Data Access with Spring Boot

JDBC

The Problems with JDBC

```
public List<Customer> findByLastName(String lastName) {
  List<Customer> customerList = new ArrayList<>();
  String sql = "SELECT FIRST_NAME, AGE FROM CUSTOMER WHERE LAST_NAME=?";

  try (Connection conn = dataSource.getConnection();
    PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, lastName);

    try (ResultSet rs = ps.executeQuery()) {
      while (rs.next()) {
        customerList.add(new Customer(rs.getString("FIRST_NAME"), …));
      }
    }
  } catch (SQLException e) {
    /* ??? */
  }
  return customerList;
}
```

Template Design Pattern

- Spring provides many template classes
  - `JdbcTemplate`, `JmsTemplate`
  - `MongoTemplate`, `ElasticTemplate`, `Neo4jTemplate`, `RedisTemplate`, …
  - `RestTemplate`, `WebServiceTemplate` …
  - Hides low-level resource management
  - Simplifies exception handling
  - Provides a consistent API

Getting started

- All steps are handled by the `JdbcTemplate`
  - Acquisition of the connection
  - Participation in the transaction
  - Execution of the statement
  - Processing of the result set
  - Handling exceptions
  - Release of the connection

JdbcTemplate Repository

```
public class JdbcCustomerRepository implements CustomerRepository {
  private JdbcTemplate jdbcTemplate;

  public JdbcCustomerRepository(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  public int getCustomerCount() {
    String sql = "select count(*) from customer";
    return jdbcTemplate.queryForObject(sql, Integer.class);
  }
}
```

No need to deal with `Connection`, `PreparedStatement`, `ResultSet`, `SQLException` and `DataSource` anymore.

Simple Java Types

Querying for Simple Java Types

- `JdbcTemplate`

```
public int getCountOfNationalsOver(Nationality nationality, int age) {
  String sql = "SELECT COUNT(*) FROM CUSTOMER WHERE AGE > ? AND NATIONALITY = ?";
  return jdbcTemplate.queryForObject(sql, Integer.class, age, nationality.toString());
}
```

- `NamedParameterJdbcTemplate`

```
public int getCountOfNationalsOver(Nationality nationality, int age) {
  String sql = "SELECT COUNT(*) FROM CUSTOMER WHERE AGE > :age AND NATIONALITY = :nationality";
  Map<String,Object> map = new HashMap<String,Object>();
  map.put("nationality", nationality);
  map.put("age", age);
  return namedParameterJdbcTemplate.queryForObject(sql, Integer.class, map);
}
```

Database Inserts

- Inserting a new row
  - Returns the number of rows modified

```
 public int insertCustomer(Customer customer) {
  return jdbcTemplate.update(
    "INSERT INTO CUSTOMER (FIRST_NAME, LAST_NAME, AGE) VALUES (?, ?, ?)",
    customer.getFirstName(),
    customer.getLastName(),
    customer.getAge());
}
```

Database Updates

- Updating an existing row

```
public int updateAge(Customer customer) {
  return jdbcTemplate.update(
    "UPDATE CUSTOMER SET AGE=? WHERE ID=?",
    customer.getAge(),
    customer.getId());
}
```

Database Deletes

- Deleting an existing row

```
public int deleteCustomer(Customer customer) {
  return jdbcTemplate.update(
    "DELETE CUSTOMER WHERE ID=?",
    customer.getId());
}
```

Generic Queries

Querying for Generic Maps

- Querying for a single row

```
public Map<String,Object> getCustomerInfo(int id) {
  String sql = "SELECT * FROM CUSTOMER WHERE ID=?";
  return jdbcTemplate.queryForMap(sql, id);
}
```

- Returns

  `Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }`

Querying for List of Maps

- Querying for multiple rows

```
public List<Map<String,Object>> getAllCustomerInfo() {
  String sql = "SELECT * FROM CUSTOMER";
  return jdbcTemplate.queryForList(sql);
}
```

- Returns

```
List {
0 - Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }
1 - Map { ID=2, FIRST_NAME="Jane", LAST_NAME="Doe" }
2 - Map { ID=3, FIRST_NAME="Junior", LAST_NAME="Doe" }
}
```

Domain Object Queries

Rowmapper for Mapping a Row

```
public interface RowMapper<T> {
  T mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

Querying for Domain Objects

- Querying for a single row with JdbcTemplate

```
public Customer getCustomer(int id) {
  return jdbcTemplate.queryForObject(
    "SELECT FIRST_NAME, LAST_NAME FROM CUSTOMER WHERE ID=?",
    (rs, rowNum) ->
        new Customer(rs.getString("FIRST_NAME"), rs.getString("LAST_NAME"))
    , id);
}
```

Querying for Domain Objects

- Querying for multiple rows

```
public List<Customer> getAllCustomers() {
  return jdbcTemplate.query(
    "SELECT FIRST_NAME, LAST_NAME FROM CUSTOMER",
    (rs, rowNum) ->
        new Customer(rs.getString("FIRST_NAME"), rs.getString("LAST_NAME"))
  );
}
```

Summary of Callback Interfaces

- `RowMapper`
  - Maps a row of ResultSet maps to a single domain object
- `ResultSetExtractor`
  - Maps multiple rows of a ResultSet map to a single object
- `RowCallbackHandler`
  - Another handler that writes to alternative destinations like a CSV file

Exception Handling with Spring

Checked and Unchecked Exceptions

The Problems with Exceptions

- Checked Exceptions
  - Force developers to handle errors or declare them
  - ❌ intermediate methods must declare exception(s) from all methods below
- Unchecked Exceptions
  - Can throw up the call hierarchy to the best place to handle it
  - ✔ Methods in between don't know about it
- **Spring always throws Runtime (unchecked) Exceptions**

SQL Exceptions

- SQLException
  - Checked exception
  - Too general – one exception for every database error
  - Calling class 'knows' you are using JDBC
  - Tight coupling

Data Access Exceptions

Spring provides a `DataAccessException` hierarchy

- Hides whether you are using JPA, Hibernate, JDBC …
- A hierarchy of unchecked exceptions
- Not just one exception for everything
- Consistent across all supported Data Access technologies

See also: https://github.com/spring-projects/spring-framework/blob/main/spring-jdbc/src/main/resources/org/springframework/jdbc/support/sql-error-codes.xml

DataAccessException

BadSqlGrammarException

UncategorizedSQLException

CannotAquireLockException

CleanupFailureDataAccessException

ConcurrencyFailureException

...

DuplicateKeyException

EmptyResultDataAccessException

OptimisticClockingFailureException

PermissionDeniedDataAccessException

QueryTimeoutException

TransientDataAccessException

...

Transactions with Spring

Transactional Code Pattern

- Many different APIs, but a common pattern

```
try {
    // beginTransaction

    …

    // commitTransaction
 } catch (Exception e) {

    // rollbackTransaction

 }
```

Spring Transaction Management

- Spring separates transaction demarcation from transaction implementation
  - Demarcation expressed declaratively (Annotations) via AOP
  - `PlatformTransactionManager` abstraction hides implementation details
- Spring uses the same API for global vs. local.
  - Change from local to global or vice versa is minor
    - It's just a configuration change for the transaction manager

Deploying the Transaction Manager

```
@Bean
public PlatformTransactionManager transactionManager(DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
```

Accessing JTA Transaction Manager

- JNDI lookup for container-managed DataSource

```
@Bean
public PlatformTransactionManager transactionManager() {
    return new JtaTransactionManager();
}

@Bean
public DataSource dataSource(@Value("${db.jndi}" String jndiName) {
    JndiDataSourceLookup lookup = new JndiDataSourceLookup();
    return lookup.getDataSource(jndiName);
}
```

Declaring Transaction Boundaries

@Transactional Configuration

```
@Transactional
public class CustomerServiceImpl implements CustomerService {

    public BookingConfirmation bookTransfer(Route route, Customer customer) {
        // Atomic unit-of-work
    }
}
```

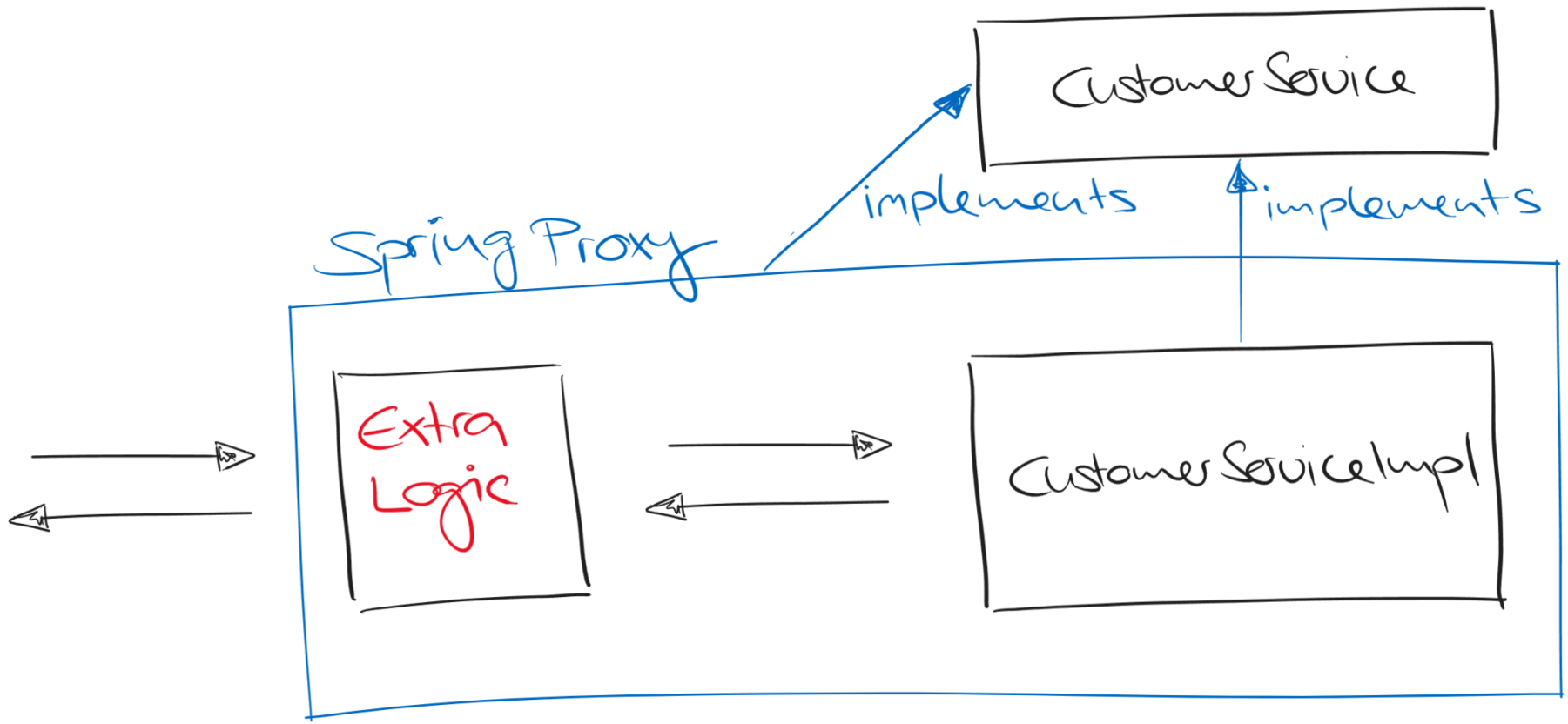
```
@Transactional(readOnly = false)
public class CustomerServiceImpl implements CustomerService {

    @Transactional(readOnly = true)
    public BookingConfirmation bookTransfer(Route route, Customer customer) {
        // Atomic unit-of-work
    }
}
```

Enabling Transactions

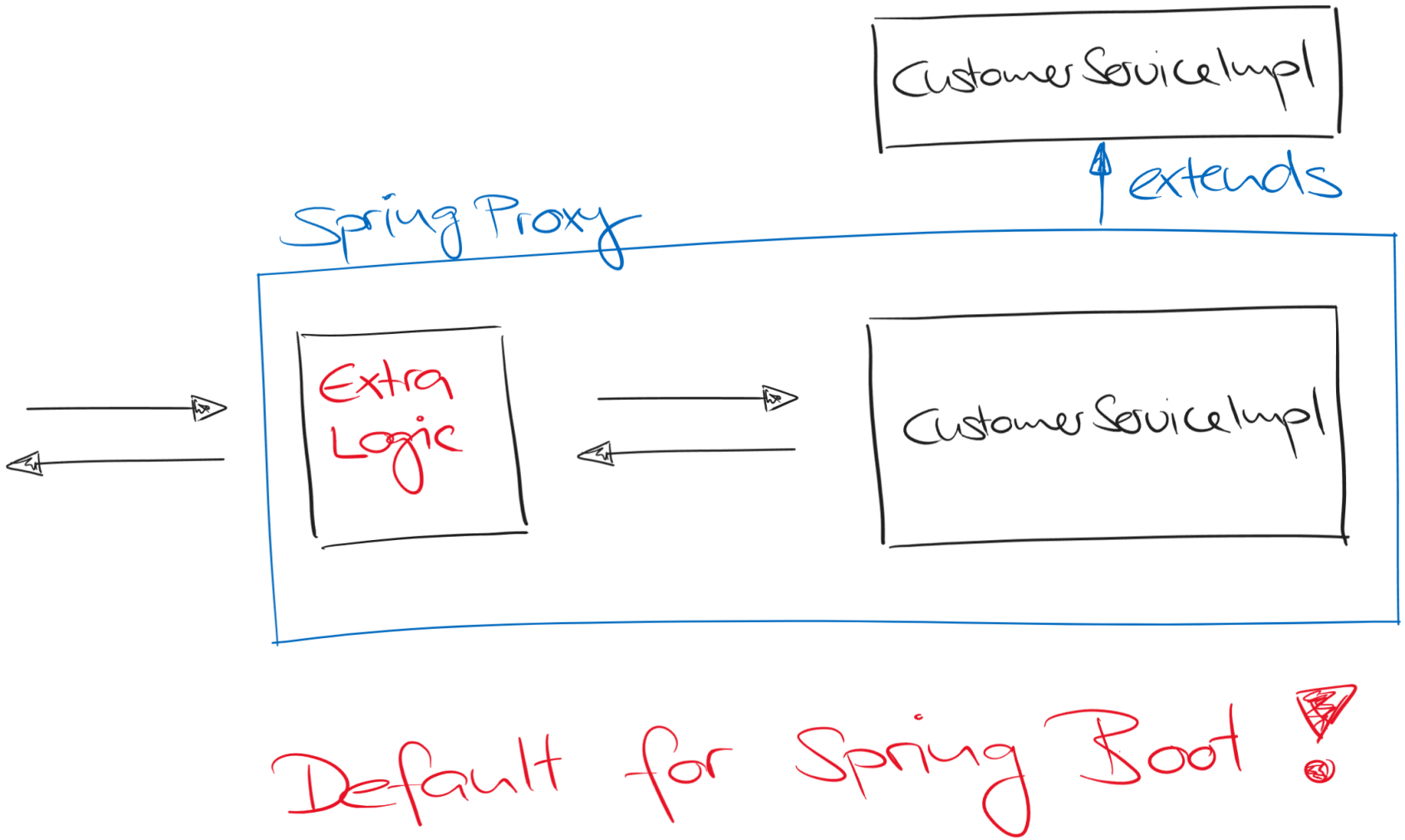
```
@Configuration
@EnableTransactionManagement
public class TxnConfig {

    @Bean
    public PlatformTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

Spring Proxies

JDK Proxy


CustomerSevice
implements
implements
Spring Proxy
Extra Logic
CustomerSeviceImpl

CGLIB Proxy



Default for Spring Boot !

Transactional Behaviour

- Proxy implements the following behavior
  - The transaction starts before entering the method
  - Commit at the end of the method
  - Rollback if a method throws a RuntimeException
    - Default behavior
    - Can be overridden
  - Checked exceptions do not cause Rollback
- All controlled by configuration

Transaction Propagation

What Happens Here?

```
@Component
public class BookingServiceImpl implements BookingService {
    @Autowired // Side note: This is evil! :)
    private CustomerService customerService;

    @Transactional
    public void updateWith(Client client) {
        // …
        this.customerService.updateCustomers(client.getCustomers());
    }
}
```

```
@Component
public class CustomerServiceImpl implements CustomerService {

    @Transactional
    public void updateCustomers(List<Customer> customers){
        // …
    }
}
```

Transaction Propagation with Spring

- 7 levels of propagation
  - The following examples show only `REQUIRED` and `REQUIRES_NEW`
- Can be used as follows:

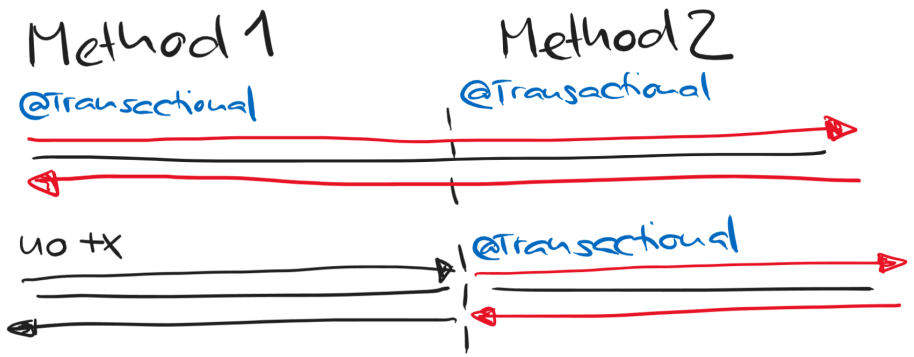
```
@Transactional(propagation=Propagation.REQUIRED)
```

7 Levels of Propagation

- **`REQUIRED`**: Execute within a current transaction; create a new one if none exists
- **`REQUIRES_NEW`**: Create a new transaction; suspend the current transaction if one exists (and use a different connection)
- **`SUPPORTS`**: Execute within a current transaction; execute non-transactionally if none exists
- **`NOT_SUPPORTED`**: Execute non-transactionally; suspend the current transaction if one exists
- **`MANDATORY`**: Execute within a current transaction; throws an exception if none exists
- **`NEVER`**: Execute non-transactionally; throw an exception if a transaction exists
- **`NESTED`**: Execute within a nested transaction if a current transaction exists; behave like **REQUIRED** otherwise

REQUIRED

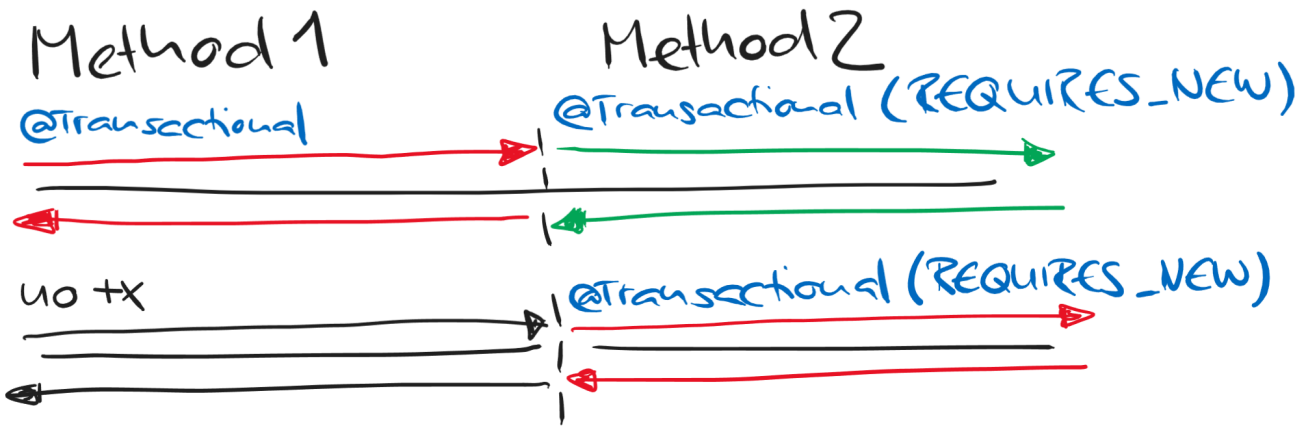
- Default propagation level if none is specified
- Execute within a current transaction; create a new one if none exists




```
@Transactional(propagation=Propagation.REQUIRED)
```

REQUIRES_NEW

- Create a new transaction, suspending the current transaction if one exists




```
@Transactional(propagation=Propagation.REQUIRES_NEW)
```

Rollback Rules

- RuntimeException triggers rollback

```
public class CustomerServiceImpl implements CustomerService {

    @Transactional
    public BookingConfirmation bookTransfer(Route route, Customer customer) {
        // …
        throw new RuntimeException();
    }
}
```

Customizing Rollback Rules

With `rollbackFor` and `noRollbackFor`

- Customizing behavior for checked exceptions and unchecked exceptions

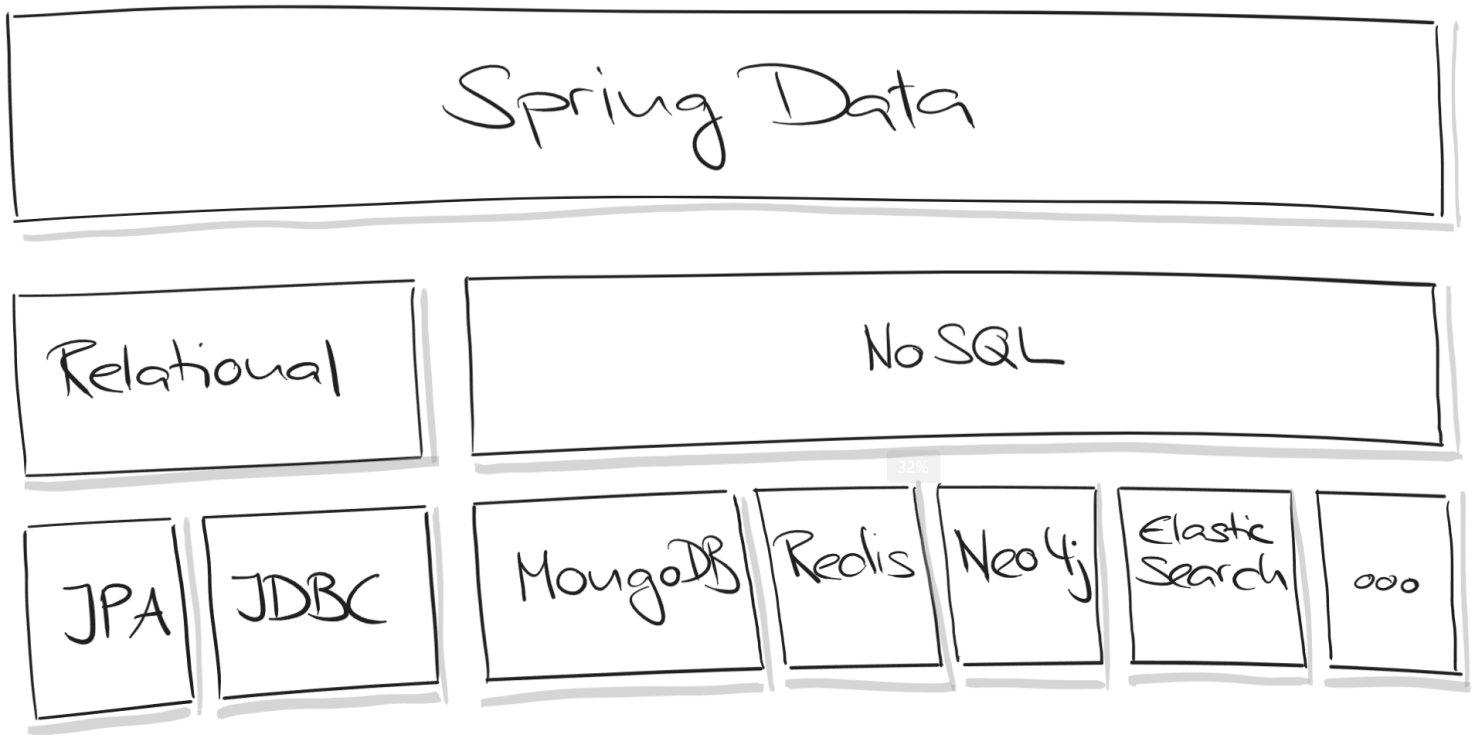
```
public class CustomerServiceImpl implements CustomerService {

    @Transactional(rollbackFor=MyOwnCheckedException.class,
        noRollbackFor={NotificationSendingException.class, MailException.class})
    public BookingConfirmation bookTransfer(Route route, Customer customer) {
        // …
    }
}
```

Spring Data JPA

What is Spring Data?

What is Spring Data?

- Umbrella project for data access technologies
- Reduces boilerplate code for data access



Spring Data JPA Setup

Dependencies

- Adding Spring Data JPA Starter to your project and getting all the necessary dependencies for Transactions, AOP, JDBC, JPA, Hibernate and others.
- Adding H2 (or HSQLDB or Derby) as an in-memory DB for faster development.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Spring Bean Setup

- Starter configures infrastructure Spring beans automatically
  - `DataSource`
  - `LocalContainerEntityMangerFactory`
  - `PlatformTransactionManager`
  - Scans for `@Entity` -> JPA Mapping

- For a default development configuration, nothing has to be done here. Customization happens by overriding

Entities and Repositories

Setting up a JPA Entity

```
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;

    protected Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format("Customer[id=%d firstName='%s', lastName='%s']", id, firstName,lastName);
    }
}
```

Setting up a JPA Repository

- Just an interface, no implementation is needed
- Spring Data proxy with implementation
- CRUD, paging and sorting methods provided
- Dynamic finders and `@Query` support

```
public interface CustomerRepository extends JpaRepository<Customer,Long>{

    List<Customer> findByLastName(String lastName);

    @Query("select e from Customer e where e.lastName = :lastName")
    List<Customer> findQueryByLastName(@Param("lastName")String name);
}
```

Repository and Repository Hierarchy

- `Repository` marker interface

  `Repository<T, ID extends Serializable>`

- Repository Interface Hierarchy :
  - `JpaRepository<T,ID>`
    - `ListPagingAndSortingRepository<T,ID>`
      - `Repository<T,ID>`
    - `ListCrudRepository<T,ID>`
      - `Repository<T,ID>`

Proxy Implementation Uses Interface

- Scanning for interfaces that extends `Repository<T, ID>`
- Methods from interfaces auto-generated
  - CRUD methods
  - Paging and Sorting methods
  - Dynamic finders & `@Query`

Test Run or How to Use It

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }

    @Bean
    public CommandLineRunner demo(CustomerRepository repository) {
        return(args) ->{
            repository.save(new Customer("Carmen","Bianchi"));
            Customer bianchi = repository.findByLastName("Bianchi");
            log.info(bianchi.toString());
        };
    }
}
```

Customizing Spring Data JPA

Customizing Spring Data JPA (1)

- To override the default configuration you might use `@EnableJpaRepositories` and `@EntityScan`
- Was stored data at startup deleted? AutoConfiguration runs embedded Databases "drop & delete".

```
spring.jpa.hibernate.ddl-auto=none
```

- Want to initialize your in-memory DB with data? Add a `schema.sql` and/or `data.sql` into your resource folder.
- Want a more sophisticated database migration tool? Look at the Flyway or Liquibase integration.

Customizing Spring Data JPA (2)

- Spring Data is highly customizable. Read the documentation. (e.g. Custom Mix-in Repositories)
- Dynamic finder methods names might get very long
  - Use `@Query` and JPQL Syntax and provide a nice name
- Use native queries if needed `@Query(value="SELECT * FROM CUSTOMER WHERE FIRST_NAME=?1", nativeQuery=true)`
- CRUD methods on repository instances are transactional
  - Use `@Transactional` in the repository to override the default behavior
  - You should use them anyway on `@Service` methods

Caching with Spring

Why Do We Need Caching?

Why Do We Need a Cache Abstraction?

- Having operations like
  - Putting data into the cache
  - Reading data from the cache
  - Updating data in the cache
  - Deleting data from the cache
- Mulitple cache providers
  - like EhCache, Hazelcast, Infinispan, Couchbase, Redis, Caffeine, Cache2k, Simple and more

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#io.caching.provider

Enabling Spring's Cache Abstraction

- Adding a starter dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

- Enable the `@Cachable` infrastructure

```
@Configuration
@EnableCaching
class EmbeddedCacheConfig {

  // Other methods omitted

}
```

Cache Manager

- `@EnableCaching` searches for a `CacheManger`
- Simple (in-memory) will be used by default since we did not specify one
- Otherwise define a specific implementation

Cache Manager - Reading Data

- The parameters of the method are keys and the return value, values
- If the value is found in the cache, the method will not be executed

```
@Cacheable("books")
public Book findBookByIsbn(String isbn) { … }
```

```
@Cacheable(value="topBooks", key="#isbn.toUpperCase()")
public Book findBookByIsbn(String isbn) { … }
```

```
@Cacheable(key="T(example.KeyGen).hash(#author)")
public Book findBookByAuthor(Author author) { … }
```

```
@Cacheable(key="#author.name")
public Book findBookByAuthor(Author author) { … }
```

Cache Manager - Updating Data

- The body of the `@CachePut` method will always be executed
  - Spring will put the result of the method into the cache

```
@CachePut(value = "authors", key = "#author.id")
public Author updateAuthor(Author author) { … }
```

Cache Manager - Clearing Data

- `@CacheEvict` deletes the data from the cache
- If we set the attribute `allEntries` to true, we can delete the whole cache

```
@CacheEvict(beforeInvocation=true)
public void loadBooks() { … }
```

```
@CacheEvict(value = "authors", key = "#author.id")
public void deleteAuthor(Author author) { … }
```

EHCache Cache Manager

```
@Autowired
ApplicationContext context;

@Value("${ehcache.xml.location:'classpath:eh-cache.xml'}")
String location;

@Bean
public CacheManager cacheManager() {
    Resource cacheConfig = context.getResource(location);
    net.sf.ehcache.CacheManager cache = EhCacheManagerUtils.buildCacheManager(cacheConfig);
    return new EhCacheCacheManager(cache);
}
```

Customizing Key Generation

- Spring Cache uses by default `SimpleKeyGenerator`
  - To override we can define a Bean with interface `KeyGenerator`

```
@Configuration
@EnableCaching
class EmbeddedCacheConfig {

    @Bean
    public KeyGenerator bookKeyGenerator() {
        return new BookKeyGenerator();
    }
}
```

Using the KeyGenerator

```
@Service
class BookService {

  @Cacheable(value = "books", keyGenerator = "bookKeyGenerator")
  public Car findBookByIsbn(String isbn) {
    return bookRepository.findById(isbn)
        .orElseThrow(() -> new IllegalStateException("Book not found"));
  }
}
```