Software Architectures for Distributed Systems

Prof. Dr. Peter Braun

4. Oktober 2024

1 Titlepage

Software Architectures for Backend Systems

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

2 Learning Goals

Learning Goals

- Understand the concepts of software architecture
- Monolithic vs. distributed architectures
- Client/Server, Services, and Microservices

Welcome to this unit about Software Architectures for Backend Systems. After you have learned in the last unit what a distributed system is and that we will focus on Client/Server systems in this class, we will further clarify this semester's contents. The name of this module is "Backend Systems," and you might already guess that we will, in particular, look at the server side of the Client/Server System. This must be seen in contrast to the client, which we can call the "Frontend." The "frontend" will be covered in a module next semester. The backend has many responsibilities and can be designed in many ways. We first introduce the notion of "Software architectures," then we discuss many options for software architectures of backend systems. At the end of this unit, you will understand these differences, and you will learn that we will focus on one specific type of architecture, which is called "Hexagonal architecture."

3 What is Software Architecture?

What is Software Architecture?

Architecture is defined

- as the **fundamental organization** of a system,
- embodied in its **components**,
- their **relationship** to each other
- and to the environment,
- and the **principles** governing its design and evolution.

Source: ISO/IEC 42010

When we talk about "Software Architecture," we refer to a system's fundamental organization. To explain it in more tangible terms, think of it like the blueprint of a building. It provides a solid, structured layout that sets the groundwork for the entire system. This groundwork is embodied in distinct components, the parts that make up the overall software. Each component is an individual entity that performs specific functions, making up a cohesive whole. If you are not familiar with the word "component," we will clarify it in a few minutes. It's like the rooms in a building; each has a dedicated function, but overall, they come together to serve the purpose of the whole building. In Software Architecture, the relationship between these components is of utmost importance. How these components interact, collaborate, and depend on each other determines the functionality and efficiency of the overall system. Apart from this internal interaction, we also consider the system's external relationship with the environment. This includes how the system responds to external inputs and communicates with its users. Lastly, architecture also consists of the principles that guide the system's design. They also dictate how the system will progress, allowing it to grow and adapt to emerging changes. I guess you don't have much experience in software architecture yet, but you should know about software design already. Let's put these two terms in contrast to each other.

4 Software Design

Software Design

- **You know the elements of software design**

- Statements, control structures like loops
- Methods or functions
- Classes and packages
- Design pattern (like visitor, observer, etc.)

- **How do you structure your system beyond that?**

After two semesters of learning a programming language, you must be familiar with these concepts, which you can see next to me. You know about single statements like assignments, loops, and conditional statements. You structure your code using functions or methods to reuse code and make the source code more readable. You model the problem domains using classes and summarize classes into packages. I hope you have already heard about design patterns, which are "best practices" in solving specific problems in "designing" software. So, you should be familiar with all these words and notions and be able to "design" software. Is this enough to create a big software system like the backend of the video streaming platform Netflix? Definitely not; you need more ways to structure your code. So, let's talk about software components next.

5 Software Component

Software Component

A software component ...

- ... is a **unit of composition** with
- contractually specified **interfaces**
- and **explicit context dependencies** only.

- ... can be **deployed independently**
- and is **subject to composition** by third parties.

Source: Clemens Szyperski. Component Software - Beyond Object-Oriented Programming, Second Edition, 2002.

When discussing a software component, we refer to a composition unit. This composition unit has an interface to communicate with other software components. You might have heard about the two metrics already: cohesion and coupling. Cohesion means how the elements of a component belong together on a logical level. Cohesion should be high. Coupling means how much a component depends on other components. This value should be low. Another key characteristic of a software component is that it can be deployed independently. The word "deployed" is confusing here in this definition. This means that the software can be used and incorporated into software systems. This means it can operate and execute its designed functions without depending on other components much. Lastly, a software component is subject to composition by third parties. When we say this, we mean that third parties can utilize this component and arrange it with other elements, making it an important part of greater and potentially more complex systems. Let's look at an example of a software component.

6 Example: User Management

Example: User Management

Data structure (e.g., a Java class): `User`

- Login name
- Password
- Full name and address

Functions (e.g., a Java class): `UserManagement`

- Register new user
- Update user profile
- Login user and check authorization
- Reset password
- Delete user

We can summarize all functions and data structures necessary for user management into such a component. We have to store some information about users, for example, login name and password, roles in the system, full name and address, and maybe some metadata like the time of the last login. You should write a class for this in Java programming language. The list of functions provided by this component is pretty straightforward. First, every system needs a login section with a username and password. This credential system ensures that only authorized individuals can access the system. The system must allow users to update their profile information. And, of course, systems must include a way to reset a forgotten password or to delete a user account when no longer needed. When you implement this software component in Java, you devise an interface defining all the functions and a Java class, and another class implements this interface. You group these Java classes into a package, and you might have many more internal classes to realize all the requirements. So, now that you know about software components, the next question is how you structure your big application into smaller components. Are there also some patterns available that can be applied, comparable to design patterns used inside of components?

7 Big Ball of Mud – No Architecture

Big Ball of Mud – No Architecture

- You structure your code in classes
- You might also have components
- Many dependencies between the components

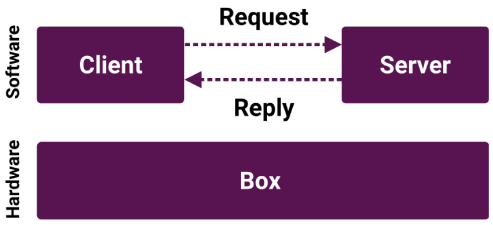
- **No structure – no architecture**

Let's start with a counter-example. Yes, you structure your code in classes, and you might also have components. But then, these software components communicate without a clear strategy. Imagine a picture of all the software components in your system, and each component needs to communicate with/all/ other components. The values for cohesion and coupling would be horrible. There is no structure in using the components and no architecture. We call this a "Big Ball of Mud."

8 Single Host Client/Server Architecture

Single Host Client/Server Architecture

- Two-tier architecture with clear roles
- Communication starts with the client



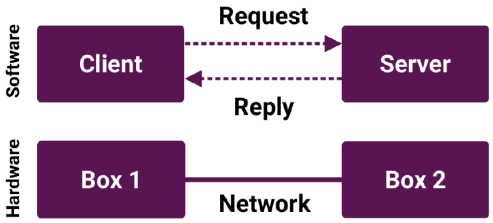
- Example: Application + Database Management

Moving on to the first type of software architecture, we are dealing with what we commonly refer to as two-tier architecture. It's an architectural configuration where we have two clearly defined roles: the Client and the Server. You know this from the last unit already. However, in the previous unit, it was meant as a pattern to structure a distributed system. Here, it is intended as a software architecture that consists of components. Hence, there is only a single computer, a single box on which both client and server are running. Looking at the classic example of an application integrated with a Database Management System, or DBMS - the application, in this case, acts as the client. It sends requests to the DBMS, for instance, to retrieve or update data. The DBMS, in its server role, processes these requests and responds accordingly by providing the requested data or confirming the updating of data.

9 (Distributed) Client/Server Architecture

(Distributed) Client/Server Architecture

- Physical distribution of client and server



- Web browser + Web server
- Web server + Database Management

Next, we extend this software architecture and place the components on physical or virtual computers. This is comparable to what we learned in the last unit. A common example is the relationship between a web browser and a web server, as shown in our example in the last unit. Another example is a Web server that uses a database on a different host.

10 Backend System = Server

Backend System = Server

- **API** – the interface to the client (frontend)
- **Business logic** – how data is processed
- **Caching** – to increase the performance
- **Database management system** – how data is stored
- **Domain** – the core of the application
- **Load balancer** – to improve scalability
- **Logging** and monitoring
- **Security** – authorization and authentication
- **Task queues** – for background or batch processing

We want to discuss the software architecture of a backend system. Let's start by collecting the primary components of such a system. Firstly, we have the API, the Application Programming Interface. This is the communication bridge between the "backend" and the "frontend." In other words, clients can interact with the server through the API. This semester, we will discuss several technologies for implementing an API, namely GraphQL, Google RPC, Web API, REST APIs, and Web Sockets. Then, we delve into the Business logic. It's how data is processed and managed to fulfill specific requirements or needs. This logic can include algorithms, rules, operations, processes, and more. Next, we discuss Caching, a critical tool for boosting system performance. By storing frequently accessed data closer to the client, caching reduces the need to fetch it repeatedly, hence speeding up processes. Following this, we have the Database Management System. Simply put, this is where all your data is stored. Depending on your requirements and the nature of the data, it can be relational, NoSQL, or other. This semester, we will mainly discuss relational database systems and touch on high-performance systems. Next is 'Domain'—the heart of any application. It defines the business's core concepts, rules, and logic and is grouped based on the fields and categories. You might wonder what the difference between business logic and domain is. A classic example is that a bank account with its rules on how money can be withdrawn from an account is part of the domain. Implementing a money transfer from one account to another is part of business logic. Meanwhile, a Load balancer comes into action when the system has a lot of work. It helps distribute network traffic evenly across numerous servers, achieving optimal resource utilization and avoiding server overloads. Now, let's discuss Logging and Monitoring. Keeping track of user interactions, errors, and system performance is vital. This will help ensure your application runs smoothly and quickly troubleshoots any problems. Following

this, we have Security. User authorization, authentication, encryption, and protecting sensitive data are significant challenges that every Backend System needs to overcome. Lastly, we consider Task Queues. Excellent for background or batch processing, these queues help manage tasks that can be completed independently and need not be executed immediately, improving application performance. Of course, this list is not exhaustive. Together, these components contribute to a well-structured and efficient Backend System. The next question is how we structure these components. Are there any best practices?

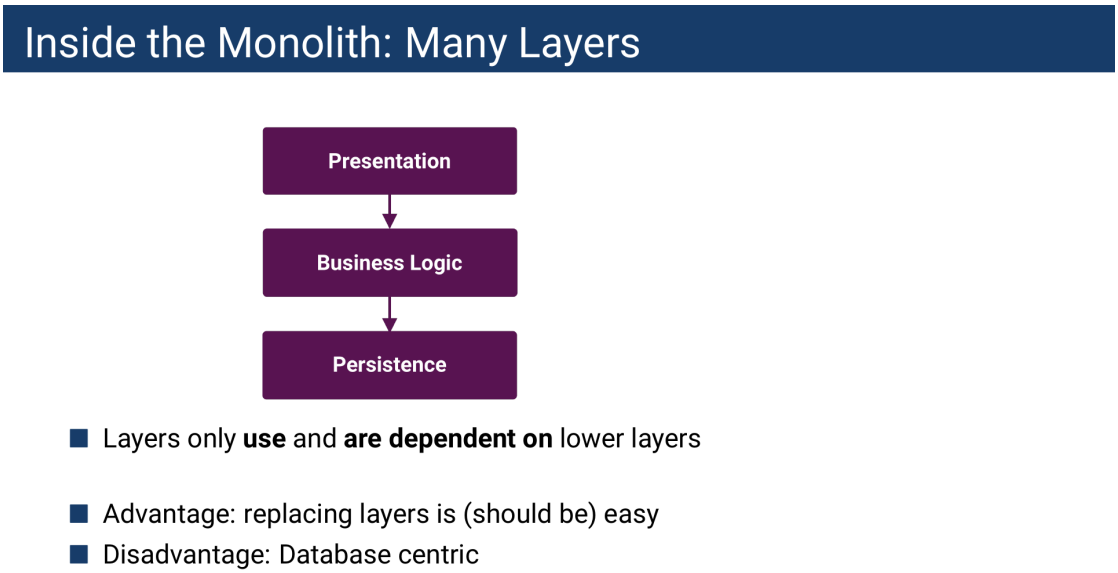
11 Monolithic Architecture of Backend Systems

Monolithic Architecture of Backend Systems

- **Single-tiered software application architecture**
- The key point is: One deployment unit
- **It does NOT mean**: no layers, no components
- Advantages: simplicity, ease of deployment
- Disadvantages: scalability is effort, less flexibility
- **Keep in mind**: for many applications, it is still fine

Let's start with the monolithic architecture. Monolithic means that the whole application, which consists of all the components mentioned earlier, is bundled into a single app. Instead of an "app," let's better use the word "deployment unit." A deployment unit is installed on the backend server as part of a build process. Having only one deployment unit means that all the functionalities of an application are tightly coupled and run in the same space. But remember, when we say monolithic, it does not mean there is no further structure inside the monolith. There will be layers and components. We talk about this later. Monolithic architectures are under pressure right now—sometimes unjustified, in my opinion. Yes, they have some scalability issues and are less flexible. On the other hand, they are simple and easy to deploy. There is no one-size-fits-all architecture in system design. Believe it or not, Monolithic architecture is fine and, indeed, an apt choice for many applications. It provides a simple, easy-to-understand structure, making life much easier when understanding, building, and troubleshooting your application.

12 Inside the Monolith: Many Layers

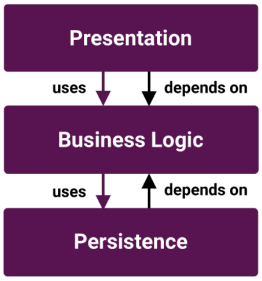


We will now examine the internal structure of a monolithic architecture. For many decades, layered architecture was the standard approach for structuring such a system. I've included a picture of the classic 3-layer architecture next to me. The top layer is called the "presentation layer." This layer contains the components responsible for controlling user interaction. This can be the graphical user interface but is now mostly the API implementation. The second layer contains all the components to control the application's business logic. This includes the domain object, the business rules, processes, and all the technical things like security authorization and authentication. Finally, the lowest layer connects to the database management system. This mustn't be the database management system, but it contains the components to connect the business logic to the external database. A fundamental rule in a monolithic architecture is that every layer depends only on the lower layers. It borrows essential services, data structures, and functionalities from them. Likewise, each layer only uses the facilities provided by the layers beneath it. An upper layer, for example, would not directly communicate or interact with a much lower layer, bypassing the layers in between. There is also no communication from a lower to an upper layer. You are also familiar with the concepts of layers from the ISO/OSI model in data communication. The advantage here is that replacing an individual layer should be relatively easy. If one layer malfunctions or requires an upgrade, you can handle it without disturbing the entire system. Since it uses and depends only on the lower layers, it can be pulled out and replaced with a new one that provides similar functionalities. This eases maintenance and allows for more flexibility when updating individual components. However, it's important to note a key drawback of this layered approach. It is often database-centric, meaning much importance is given to the database layer, which might lead to some issues. For instance, think of database normalization. It is necessary

for databases but not a good means to model a business domain. So, while a monolith's structure and way of segregating functionalities into layers can benefit replaceability and maintenance, its innate focus on the database layer could be a bottleneck in overall performance. When designing your architecture, aim to balance ease of maintenance and system performance.

13 Dependency Inversion

Dependency Inversion



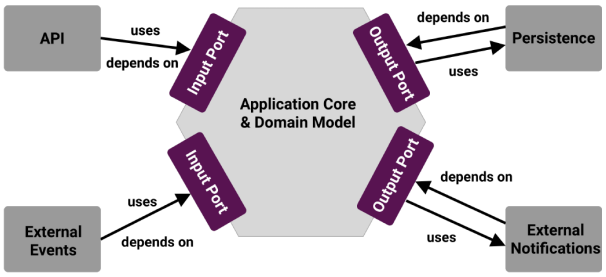
- Layers only **use** lower layers
- But all layers **depend on** the business logic

- **Consequence: the business logic is the core**

What can we do to overcome this database-centric approach? Please have a look at the picture next to me. We still have the same three layers. Upper layers still only communicate with lower layers. However, there is a big difference here. Look at the arrows on the right ride. The implementation of the presentation layer depends on implementing the business logic layer. The persistence layer now depends on the business logic. The arrow points upwards. This is a huge difference from the first approach on the last slide. What does it mean? Think about this: In the old approach, the persistence layer provides an interface for all its functions. Data structures defined by the persistence layer must be used in these functions. The business layer needs to use this interface and, therefore, needs to use this data structure. Hence, the business logic depends on the persistence layer. Now, we inverse this dependency. In this approach, the business logic is the application's core and defines the data structures. The persistence layer must provide its services using the data structures at the interface. // With the dependency inversion principle, higher-level modules, those closer to the business logic, should not depend on lower-level modules. Instead, they should depend on abstractions, making our software more flexible, reusable, and easier to maintain.

14 Hexagonal Architecture – Ports and Adapters

Hexagonal Architecture – Ports and Adapters



- Input ports accept incoming requests
- Output ports forward outgoing requests
- Outer components depend on the ports!

Let's delve into the Hexagonal Architecture, also known as Ports and Adapters. This architecture puts the Domain Model and Application Core at the heart of the system, allowing it to interact with outside elements via ports. This architecture is an example of the dependency inversion principle shown on the last slide. It's a design method that helps keep your application flexible and resilient to changes in external factors. The term 'Input Ports' refers to devices specifically designed to accept incoming requests. These devices act as receivers, processing information from outside the application's core. Whether this is user interaction, data from external systems, or even other applications, it's the responsibility of these input ports to funnel these inputs in a way our core can understand. On the other hand, the 'Output Ports' handle the outgoing requests. They're our communicators to the outside world. Whether interacting with databases or external systems, these output ports ensure that our core's response is sent and interpreted correctly. What is key to remember here, and somewhat counterintuitive, is that our software's outer components depend on our ports. This is a key principle of "Ports and Adapters" methodology. Instead of having our core dependent on the implementation details of the outer layers, we invert that dependency. This approach allows for changes in technology or the requirements of the outer layer without major adjustments to the core of our system. In essence, Hexagonal architecture provides a structured way for the system to interact with the outside world while keeping its core independent and flexible. It's an effective strategy for ensuring a clean separation of concerns between the external elements and the heart of your system. We will use this architecture in many of our examples and exercises.

15 Distributed Architectures

Distributed Architectures

- In contrast to the monolithic architecture

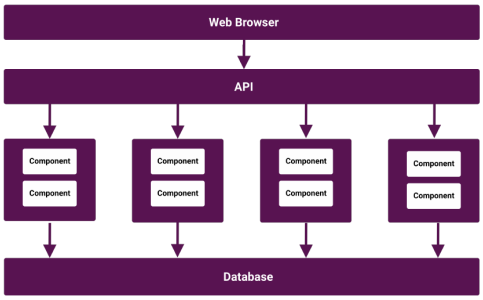
- The application is split into components
- Each component is a deployment unit
- Each component runs on a (virtual) computer

- Components communicate via network

As we shift focus from monolithic architecture, we delve into distributed architectures. Unlike in monolithic structures, where the application is developed as a single unit, we break it into smaller functional components here. Essentially, we are separating or distributing our application across different nodes. Every component we create is considered a separate deployment unit. Imagine these components as distinct entities doing their tasks. In the context of distributed software architecture, they are individually deployable and can be distributed across various machines or virtual computer systems. Now, these components can't work in silos. They need to communicate and work together to ensure the software functions. This is where the network comes in. The network facilitates communication between these components. We will now briefly introduce three types of distributed architectures.

16 Example: Service-based Architecture

Example: Service-based Architecture

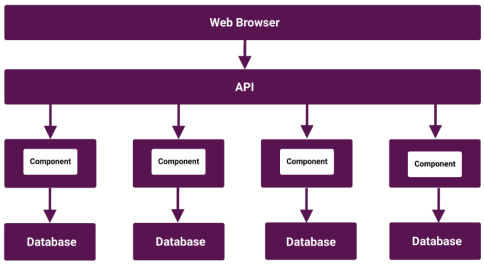


- Each purple box is one (virtual) computer
- Components are grouped (e.g., logically)
- Important: Central database

We start with the service-based architecture. Direct your attention to the slide. You'll notice several purple boxes. Each represents a virtual computer or device. These are the cores of our distributed system, all working together, yet each performing its specific function. We distribute the components on these computers. One computer is responsible for hosting the API component. One computer is responsible for the database. In the middle, we assign the components to many computers. Here, we group components to services. The picture only shows two components on a single computer, but this can be more or less. Such grouping depends on various factors, including their functionalities, dependencies, and role in the overall system. Cohesion and coupling are now important again. Another key aspect, arguably the most critical one in a service-based architecture, is the central database. All components use this central database. // As the name suggests, service-based architecture provides specific services efficiently and effectively according to the user's needs. It's modular, flexible, and scalable, ideal for many modern distributed systems.

17 Example: Microservices Architecture

Example: Microservices Architecture



- Each service is small(er) and contains the database
- The microservices communicate with each other

- Advantages: scalability, independent deployment
- Disadvantages: complexity in management

One step further is the micro-service architecture, which is quite fashionable right now. The picture next to me is very similar to the last one. There are two differences. There is only a single component on each computer in the middle layer, where the services are located. Second, each so-called microservice has its own database. I chose these two differences to make it easy to understand. In practice, this might be an oversimplification. Developing microservices is extremely difficult, and you must have a lot of experience in the business domain and software architecture in general to create a good microservice-based architecture. This model provides several key advantages. First, it is scalable. Each microservice can be scaled up or down independently based on your software's requirements. If one service is seeing more usage, you can allocate more resources to that service without needing to scale the whole system. Second, microservices offer independent deployment. Any changes or updates can be executed on each service individually, minimizing the risk to the overall system. However, like all architectures, microservices have their challenges. The most prominent is the complexity in management. This complexity arises from the fact that you are managing many small services instead of one monolithic system. Netflix uses such a microservice architecture that consists of about one thousand services. You have to ensure these independent services work perfectly together, which can require careful management and oversight.

18 Example: Event-Driven Architecture

Example: Event-Driven Architecture

- Architecture that reacts to events or changes in state
- No request-reply communication pattern

- Advantages: asynchronous processing, decoupling
- Disadvantages: message handling complexity

The last example is the event-driven architecture. It is also a common architectural style for distributed systems. Please consider this a system that reacts or responds to events or changes in its state. Unlike the traditional request-reply communication pattern we discussed before, in event-driven architecture systems, communication is more about notifying involved components about the occurrence of events. So, rather than waiting for a response after making a request, the system responds to changes. We keep this introduction short because we will not discuss this architecture further in this class or use any technology for event processing. // An outstanding advantage is the asynchronous processing feature. This ensures the system does not have to wait for an operation to complete before moving on to another. Also, this architecture type supports decoupling. A key challenge you might experience in this architecture rests in its inherent complexity in handling messages. Because of the asynchronous nature, troubleshooting and tracking issues can become a more difficult task since we no longer follow a linear flow of operations.

19 Summary

Summary

- Software architecture defines the structure
- Monolithic architectures are simple but less flexible
- Distributed architectures offer scalability
- Choosing the suitable architecture is important

That's it for today. We introduced you to the topic of software architectures of backend systems. Let's summarize the key points we've covered. At the core, remember that software architecture essentially defines the blueprint or structure of a system. We've discussed the pros and cons of different types of architecture, and one of the earliest forms we covered was monolithic architecture. This type of software architecture is simple and often easier to design and deploy. However, the downside is that it is less flexible. It can be challenging to scale up or modify a system built in a monolithic fashion because any alterations often require changes to the entire system. // In contrast, we have distributed architectures, which distribute different software components across multiple systems or servers. These architectures offer better scalability, one of their most significant advantages. It's like building with Lego blocks; we can scale up the system by adding more blocks as needed. // However, remember that choosing the suitable architecture for a software system is critical. There's no one-size-fits-all solution here. The choice of architecture largely affects the system's efficiency, performance, and maintainability in the long run. Therefore, it's essential to analyze the requirements and constraints of your specific project before settling on an architectural style. We will focus on monolithic architecture in the hexagonal interpretation for the rest of this class.