have to choose and justify the architectural style (or mixture of styles) most suitable according to the requirements found in a case study. Finally, the last exercise will evaluate some basic concepts

of distributed architectures.

CAA 1 covers the contents of the book Fundamentals of Software Architecture (see the Resources section for the chapters to be studied).

Competences

The main competence of the Degree in Software Development to be acquired through this CAA is:

* Know how to propose and evaluate various technological alternatives to solve a specific problem.

Objectives

The objectives of the CAA 1 are to:

* Explain the fundamentals of software architecture and the characteristics of architecture.
* Identify and argue the strengths and weaknesses of the different architectural styles and how they influence the decision of the most appropriate architectural style for a targeted application.
* Place the newly emerging paradigms of software engineering in the framework of software architectures.

Exercise 1

Carry out a comparative study of the Microkernel and Orchestration-Driven Service-Oriented architectural styles based on the architectural characteristics of each of them. Use the table below.

Finally, propose an example of an application that fits each of the two architectural styles.

|  |  |  |
| --- | --- | --- |
| ------------------- | **Microkernel** | **Orchestration-Driven**  **Service-Oriented** |
| Partitioning type | Domain and Technical  *\*The learning material point that this Is the only architecture that can be both domain and technical partitioned.*  The core system provides the minimal requirements while additional functionalities (plug-ins) can be added (also deleted) dynamically allowing flexibility and personalization for the client. | Technical:  Services are partitioned and coordinated in a centralized way through an “orchestrator”. This represents easier workflow supervision and control over complex processes. |
| Number of quanta | 1 → Core System.  Singular because all the petitions are managed by the core system. | 1 → Centralized Management by the orchestrator which controls the services workflow.  The trade-off in quanta is given by the simplicity of elements coordination at the cost of limited flexibility. |
| Deployability | ★★★ Moderate  Each module can be deployed individually and independently of the core, allowing for the integration of new functionalities without impacting the rest of the system. However, this remains a monolithic architecture. | ★ Low  Services can be deployed independently but still depends on the orchestrator for its coordination requiring careful planning to ensure that services interactions are not affected. |
| Elasticity | ★ Low  This capacity is limited by the core capabilities. This microkernel type of deployment makes it difficult to adapt to changes. | ★★★ Moderate  Services can be escalated independently, but the orchestrator can also become a bottleneck under high demands. |
| Evolutionary | ★★★ Moderate  The independence of plug-ins makes it easy to adapt to new requirements without changing the core. This allows the system for certain capacity to quickly respond to changes. | ★ Low  This architecture is not very flexible to dynamic changes and updates due to its centralized management by the orchestrator resulting in a complexity for service evolution. |
| Fault tolerance | ★ Low  The core system is the only fault tolerance point, making this architecture vulnerable: If the core fails, the entire system will fail. | ★★★ Moderate  In this architecture, the orchestrator is also the only fault tolerance point, but the services may still be operating improving resilience of the system. |
| Modularity | ★★★ Moderate  This architecture can be considered modular due to its plug-ins, but they still depend on the core system for its functionality limiting its independence. | ★★★ Moderate  As we mentioned before, in this architecture the services are independent in its design, but depend on the orchestrator for its coordination, limiting also the autonomy of the entire system. |
| Overall cost | ★★★★★ Very Low  The simplicity of this architecture mainly for its flexible design of customizable plugin makes this characteristic one of the main strengths of this architecture. | ★ High  Being a distributed architecture and the need of a central orchestrator, makes this system relatively expensive in terms of implementation but also for its maintenance. |
| Performance | ★★★ Moderate  Overall, the performance of this architecture is good because it’s compact, flexible and with the capacity of eliminating unnecessary modules that can improve performance. | ★★ Low  The performance in this architecture is a challenge because it has high latency and for the complexity of coordinating multiple distributed services through the orchestrator. |
| Reliability | ★★★ Moderate  Given that this architecture depends mainly on its core, if a module fails the system can continue working, making it stable (of course, if the core is well designed) | ★★ Low  Reliability is highly linked with the robusticity of the orchestrator, adding a high risk of failure to this architecture. |
| Scalability | ★ Low  The scalability in a microkernel architecture is limited for its monolithic nature, making it vulnerable to bottlenecks by its dependency on the core system. | ★★★★ High  Given that the services can escalate individually, the overall capacity of this architecture, and the capacity to adapt to high volume and demands is very good. |
| Simplicity | ★★★★ High  This architecture is simple because it is easy to maintain and deploy among others. As we said, the core system remains static while plug-ins are being added and deleted depending on the needs. | ★ Low / Very Low  Its central coordination by an orchestrator is always adding complexity to this architecture. The design, maintenance and updates requiring always careful implementations makes this a complicated |
| Testability | ★★★ Moderate  Again, independent modules make individual testing easier, but the core system remains critical, adding some vulnerability in this characteristic. | ★ Low / Very Low  Being a distributed architecture, the interaction between services makes the testing to be extremely difficult and challenging. |

Conclusion:

Microkernel monolithic architecture stands out for its simplicity and low cost, but its vulnerable in terms of elasticity, scalability and fault tolerance due to its high dependence on the core system. On the other hand, Orchestration-Driven architecture, being distributed, is very good for scalability purposes but also expensive and complex, making it very difficult to deploy and test.

Example Application:

* Microkernel: A type of application that fits really good for this architecture can be an IDE (like Visual Studio Code for example). The core system will handle the basic functionalities, and the client/users can add and remove specific functionalities based on individual purposes by downloading the available plug-ins.
* For Orchestration-driven architecture, it is hard for me to find an example that fits, because all the material read highlights that this was an architecture very used in the 90s, that it serves as a good foundation for understanding the trade-offs of distributed architectures, but it is currently not much implemented. Anyway, for the sake of this exercise, I will propose an e-commerce platform that uses orchestration to manage the workflows between the different services such as catalog management, payment processing, order fulfilment and shipping. The main highlight here is that the orchestrator will ensure all these services work together to process customer orders efficiently.

Exercise 2

A family restaurant business that has three pizzerias on the outskirts of the city of Madrid has decided to start offering home delivery.

To implement the application, a small company has been hired that has its development team consisting of 1 UI expert and 2 Node.js developers (one of them with knowledge of MongoDB). The most experienced member of the development team will also act as the architect.

The entire application will be developed with a Node.js backend, a React.js frontend, and the data will be persisted in a MongoDB database. Both the Node.js server and the database will be hosted on an on-premise infrastructure.

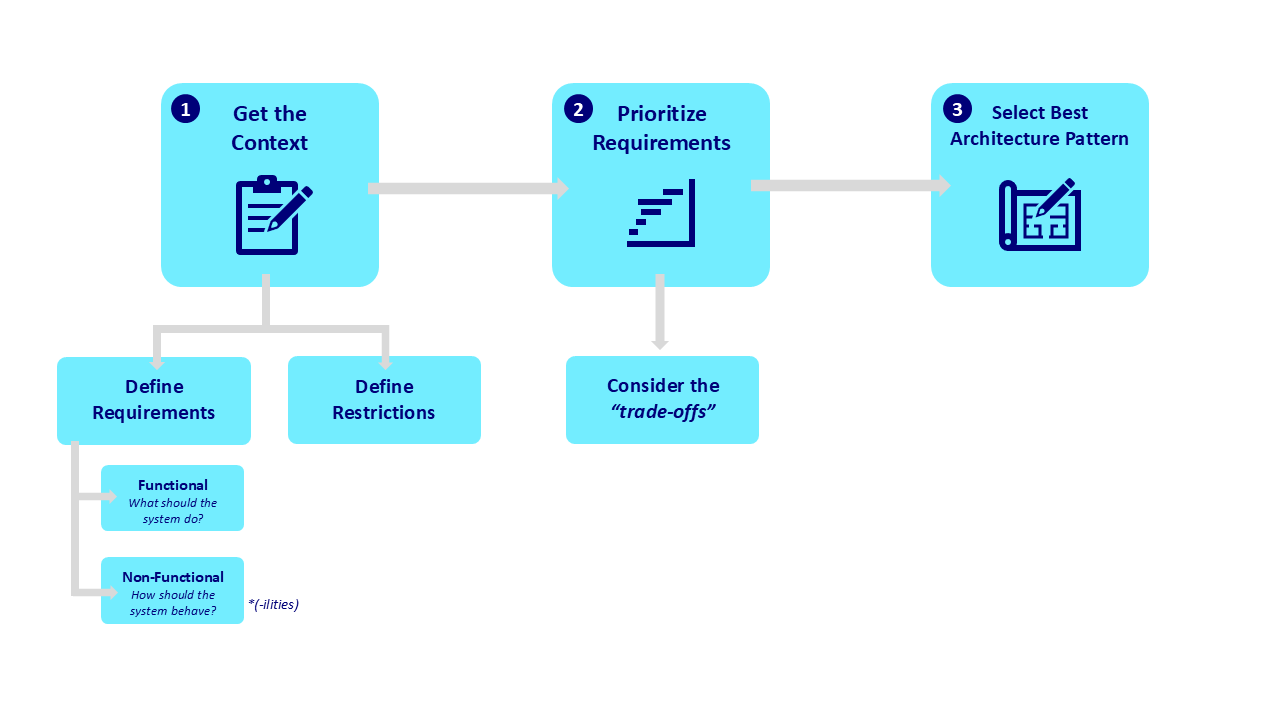
Finally, it must be taken into account that the economic situation of the catering group is quite critical, so we are much more interested in having something up and running quickly and at a low cost, rather than thinking about possible performance, maintainability, or modularity issues. We are also not concerned about the scalability of the application (with 3 pizzerias we don't seem to have problems in that aspect).

Which of the architectural styles that you have studied in Chapter II: Architecture Styles of the Fundamentals of Software Architecture book do you think fits best with the nature of the project?

Reason the answer by comparing the requirements and constraints of the project to be developed with the pros and cons of the chosen architectural style.

**Before Starting:**

After reading all the material and based on the proposed path of the exercises from the videos, I have created this mental map with the aim to serve as a guidance for performing the next two exercises in a correct and structured manner:



1. **Get the Context:**
   * Functional Requirements or What the system needs to do?

*These are the core capabilities and processes the system must perform.*

* + Non-Functional Requirements or How the system behave?

*These involve including considerations for scalability, reliability, performance and any other “-ility”.*

* + Restrictions

*These involve recognizing any constraints that must be considered, such as GDPR regulatory compliance or tech limitations.*

1. **Prioritize Requirements “Trade-offs”:**
   * If we understand the importance of the different requirements and the restrictions, we will weight and decide which features and functionalities are critical and which can be put away if necessary. This Prioritization will help us managing resources effectively and ensures that the most valuable aspects of the system are the ones that will be developed first.
   * Simplification and Focus:
     1. YAGNI (You Are not Gonna Need It): This principle say that we should not be adding any functionality unless it is strictly necessary.
     2. One Thing at a Time: By focusing on one component or process at a time, we will ensure a correct order in the execution, and we will also reduce complexity in the development process.
2. **Select the best Software Architecture Pattern:**
   * Once we follow all the above steps, we can decide between different architectural patterns whether a Monolithic or Distributed based on the needs and scope of the project.

**1. Understanding the Context and Requirements**

**Business**: A family Pizzeria chain in the outskirts of Madrid

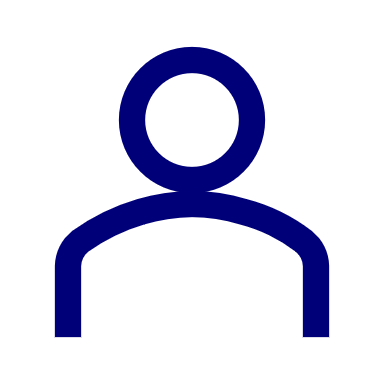
**Objective**: To develop an application for Pizza Home Delivery

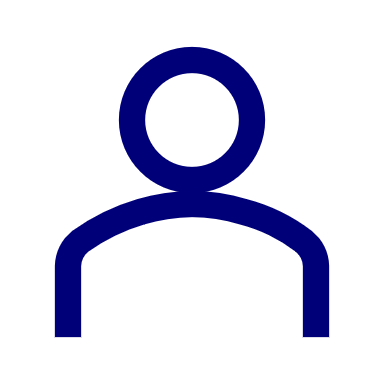
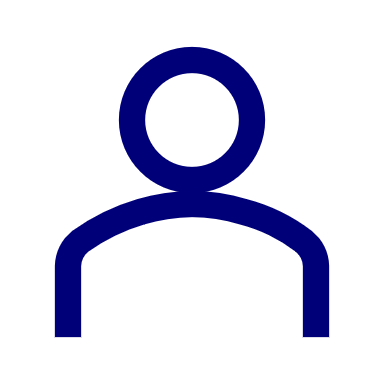
**Tech Stack**:

Database → MongoDB

Backend → Node js.

Frontend → React js.

**Taskforce**:

1 UI expert

2 Node js. developers (1 with experience in MongoDB)

* 1. **Functional Requirements**
* The system must handle online home delivery for a restaurant business with three pizzerias.
* Users will place orders online, and these orders will be processed by the system.
* The system should allow the clients to interact with the menu for selecting pizzas and other products from the catalogue.
* The system must allow customers to track delivery status.
  1. **Non-Functional Requirements**
* Simplicity: The business has a critical economic situation, so building a simple, fast and low-cost solution is the top priority.
* Deployability: The client expects the solution to be operative in short time.
* Maintainability: Given the constraints, the client is not worried about long-term maintainability. However, the architecture selected should be flexible enough to allow for future improvements.
* Other Considerations: Since the system will process orders for only three locations, and due to the limited budget, scalability and high performance are not immediate priorities.
  1. **Restrictions**
* Limited resources and technology defined: As described above, the team consists of 1 UI expert and 2 Node.js developers (one with knowledge in MongoDB), so the chosen architecture must align with this.
* On-premises infrastructure: The application must run on local servers, so cloud-based solutions are not an option.
* Budget: Critical economic situation, so it is preferred a simple and low-cost solution.

1. **Prioritization and Trade-offs**
   1. **Key Priorities**

* Simplicity - High priority: The system must be simple to develop and maintain adjusted to the tech stack already described.
* Low cost - High priority: The budget is limited, so we must minimize costs.
* Deployability - High priority: The client needs the application to be operational as soon as possible.
  1. **Trade-offs**
* Performance vs. Simplicity: Since the business doesn’t expect to have high traffic, we can prioritize simplicity over scalability and high performance.
* Maintainability vs. Deployability: Even that maintainability is important, given the described financial situation, we must focus on deploying a functional system in short time, even if it may lead to technical debt in the future

1. **Simplification and Focus**
   1. **YAGNI**

The system only needs to manage orders for three pizzerias, so we don't need to invest in a distributed system or scalable system.

* 1. **One Thing at a Time**

We must focus on developing the core functionalities \*user, registration, place orders, order tracking and delivery processing and then if the business grows, we will take care of developing new requirements.

1. **Choosing the Right Software Architecture Pattern**

Based on this analysis, and after evaluating all the architectures from the book, my selected option is the Layered Architecture where we will separate the system into layers:

* Presentation Layer → React.js frontend
* Business Logic Layer → Node.js backend
* Data Layer → MongoDB database

This structure will keep the system simple, cost-effective and can help with maintainability and future optimizations:

A diagram of a software process

Description automatically generated with medium confidence

**Why we select a Monolithic Architecture?**

Fast, Simple, and Cheap: Aligned with what we defined as the primary goal, which is to deliver a working system quickly at a low cost. A monolithic architecture can achieve this by allowing the team to focus on core functionalities without the need of managing multiple services.

Easy for the Team: Our constraint that the development team is small and already familiar with Node.js and MongoDB, making it easy to implement a monolithic solution.

**How this aligns with the defined context?**

Functional Requirements: All components defined (menu, ordering, delivery tracking, etc.) can be handled with only a single system.

Non-Functional Requirements: Since scalability and performance are not immediate requirements, we don’t need a distributed system. The monolithic design will offer simplicity, low cost and fast deployment.

Restrictions: The technology stack available fits perfectly with a monolithic architecture, and it can be easily deployed on the on-premise infrastructure without issues.

**Why Layered Architecture is the best choice?**

Fits with the defined needs: The layered architecture offers simplicity and is perfect for the small scale and low complexity nature of the restaurant home delivery order system.

Fits with the available technology: The development team is familiar with React js., Node.js and MongoDB, and the layered architecture fits good with this technology stack.

Easy Scalable: While scalability is not a priority now, this architecture can anyway be expanded easily if the business grows, without adding much complexity.

Separation of concerns: This architecture will give a clear separation between the frontend - React.js, business logic - Node.js, and data MongoDB, making it easier to maintain and improve in the future if needed.

**Why not other monolithic architecture?**

Pipeline Architecture will add unnecessary complexity for this simple system.

Microkernel Architecture is better for systems that require extensive modularity and extensions, and this is not our use case.

Exercise 3

The investment group that launched the photo store ("Photo&Film4You") that you will find attached with this description, managed to launch the application in record time and this has allowed them to successfully exceed business expectations. Not only that, but the initiative has been very well received; its users already number in the tens of thousands and have a catalog of more than 1,500 references. The success has been so great that they are even considering leaping into the American

and Asian markets.

But not everything is so great; due to this rapid growth, they have already started to find some problems and new challenges with the current equipment rental application. We list them briefly:

* There are teams where the number of users wanting to make a reservation is very high. In this scenario the application is already very slow and the "on-premise" servers are very saturated. High concurrency is starting to be a serious problem for the platform.
* The application has periods of very intensive use in which the current infrastructure is not enough, but we cannot afford to dimension the infrastructure for those specific periods of high demand because we would be wasting many resources the rest of the time.
* It seems more and more clear that the technological requirements are not homogeneous for the entire application.
* We have encountered regression errors that have left the platform out of service for hours, with the consequent loss of money this implies. This is because we do not have a suite of automatic tests that assure us that the releases will work correctly and will not break anything.
* There are increasingly complex and specialized processes that require more specific domains both functionally and technically. It is no longer possible for a small development team to master all of these concepts. There are complex processes that are best solved using functional programming techniques with functional programming languages and NoSQL data sources while billing processes require to be implemented with more traditional technologies.
* There are numerous scheduled processes that perform actions on the database, such as updating the stock of products in the available catalog.
* It is necessary to deploy a mobile application that will increase traffic to the servers.
* To be competitive we need to develop new features and put them into production very quickly, if we don't do it, the competition will Ideally, a feature should be promoted to production safely without manual intervention, and this cycle could be repeated several times a day. This implies that deployments of new features should be independent and follow their own life cycle.

Which architectural style of the ones you studied in *Chapter II. Architecture Styles* from the *Fundamentals of Software Architecture* book do you think best fits the new requirements of the project?

Reason the answer by comparing the requirements and constraints of the project to be developed with the pros and cons of the chosen architectural style.

**1. Understanding the Context and Requirements**

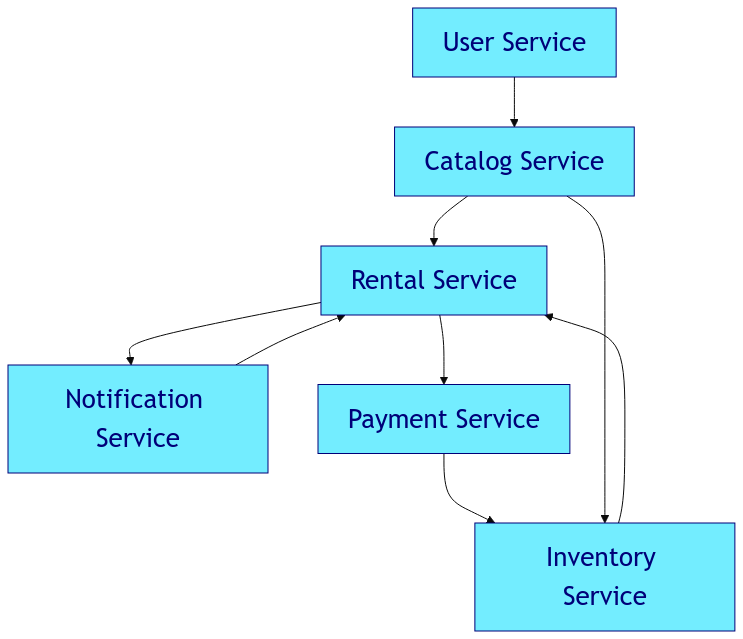
**Business:** **Photo&Film4You** a store for audiovisual equipment specialized in film shooting and professional photography sessions.

**Objective:** An online audiovisual rental service. Expand their current equipment rental service to address high demand and new challenges due to business growth, including supporting a high number of users, managing varied processes and technology requirements, and ensuring fast, frequent deployments.

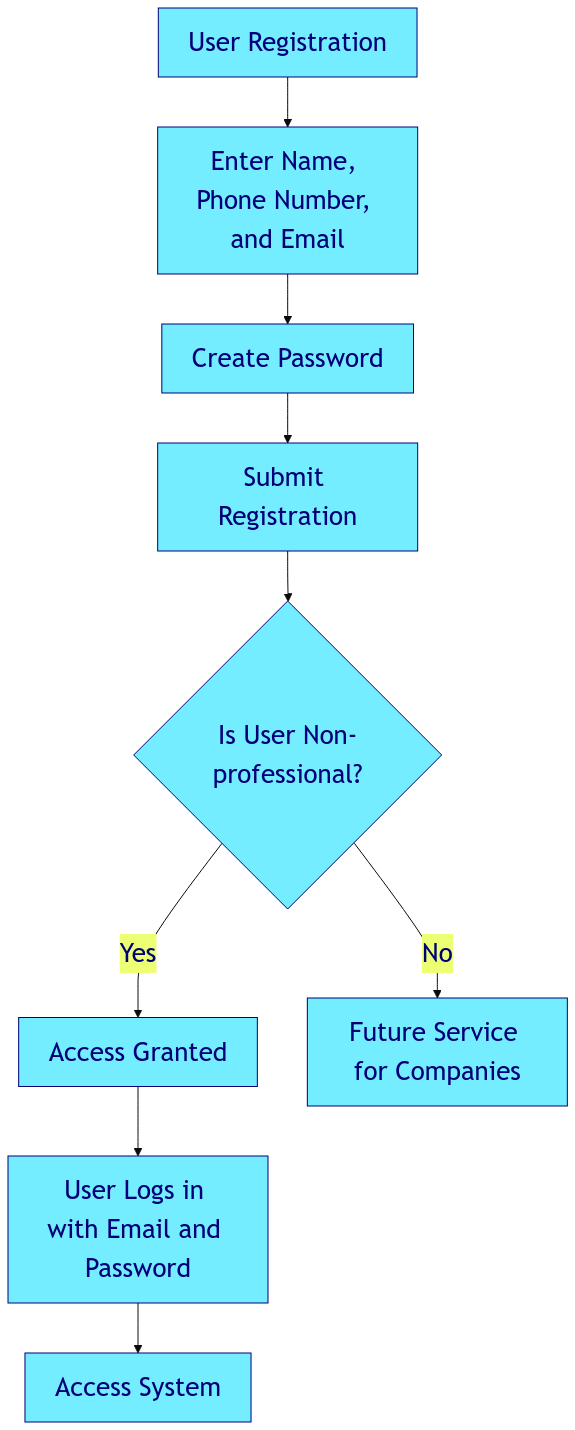
**Tech Stack:** Current stack includes an on-premises server environment. There are indications that cloud options may be considered to handle demand current high demand.

**Taskforce:** Administrators managing the catalog and developers responsible for maintenance and upgrades, with the potential need for cross-functional teams due to domain-specific functionalities.

**Services Needed in Architecture:**



**User Service:**



**Catalog and Inventory Services:**

├── Sections/

├── Cameras

├── Subsections/

├── Video

├── Products/

├── Name: Sony PXW-Z150 4K XDCAM

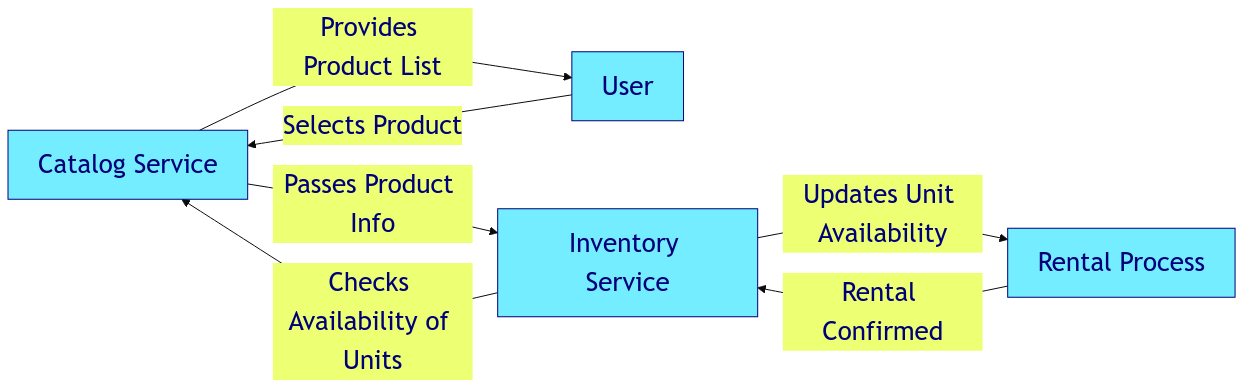
├── Rental Price: 50 €/day

├── Units/

├── 1: Serial #12345 - Status: Operational

├── 2: Serial #12346 - Status: Non-operational

└── 3: Serial #12347 - Status: Operational

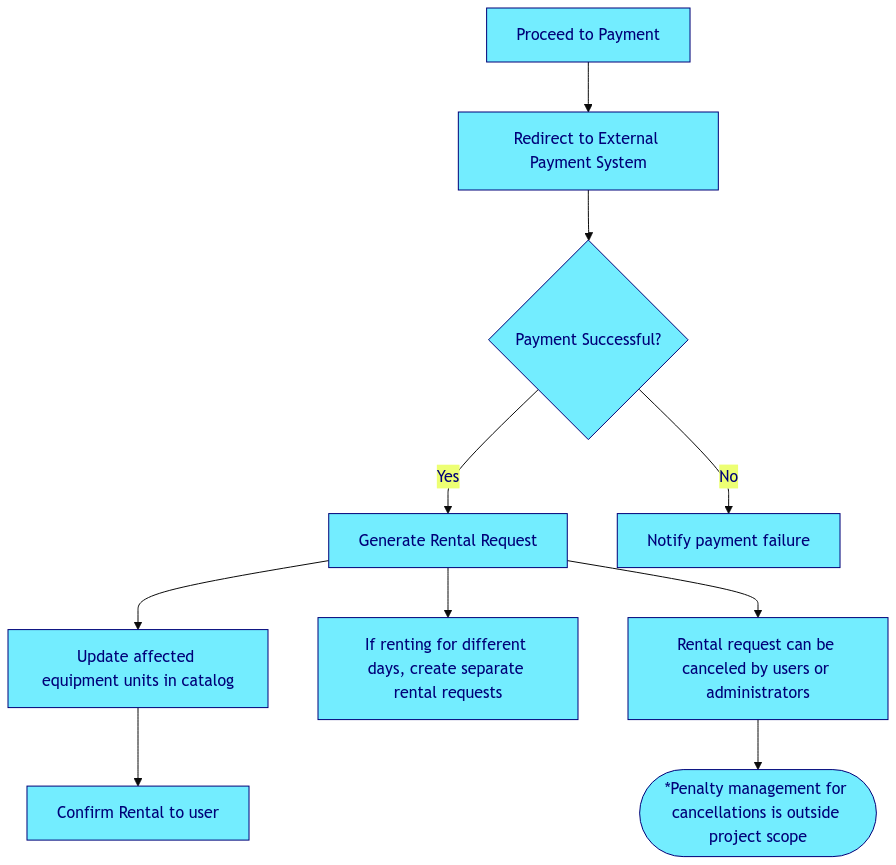


**Rental Service:**

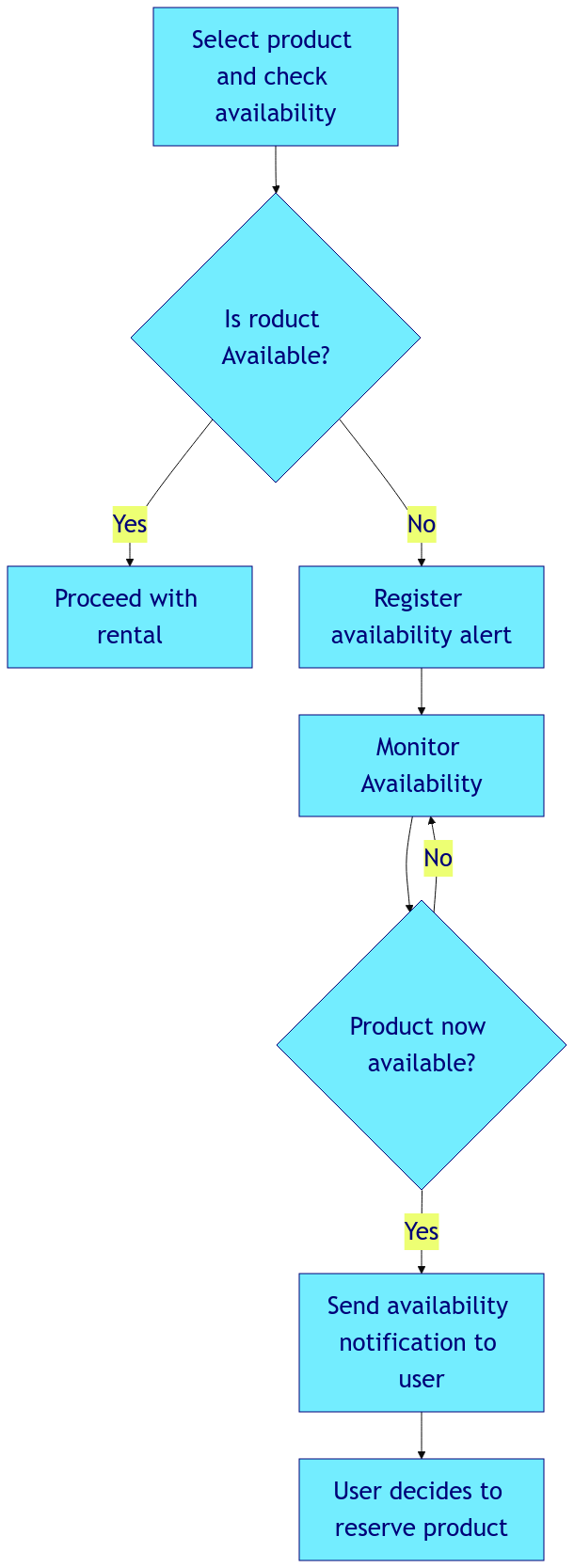
A diagram of a flowchart

Description automatically generated

**Payment Service and Rental Confirmation:**



**Notification and Availability Alert Service:**



* 1. **Functional Requirements**
* The system should support a high volume of concurrent users and rentals.
* Administrators need to manage a large catalog of around 1,500 items, including their status updates.
* Registered users should be able to navigate the catalog, reserve items, and make payments.
* Users must be notified when unavailable items become available.
* The service must be available via both web and mobile platforms and different browsers.
* Features must be released independently and frequently to keep the system updated.
  1. **Non-Functional Requirements**
* **Scalability**
* **Fault Tolerance**
* **Modularity**
* **Deployability**
  1. **Restrictions**
* The current infrastructure is on-premise, but it is mentioned that they are experiencing saturation during periods of high demand.
* Mobile accessibility is required because the next phase includes deploying a mobile app.
* Payments are handled via external system because no data from payments is expected to be stored in our database, probably due to GDPR compliance.

1. **Prioritization and Trade-offs**
   1. **Key Priorities**

* Scalability - High Priority: Probably the most important requirement to be able to handle the high traffic and periods of peaks in demand.
* Fault Tolerance - High Priority: It is requested that the system should provide reliability and minimizing any possible disruption.
* Modularity - High Priority: Provide different technical solutions for the specific functionalities already described.
  1. **Trade-offs**
* Performance vs. Modularity: Modularity allows diverse technical solutions, but it may reduce the performance if the components are not optimized for the integration between them.
* Rapid deployment vs. Fault Tolerance: A fast deployment is mentioned as very important, but this may increase the risk of errors if there is not a proper testing suite.

1. **Simplification and Focus**
   1. **YAGNI**

The system should focus on immediate functional needs like scaling for user capacity, modular design for different tasks and fault tolerance to prevent disruptions.

* 1. **One Thing at a Time**

Prioritize the required process like user registration, rental processing, catalog management, and payment.

1. **Choosing the Right Software Architecture Pattern**

Based on the analysis above, we can conclude that a **Microservices Architecture** is the best fit for our use case.

**Why Microservices Architecture?**

* **Scalability and Modularity:** A Microservices architecture will provide high modularity, and it will allow the different parts of the system to scale independently. For example, we may scale the catalog service without the need to scale the rest of the services.
* **Fault Tolerance:** If one service fails, like for example the payment service, the other systems can still function.
* **Deployability:** All the services required can be deployed independently without affecting the entire system aligned with the requirements of adding new features quickly and constantly.

**How this aligns with the defined context?**

Microservices can handle very good the complex environment presented in this use case, with specific services for catalog management, rental processing, user management, and notifications.

**Why not other architectures?**

* **Monolithic Architecture:** Not the best option because of the scalability required and the difficulties in maintaining fault tolerance at high traffic volumes as is expected.
* **Service-Oriented Architecture:** this is like microservices, but as we mentioned in exercise 1, the complexity of this architecture and the communication between servicers, may not be the best options due to the rapid Deployability requirements.
* **Event-Driven Architecture:** This may be good fit for the asynchronous process of this use case, but we mentioned that a modular structure is preferred to have better control over each service or process.

**Final considerations: Migrate to Cloud for Scalability**

The company mention this possibility, and we consider a really good option since the **on-premises servers** are now saturated during periods of high demand. We know that maintaining fix infrastructure is very expensive because we may not use all the resources all the time, so moving to **cloud-based infrastructure** will provides a more flexible solution.

Exercise 4

Briefly answer the following questions about the concepts seen in the book “Fundamentals of software architecture”.

4.1. Briefly describe each of the 4 main prisms or dimensions that make up the definition of an architecture and give an example of each of them. The examples chosen must be different from those appearing in the book "Fundamentals of Software Architecture".

1. **Architecture Characteristics**: They are the non-functional properties that the system must handle. In the book they are also called many times as “-ilities” of the system. These are for example, scalability, availability, Deployability, etc.

An example of how this defines a dimension of architecture can be an e-commerce application, which must provide high availability to handle periods of high traffic like Black Friday.

1. **Structure**: This is the type of architecture, or the structure that defines the architecture style of our system and how the elements and components are organized and communicate with each other. For example, a social media app might have event-driven structure, so user interactions trigger asynchronous events that communicate with other services like notifications, recommendations, etc.
2. **Architecture Decisions**: They are the rules that guide the decision about how to build the system, defining what is allowed and what is not. For example, a system that requires high security, like a bank platform, we may decide that only security microservices have direct access to the database of users, and this way, improve the security.
3. **Design Principles**: They are general guidelines to help build the system, but they don’t imply strict rules. For example, for an e-learning platform, one design principle must be whenever it is possible ensure accessibility for users with disabilities, so as the example from the book on leveraging async communication, in our use case, the principle will guide the decisions of the development team such as ensuring that color contrasts in the interface design are strong enough to facilitate reading for users with low vision or color blindness, etc.

4.2. Which of these associated tasks are the responsibility of an architect and which

are the responsibility of the development team:

* Create a state diagram for the stages an order goes through.
* Create a prototype with the user interface to verify the users’ experience with the application before developing it.
* Ensuring software quality.
* Define how to manage the increase in usage at the infrastructure level during marketing campaigns.
* Perform functional testing.
* Test automation.
* Mentorship and collaboration.

|  |  |  |
| --- | --- | --- |
| Task | Responsible | Justification |
| Create a state diagram for the stages an order goes through | Architect | Defining states and transitions is one main architecture tasks. |
| Create a prototype with the user interface to verify the users’ experience with the application before developing it | Development Team | This task is typically done by the development team, or a UX/UI expert. |
| Ensuring software quality | Architect / Both | The architect defines the quality attributes, for example scalability, performance, etc. and then ensures the system meets these criteria. |
| Define how to manage the increase in usage at the infrastructure level during marketing campaigns | Architect | The architect was responsible for designing a scalable system that can handle these events. |
| Perform functional testing | Development Team | The development team handles this since they are familiar with the implementation. |
| Test automation | Development Team | The architect may specify the need for an automation test, but the development team will be responsible for implementing it. |
| Mentorship and collaboration | Architect | Another key role of an architect is to provide technical leadership, mentoring and improve the capabilities of the team. |

4.3. Give a concrete example of an application that follows an architectural style based on microservices where it is shown that it is not possible to maximize ALL the desirable characteristics of this architecture.

A concrete example of an application that follows microservices architecture and is clearly shown that it is not possible to maximize all the desirable characteristics is the **Amazon e-commerce platform**. I’ve worked in Amazon for four years and I always remember hearing the story on how Amazon moves from a Monolithic infrastructure to the current distributed microservices. As far as I understand, they focused mainly on Scalability, Resilience and Agility, but doing so they faced a lot of challenges that made it impossible to maximize all the desirable characteristics of microservices. For example, during the “peak period”, Black Friday or prime days, Amazon prioritize data partition tolerance over data consistency, meaning that different services like recommendations or prices that works on distributed databases that are capable to handle the requirements of performance and scalability, ends-up with occasional temporal inconsistencies in inventory level being showing to users.

This trade-off is not the only one in the Amazon´s architecture, because despite microservices allows the company to easily scale and deploy individual services according to demand, it sacrifices the cost-effectiveness of it. For example, every service has its own database, leading to some data redundancy, prioritizing resilience, availability and low latency but with high cost and overload in comparison with a Monolithic architecture.

4.4. Taking into account the description of the project and the architecture chosen in

exercise 3, choose three of the characteristics of the architecture of each class (operational, structural, transversal) and argue why and what precedence you would place among the nine candidates, justifying the position granted.

**1. Scalability → Operational:**

This is the most important because Photo&Film4You is having trouble with high traffic during busy periods. The microservices architecture selected is because it helps to scale each part of the system independently, so if the catalog service receives a lot of traffic, we can add more resources to that service without affecting others.

**2. Modularity → Structural:**

This is important because Photo&Film4You has different parts (product catalog, payments, user accounts, etc.). By keeping each part separate in this modular structure, we work with one service without affecting others, making the system more flexible and this way it is easier to work with it.

**3. Maintainability → Transversal:**

This is also important because as we already mentioned, with microservices, we can make changes to one service without affecting others and this helps the developers maintain or release updates with less risk.

**Ranking:**

1. Scalability → Top priority to handle high traffic.

2. Modularity → To make the system flexible,

3. Maintainability → Important but less urgent than the other two.