Introduction to .NET

Florin Olariu

“Alexandru Ioan Cuza”, University of Iași

Department of Computer Science

Agenda

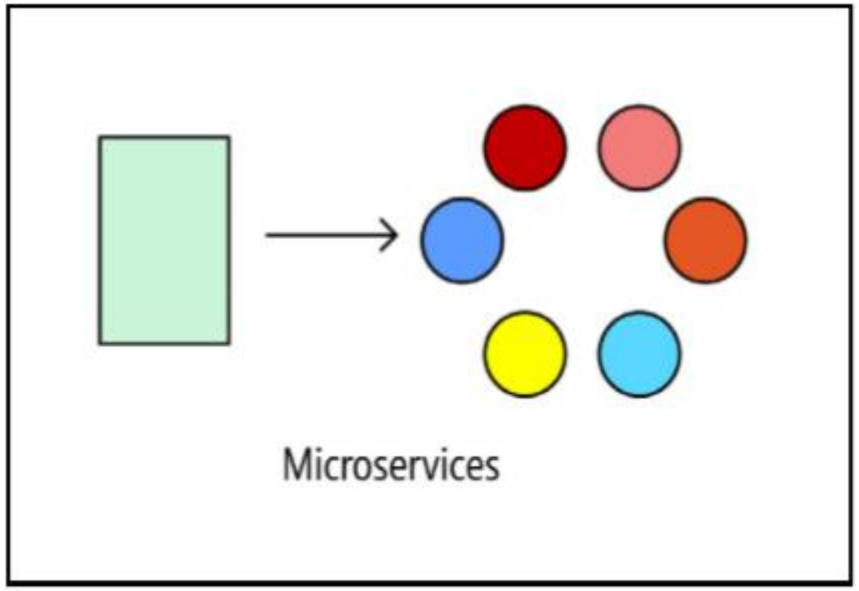
- Microservices architecture style
  - When to use microservices
  - Microservices benefits
  - Microservices challenges
- CQRS architecture style
  - When to use CQRS
  - CQRS benefits
  - CQRS challenges
- CQRS in microservices
- Understanding the deployment terminology
- Demo

Microservices architecture

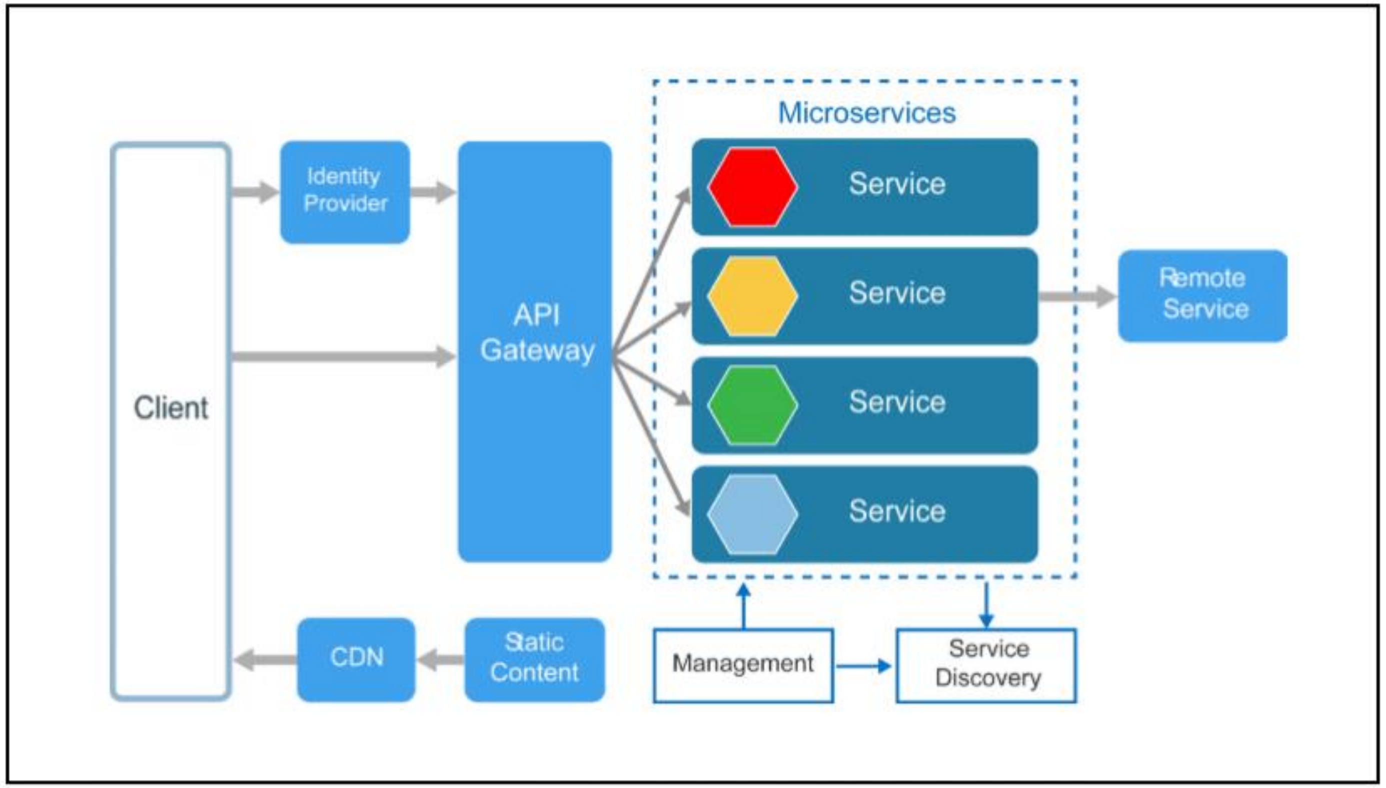
Microservices architecture style

- A microservices architecture consists of a collection of small, autonomous services.
- Each service is self-contained and should implement a single business capability

Microservices architecture



Microservices architecture



Microservices architecture

- In a microservices architecture, services are small, independent, and loosely coupled.

Microservices architecture

- In a microservices architecture, services are small, independent, and loosely coupled.
- Each service is a separate codebase, which can be managed by a small development team.

Microservices architecture

- In a microservices architecture, services are small, independent, and loosely coupled.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.

Microservices architecture

- In a microservices architecture, services are small, independent, and loosely coupled.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.

Microservices architecture

- In a microservices architecture, services are small, independent, and loosely coupled.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.

Microservices architecture

- In a microservices architecture, services are small, independent, and loosely coupled.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Services don’t need to share the same technology stack, libraries, or frameworks.

When to use microservices

When to use microservices

- Large applications that require a high release velocity.

When to use microservices

- Large applications that require a high release velocity.
- Complex applications that need to be highly scalable.

When to use microservices

- Large applications that require a high release velocity.
- Complex applications that need to be highly scalable.
- Applications with rich domains or many subdomains.

When to use microservices

- Large applications that require a high release velocity.
- Complex applications that need to be highly scalable.
- Applications with rich domains or many subdomains.
- An organization that consists of small development teams.

Microservices benefits

Microservices benefits

- Independent deployments. You can update a service without redeploying the entire application, and roll back or roll forward an update if something goes wrong. Bug fixes and feature releases are more manageable and less risky.
- Independent development. A single development team can build, test, and deploy a service. The result is continuous innovation and a faster release cadence.
- Small, focused teams. Teams can focus on one service. The smaller scope of each service makes the code base easier to understand, and it’s easier for new team members to ramp up.

Microservices challenges

Microservices challenges

- Complexity. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.

Microservices challenges

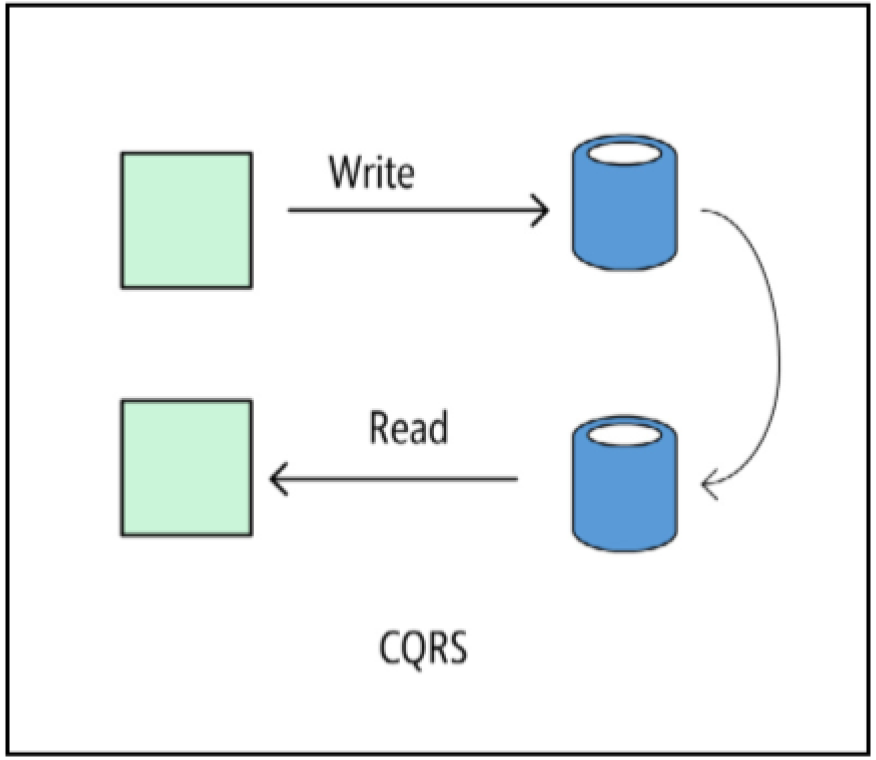
- Complexity. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- Data integrity. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.

Microservices challenges

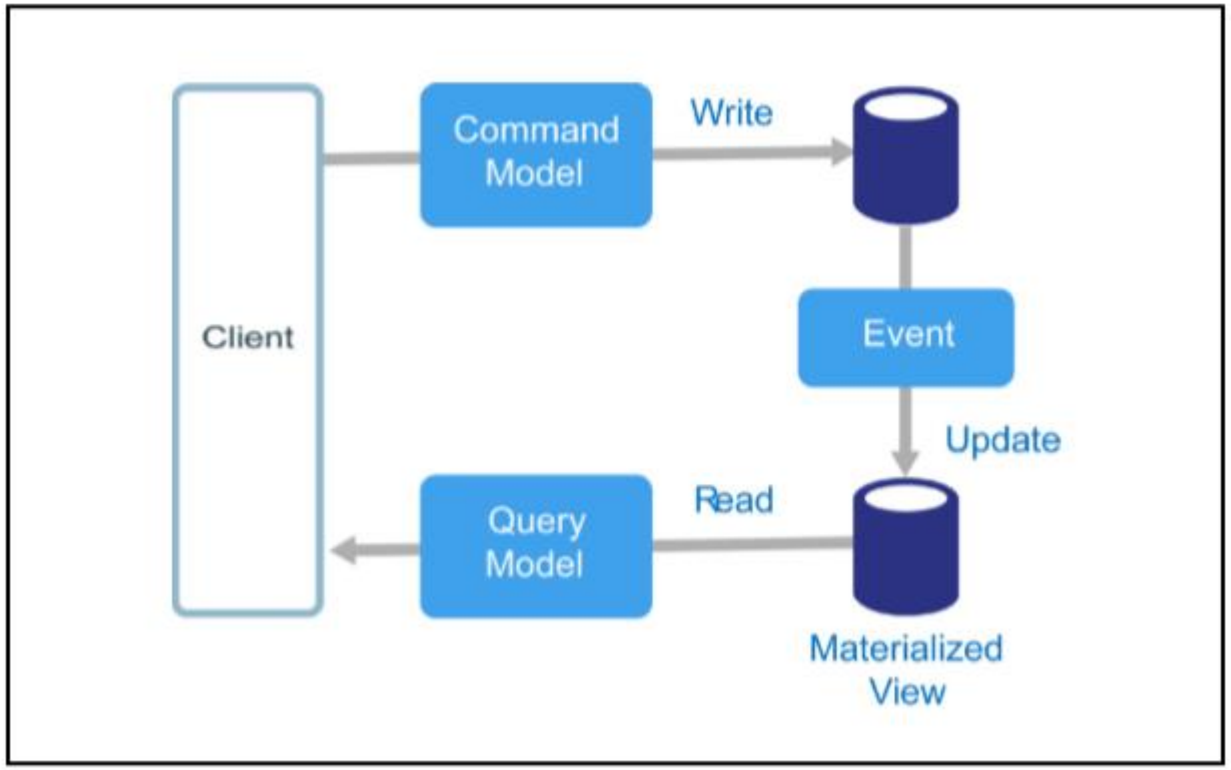
- Complexity. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- Data integrity. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- Skillset. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

CQRS architecture style

CQRS architecture style



CQRS architecture style



CQRS architecture style

- Commands should be task based, rather than data centric. (“Book hotel room,” not “set ReservationStatus to Reserved.”) Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

When to use CQRS

When to use CQRS

- Consider CQRS for collaborative domains where many users access the same data, especially when the read and write workloads are asymmetrical. CQRS is not a top-level architecture that applies to an entire system.

When to use CQRS

- Consider CQRS for collaborative domains where many users access the same data, especially when the read and write workloads are asymmetrical. CQRS is not a top-level architecture that applies to an entire system.
- Apply CQRS only to those subsystems where there is clear value in separating reads and writes. Otherwise, you are creating additional complexity for no benefit.

CQRS benefits

CQRS benefits

- Independently scaling. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.

CQRS benefits

- Independently scaling. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
- Separation of concerns. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.

CQRS benefits

- Independently scaling. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
- Separation of concerns. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
- Simpler queries. By storing a materialized view in the read database, the application can avoid complex joins when querying.

CQRS challenges

CQRS challenges

- Complexity. The basic idea of CQRS is simple. But it can lead to a more complex application design, especially if they include the Event Sourcing pattern.

CQRS challenges

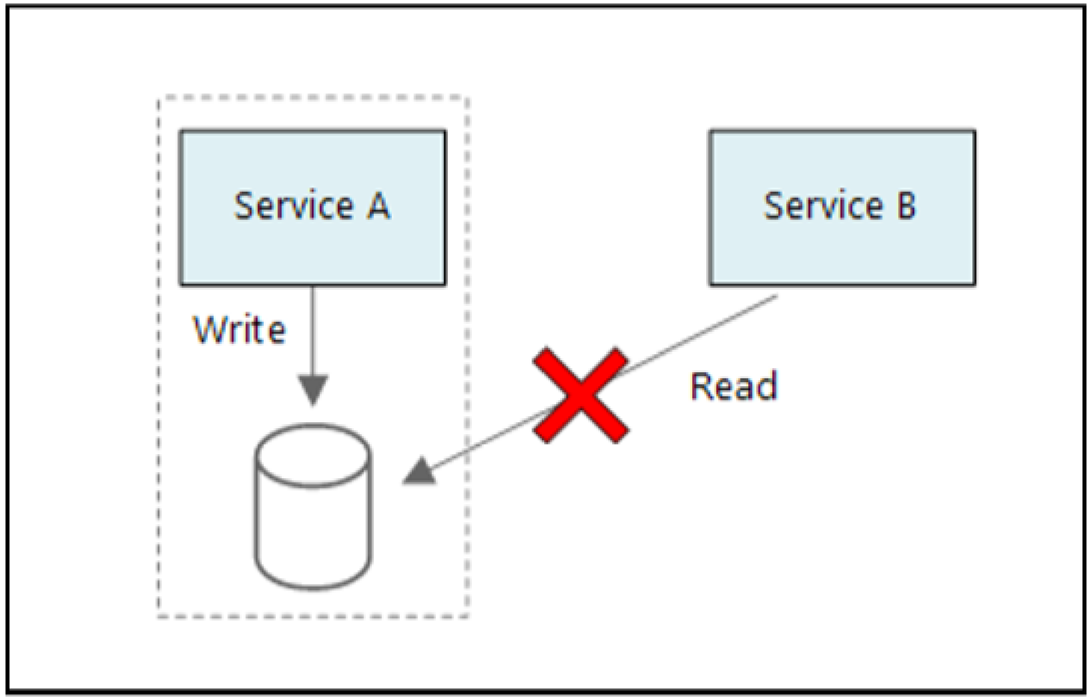
- Complexity. The basic idea of CQRS is simple. But it can lead to a more complex application design, especially if they include the Event Sourcing pattern.
- Messaging. Although CQRS does not require messaging, it’s common to use messaging to process commands and publish update events. In that case, the application must handle message failures or duplicate messages.

CQRS challenges

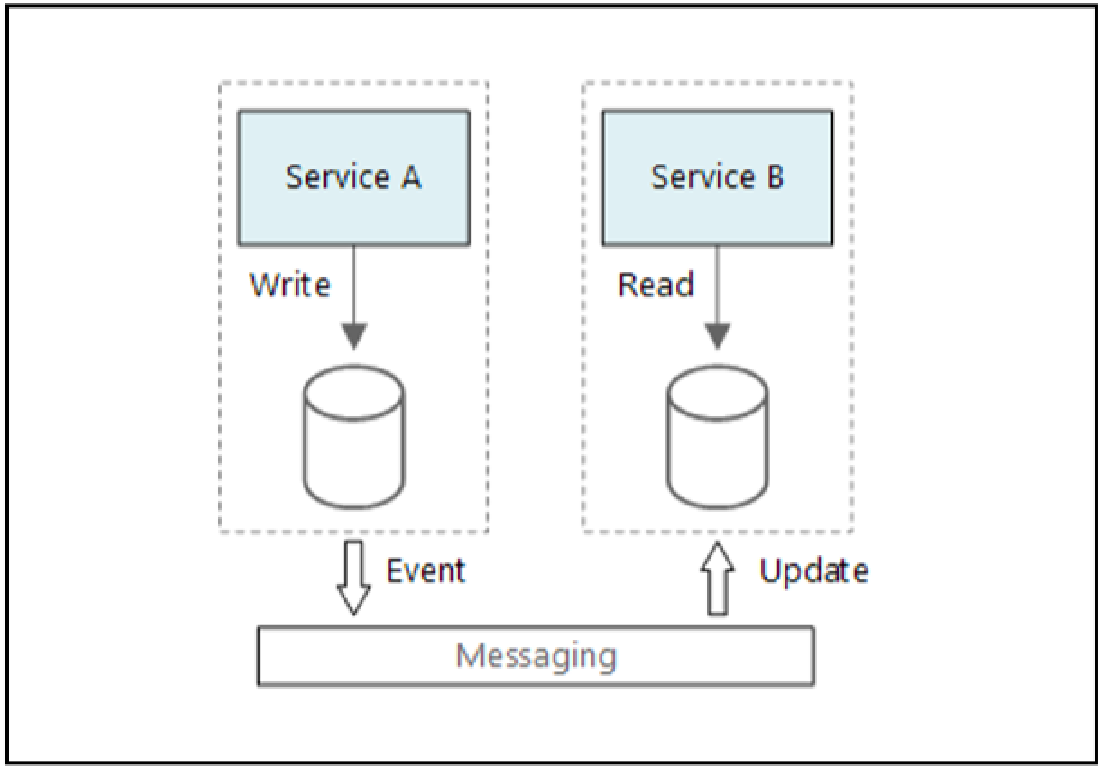
- Complexity. The basic idea of CQRS is simple. But it can lead to a more complex application design, especially if they include the Event Sourcing pattern.
- Messaging. Although CQRS does not require messaging, it’s common to use messaging to process commands and publish update events. In that case, the application must handle message failures or duplicate messages.
- Eventual consistency. If you separate the read and write databases, the read data may be stale.

CQRS in microservices

CQRS in microservices



CQRS in microservices



Understanding the deployment terminology

Understanding the deployment terminology

- **Build**: In the build stage, the service source gets compiled without any errors along with the passing of all corresponding unit tests. This stage produces build artifacts.

Understanding the deployment terminology

- **Build**: In the build stage, the service source gets compiled without any errors along with the passing of all corresponding unit tests. This stage produces build artifacts.
- **Continuous Integration (CI)**: CI forces the entire application to build again every time a developer commits any change—the application code gets compiled and a comprehensive set of automated tests are run against it. This practice emerged from the problems of frequent integration of code in large teams. The basic idea is to keep the delta, or change to the software, small. This provides confidence that the software is in a workable state. Even if a check-in made by a developer breaks the system, it is easy to fix it this way.

Understanding the deployment terminology

- **Deployment:** Hardware provisioning and installing the base OS and correct version of the .NET framework are prerequisites for deployment. The next part of it is to promote these build artifacts in production through various stages. The combination of these two parts is referred to as the deployment stage. There is no distinction between the deployment and release stage in most monolithic applications.

Understanding the deployment terminology

- **Deployment**: Hardware provisioning and installing the base OS and correct version of the .NET framework are prerequisites for deployment. The next part of it is to promote these build artifacts in production through various stages. The combination of these two parts is referred to as the deployment stage. There is no distinction between the deployment and release stage in most monolithic applications.
- **Continuous Deployment (CD)**: In CD, each successful build gets deployed to production. CD is more important from a technical team's perspective. Under CD, there are several other practices, such as automated unit testing, labeling, versioning of build numbers, and traceability of changes. With continuous delivery, the technical team ensures that the changes pushed to production through various lower environments work as expected in production. Usually, these are small and deployed very quickly.

Understanding the deployment terminology

- **Continuous delivery**: Continuous delivery is different from CD. CD comes from a technical team's perspective, whereas continuous delivery is more focused on providing the deployed code as early as possible to the customer. To make sure that customers get the right defect-free product, in continuous delivery, every build must pass through all the quality assurance checks. Once the product passes the satisfactory quality verification, it is the business stakeholders' decision when to release it.

Understanding the deployment terminology

- **Continuous delivery**: Continuous delivery is different from CD. CD comes from a technical team's perspective, whereas continuous delivery is more focused on providing the deployed code as early as possible to the customer. To make sure that customers get the right defect-free product, in continuous delivery, every build must pass through all the quality assurance checks. Once the product passes the satisfactory quality verification, it is the business stakeholders' decision when to release it.
- **Build and deployment pipelines**: The build and deployment pipeline is part of implementing continuous delivery through automation. It is a workflow of steps through which the code is committed in the source repository. At the other end of the deployment pipeline, the artifacts for release are produced. Some of the steps that may make up the build and deployment pipeline are as follows:

Understanding the deployment terminology

- **Continuous delivery**: Continuous delivery is different from CD. CD comes from a technical team's perspective, whereas continuous delivery is more focused on providing the deployed code as early as possible to the customer. To make sure that customers get the right defect-free product, in continuous delivery, every build must pass through all the quality assurance checks. Once the product passes the satisfactory quality verification, it is the business stakeholders' decision when to release it.
- **Build and deployment pipelines**: The build and deployment pipeline is part of implementing continuous delivery through automation. It is a workflow of steps through which the code is committed in the source repository. At the other end of the deployment pipeline, the artifacts for release are produced. Some of the steps that may make up the build and deployment pipeline are as follows:
- 1)Unit tests
- 2) Integration tests
- 3) Code coverage and static analysis
- 4) Regression tests
- 5) Deployments to staging environment
- 6) Load/stress tests
- 7) Deployment to release repository

Understanding the deployment terminology

- **Release**: A business feature made available to the end user is referred to as the release of a feature. To release a feature or service, the relevant build artifacts should be deployed beforehand. Usually, the feature toggle manages the release of a feature. If the feature flag (also called feature toggle) is not switched on in production, it is called a dark release of the specified feature.

One more thing…

One more thing…

Rules of Optimization:
Rule 1: Don't do it.
Rule 2 (for experts only): Don't do it yet.

Bibliography

- Pluralsight
- Application Architecture Cloud Guide - Microsoft

Questions

- Do you have any other questions?

Thanks!

See you next time! ☺