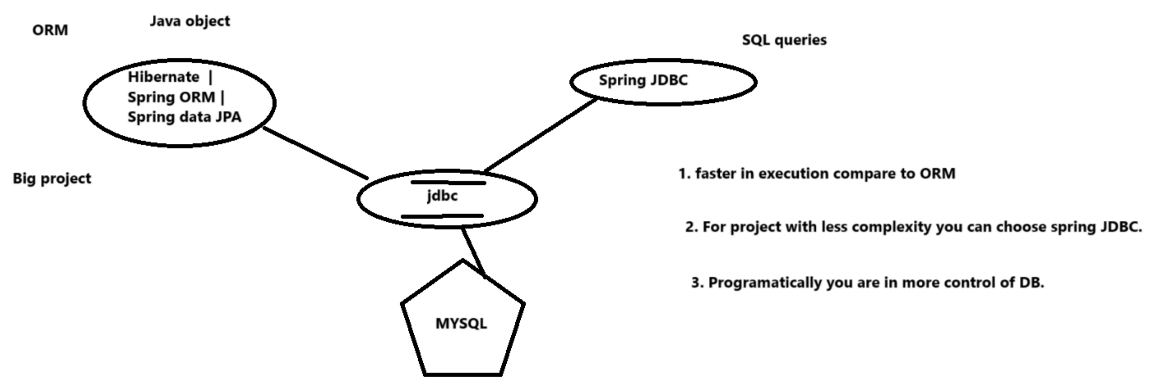
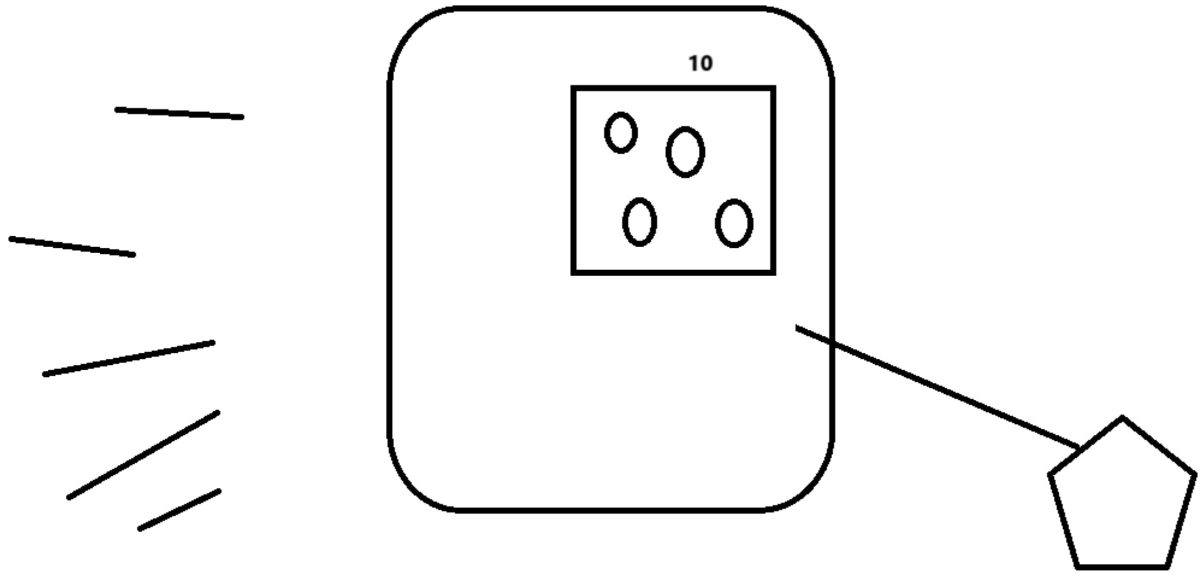
DIAGRAM


ORM
Java object
Hibernate | Spring ORM | Spring data JPA
Big project
SQL queries
Spring JDBC
jdbc
MYSQL
1. faster in execution compare to ORM
2. For project with less complexity you can choose spring JDBC.
3. Programatically you are in more control of DB.



LIVE NOTES

Technical good + Attitude.

1. Working with Spring JDBC
2. Apply this in project use-cases

Yes, I will figure out & work with it.

----

1. pom dependencies
   spring jdbc
   db specific (mysql)

   Spring boot autoconfig is activated.
   create java objects, are trying to point to right DB.

2. We need to inform the right DB configuration to connect to DB.

```
spring.datasource.url=jdbc:mysql://localhost:3306/payments
spring.datasource.url=jdbc:mysql://<physical IP>:<port>/<logical db name>
```

- for each env specific property file, add required DB configuration.

- Sensitive information, DB IP, Stripe APIKey should not be there in property file/not in source code (not committed to bitbucket)

  AWS Secrets Manager

3. 2 primary objects which are AutoConfigured
   - NamedParameterJDBCTemplate (CRUD)
   - Connection Pool (Hikari DS) - Not required to directly work with this. Spring boot JDBC framework, will internally do it for us.

   Sensible Defaults

For every request its not practical to create a new connection object. In concurrent scenario when 1k requests are coming at the same time, then opening 1k connections is not practical. network failure, db might reject connections.. memory hit....

Connection Pool.

you create pool of connection object. 10 object, 20 object.
for every request you use object from this pool. ONce your work is done, you release the object back into pool.

Write to achive connection & to release connection.
open connection
close connection

4. How to programmatically write code to work with DB.
.dao / .repository

service => dao
via interface

TransactionDao

@Repository
TransactionDaoImpl

it an be Dependency injected or lookedup

Usecase: When createpayment request comes, then save the Txn Object into Transaction table in DB.
CreatedStatusHandler class.

```
public interface TransactionDao {

	public TransactionDTO createTransaction(TransactionDTO txnDto);

}
```

in DaoImpl

1. We received TransactionDTO
2. Convert TransactionDTO to TransactionEntity
3. jdbcTemplate.update(needed param)

while converting DTO to Entity,
String APM is not converted to its id=1 value.
- We need to establish this mapping (Enums)
    created total 4 enums.

- Customize ModelMapper behaviour when trying to convert DTO to Entity
    how to use this mapping of Enums.

```
import org.modelmapper.AbstractConverter;

import com.hulkhiretech.payments.constant.PaymentMethodEnum;

public class PaymentMethodEnumConverter extends AbstractConverter<String, Integer> {
   @Override
   protected Integer convert(String source) {
      return PaymentMethodEnum.getByName(source).getId();
   }
}
```

replicate the convert for all 4 enums.

- yes, the string values are mapped to corresponding id & TransactionEntity is ready to be saved in DB.

====================

```
@Override
	public TransactionDTO createTransaction(TransactionDTO txnDto) {
		log.info("Creating Transaction in DB||txnDto:" + txnDto);

		TransactionEntity txnEntity = modelMapper.map(txnDto, TransactionEntity.class);
		log.info("Converted to Entity txnEntity:" + txnEntity);
```

```
                    String sql = "INSERT INTO payments.`Transaction` (userId, paymentMethodId,
providerId, paymentTypeId, txnStatusId, " +
                                        "amount, currency, merchantTxnReference, txnReference,
providerReference, errorCode, errorMessage, retryCount) " +
                                        "VALUES (:userId, :paymentMethodId, :providerId,
:paymentTypeId, :txnStatusId, :amount, :currency, " +
                                        ":merchantTxnReference, :txnReference, :providerReference,
:errorCode, :errorMessage, :retryCount)";

                    SqlParameterSource params = new
BeanPropertySqlParameterSource(txnEntity);
       KeyHolder keyHolder = new GeneratedKeyHolder();

                    jdbcTemplate.update(sql, params, keyHolder, new String[]{"id"});

                    int id = keyHolder.getKey() != null ? keyHolder.getKey().intValue() : -1;
                    txnDto.setId(id);

                    log.info("Transaction created in DB||txn id:{}|txnReference:{}",
                                        txnDto.getId(),
                                        txnDto.getTxnReference());

                    return txnDto;
          }
```

================

Read the autoincremented Id value back.

Make flow of initiatedTxn work

1. Pass String txnReference to service layer.
2. PaymentServiceImpl.initiatePayment
   Make DB call to get TransactionDTO back

   public TransactionDTO getTransactionByTxnReference(String txnRefs);

   write Spring JDBC code to read the values.
   TransactionEntity

Convert it into TransactionDTO

String => Int

Entity => DTO
id => String

4 converts <String, Integer>
new 4 converters <Integer, String>
    using enums to do code.

```
public class PaymentMethodEnumIdToStringConverter extends AbstractConverter<Integer,
String> {
   @Override
   protected String convert(Integer source) {
      return PaymentMethodEnum.getById(source).getName();
   }
}
```

ModelMapper
PropertyMap<TransactionEntity, TransactionDTO>

-
getTransactionByReference()
updateTransaction
- update status
- providerReference
- errorCode
- errorMessage