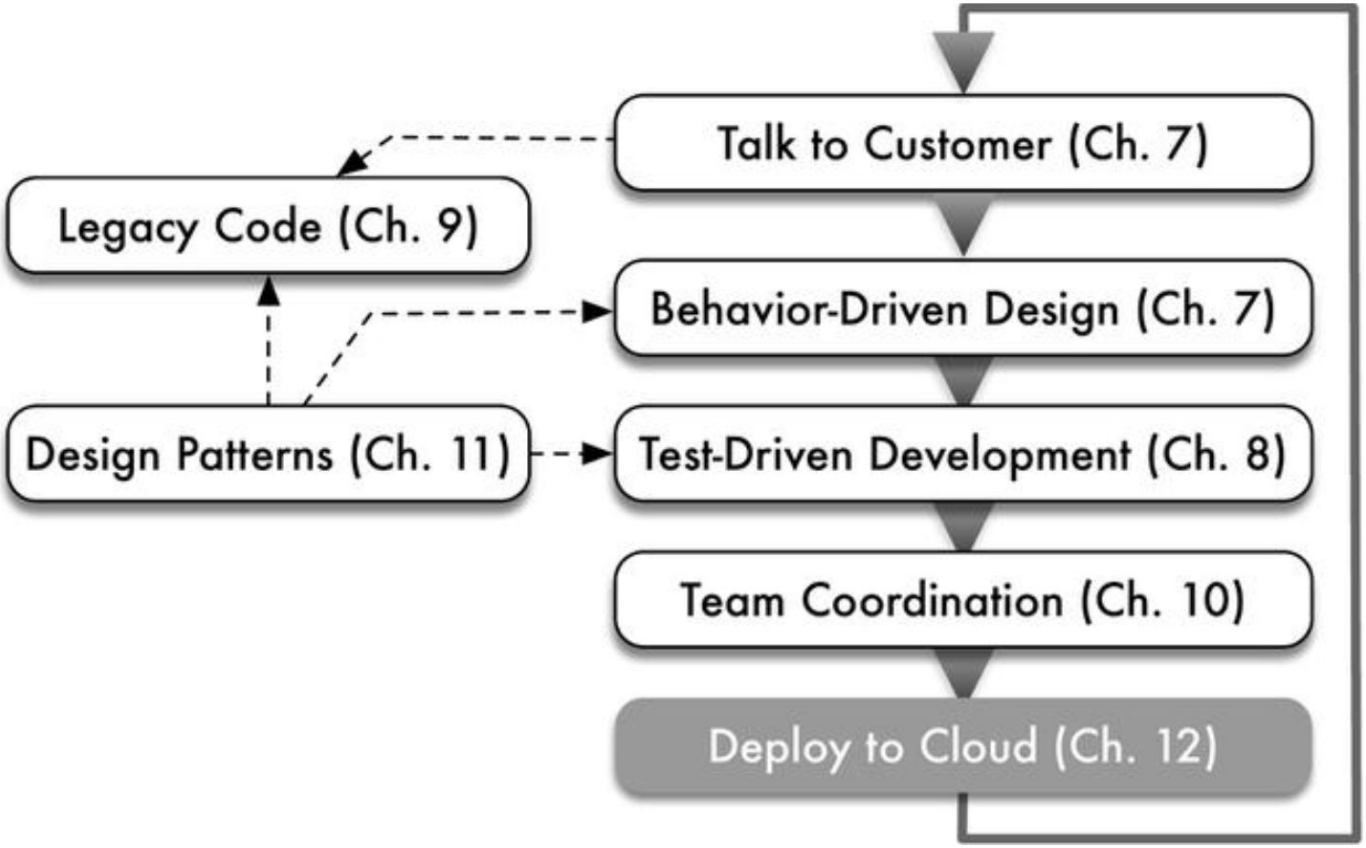
Module 12 (Dev/Ops)

CS169A

Chapter 12: Dev/Ops: Deployment, Performance, Reliability, and Practical Sec...

Enable All | Disable All | Show

Name	Pin	Due Date	Started	Completed
12.0 Concepts and Prerequisites			1	0
12.1 From Development to Deployment			1	0
12.2 Three-Tier Architecture			1	0
12.3 Responsiveness, Service Level Objectives, and Apdex			1	0
12.4 Releases and Feature Flags			1	0
12.5 Monitoring and Finding Bottlenecks			1	0
12.6 Improving Rendering and Database Performance With Caching			1	0
12.7 Avoiding Abusive Database Queries			1	0
12.8 CHIPS: The benefits of caching in SaaS			1	0
12.9 Security: Defending Customer Data in Your App			1	0
12.10 The Plan-And-Document Perspective on Operations			1	0
12.11 Fallacies and Pitfalls			1	0
12.12 Concluding Remarks: Beyond PaaS Basics			1	0


Talk to Customer (Ch. 7)
Legacy Code (Ch. 9)
Behavior-Driven Design (Ch. 7)
Design Patterns (Ch. 11)
Test-Driven Development (Ch. 8)
Team Coordination (Ch. 10)
Deploy to Cloud (Ch. 12)

Development vs Deployment

- The moment a SaaS app is deployed, its behavior changes because it has actual users.
- Applications should be designed for deployability and monitorability.

Development -

- Testing to make sure your app works as designed

Deployment -

- Testing to make sure your app works when used in ways it was *not* designed to be used

Development to Deployment

Criteria for Performance and Security that we will address:

- **Responsiveness:** How long do users wait before app delivers response?
- **Release Management:** How do we deploy/upgrade app in place without compromising availability and responsiveness?
- **Availability:** What percentage of time is the app correctly serving requests?
- **Scalability:** As users increases, does the app maintain availability + responsiveness without increasing cost per user?
- **Privacy+Data Integrity:** Is data accessible/mutable to/by authorized parties?
- **Data integrity :** Can the app prevent customer data from being tampered with, or at least detect tampering has occurred or that the data has been compromised.
- **Authentication**: Can we trust that the user is who they claim to be?

Such non-functional characteristics can be more important than functional features since such headaches can drive users away.

Development - Production Parity

- Unexpected behavior due to differences between the development and production environments.
- Not possible to foresee all such scenarios during deployment;
- Agile & automation is <3
- When it comes to deployment - Automate Everything

- If deployment of a new version of the software causes unexpected problems, how easily can you roll back to the previous version?
- If a schema migration or data migration causes unexpected problems, can you easily restore a database snapshot taken immediately before the migration was run?
- If every task was automated, you just type one line and rerun the script that takes care of everything. Imagine answering the above questions with a manual only process X

SaaS Goals:

- High availability & responsiveness,
- Release managements without downtime
- Scalability without increased user costs
- Defend customer data in the app

Good PaaS providers can provide infrastructure mechanisms to automatically handle some of the details of maintaining performance stability and security, but as a developer you must also address these concerns in various aspects of your app’s design and implementation.

Three Tier Architecture

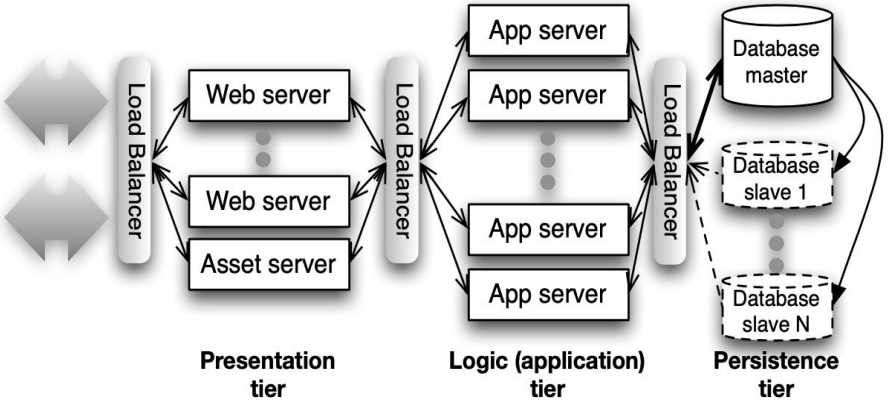
Separates the responsibilities of different SaaS server components (Web server, application server, and database tiers) and enables ***horizontal scaling*** to accommodate millions of users

Tiers:

- Presentation: Render views + interact with users
- Logic/Application: Runs SaaS app code + app logic
- Persistence: Stores app data

“Shared Nothing”: Can add more computers to presentation + logic tiers for scale -> Horizontal Scaling

Deployment and maintenance greatly simplified by deploying your app on a **Platform as a Service (PaaS).** The database cannot benefit from horizontal scaling in a 3-tier architecture as readily as the web server or app server, database capacity is an issue.



The master-slave configuration is appropriate for applications that experience what kind of workload?

A. Read heavy
B. Equal distribution of read and write operations.
C. Write heavy
D. None of the above.

The master-slave configuration is appropriate for applications that experience what kind of workload?

A. **Read heavy**
B. Equal distribution of read and write operations.
C. Write heavy
D. None of the above.

Responsiveness, SLOs, Apdex

Motivation: How do we improve availability?

Defensive Programming: Add error handling + resilience to code to avoid crashes

Software Rejuvenation: Restart subset of identical processes to neutralize leaks

Overprovisioning: Improve latency with just more computers! (Doesn't scale well)

Availability Metrics:

- Service Level Objectives: "95% requests in 5 min window have latency<100ms
- Apdex: SLO; 0 ≤ value ≤ 1; assigns full/half/no credit based on latency
    - is an open standard that computes a simplified SLO as a number between 0 and 1 inclusive representing the fraction of satisfied users. Given a user satisfaction threshold latency T selected by the application operator, a request is satisfactory if it completes within time T, tolerable if it takes longer than T but less than 4T, and unsatisfactory otherwise.

Releases & Feature Flags

Motivation: Deployment is tricky! Many problems may manifest at deployment time, so making sure we have testing guard rails + monitoring in place is important.

- In the roll-out process of a new version in a deployment of multiple server instances - some servers will have version n and others will have version n-1
- This is worse if the upgraded code needs a new schema version as well - cos if you roll out the schema first the old server code fails. And if you roll out the server code first it won't work without the new schema

Incremental Features Roll Out:

- Migrate ***atomically***: Take service offline, perform changes, bring back online. Simple, but directly causes unavailability.
- **Feature Flags**: Use configuration variables to indicate which code paths in an app are executed (i.e. preflight checking, A/B testing, complex features)

Monitoring & Finding Bottlenecks

Motivation: How do we measure responsiveness and availability in production app.

Monitoring consists of collecting app performance data for analysis and visualization. In the case of SaaS, application performance monitoring (APM) refers to monitoring the Key Performance Indicators (KPIs) that directly impact business value.

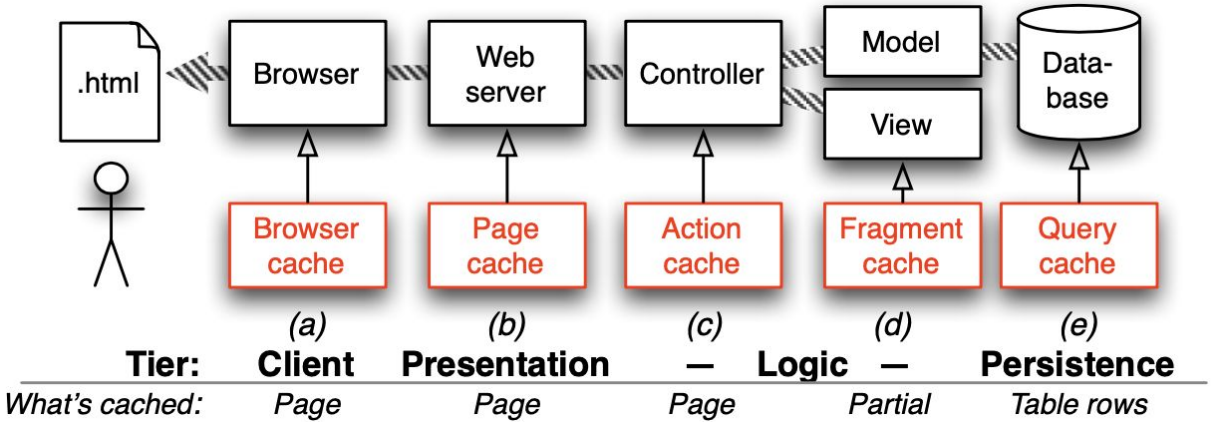
Techniques for monitoring SaaS apps:

- Active (intentional, external stimulus) or Passive (Only monitors until asked)
- External (Reports app behavior as seen from outside) or Internal (Hook into the code and see gauge the performance of internal workings)
- App performance or User Behavior?

Techniques

- **Profiling**: Internal monitoring during development
- **Stress/Longevity Testing**: Figure out costs of slow or expensive requests
- **Monitoring User Behavior:** Clickstreams, Think times, Abandonment

Caching



Motivation: Save cost of repeated actions by "caching" aka saving the data in an accessible place.

Challenges: Naming and Expiration

ActiveRecord has a variety of tools for performing caching at both software and hardware levels. When using it, make sure to separate cacheable from non-cacheable units.

Avoiding Abusive Queries (~CS186)

Problem: Poorly written queries can result in inefficient queries that needlessly take more time and resources than necessary to retrieve the desired results.

2 Common Mistakes:

1. **“n+1” Query Problem**: Association performs more queries than necessary
2. **Table Scan Problem**: Lack of proper indices for speeding up queries

Technical Terms:

- **Full Table Scan**: Scanning an entire table to perform a query (not optimal)
- **Database Index**: Data structure distinct from database that uses *hashing* to afford constant time access on database rows for conditioned queries. Offers more time efficient queries at expense of more disk space.

Defend customer data in the application.

Security Principles:

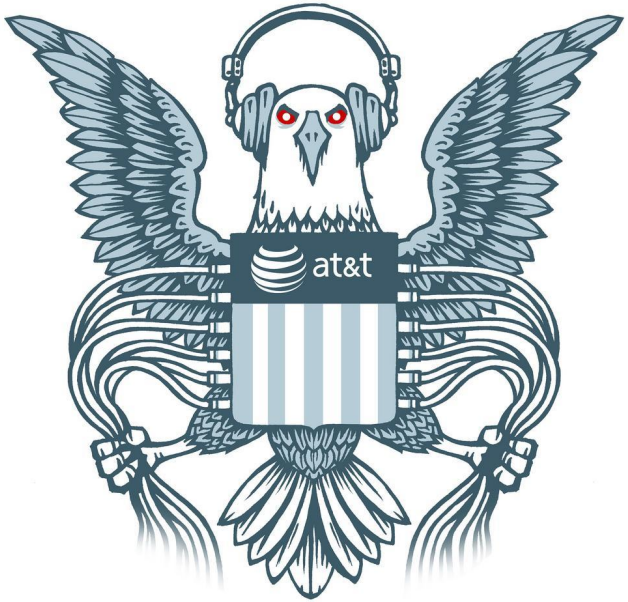
1. **Least Privilege:** User should have no more privilege than necessary for task.
2. **Fail-Safe Defaults:** User should be denied access unless explicitly authorized.
3. **Psychological Acceptability**: Protection should *not* make app harder to use.

We are the CIA

Confidentiality: preventing sensitive information from unauthorized access attempts

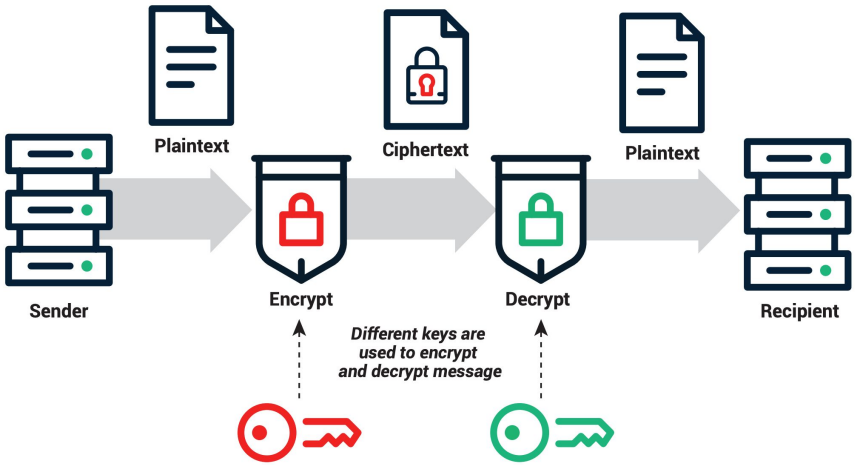
Integrity: data must not be changed in transit or tampered with in any step of transit.

Availability: information should be consistently and readily accessible for authorized parties

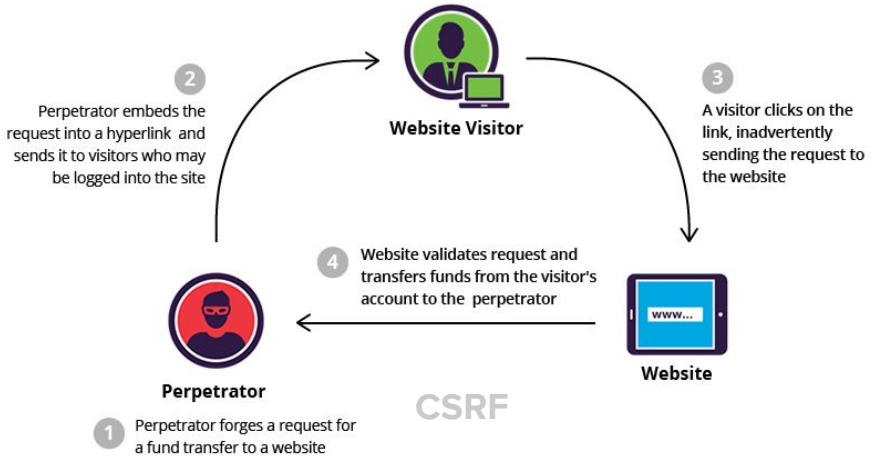


The 6 Security Vulnerabilities:

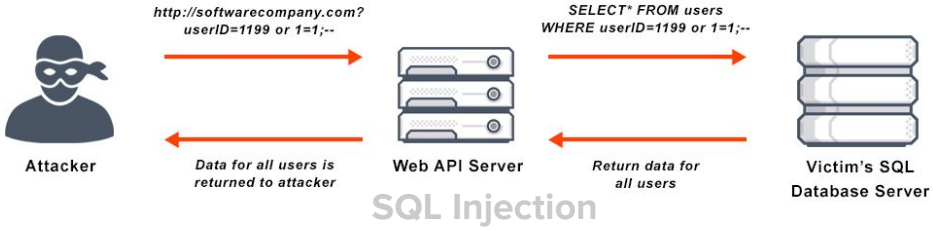
1. **Protecting Data Using Encryption**: Privacy, Authenticity, Integrity of data.
   - TLS encrypts HTTP traffic with asymmetric+symmetric encryption
   - Public Key Crypto: Protocol for two parties to exchange secrets
   - Certificate Authorities + Public Key Certificates = Prove Identity



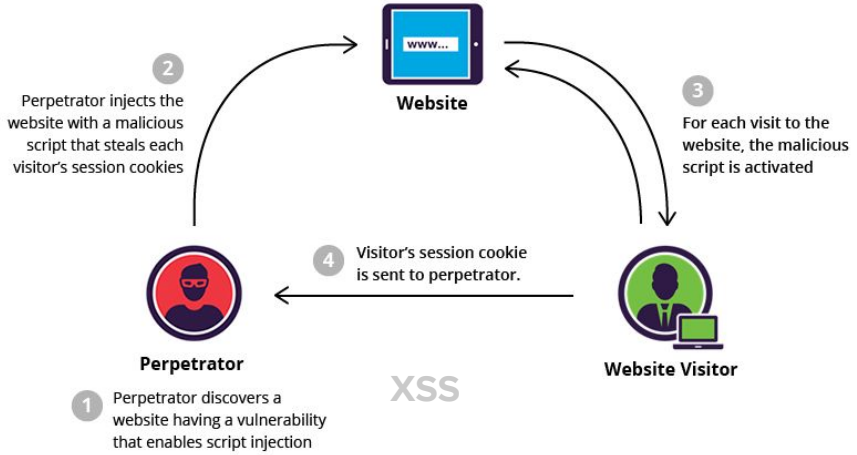
2. Cross Site Request Forgery: Trick user's browser into visiting a different website for which user has a valid cookie; allows browser to act on behalf of the malicious entity using the user's privileges.



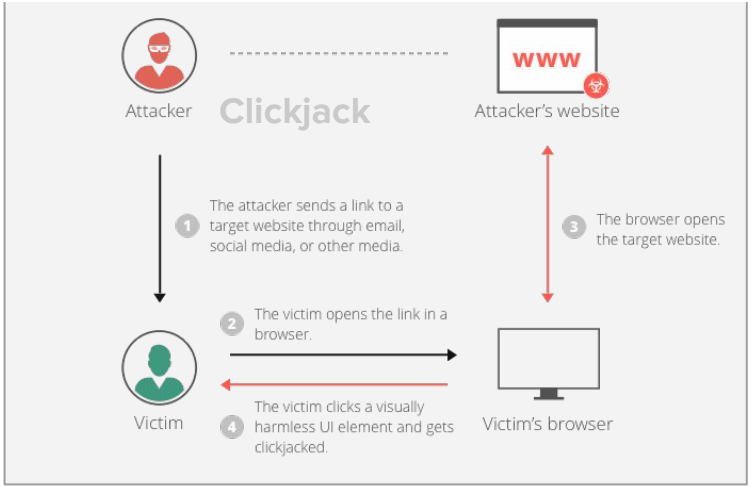
3. SQL Injection: Poorly written queries + unsanitized inputs allows attackers to manipulate a database by passing in parts of SQL queries as form data.



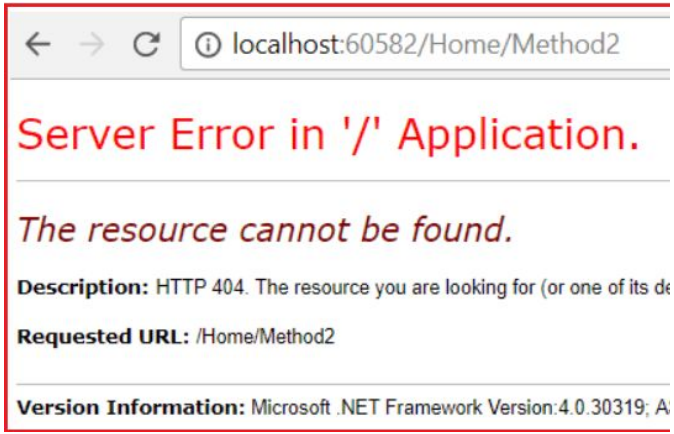
3. Cross Site Scripting: Javascript Code added and re-rendered by a website can trigger execution of malicious scripts on other users' browsers.



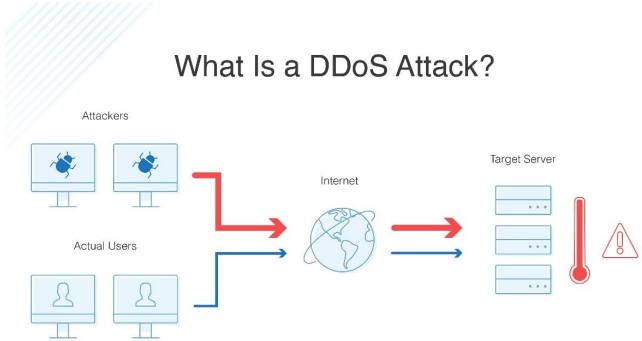
4. Clickjacking (a.k.a. UI Redress): Attacker mimics legitimate web pages with carefully overlaid UI elements that, when interacted with, cause users to unintentional perform unwanted actions. (bait buttons with invisible layers)



5. **Prohibiting Calls to Private Controller Methods**: If controllers have methods that aren't meant to be called by the user directly (just through an inside action), consider prohibiting the users from making these calls using "protected" keyword.



6. **Self-denial-of-service**: A malicious denial-of-service attack seeks to keep a server busy doing useless work, preventing access by legitimate users. You can inadvertently leave yourself open to these attacks if you allow arbitrary users to perform actions that result in a lot of work for the server, such as allowing the upload of a large file or generating an expensive report. For this reason, "expensive" actions are usually handled by a separate background process.



Which of the following are valid ways of enhancing database performance?

- Caching commonly used queries
- Indexing useful through-associations
- Creating a large table to store all the data together
- Declaring index uniqueness if possible

Which of the following are valid ways of enhancing database performance?

- **Caching commonly used queries**
- **Indexing useful through-associations**
- Creating a large table to store all the data together
- **Declaring index uniqueness if possible**

What is the purpose of indexing in a relational database?

- To speed up some types of queries
- To make filtering and searching in indexed attributes much faster
- To make data storage in the database more efficient, thus saving space
- To speed up table scans

What is the purpose of indexing in a relational database?

- **To speed up some types of queries**
- **To make filtering and searching in indexed attributes much faster**
- To make data storage in the database more efficient, thus saving space
- To speed up table scans

Attendance

Link: https://tinyurl.com/disc-week13

Passcode: devops