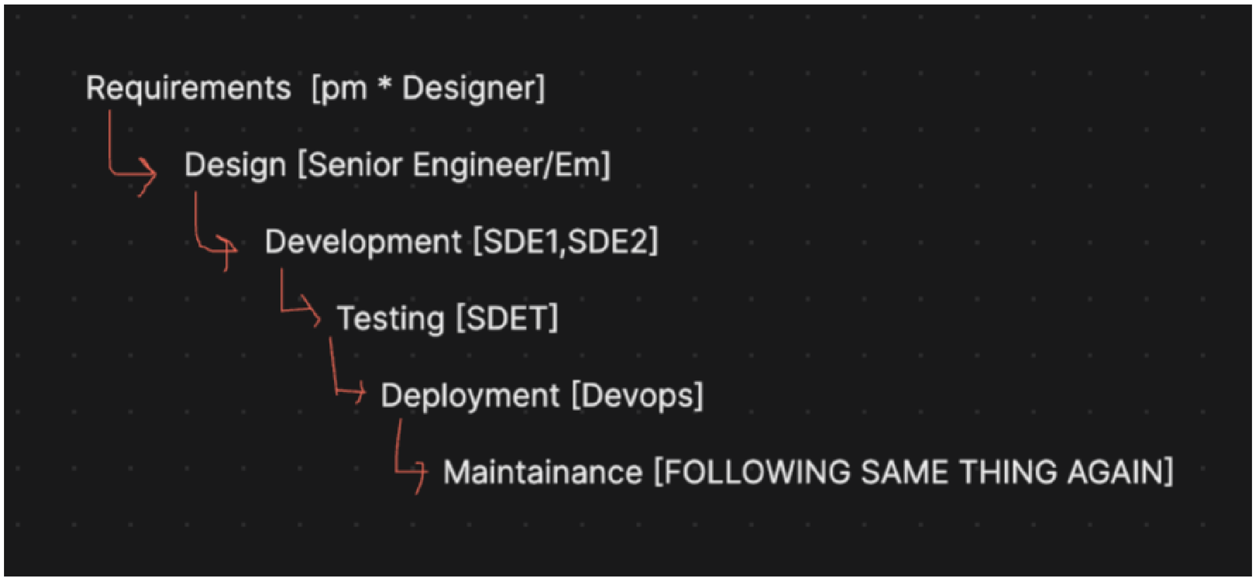
Episode-01 | Microservices vs Monolith - How to Build a Project

Waterfall Model (SDLC)



Requirements [pm * Designer]

Design [Senior Engineer/Em]

Development [SDE1,SDE2]

Testing [SDET]

Deployment [Devops]

Maintainance [FOLLOWING SAME THING AGAIN]

Monolith vs Microservices

Monolithic Architecture

A **monolith** refers to a large, single codebase that contains everything a project needs to function. This includes:

- Database (DB)
- Frontend (user interface)
- Backend (business logic)
- Authentication
- Code for additional features like analytics

In a monolithic architecture, **all components are tightly integrated** and exist within a single repository. While this structure can be simple to start with, it can become harder to maintain as the project grows larger. Updates or changes to one part of the system can affect other components, making the development and deployment process more complex.

Microservice Architecture

In **microservice architecture**, the application is divided into smaller, independent services, each responsible for a specific feature or functionality. For example, you might have:

- A **notification** microservice that handles all user notifications
- An **authentication** microservice that manages user login and security

Each microservice is developed, deployed, and maintained separately, and often by different teams. This allows for greater flexibility, scalability, and ease of maintenance.

In large companies like **Uber**, there are dedicated teams managing individual microservices. For example:

- One team handles the **fare calculation** microservice.

- Another team handles the **cab booking** microservice.

This separation of concerns improves efficiency and enables teams to work independently, leading to faster development cycles and more robust applications.

---

1. **Development Speed**
    - **Monolith**: Development is slower in monolithic architecture because all developers work on a single, large codebase. This can lead to code conflicts and longer merge times.
    - **Microservices**: Development is faster because each service has its own code repository, allowing teams to work independently without stepping on each other's toes.
2. **Code Repositories**
    - **Monolith**: Since there's a single repository for the entire project, it slows down the development process, especially with larger teams.
    - **Microservices**: Multiple repositories, each dedicated to a different service, enable faster development and better version control.
3. **Scalability**
    - **Monolith**: Scaling a monolithic application in large companies is difficult because all components are tightly coupled.
    - **Microservices**: Microservices are inherently easier to scale since each service can be scaled independently based on demand.
4. **Deployment**
    - **Monolith**: Deploying a monolithic application means redeploying the entire system, even for minor changes, such as a frontend feature update.
    - **Microservices**: You can deploy services independently. For example, you can deploy just the frontend service without affecting the rest of the system. While this is an advantage, it can also become a con, as managing multiple deployments can introduce complexity.
5. **Tech Stack Flexibility**

- **Monolith**: You're typically restricted to a single tech stack across the entire application, which may limit flexibility.
- **Microservices**: Different services can use different technologies. For example, you might use **React** for the admin dashboard and **Angular** for the login page. This allows teams to choose the best tool for each job, commonly seen in large companies.

6. **Infrastructure Cost**
    - **Monolith**: Generally, a monolith has lower infrastructure costs because you only need to manage a single server or environment.
    - **Microservices**: Microservices often require multiple servers and separate teams, leading to higher costs in terms of both infrastructure and personnel.
7. **Complexity**
    - **Monolith**: Easier to manage for small projects, but as the project grows, it becomes more difficult to maintain and evolve.
    - **Microservices**: More complex to set up for small projects due to the need for inter-service communication and management, but for large applications, microservices are less complex to scale and maintain.
8. **Fault Isolation**
    - **Monolith**: A failure in one part of the system can lead to the entire application crashing.
    - **Microservices**: Faults are isolated to individual services, meaning only the affected service will crash, while the rest of the system remains operational.
9. **Testing**
    - **Monolith**: Writing test cases is generally easier since everything is in one place, but as the codebase grows, maintaining tests can become challenging.
    - **Microservices**: Testing becomes more difficult due to the distributed nature of services, but it allows for testing services independently.

10. **Ownership**
   - **Monolith**: Typically, there's centralized ownership, with all teams contributing to a single codebase.
   - **Microservices**: Each team takes ownership of its respective service, providing more accountability and autonomy.
11. **Maintenance**
   - **Monolith**: Maintenance becomes harder over time as the codebase grows.
   - **Microservices**: Maintenance is easier because services are decoupled and smaller, making updates more manageable.
12. **Reworks and Revamps**
   - **Monolith**: Large-scale changes are challenging because everything is tightly coupled.
   - **Microservices**: Revamping one part of the system is easier since you can modify one service without affecting the others.
13. **Debugging**
   - **Monolith**: Debugging can be simpler since everything is contained in one place, but large codebases can still be difficult to navigate.
   - **Microservices**: Debugging is tougher because it involves multiple services, and issues can lead to a 'blame game' where teams may accuse each other's services of causing the problem.
14. **Developer Experience**
   - **Monolith**: Developers can feel restricted by the single codebase and limited tech stack.
   - **Microservices**: Many developers prefer working with microservices due to the flexibility, independence, and varied tech stacks.

NamasteDev.com Architecture

1. **Microservices Setup**:

NamasteDev.com operates with multiple microservices, where each part of the system is built with different technologies tailored for specific roles. Here's a breakdown:

- **Student Web (Frontend)**:
    - Built with **Next.js**, the Student Web service handles the student-facing part of the application. It provides a fast and SEO-friendly user interface for students to interact with.
- **Admin Dashboard (Frontend)**:
    - Developed in **React.js**, the Admin Dashboard is used by administrators to manage content, students, and other aspects of the platform. This dashboard gives the admin full control over the system in a user-friendly environment.
- **Backend Application**:
    - Written in **Node.js**, the backend service acts as the bridge between the frontends (Student Web and Admin Dashboard) and the database. It handles API requests, processes data, and manages the core logic of the application.
- **Student Mobile App (Frontend)**:
    - The mobile app, developed using **React Native**, provides students with a mobile-friendly experience, syncing with the same backend as the web applications.

2. **Communication Between Services**:
    - The **Node.js backend** communicates with both the **Student Web** and the **Admin Dashboard** through APIs. It also fetches and processes data from the database, sending appropriate responses back to the respective frontend applications.

DevTinder Architecture

In **DevTinder**, we'll adopt a similar microservice-based architecture but with a simpler setup. Here's how it works:

1. **Frontend (Microservice)**:
    - Built using **React.js**, the frontend will be responsible for rendering the user interface. It will communicate with the backend through API calls to display user profiles, matches, and other features.
2. **Backend (Microservice)**:
    - The backend is written in **Node.js**. This service will handle requests from the frontend, interact with the database, and serve the necessary data to the frontend.

Example: Accessing User Profiles

Let's break down what happens when a user visits `devtinder.com/profile`:

1. The frontend (React) makes an **API call** to `/getprofile` to request the user's profile information.
2. The **Node.js backend** receives the API request and processes it.
3. The backend then **retrieves the user data** from the database.
4. Once the data is fetched, the backend **sends the profile information** back to the frontend.
5. The frontend takes this data and **renders the profile** on the UI for the user to see.

This microservice structure allows the frontend and backend to work independently, ensuring that the backend can handle multiple types of requests efficiently, while the frontend focuses on delivering a smooth user experience.