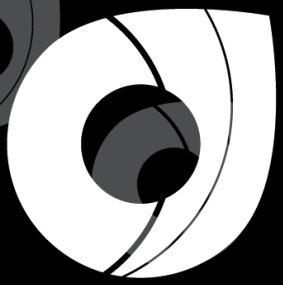
Corndel DevOps Engineering Programme

in association with Softwire

Module 9: Data & Security 1



Corndel Digital.

DB1A

Introduction

All but the simplest of applications require some sort of data. The use (and abuse) of data is pervasive in the modern world. As DevOps engineers you have great responsibility over how that data is used and secured.

In modules 9 and 10 we will cover a variety of topics relating to data and its security. This includes the practical matters of *how* we **store** and **secure** data; but also looking at the **legal obligations** we are under and the **ethics** of data stewardship.

While we will talk in detail about **databases** the contents of this module applies to data however it is stored: in file systems; in documents or spreadsheets; in physical filing cabinets or written on a Post-it note.

These subjects are important. It is easy to get them wrong, which can result in far reaching, significant *real world* consequences.

What is data?

According to the Cambridge English dictionary, `data` is

> *Information, especially facts or numbers, collected to be examined and considered and used to help decision-making, or information in an electronic form that can be stored and used by a computer.*

While your software handles the "usage" aspect of data, you will most likely need to store your data outside of your software. This can be done in many ways, depending on the software. For example:

- Image editing software saves images directly to the local file system.
- Online document editing software such as Google Docs saves entire documents in a cloud-based file system.
- Websites with a user login area save data about their users in a database.

While there are useful programs that don't store data you will certainly need to deal with data storage of a variety of types.

Table of Contents

Core Material

Additional Material

What to read?

This module is a little different to others. While the additional reading is still technically optional it covers **SQL**, which will be used during the workshop. If you're not already familiar with SQL you will find the workshop significantly easier if you do cover the additional reading at the end of this module.

Chapter 1

Data and Databases

What is a database?

Databases are specialised programs to store and manipulate data in a well-defined, structured manner. The database software is itself responsible for storing the data in a way that is efficient to write to, read from and search. In most cases you will not need to worry about exactly *how* it does this.

Databases can be broken down into two major types: *Relational* databases and *Non-Relational* databases.

Relational databases

These are the most commonly used databases for data storage today. Relational databases store data in pre-defined *tables* of data. A database will usually contain multiple tables, which reference, or are related to, each other - hence the name. The structure for how data is stored in a relational database has to be defined *before* data is added to the database.

Tables

A table in a relational database is defined by a *schema*, which consists of one or more *columns*. The schema describes the shape of the stored data.

The data itself is stored in *rows*, or *records*, which must all conform to the column definitions. For example, a table representing employees in an HR system can start out as:

User ID	Name	Hire Date	Leaving Date	Is Contractor
1	John Smith	2018-02-23	2020-04-12	`true`
2	Jane Doe	2020-03-04		`false`

In this example, the schema for the `employees` table is defined to have five columns: `User ID`, `Name`, `Hire Date`, `Leaving Date` and `Is Contractor`. The table contains two rows of data, representing employees John Smith and Jane Doe.

SQL

A language called *SQL* (**S**tructured **Q**uery **L**anguage) is used to communicate with the databases. It allows users to:

1. Modify database structure and schemas
2. Grant and revoke permissions for users
3. Create queries to read and write data

Its query syntax is very flexible and allows for highly complex data manipulation. SQL queries are composed of one or more *statements*, where each statement performs a single action on the database or data. For example:

- `SELECT` statements read data from one or more tables.
- `INSERT` statements write new rows to a table in the database.
- `UPDATE` statements edit existing rows in the database.
- `DELETE` statements delete rows from a table in the database.

For example, to find the names and hiring dates of all employees you could use the following query.

```
SELECT
[Name], [Hire Date]
FROM employees
```

CRUD

This acronym crops up a lot around databases, representing four types of operation that can be applied to data:

- Create
- Read
- Update
- Delete

CRUD can also be used as an adjective and a verb, e.g. a CRUD app is an application whose primary purpose is to let users CRUD data.

There are also statements such as `CREATE` or `ALTER` which change the database's schema by *creating* new tables or *altering* existing ones.

Transactions

A critical feature of relational databases is the concept of *transactions*, or single, self-contained units of work. A user of the database can be guaranteed that any actions performed within a transaction will either complete fully, or be reverted fully if the transaction fails, and cannot be interfered with by other users. This can be used to ensure that the database is never in an unknown state.

Example: *Imagine a banking application transferring £100 between two accounts. This involves deducting £100 from Account A and adding £100 to Account B. Without a transaction it is possible for the money to be deducted from Account A but* fail *to be added to Account B; this would leave the database in an inconsistent state, requiring some clean up. With a transaction if adding the money failed then the deduction would also fail; we would have to try to transfer the funds again but there wouldn't be any clean up to do.*

These properties of transactions are known as *ACID*:

- **A**tomicity: a transaction will either complete in its entirety or will have no effect at all on failure.
- **C**onsistency: a transaction will never violate any constraint rules of the database.
- **I**solation: multiple transactions running at the same will not affect each other.
- **D**urability: Once *committed* (complete), the effects of a transaction will persist even after a system failure.

A transaction can contain any number of SQL statements of any type. For example, this could be a valid transaction:

- A `DELETE` statement to remove employee with `User ID` equal to `1`
- An `ALTER` statement to add a `Job Title` column to the `employees` table above.
- An `UPDATE` statement to set `Job Title` to "Consultant" for employee with `User ID` equal to `2`.

The *ACID* properties are important. For example, if we have a database rule that prevents our account balance from dropping below zero, a transaction could guarantee *consistency* by failing if the result would violate this rule. *Isolation* is a little more

complicated: imagine there is an automated process that calculates interest on our savings account every night. By using transactions we can ensure that that interest is calculated either *before* **or** *after* we transfer money between accounts; however the calculation will never happen between the money leaving Account B and the money arriving in Account B.

Application logic

Most relational databases allow users to not only store data, but to write logic to be executed on the data directly in the database. The logic is usually written using SQL and is packaged into *functions* and *stored procedures*.

This logic can be triggered either manually or automatically:

- Manually: a user runs a short SQL statement which triggers a stored procedure. This can be useful when a complex piece of SQL code is run frequently, and is packaged into stored procedure instead of having users run the SQL directly.
- Automatically: A user sets up a *trigger*, which reacts to certain events in the database and automatically runs a given function or stored procedure. For example, an update to a database table can trigger a stored procedure to add an entry to an audit table with details of the change.

Examples

Most well-known database names are relational databases:

1. Microsoft SQL Server
2. MySQL
3. PostgreSQL
4. Oracle
5. SQLite
6. Microsoft Access

Non-relational databases

Any database which doesn't store data in relational tables with a defined schema is known as *non-relational*, or sometimes *NoSQL*.

The use of NoSQL to describe all non-relational databases isn't entirely accurate as many such databases support the use of SQL.

There many kinds of non-relational database, but some key types of non-relational

database you should be aware of include the following:

Kinds of non-relational databases

Document Stores

Instead of tables, these databases store data in *collections* of *documents*, where a given document uses a standard encoding such as JSON. The documents are not required to follow a fixed structure, and sometimes can have metadata associated with them.

These types of databases are useful when the structure of the data is unknown or changes rapidly, or when different kinds of data are stored in the same collection.

MongoDB is an example of a document store.

Key-Value Stores

These databases can be thought of as large lookup tables, where a given unique key is associated with a single value.

Key-value stores tend to be very light-weight and extremely fast at both reading and writing. Their main use case is to store and retrieve data which by the unique key, hence are widely used to store temporary data such as user sessions on a website, or as a cache layer for complex web pages to improve performance.

The line between key-value and document stores can be blurred, as it is possible to store complex objects (documents) as values in a key-value store, or pairs of values in a document database.

Examples of key-value stores include Redis, Memcached and DynamoDB.

Graph Databases

Instead of storing data as independent, self-contained structures such as rows in a table or documents in a collection, graph databases store *relationships* between data entries.

Such databases have a much smaller number of use cases than traditional databases, but are very powerful in these cases. Examples of these include

- Tracking relationships between users in a social network.
- Analytics of relationships between financial records for fraud detection.
- Recommendation engines for data with interconnected categories, such as streaming media and retail.

Comparison With Relational Databases

Given the variety in non-relational databases a comprehensive comparison is beyond the scope of this course; however, we can make some broad generalisations about the differences.

Non-relational databases do not enforce strict tabular schemas for data like relational databases do. They may still enforce some other type of schema, depending on the type of database.

Non-relational databases are often much easier to scale horizontally, i.e. to spread the database over multiple physical or virtual machines. This means that non-relational databases can still be performant at sizes where a relational database would struggle (or require prohibitively expensive hardware).

Non-relational databases may not offer ACID guarantees or transactions. This is generally a trade-off for faster performance or scalability.

Non-relational databases tend to be lighter-weight and efficient than relational databases, at the expense of having a much smaller feature set. This is useful for using databases for specialised use cases, instead of a general-purpose data store. They also tend to only store data and do not provide users the ability to run logic inside the database. Many powerful features of relational databases are less desirable in modern architectures, which prefer to keep application logic out of the database unless necessary for performance.

Other Data Stores

There are a number of other storage mechanisms that aren't usually referred to as databases. Typically these are tailored to a specific purpose. Examples include:

- **File storage** on the local file system or a cloud file storage option, like AWS' S3
- **Caches**, temporary storage for data that is expensive to access normally (e.g. Redis).
- **Messaging queues**, transient storage, for messages being exchanged between systems. These are recorded by one system, then picked up and deleted in order by another, (e.g. Kafka).
- **Search engines**, storage optimised for text searches of documents, (e.g. Elasticsearch).

Choosing a database

Given the plethora of choices of databases available, picking a database for a new software project can be daunting. In many cases a straightforward relational database will suffice. However, you should consider the nature of your data and how you expect it to be used. Often, the decision will be as much about how to configure and host a given technology as which technology to use.

It is possible that multiple databases are used for different parts of the application. For example, we can use a standard relational database to store details about artists, genres, tracks and playlists for a streaming music application, and use a graph database to power a recommendations engine based on relationships between artists, genres and songs.

Chapter 2

Database Maintenance

How to care for your database

There is more to maintaining a database than setting up a database schema and populating it with your data. We must ensure that the database is available whenever it needs to be (reliable), that it handles the workload placed upon it (performant) and that data is not lost when something does go wrong (recoverable).

Performance

Indices

Ensuring that your data is indexed correctly will mean queries of your database execute quicker and more efficiently. In order to get the most benefits, it is advisable to apply indices to columns which are used often for filtering data.

Depending on the database software, indices may need to be rebuilt periodically in order to ensure that they are stored on the file system optimally. If this is not done to tables with heavy write usage, then their data becomes disordered on disk - this is known as *defragmentation*.

Replicas

In some cases your application may have enough user traffic for the database itself to become a bottleneck. If most of the traffic reads from your database, then it may be useful to add read-only *replicas* of your database. These are automatically synchronised to the *primary* read-write database, and provide a read-only interface. This means the application can pick from multiple available databases from which to read, which reduces overall traffic to each instance if implemented correctly.

Reliability

Although database engines are carefully engineered for reliability out of the box, there are hygiene factors worth considering as part of your database provisioning setup:

- Ensure that the hardware where the database is running is sufficient for its use cases
- Ensure that the OS and database software are kept up to date to avoid security issues and bugs
- One or more read-only replicas can ensure that your application can function in read-only mode even if the primary database goes down

Disaster Recovery

Monitoring

A poorly performing database can bring an entire application to its knees. To avoid these problems we need to keep an eye on the health of the database and take action if necessary.

We want to monitor **hardware resource usage**, ensuring the database always has sufficient CPU, RAM and disk space to function properly; and **network connectivity** (it is no good having a database if it is difficult or impossible to connect to it).

Activity in **error logs** can also indicate a problem with the database not caused by the above problems. For example, trying to insert data that doesn't match the schema would cause an error but not otherwise impact the server.

A monitoring system should be able to automatically detect any system errors or resource shortages and ensure that somebody who can take action is notified. More sophisticated systems will be able to monitor database statistics and then report any sudden or significant variations from the norm. This can allow issues to be identified and corrected before they have a negative impact on the system.

Backups

In the case that the data stored within a database is corrupted, either by a system failure or development issue, it is vital that the database can be restored to a functional state as soon as possible. Regular database backups are essential in order to achieve this - without backups, the data can be lost for good.

Frequency and longevity of backups vary from database to database and depend on use cases. The trade-off is usually between the resources required to create and store back-ups versus potential data loss. For example, there is little need to have daily backups of a database which is updated once a month, while a database which deals with real-time data may need at least hourly backups.

Ideally, copies of database backups should be stored at a different physical site in case of catastrophic failure, e.g. a fire in the server room.

Restoring from backups

Keeping backups is important, but they are useless unless you can use them to restore your system after an outage. Practising restoring systems from backups is one way of demonstrating this.

Software Updates

In addition to the above it is important to keep your software up-to-date. This includes the database software itself and any other software on the machine running your database. These changes should be tested in other environments *before* being applied to the production environment.

In practice

It would be very hard to follow all of the above advice starting from scratch. Luckily, there are other options available to us.

Cloud Services

All of the larger cloud platform service providers offer some manner of managed database product. These are usually instances of well-known database software running somewhere in the cloud. The customer never needs to know the details of the hardware or platform where the database is running. Instead, they are simply given a method of connecting to the database, as well as contractually guaranteed availability.

This means that we as the customer no longer need to worry about:

- Maintaining the hardware
- Maintaining the OS of the server and applying security patches
- Keeping the database software up to date and applying security patches

Most of the services also offer options such as monitoring, automatic back-ups and replication out of the box.

There are of course downsides of using a managed database instance:

1. Cost: it can be more expensive to run a managed database instance than to rent a simple VM and manually install database software.
2. Customisability: Given that the customer is generally not allowed access to the underlying server running the database, their ability to customise the database installation itself is limited.

Generally, the "thicker" the abstraction layer, the harder it can be to diagnose and resolve issues with performance (or other bugs). This tends to be true of all abstraction layers, not just databases.

Containerisation

A managed cloud option may be unsuitable for our use case if:

- The database may require customisation beyond that offered by cloud providers.
- We may be required to store data on-premise for regulatory or legal reasons.
- The cost of using a managed cloud database is prohibitive.

In order to reduce our maintenance workload, we can apply the learnings from Module 5 and run our databases as containers.

Chapter 3

Database Security

Security is important

Having poor data security can have serious consequences for a business and its users. While we will cover security in more detail in Module 10 we will highlight some basic data security concerns in this chapter.

We hear about database breaches in the news, which damage consumer confidence in companies, and more importantly, can leak private data of its customers. Recent examples have included:

- 2015: A breach at Ashley Madison, a dating website specialising in extramarital affairs, made the identities of its users public.
- 2017: A breach at Equifax, one of the largest credit bureaus in the US, exposed personal information of around 148 million customers. The details included full names, addresses and social security numbers.
- 2018: A breach at the Marriott hotel chain results in millions of customer credit card and passport details leaked.

How to secure data

There are several principles to adhere by in order to minimise risk of data breaches.

1. Do not roll your own

It is difficult to secure your data against all possible attacks. Most naive implementations will have exploitable flaws, so it is much more sensible to rely on existing, battle-tested security frameworks. These stand a much greater chance of having had most of their vulnerabilities fixed.

2. Principle of least privilege

The principle of least privilege states that a `subject should be given only those privileges needed for it to perform its task`. For example, a reader of a news website should not have editor privileges to the articles, or a banking customer should only be able to see details of their own account.

Most database software will allow you to specify fine-grained access to resources using either access *access control lists* (ACLs) or *role-based access control* (RBAC). These will allow or deny users the right to access or modify specific database objects. Depending on the database provider, it is possible to set up access rules on a per-row basis.

In order to comply with the principle of least privilege, the default setting should be to deny users all privileges initially and grant access on a case-by-case basis.

3. Encryption

If a data breach does occur, its impact can be greatly mitigated by ensuring data is sufficiently encrypted, and hence worthless to the attacker. Encryption of data falls into two broad categories: encryption at rest and encryption in transit.

3.1 Encryption at rest

When data is at rest*, it is not actively moving between devices or networks. This usually refers to data stored on disk.*

Encryption of data where it is stored ensures that if an attacker can somehow access our data, they would not be able to read it without a decryption key. Hence, in order for encryption at rest to be effective, the encryption key for the data must be stored in a separate location.

There are different levels of encryption, including:

- Encryption of specific columns in a database
- Encryption of entire databases
- Encryption of the entire disk drive where data is stored

Most cloud-based database providers offer full database encryption-at-rest as standard.

3.2 Encryption in transit

Whenever data moves between systems, it should also be encrypted to minimise the risk of man-in-the-middle attacks. It is good practice to encrypt all connections as a principle; it is particularly important when connecting to cloud database providers over public internet.

Most cloud providers forbid unencrypted connections to their database products.

TLS (Transport Layer Security) is one of the most common protocols for encrypting data in transit. We shall cover it in more detail in the next module.

4. Connectivity restriction

We can apply a similar concept to the principle of least privilege to networks: A service should only allow connectivity to services to and/or from services which explicitly require it. For example, a database should only allow connectivity from the server running the backend of a website and not the public internet.

This can be achieved by:

1. Applying sensible firewall rules to servers running sensitive systems such as databases, preventing access from unknown locations.
2. Not exposing sensitive systems to the public internet at all, and only allowing access from private networks.

Usually a well-secured web application will use both methods:

1. The database server is only accessible via an internal network
2. The database server has firewall rules applied to prevent access from all internal machines other than the website backend server and any servers required for maintenance purposes.

5. Physical security

It is important to consider how secure is the physical hardware running not only database software itself, but also where the backups are stored. After all, the greatest network firewalls in the world cannot stop an intruder walking off with sensitive data on a flash drive.

Ensure access to server rooms is restricted to those who *need* access; ensure staff are away of strangers tailgating into secure areas; and avoiding storing sensitive data on easily removable media.

6. Just Don't Store It

The most secure data is the data you don't store. It is impossible to compromise data that you never had in the first place. Think carefully about whether sensitive data is actually necessary before storing it in your system.

Password security

Secure storage of user passwords is crucial:

1. In the event of a breach which leaks unsecured passwords, it will be possible for malicious actors to log into your system as anyone whose details have leaked.
2. People have a habit of reusing the same password for multiple logins. Hence, if you store user passwords in plain text and you have a data breach, those passwords can be used not only to log into your system but potentially dozens of others.

Password storage

You should *never* store user passwords in plain text. The industry standard solution is to use a one-way *hash function* to encrypt stored passwords. Such hash functions are designed to be irreversible, so even if you do leak your encrypted passwords, they would be useless.

In order to verify as password, the logic would check equality of the stored, hashed password against a hashed version of the given password:

```
hash(givenPassword) == storedPasswordHash
```

Examples of hashing algorithms include the `MDx` family, `SHA1`, `SHA256` and `bcrypt`. The latter has an advantage of being resistant to *brute-force* attacks as its execution speed can be reduced by adjusting its configuration.

Not all hashing algorithms are equally secure. Some, like `MD5` and `SHA1`, are vulnerable to attack and should ***not*** *be used for encryption.*

Hash Functions

Hashing is a very versatile concept in Computer Science. It involves turning an input string into a **hash**. The exact process is beyond the scope of this course but we rely on hashes to be:

- **deterministic:** the same input should produce the same hash
- **unique**: difference inputs should produce different hashes
- **easy to calculate** but **difficult to reverse**

Example `md5` hashes:

Input	Hash
empty string	`d41d8cd98f00b204e9800998ecf8427e`
“Module 09”	`bba496e39501de03adaab7b4f9681f2d`
“Module 10”	`0629c7158ec87f423ec25f02e23170c1`
Script of DreamWorks’ Bee Movie	`7e7fa0a8f39f4a580c211419a61e7134`

Salting

An additional threat to breaking password encryption is a *rainbow table*, which is a pre-computed list of hashes of common passwords for a given hashing algorithm. To defend against this, we can use a *salt* - an extra chunk of data appended to the password before hashing. The salt is stored alongside the encrypted password so it can be applied to the hash function during a password verification attempt. Our verification function above becomes

```
hash(givenPassword, storedSalt) ==
storedPasswordAndSaltHash
```

The salt is useless on its own, which makes it safe to store unencrypted alongside the encrypted password.

Third-party authentication

A safer option than storing user passwords as part of your data (even encrypted) is to delegate your authentication process to a trusted third party. This involves forwarding any login requests to a third party authentication provider, along with some secret information that authenticates your application with the provider. The authentication provider then returns the success status, along with any other data that the end user has allowed them to provide, such as a full name or profile picture.

The industry standard for this sort of mechanism is called `OAuth`.

There are many authentication providers, including Facebook, Google, Microsoft and Github.

Peppers

As well as salts, some implementations use a **pepper** - an additional constant secret input, known by the application but not the database. It is sometimes referred to as a **secret salt**.

Unlike the salt, the pepper is the same for each stored password. The idea is that the pepper provides a component that is stored outside the database, making it more difficult for an attacker who only has access to the database to reverse-engineer the salts.

However, the use of peppers is somewhat controversial, mainly because the most-used hashing algorithms don't include support for them. This means that adding peppers onto a hash usually requires you to 'roll your own' security to some extent.

Sensitive Data

As well as passwords, other types of data require special handling.

Any payment details, such as credit card numbers, must be secured using methods compliant with the **PCI DSS** industry standard. While PCI is not a legal regulation in itself, non-compliance ay be a violation of data protection rules and may also be sanctioned by credit card providers, resulting in an inability to take payments.

The **Data Protection Act** (UK) and **HIPAA** (US) outline legal requirements for storage of sensitive clinical data. The **General Data Protection Regulation** or **GDPR** (EU + UK) imposes restrictions on how personal data can be processed.

If your systems use sensitive data then care should be taken to ensure it doesn't leak through to test or development environments. Ensuring that lower environments are representative of the production environment whilst maintaining data security is a challenge. One solution is to anonymise or redact live data before using it to populate a test environment. An alternative is creating realistic looking fake datasets, although this can be a challenge.

Data Breaches

It is important to disclose security breaches promptly, so that affected individuals can take action to protect themselves. For example, any users whose sign-in details have been leaked should be warned to change their login immediately.

In some jurisdictions this is a legal requirement. For example, GDPR requires that any breach of personal data which is likely to lead to a risk of rights and freedoms of individuals must be disclosed within 72 hours.

Chapter 4

Data Ethics

Overview

Everyone in our field has responsibility for the stewardship of data. The nature of this data can vary hugely by industry and can include highly **personal and sensitive data**: financial transactions, medical records, and our private correspondence.

The *ethical* stewardship of this data is one of the most important aspects of our work, but can also be one of the most neglected. Before we discuss the mechanics of storing data and processing data, we should examine the ethical implications of doing so.

The ethical use of data is a big and rapidly developing field. So long as technology evolves so will the potential uses and abuses of that data. We will explore some real-world examples below.

Privacy and Surveillance

When users provide you with data they have a certain expectation of privacy. If your application violates this expectation it could be considered an invasion of privacy.

Unintentional invasion of privacy

Until 2019 Apple allowed contractors to review audio recordings from Siri for grading purposes. According to a whistleblower:

"*There have been countless instances of recordings featuring private discussions between doctors and patients, business deals, seemingly criminal dealings, sexual encounters and so on.*"

While the motivation for this practice was to improve the quality of service for users it was an invasion of privacy. Apple has changed its practices so users have to **opt-in** to sharing their recordings with Apple.

Amazon, Google and Microsoft were engaging in similar practices and also took some corrective measures.

Source: https://www.theguardian.com/technology/2019/aug/29/apple-apologises-listen-siri-recordings?

Intentional abuse of data

Internal users with access to data can also intentionally misuse it. In 2018 a Facebook

employee was fired for allegedly misusing data that he had privileged access to in order to stalk women online.

Source: https://www.theguardian.com/technology/2018/may/02/facebook-engineer-fired-alleged-stalker-tinder

Your organisation should have controls to ensure that data cannot be misused by employees. This can include restricting access to those who **need to know** and **auditing** when data is accessed and by whom.

Targeting

One major use of collected data is to individually tailor user experiences and provide targeted advertising. This use can be desirable for users: they receive a more personal service and see advertisements relevant to their interests.

Targeting can also fail in a variety of ways, from the frustrating (seeing adverts for a product after you've bought it); to the amusing (an engaged gay man being shown adverts for wedding dresses); to the tragic (Facebook continuing to show baby adverts after the loss of a child).

Source: https://www.tommys.org/about-us/charity-news/how-stop-pregnancy-ads-following-you-after-loss

Providing users the ability to **opt-out** of targeted advertising can help mitigate some of the downsides of the practice, by providing users with greater control over their data.

Automation, AI & Bias

Another increasingly common use of data is automated decision making and **artificial intelligence** (AI). Automation and AI is a very broad topic beyond the scope of this course. For our purposes we will use **automation** to cover any sort of decision making process without a human in the loop. Broadly, all these processes make decisions based on some sort of input or training data.

It is important to be aware of biases within data when considering applications of that data. The data itself can be biased, or the process by which the data is gathered. Any bias in the input data will be reflected in the outputs of your automated process.

> There are also legal restrictions on using certain types of data as the basis for automated decision making. We shall cover this in more detail in another chapter.

Automation applications typically lack transparency on why they make the decisions they do - this can make biases difficult to pinpoint. This is not because the process is designed to be biased; rather automation has

a tendency to amplify existing upstream biases.

There are many well documented instances of bias in automation. One common example is a tendency for facial recognition algorithms to prefer lighter faces over darker ones. This issue was highlighted in Twitter's image cropping algorithm in 2020.

Source: https://www.theverge.com/2020/9/20/21447998/twitter-photo-preview-white-black-faces

Bias in automation can also impact systems with my direct real world consequences. In 2016 the United States' Correctional Offender Management Profiling for Alternative Sanctions or **COMPAS** tool was found to incorrectly identify black defendants as having a higher rate of recidivism; while also incorrectly regarding white defendants as lower risk.

Source:

1. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing
2. https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm

Eliminating bias from automation is not an easy task. You must start by being aware that bias exists and acknowledging that both you and process will have biases.

Further reading: https://www.forbes.com/sites/joemckendrick/2019/06/28/artificial-intelligence-may-be-biased-but-it-can-also-help-bust-bias

Government intervention

Government has a significant role to play in the field of data ethics. Data protection legislation provides a minimal baseline for the ethical use of data and codifies the rights of consumers. We will cover data protection legislation in another chapter.

However, governments also collect a huge amount of data and have the ability to impose legal obligations on organisations to supply data about their users. Whether these obligations are necessary security requirements or unjustified invasions of privacy is a political issue beyond the scope of this course.

Surveillance

Governments may impose obligations on your organisation to compel you to share gathered data with government bodies. The form of government surveillance can vary. In some cases the cooperation of corporations is required, for example the US Government's PRISM program was conducted with the collaboration of Microsoft and other big tech companies.

Source: https://www.theguardian.com/world/2013/jul/11/microsoft-nsa-collaboration-user-data

Data Collection

Governments regularly collect data about citizens to help with a huge variety of issues. Information from the electoral register and the census help draw constituency boundaries; determine how to allocate funding; inform local planning policy; and much, much more.

However, collecting this data can be dangerous; governments can also misuse data for targeted persecution. Infamously the United Statues used data from the 1940 census to identify and intern Japanese-Americans during WWII. At the same time Nazi Germany used population data from across Europe to identify Jews, Roma and other ethnic minorities to be killed.

Source: https://www.abc.net.au/radionational/programs/rearvision/the-dark-side-of-census-collections/7860908

More recently data mining company Palantir has faced criticism for its work with the US ICE to identify and deport undocumented migrants to the USA.

Source: https://www.washingtonpost.com/business/2019/08/22/war-inside-palantir-data-mining-firms-ties-ice-under-attack-by-employees

Transparency & Trust

The best defence of ethical data stewardship is transparency. If your users understand their data is to be used and who their data will be used by they will be able to provide **informed consent**. Users will only provide that consent if they **trust** your organisation and its processes. If users do not trust you then they will not work with you or, if they are compelled to, will provide inaccurate or incomplete data.

There have been many recent scandals around deliberately secretive and inappropriate uses of data:

1. Cambridge Analytica created a Facebook app called "This is your digital life" that quietly collected user data, allowing them to micro-target ads in the 2016 US election. https://www.newscientist.com/article/2166435-how-facebook-let-a-friend-pass-my-data-to-cambridge-analytica/
2. AT&T, Verizon, Sprint & T-Mobile selling mobile phone users' location data to advertisers without their consent. https://www.govtech.com/network/Wireless-Carriers-Face-200M-Fine-for-Selling-Location-Data.html

While these examples are egregiously bad we cannot comprehensively define the ethicalness of all potential use cases. Not only are there existing areas of moral ambiguity, but technological changes will ensure there are always new questions to be answered.

Government legislation and industry guidelines will provide some guidance as to a minimum standard, however these will almost always be *reactive*. You will have to make your own decisions about how to ethically use data.

As a rule of thumb, we suggest the following: only collect data that is *necessary* for your stated purpose; only keep is for as long as you *need to* for that purpose; and only use it for that stated purpose.

Chapter 5

Data Protection Legislation

Many countries have introduced data protection frameworks to ensure organisations are managing data securely and ethically. These are legal obligations and should be regarded as the minimum possible standard for ethical use of data.

The EU's **General Data Protection Regulation**, commonly known as **GDPR**, is the best known of these, but many other regulations worldwide are similar to or modelled on the GDPR. The **Data Protection Act 2018** is the UK's implementation of the GDPR.

GDPR applies to data belonging to EU citizens regardless of where the processing happens. Even if UK law changes, if your application deals with EU citizens then GDPR still applies.

In this chapter we will cover the basics of GDPR as this is the most widely applicable piece of legislation. However, other legislation may be applicable to your organisation. For example, the **Freedom of Information Act 2000** applies to public bodies, but *not* to private enterprise.

None of this is legal advice. Consult the ICO or a lawyer for more authoritative information.

What does the GDPR do?

GDPR sets out principles governing how personal data should be processed. For our purposes any data handled or stored by your application is considered processed.

These principles are:

- Lawfulness, fairness and transparency
- Purpose limitation
- Data minimisation
- Accuracy
- Storage limitation
- Integrity and confidentiality
- Accountability

Before we explore the implications we should first be clear what we mean by personal data.

More reading: ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/principles

What is personal data

GDPR defines personal data as "*any information relating to an identified or identifiable natural person (‘data subject’)*". The information doesn't have to be accurate, it just needs to be related to the person.

The key part of this is "*identified or identifiable*". An **identified** person is where you explicit say who the data relates to, e.g. information relating to *Sherlock Holmes* relates to an identified person. An **identifiable** person is one can be identified *directly* or *indirectly* from the data. This could mean by identified through an *identification number* or from other factors.

This means there is some degree of subjectivity. For example, "*Consulting Detectives living on Baker Street*" is identifiable whereas "Detectives living in London" is not.

Can you identify the individual the data pertains to based on the data? If so then GDPR applies.

For a more detailed summary see: gdpr.eu/eu-gdpr-personal-data

Some quick terminology

Any person who has personal data collected about them is referred to as a **data subject**. We will use this term in the rest of this chapter.

A **data controller** is a person or organisation that has decided to collect personal data about a data subject. A **data processor** is a person or organisation that processes data on behalf of a data controller. We will refer to data controllers in this chapter, although in *most* cases the law is equally applicable to controllers and processors.

Basis for processing personal data

Under GDPR you are required to have a **lawful basis** (or *lawful reason*) to process personal data. If you do not have a lawful

Personal Data

'Personal data' means any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person.

GDPR Article 4

basis for processing data then you should not be doing so.

The clearest lawful basis for processing data is **consent**. This is where the data subject has actively **opted-in** to permit their data to be processed for a *specific purpose*.

When GDPR was introduced in May 2018 there was a rush by organisations to get their existing contacts to consent to future direct contact.

Not all bases are appropriate for all use cases. Certain types of processing and certain types of personal data have more stringent rules. Your processing needs to be *necessary* for a specific purpose; if you can reasonably achieve the same goal without the processing then you won't have a lawful basis.

We shall summarise the six lawful bases for processing below.

Explicit consent

Where the data subject has actively **opted-in** to processing for a specific purpose. Data controllers are prohibited from refusing service to users who do not consent, unless the processing is necessary for the service.

For example, a customer consents to an online retailer using their billing address to process payments. However, the retailer cannot refuse to sell a product to a customer who doesn't consent to direct marketing to that billing address.

Contract

Applies when you have a contract with the data subject, or when data processing is a prerequisite to enter into a contract.

For example, an insurance company can process data about a potential customers car in order to supply them with a quotation.

Legal obligation

Applies when you need to process the data in order to comply with the law. Restaurants keeping records of diners during the COVID-19 pandemic is an example of a legal obligation.

Vital interests

This generally only applies to life and death situations, for the purposes of saving life. It cannot be used if the data subject is capable of giving consent. This is the legal basis for divulging a patient's medical history when admitted to an Accident & Emergency department with life-threatening injuries.

Public task

Applies mainly to public authorities or some private companies carrying out tasks in the public interest as laid down in law.

Legitimate interests

Legitimate interests is perhaps the most flexible of the bases. It allows you to process personal data for a legitimate interest, provided that:

- You are pursuing a legitimate interest;
- The processing is necessary for that interest; and
- The impact on the data subject's rights is not disproportionate.

If you are using personal data in a way that the data subject would reasonably expect and there is minimal impact on their privacy then this may be appropriate.

Using legitimate interests as a lawful basis for processing data requires you take on extra responsibility for protecting the rights of the data subjects. Use this with caution.

More reading: ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/lawful-basis-for-processing

Special category personal data

Certain personal data is considered particularly sensitive and is classified as **special category** personal data. The rules about processing special category data are more stringent rules.

Special category data includes data about (or which might indicate) any of the following:

- Racial or ethnic origin
- Sexual orientation
- Political opinions
- Membership of a trade union
- Religious or philosophical beliefs
- Genetic data
- Data used for biometric identification
- Health and medical conditions
- Sexual activity and sex life

In order to process special category data, one of the stronger 'Article 9' conditions must also be met. Some of these overlap with the above lawful bases, but 'legitimate interest' in particular is not sufficient.

More reading: ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/special-category-data/what-are-the-rules-on-special-category-data

Criminal convictions

Similarly “personal data relating to criminal convictions and offences or related security measures” is subject to additional scrutiny. These can only be under the control of official authorities, or where it is explicitly authorised by law.

In the UK, the Data Protection Act 2018 defines specific conditions where this is permitted.

More reading: ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/lawful-basis-for-processing/criminal-offence-data

Your rights

The GDPR grants data subjects a number of rights, which data controllers need to uphold.

The right **to be informed** requires data controllers to disclose to data subjects the personal data they are collecting; the purposes for which they will process it, and which of the above bases it will processed under. Data controllers also need to disclose any third parties the data will be shared with and how long it will be stored. Data subjects also have the right **to access** their personal data as well as information on how it is processed. This is commonly referred to as a **subject access request** or **SAR**.

Data subjects also have rights to modify their data, under certain circumstances. The right **to rectify** requires records of personal data to be corrected; the right **to erasure**, commonly known as the **right to be forgotten**, allows for personal data to be deleted. In both cases these rights are not absolute and may be refused in certain circumstances.

The right **to object** to the processing of personal data. If the data is used for direct marketing purposes, processing must be stopped. Otherwise, for certain bases for processing (including lawful interest), processing may only continue if there is a compelling reason. Similarly data subjects have the right **to restrict processing** of their personal data *under limited circumstances*.

Finally, there is the right **to 'data portability'** - the ability to transfer their data in a machine-readable format to the subject or to another data controller.

More reading: ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/individual-rights

Automated decision making

There are additional requirements around **automated decision making** and **profiling** of data subjects. This is only permitted with

explicit consent, under the *contract basis*, or with *legal authorisation*.

GDPR also requires that you provide simple ways for subjects to **request intervention** on a decision and regularly check that the system is working as intended.

Data breaches

A personal data breach means a breach of security leading to the accidental or unlawful destruction, loss, alteration, unauthorised disclosure of, or access to, personal data.

Source: ico.org.uk

It is required that any breach of personal data likely to risk individuals rights and freedoms must be disclosed within 72 hours. If there is likely to be a *high risk* to the data subjects rights then they must be informed directly "without undue delay".

Source: ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/personal-data-breaches

Penalties

While getting data protection right is a legal obligation and ethically good, getting it wrong can be exceedingly expensive. There can be significant financial consequences for organisations that breach the conditions of GDPR. This can be up to 20 million euro or 4% of *global turnover*.

In 2018 Google was fined 50 million euros for insufficient transparency around targeted advertising.

Information Commissioner's Office (ICO)

In the UK, the body responsible for investigation enforcing these rules is the **Information Commissioner's Office** or **ICO**. The ICO is an independent regulatory body which reports to parliament.

It is also responsible for enforcing other UK data regulations, most notably the **Freedom of Information Act 2000**, which governs the right of the British public to request information held by public authorities.

The ICO website, ico.org.uk, is a good resource for further reading about your obligations in the UK.

Further reading on GDPR

GDPR: gdpr-info.eu

ICO: ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr

Additional Material

Relational Database Schemas

Relational databases store their data in **tables** according to a pre-defined schema. In this chapter we will cover some of the key concepts in relational database schemas.

Schema structures

We shall continue with the `employees` example from Chapter 1:

UserID	Name	HireDate	LeavingDate	IsContractor
1	John Smith	2018-02-23	2020-04-12	`true`
2	Jane Doe	2020-03-04		`false`
3	John Skeet	2010-03-04		`false`

Columns

A column in the schema usually contains more information than just the name of the column. It can also state:

- The type of data stored in the column, i.e. whether the data will always be a piece of text, an integer, a true/false or even raw binary data.
- Whether the column is *nullable* or not. If a column is nullable, it is possible for the data to be *null*, i.e. not have a value. Conversely, if a column is non-nullable, then it **must** contain a value - the database will not allow empty values to be written to this column.
- Any constraints imposed on the data. If a user or software attempts to write data into a table which breaks a constraint rule, then the database will not allow this.

We can extend the example above to include some of this information:

- `UserID` is non-nullable, has type `Integer` and has a *uniqueness* constraint applied.
- `Name` is non-nullable and has type `Text`. We do not apply a uniqueness constraint as two employees may have the same name.
- `HireDate` is non-nullable and has type `Date`.
- `Leaving Date` is *nullable* and has type `Date`. We mark this column as nullable and can represent employees still employed by our company with an null value.
- `IsContractor` is non-nullable and has type `Boolean`. Boolean columns can only have values `true` or `false`.

Indices

An *index* is a data structure which improves the speed with which information can be found in one or more columns in a table. Without an index a database would need to query every row in a table until it found rows matching the search criteria, while an index would mean a more efficient algorithm could be used to look up the data.

To make searching even more efficient, certain index types not only create a supporting data structure to aid searching, but also reorganise data for the entire table to be stored on disk in a particular order. In a Microsoft SQL Database, for example, these are called *clustered* indices.

Note that creation of indices requires extra storage space and slows down writing to the database - any indices affected by write operations would need to be updated as well. Hence, indices should be used only when read performance bottlenecks are identified.

In our example, it is feasible that the HR department would want to perform frequent searches of data by employee name, so we should add an index to the `Name` column.

Primary Keys

It is good practice for a table to have a column declared as the *primary key*. This marks the column as the ID of the entire row. Usually the value of this field is auto-generated by the database itself, but it can be left up to the user to populate.

Marking a column as the primary key automatically applies certain conditions:

- The column must be non-nullable - and ID for a row must exist.
- The column has a unique constraint applied, as a given ID must uniquely identify a single row.

- In some databases, an index is automatically applied to the column, as searching by ID is a very common operation.

In our example, we should mark the `UserID` column as the primary key.

It is possible for multiple columns to be marked as a primary key. This defines a composite *key, and the conditions above are applied to all columns taken together.*

Foreign Keys

Foreign keys are crucial in defining links between tables in a relational database. They are constraints applied to a column (or columns) in a table, which reference the primary key column(s) of another table.

A foreign key constraint enforces that the value of an item in the column exists in the referenced column. This helps ensures integrity in the database by preventing references to non-existent records.

For example, we add a `Department` column to `employees` table and assume we have a `departments` table:

`employees`

UserID	Name	HireDate	LeavingDate	IsContractor	Department
1	John Smith	2018-02-23	2020-04-12	`true`	2
2	Jane Doe	2020-03-04		`false`	1
3	John Skeet	2010-03-04		`false`	2

`departments`

DeptID	Name
1	HR
2	IT

In this case we can define a foreign key from `employees.Department` to `departments.DeptId`. This would prevent any values being inserted into the `Department` column that do not appear in the `DeptID` column. (Depending on your database settings it could also prevent values being removed from `DeptID` if they were referenced in the `Department` column.)

The syntax `[table name].[column name]` *is useful if we want to be explicit about which table a given column is in, and is required when multiple tables have a column with the same name.*

Migrations

Over time our application's requirements will change. When this happens we may need to make alterations to our database's schema. This can be done by running SQL scripts against the database, introducing new tables or updating existing tables to support new functionality.

It is important to ensure database changes are run consistently across different environments, so that test environments are representative of live environments.

Using a database migration framework, the SQL to update a database to match a new version of the application code can automatically be generated and run as part of your CI/CD process.

This allows you to store and version the instructions for generating a database along with the application code that depends on it.

Additional Reading

SQL

SQL is made up of statements, which when executed against a database, will perform an operation of a single type on the database.

We shall continue using the `employees` and `departments` tables from the previous chapter:

`employees`

UserID	Name	HireDate	LeavingDate	IsContractor	Department
1	John Smith	2018-02-23	2020-04-12	`true`	2
2	Jane Doe	2020-03-04		`false`	1
3	John Skeet	2010-03-04		`false`	2

`departments`

DeptID	Name
1	HR
2	IT

Reading data

The most common use case for SQL when starting off is for reading data. All such operations are performed using the SELECT operator. The syntax for this clause looks like:

```
SELECT
[column 1], [column 2], ..., [column N]
FROM [data source]
[Optional JOIN clauses]
[Optional WHERE clause, or filter]
[Optional GROUP BY aggregation clause]
```

For example, to find all employee names with a hiring date on or later than 1st Jan 2019, the query would be:

```
SELECT
UserID, Name
FROM employees
WHERE "Hire Date" => "01-01-2019"
```

which would return the result:

UserID	Name
2	Jane Doe

In the example:

- The column list contains a single column, `Name`
- The data source is the `employees` table. This can be more complex, e.g. the result of another query (sub-query).
- The `WHERE` clause filters the data by the `Hire Date` column using the *greater or equals* operator with the 1st January 2015 as the filter value.
- There are no `JOIN` or `GROUP BY` clauses.

  The `WHERE` clause can grow to be very complex and make use of a variety of comparison operators. This introduction is too brief to go into detail of all of them!

  It is also possible to select *every* column from a table by writing `*` instead of a list of columns.

Joins

In order to select data from more than one table at the same time, a `JOIN` clause is required. This allows rows from one table to be 'joined' to rows from another table using specific columns from both tables. The most common use case is to join via foreign-keyed columns.

In our example, suppose we want to return the name of the department each employee is in as well as employee data. In this case, we would join the `departments` table onto the `employees` table using the columns `departments.DeptId` and `employees.Department`:

```
SELECT
employees.Name as EmployeeName, departments.Name as DepartmentName
FROM employees
JOIN departments ON employees.Department = department.DeptId
```

This would return the results:

UserName	**DepartmentName**
John Smith	IT
Jane Doe	HR
John Skeet	IT

In the example we are renaming the output columns using the alias *syntax, or the* `as` *keyword. This is useful for generated columns or when we have ambiguous source column names (as just "Name" would otherwise be above).*

Aggregation

Aggregation functions include `COUNT()`, `MAX()`, `SUM()` and `AVG()` which condense data from multiple rows into a single row.

For example, if we want to count how many employees whose names start with 'John' we can run:

```
SELECT
COUNT(*) as JohnCount
FROM employees
WHERE Name LIKE 'John%'
```

This returns:

JohnCount
2

The `LIKE` *operator in a* `WHERE` *clause uses the* `%` *symbol as a* wildcard*, which means that it can have any value. In our case,* `'John%'` *represents a string which starts with* `'John'` *and then has any number of other characters.*

Group By

An optional `GROUP BY` clause can be used to list one or more columns by which to group the results. For example, if we want to get the earliest date someone was hired in any department, we can use the following:

```
SELECT
Department, MIN(HireDate) As EarliestDate
FROM employees
GROUP BY Department
```

This returns:

Department	EarliestDate
1	2020-03-04
2	2010-03-04

Writing Data

Writing data usually involves simpler queries than reading. It can be broken down into two categories:

1. Creating new data, or adding new rows to a table
2. Updating existing data, or changing values of items in existing rows

Creating new rows

In order to add new rows to a table, we use the `INSERT INTO` statement. It has the following syntax:

```
INSERT INTO [table name] ([column 1], [column 2], ... , [column N])
VALUES ([value 1], [value 2], ..., [value N])
```

For example, if we want to add a new entry to the `employees` table we can write:

```
INSERT INTO employees (UserID, Name, HireDate, LeavingDate, IsContractor,
Department)
VALUES (4, 'The Doctor', '2003-11-10', '2003-12-15', true, 2)
```

If we then run `SELECT * FROM employees`, the query returns:

UserID	Name	HireDate	LeavingDate	IsContractor	Department
1	John Smith	2018-02-23	2020-04-12	`true`	2
2	Jane Doe	2020-03-04		`false`	1
3	John Skeet	2010-03-04		`false`	2
4	The Doctor	2003-11-10	2003-12-15	`true`	1

If we want to write a value for every row of the table, in the order the columns appear in the schema, we do not have to explicitly write out the column names. For example, the above statement can be re-written as:

```
INSERT INTO employees
VALUES (4, 'The Doctor', '2003-11-10', '2003-12-15', true, 2)
```

Updating existing rows

In order to edit existing rows, we use the `UPDATE` statement. It has the following syntax:

```
UPDATE [table name]
SET [column 1] = [value 1], [column 2] = [value 2], ..., [column
N] = [value N]
WHERE [condition]
```

For example, if Jane Doe decided to leave the company we would want to set her leaving date as follows:

```
UPDATE employees
SET LeavingDate = '2021-03-12'
WHERE UserID = 2
```

We recommend you always *add a* `WHERE` *clause to an* `UPDATE` *statement. Otherwise, the change will affect* every *item in the table, which is rarely what you intend.*

Deleting data

In order to remove rows from a table, we use the `DELETE FROM` statement. It has the following syntax:

```
DELETE FROM [table name]
WHERE [condition]
```

In our case, it turned out that The Doctor was added as a prank and we want to remove this person from our employees list. We would write:

```
DELETE FROM employees
WHERE UserID = 4
```

If we then run `SELECT * FROM employees`, the query returns

UserID	Name	HireDate	LeavingDate	IsContractor	Department
1	John Smith	2018-02-23	2020-04-12	`true`	2
2	Jane Doe	2020-03-04		`false`	1
3	John Skeet	2010-03-04		`false`	2

Warning: *running* `DELETE FROM employees` *will remove **all** the data in the table. As when updating, you should* always *add a* `WHERE` *clause, or risk deletion of **all data** in a table.* **This can be disastrous in a production environment!**