**JDBC**

JDBC is an API for Java, allowing to programatically connect to a multitude of different databases, all from within your Java application. As opposed to an ORM like EFCore for C# or Hibernate for Java, JDBC simply sets up a connection which allows for executing straight-up SQL queries on the database.

**Inversion-of-control**

Inversion-of-control is also known as Dependency injection, is simply the act of passing pre-constructed objects into a class, instead of the class controlling the instantiation of its dependencies.

**Spring Boot**

`https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth` Spring is a framework for creating applications in Java. Spring Boot is an extension of this, which follows the convention-over-configuration paradigm, attempting to reduce the number of decisions a developer needs to make.

**Transaction Abstraction Framework**

Transaction management in this case simply means "How does spring start, commit and rollback JDBC transactions"

**The 'Transactional' annotation**

Aspect-oriented-programming (splitting the implementation of some 'aspects' of the code across multiple different classes). Any public method annotated with the '@Transactional' tag will execute inside a database transaction. This relies on you having setup the transaction manager in Spring's 'Configuration' settings.

```
1  @Transactional
2  public Long registerUser(User user) {
3    // execute some SQL that e.g.
4    // inserts the user into the db and retrieves the autogenerated id
5    // userDao.save(user);
6  return id;
7      }
```

Can be translated roughly into

```
1  public class UserService {
2
3      public Long registerUser(User user) {
4          Connection connection = dataSource.getConnection(); // (1)
5          try (connection) {
6              connection.setAutoCommit(false); // (1)
7
8              // execute some SQL that e.g.
9              // inserts the user into the db and retrieves the autogenerated id
10             // userDao.save(user); <(2)
11
```

```
12              connection.commit(); // (1)
13          } catch (SQLException e) {
14              connection.rollback(); // (1)
15          }
16      }
17  }
```

This means that Spring (and indeed Spring Boot) automatically inserts the code for opening and closing of the connection, comiting the transaction, as well as handling rollbacks. É Bene.

But, Spring cannot modify your classes directly, so how does it work? Well, at its core, Spring is an Inversion-of-control container, meaning it is in charge of doing all the dependency injections. Therefore, Spring can "substitute" all injections of your class UserService with a proxy that it itself has generated. This proxy then handles the connections, while still calling your created UserService class for the logic.

The upside of this is obvious, as the programmer does not have to handle setting up and closing databases. However, it also has as result that the code you wrote is not necessarily the code that is executed. Take for instance the following class:

```
1   @Service
2   public class UserService {
3
4       @Transactional
5       public void invoice() {
6           createPdf();
7           // send invoice as email, etc.
8       }
9
10      @Transactional(propagation = Propagation.REQUIRES_NEW)
11      public void createPdf() {
12          // ...
13      }
14  }
```

In this case, the user clearly intended for 2 transactions to be opened. As the proxy UserService calls UserService.invoice(), it does so from within new transaction. Control is then passed to the 'real' UserService class, which doesn't have the power to open new connections. Therefore, the createPDF method does not open a new connection. To achieve what the user probably wanted, we'd have to rewrite the code of createPdf into its own service, which would then itself be proxied, allowing for opening a new transaction:

```
1   @Service
2   public class UserService {
```

```
3
4       @Autowired
5       private InvoiceService invoiceService;
6
7       @Transactional
8       public void invoice() {
9           invoiceService.createPdf();
10          // send invoice as email, etc.
11      }
12  }
13
14  @Service
15  public class InvoiceService {
16
17      @Transactional
18      public void createPdf() {
19          // ...
20      }
21  }
```

**DAO**

In Java Spring, the DAO pattern is often used when working with databases. A DAO object basically defines the operation that can be done to a class, holding all of the SQL queries and calls to the entity manager. This makes the java code more resilient to changes in the DB provider, as well as keeping up seperation of concerns, having all SQL queries in a single class. Modifying the DAO will by default modify the database, but if the DAO is modified inside a transaction, the changes will only be written once the .commit() is called on the transaction.

**DotNet**

The DotNet framework uses EFCore to manipulate databases. EF-Core allows for creating a "Context" object to represent a database, with a collection of type 'DBSet¡Person¿' to to represent the table "People" in the database. Manipulating the database then simply means manipulating the Context object programatically. EF-Core then allows for calling 'Context.SaveChanges()', which opens a transaction, and performs all the changes that were done before calling 'SaveChanges()'. It is possible to use rollbacks and use multiple SaveChanges in a single transaction by calling Context.Database.BeginTransaction().

**Database-per-service**

The database-per-service pattern in microservice architecture dictates that each service that manages persistent data must have its own database, and data from within that database is only accessible to other services through the API of the service owning the database. This pattern makes it easier to scale and deploy these services independently, but makes it tough to have consistency in the data, since the 'UserService' service must now

delegate the work of updating the 'Orders' table to the 'OrderService' service when a user places an order.

**2PC**

The 2 Phase commit is a protocol used to distributed transactions, meaning coordinating transactions across multiple different services. The protocol has a coordinator node, and some worker nodes. When a worker wants to write a change to its database, it will need to send a *commit-request* message to the coordinator node, then apply the change up to the point right before comitting the change. The coordinator then sends out a "commit-request" message to all other related nodes, which then also apply the changes right up until before the commit step. Each worker then sends an *accept* message if the changes can be committed, or a *reject* message otherwise. If the coordinator receives only *accept* messages, it sends out a *commit-accept* message to all participating worker nodes, which all commit their changes. Otherwise, the coordinator sends out a *rollback* message, and all workers rollback their changes. The 2PC protocol is a blocking protocol, meaning workers cannot do any other changes while waiting for a **commit-accept** message from the coordinator. For this reason, it assumes a *fail-restart* failure model, in which processes which fail will eventually restart into the same state, and therefore eventually send a correct *accept/reject* message.

**3PC**

This protocol is an extension of the 2PC protocol, which adds an extra step in the middle. The coordinator still sends out a commit-query, and collects the responses from the workers. If all workers send *accept* messages, the coordinator first sends out a *prepare-to-commit* message, informing the workers of the unanimous decision. When a worker recieves the *prepare-to-commit* message, it knows it can safely commit, but will not do so until some timeout has run out, or until it gets a *commit* message from the coordinator.

When all workers have acknowledged the *prepare-to-commit* the message, the coordinator can send out a *commit* message, causing all workers to commit.

If the coordinator fails, and even a single worker was informed of the unanimous decision, then the coordinator can simply be replaced by another process, which will re-send the *prepare-to-commit* messages.

**Saga Pattern**

The saga pattern is used for the same reson as the 2PC pattern, that is, to coordinate transactions across a distributed database. It works by having a service initiate a *saga* when it wants to make a change to its local database. Service creates *PENDING* update to its own database, then sends out a HTTP (or some other channel) to other services to also update their databases. If these are updated successfully, a reply is sent to the saga initiator, which will also commit its changes. If they fail, the initiator will roll back its changes.

This pattern seems a lot like 2PC, expect its missing the central coordinator node, having the service which wants to update its database act as the coordinator.

The saga pattern has the issue that all other services participating in the saga must both commit, as well as send a message that they have committed successfully. If this message is lost, the databases of the two services will enter an inconsistent state, as the initiator will rollback its changes since it didn't hear from other services in the saga. For this reason, either the *Transactional Outbox* or *Event Sourcing* pattern needs to be implemented if the saga pattern is to be used.

It also has the problem that completing the saga can take a long time, so telling the user if the action succeeded can not be synchronous.

**Event Sourcing**

Event Sourcing is a pattern in which data in a database is not stored as entities, but rather as a series of messages describing how to change the state of the original entity. This solves the problem of each service having to both commit changes as well as send a message to other services, as the sending of the message IS the commit. I find it difficult to understand, but think of it as a publish-subscribe pattern I guess.

**Transactional Outbox**

Where the Event Sourcing pattern made the messages into persisted storage, the transaction outbox pattern stores the message in persistent storage in the same transaction as the entity is stored. The message can then be read from the outbox by a message relay service, which passes it on to any related services.

**Kafka**

Kafka is a distributed event streaming platform. Its intended use is allowing for real-time communication through pub/sub or message queues. Kafka does not rely on a message queue, but stores all messages in a log, allowing subscribers to read from the log at some offset. When a subscriber sends an ack to acknowledge that it has read a given message, Kafka increments the offset, allowing a subscriber to read the next message. Kafka ships with a Java client, as well as SDKs for most large languages. An analysis I read compared Kafka to a library, where books can be stored until some subscriber asks for a book on a specific topic, after which that book is loaned out to the user.

**RabbitMQ**

RabbitMQ is a distributed message broker, relying on message queues to push messages from publisher to subscribers. This push-model works well with longer-running tasks, allowing for responding to HTTP requests immediately while ensuring that the message will be processed at some future point. RabbitMQ can also be used to distribute the load of processing messages between multiple subscribers to the same queue, also handling routing. RabbitMQ can be compared to a post office, receiving and rerouting mail. RabbitMQ seems like the best fit for our current project, as it seems like working with messages is simpler than working with pub/sub in this case.