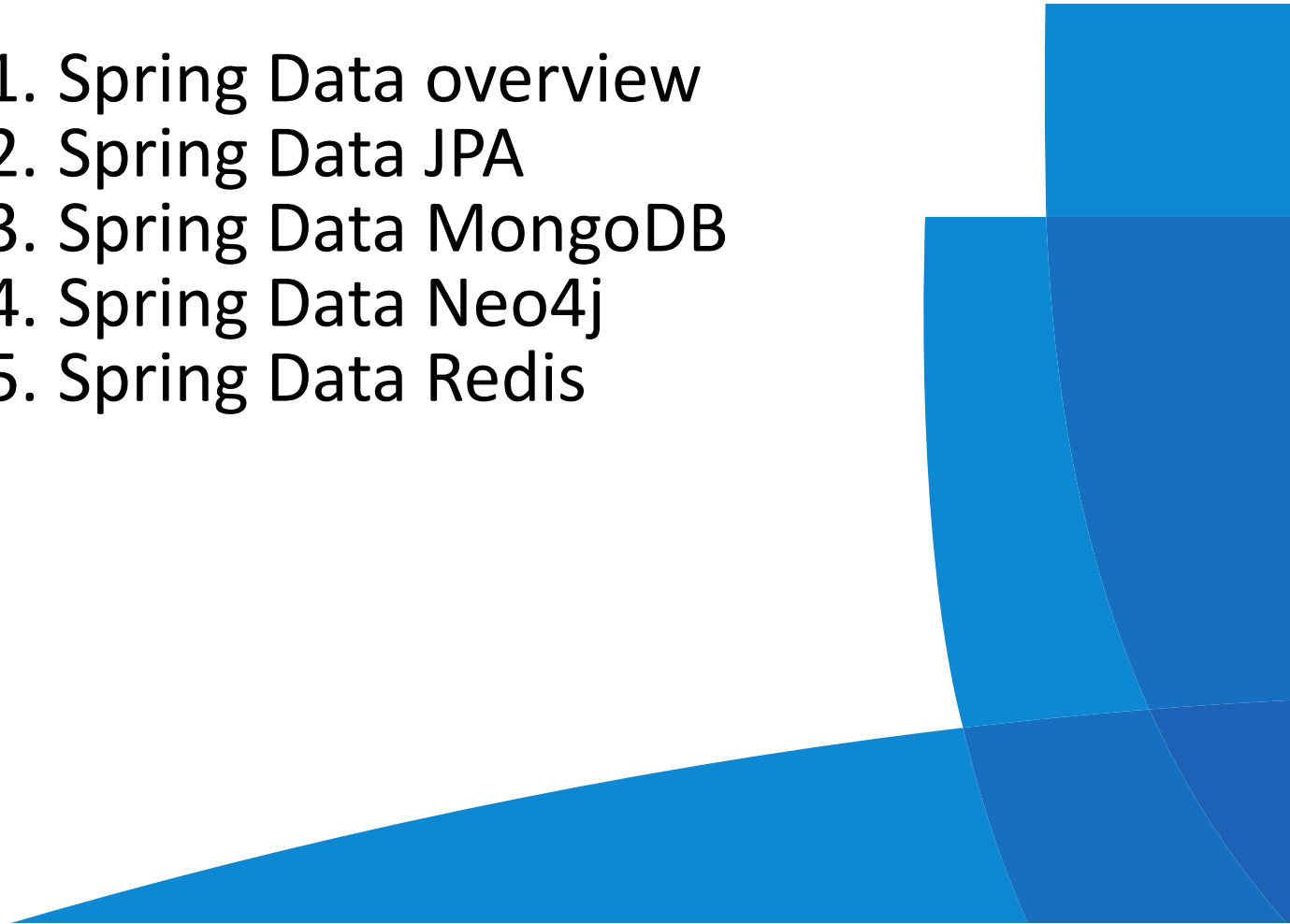
Agenda

1. Spring Data overview
2. Spring Data JPA
3. Spring Data MongoDB
4. Spring Data Neo4j
5. Spring Data Redis



Structure of Spring Data

- open source umbrella project
- supporting different models of data storage
- familiar repository abstraction while preserving special traits of underlying model
- support for a range of popular databases
- RDBMS and NoSQL

Common ingredients

- entity class -> represting an object in your domain model
- repository -> primary method to access data

A user entity

```
import lombok.*

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {

 private Long id;

 private String firstName;

 private String lastName;

 private String email;

}
```

project Lombok

- project Lombok will be used
- a compiletime annotation processor
- free to write your own getter/setters ...
- or overriding methods

Lombok annotations

- Data -> getter/setters/toString/hashCode/equals
- @AllArgsConstructor -> default constructor
- @NoArgsConstructor -> constructor for all fields

@Repository

- Stereotype of @Component
- db's specific exceptions are mapped to unified spring exception model
- Spring Data -> provides implementations for interface definitions

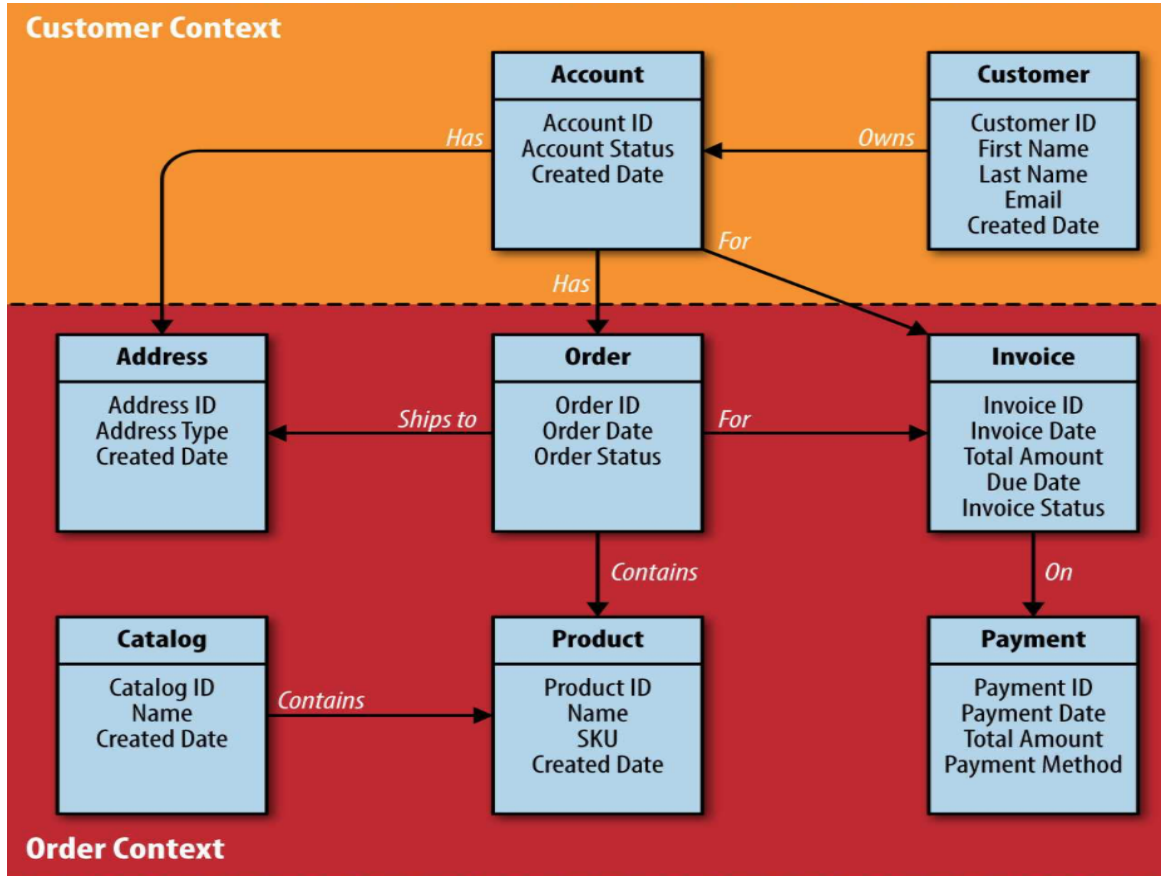
CrudRepository

- most basic pre-defined repository interface -> CrudRepository
- CRUD
  1. CREATE
  2. READ
  3. UPDATE
  4. DELETE

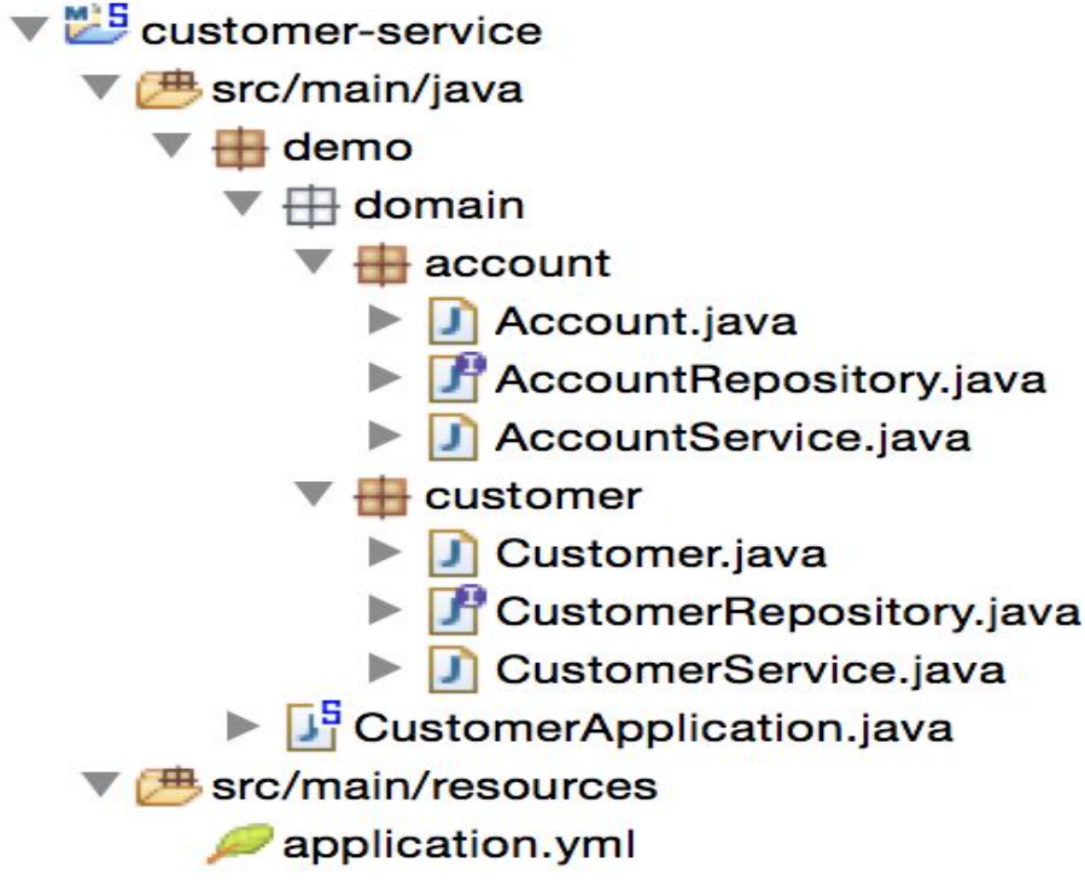
CrudRepository<User,Long>

- supports CRUD operation given the entity User and it's Id column (id of type Long)
- Spring Data provides a bean that implements this interface

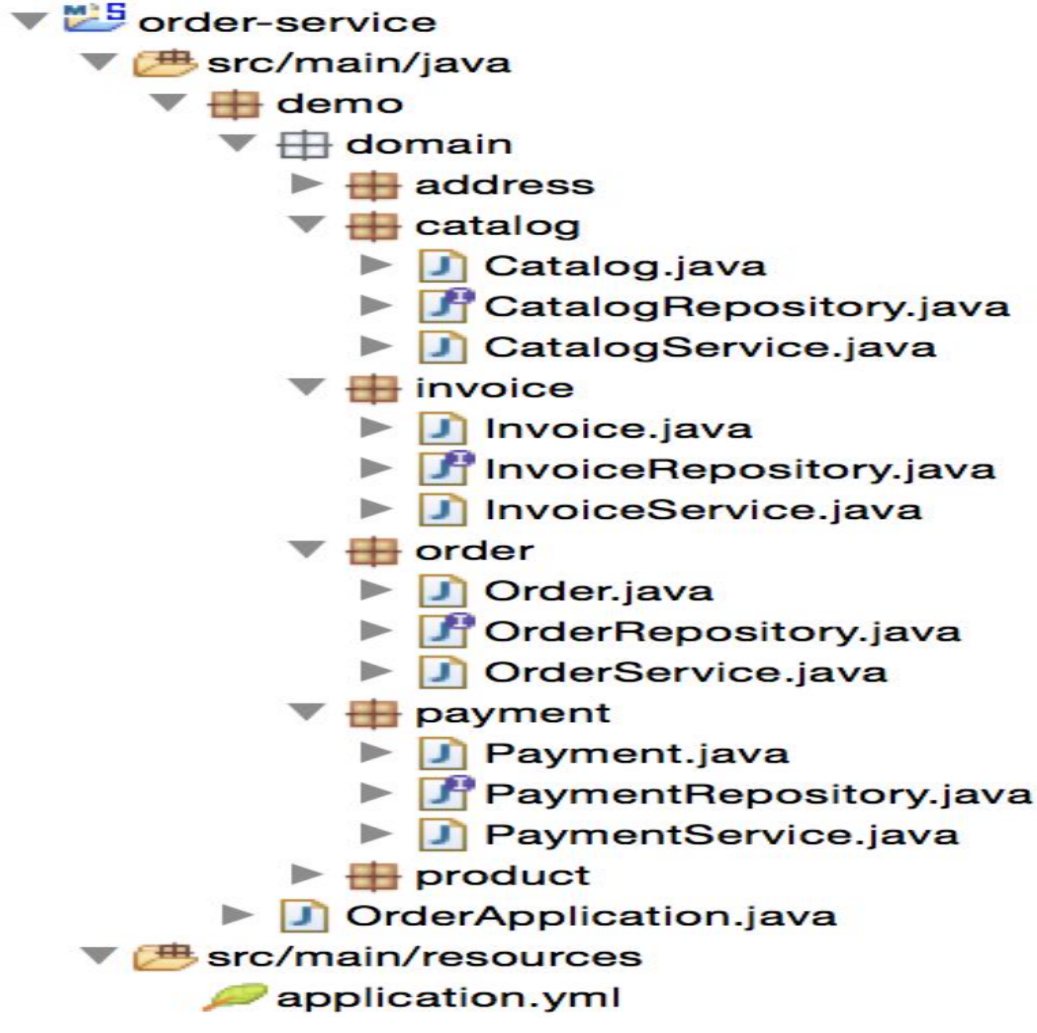
Organizing Java packages for Domain Data



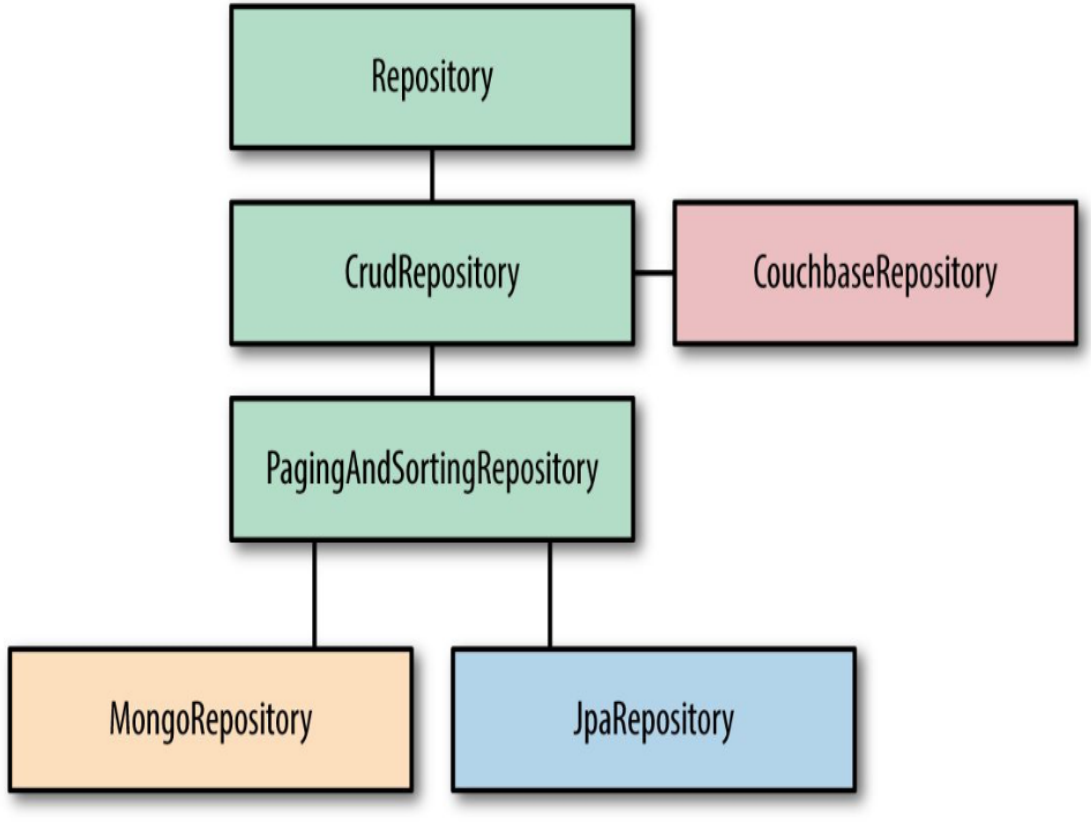
Customer BoundedContext



Order BoundedContext



Repository Hierarchy



JDBC Templates

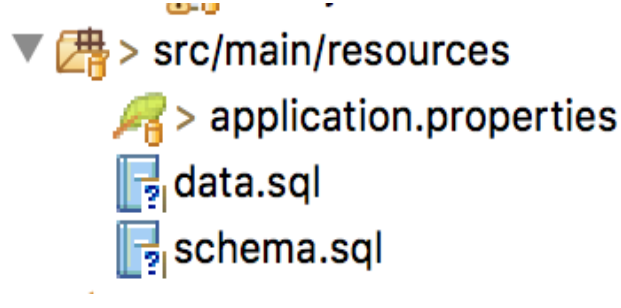
- JDBCtemplate most well-known implementation of template designpattern
- Templates are available for other stores as well

Code highlights

```
jdbcTemplate.execute("DROP TABLE user IF EXISTS");

List<Object[]> userRecords = ....

jdbcTemplate
  .batchUpdate(
   "INSERT INTO user(first_name) VALUES (?)",userRecords);

RowMapper<User> userRowMapper = (rs, rowNum) ->
    new User(rs.getLong("id"),rs.getString("first_name"));

List<User> users = jdbcTemplate.query(
  "SELECT id, first_name FROM user WHERE first_name = ?",
  userRowMapper, "Thom");
```

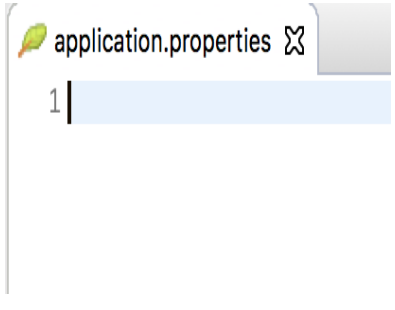
Creating a database

- By convention spring will evaluate 2 files to create db
  1. schema.sql
  2. data.sql



Applying defaults

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```



3 Bounded Contexts

- Shop model with 3 bounded contexts
- Each bounded context will use a different type of storage

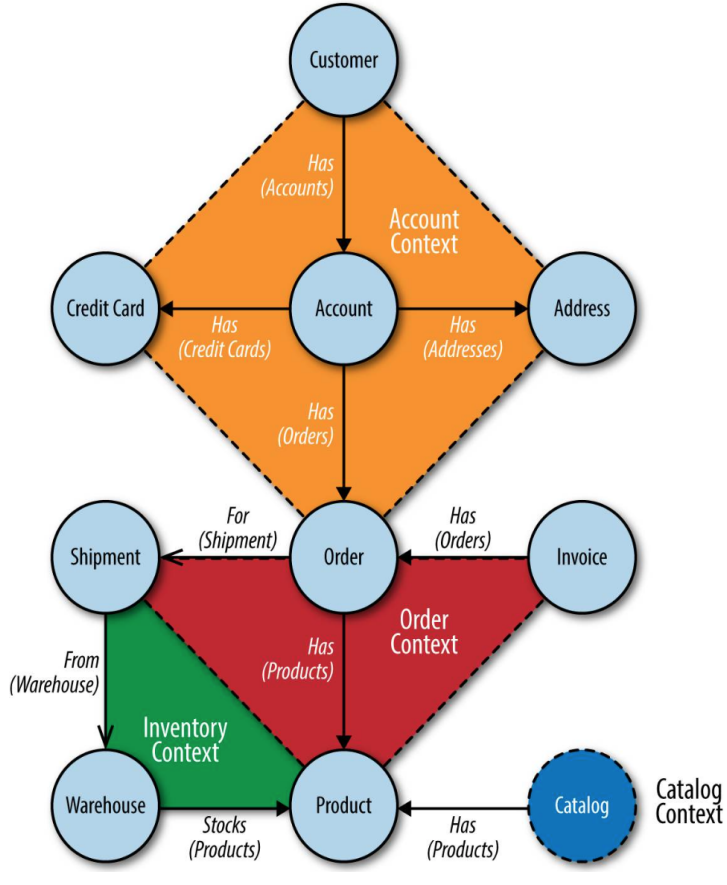
Different kinds of Storage

1. Relational storaged RDBMS (SQL)
2. NoSql storage; Not only SQL storage

NoSql storage

- model used:
- document -> MongoDb
- graph -> neo4j
- key value -> Redis

The domain


Customer
Has
(Accounts)
Account
Context
Credit Card
Account
Address
Has
(Credit Cards)
Has
(Addresses)
Has
(Orders)
Shipment
For
(Shipment)
Order
Has
(Orders)
Invoice
Order
Context
From
(Warehouse)
Has
(Products)
Inventory
Context
Warehouse
Product
Catalog
Catalog
Context
Stocks
(Products)
Has
(Products)

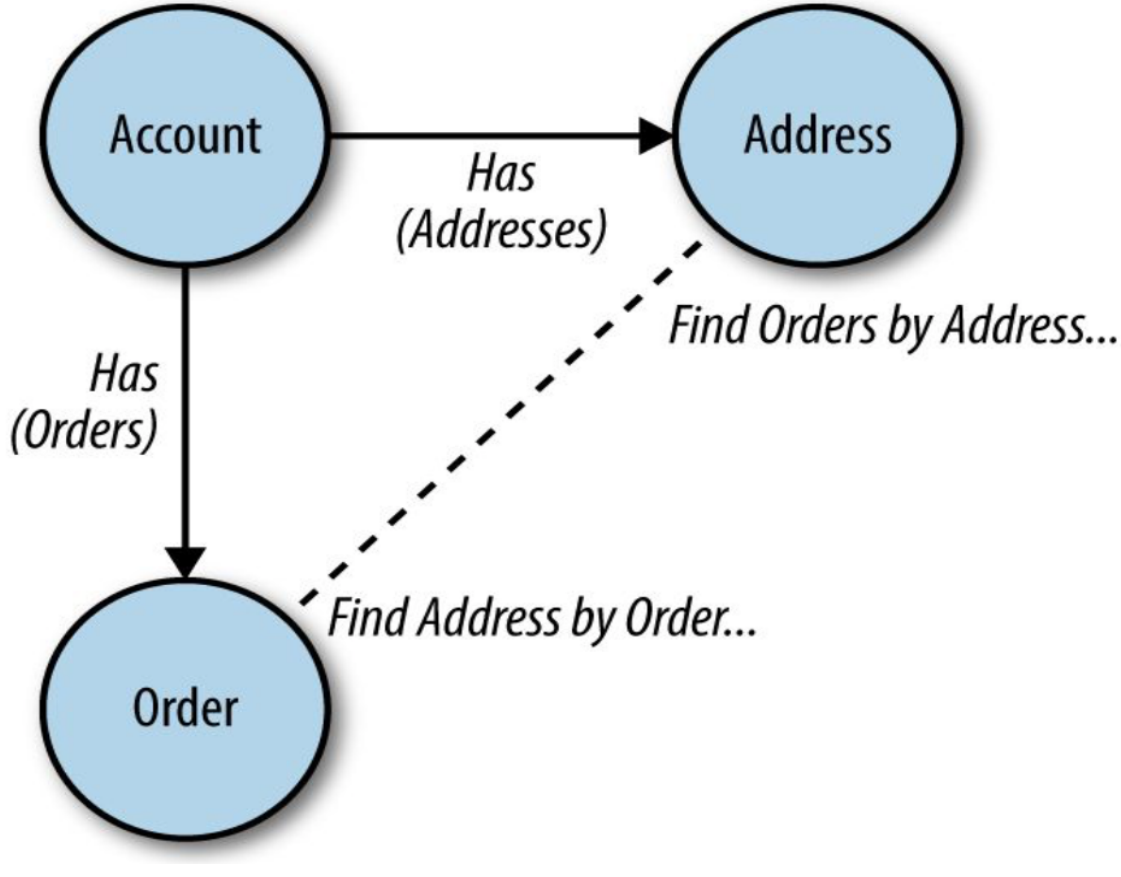
Influence of DDD

- 3 Bounded contexts
  - Making explicit the shared concepts -> 2 colors
  - Making explicit unrelated concepts -> 1 color
  - each domain class is represented with a circle
  - each domain class that should be queried in this context is accompanied by a repository
  - here, each circle is a domain class and a repository

Relationships

- solid lines -> a directed relationship (i.e. has a)
- dotted lines -> inferred relationship through a repository join
- a dotted line indicates that there is a repository query that acts as a join bridge between related concepts
- this bridge is only valid if there is a shared concept that exists between the 2 repositories

Relationship in RDBMS 1



Relationship in RDBMS 2

```
  "SELECT o FROM Orders o"
+ "INNER JOIN Account ac ON ac.accountId=o.accountId"
+ "INNER JOIN Address ad ON ad.accountId=ac.accountId"
+ "WHERE ad.=id=1",
```

The AccountService

- Use Spring Data JPA for AccountService
- The AccountService covers the AccountContext
- Unique to Spring Data JPA -> multiple (JDBC compatible) implementations
  - use embedded datasource for testing
  - use full-featured datasource for development
- Use profiles to cleanly separate the 2

Using profiles

application.yml

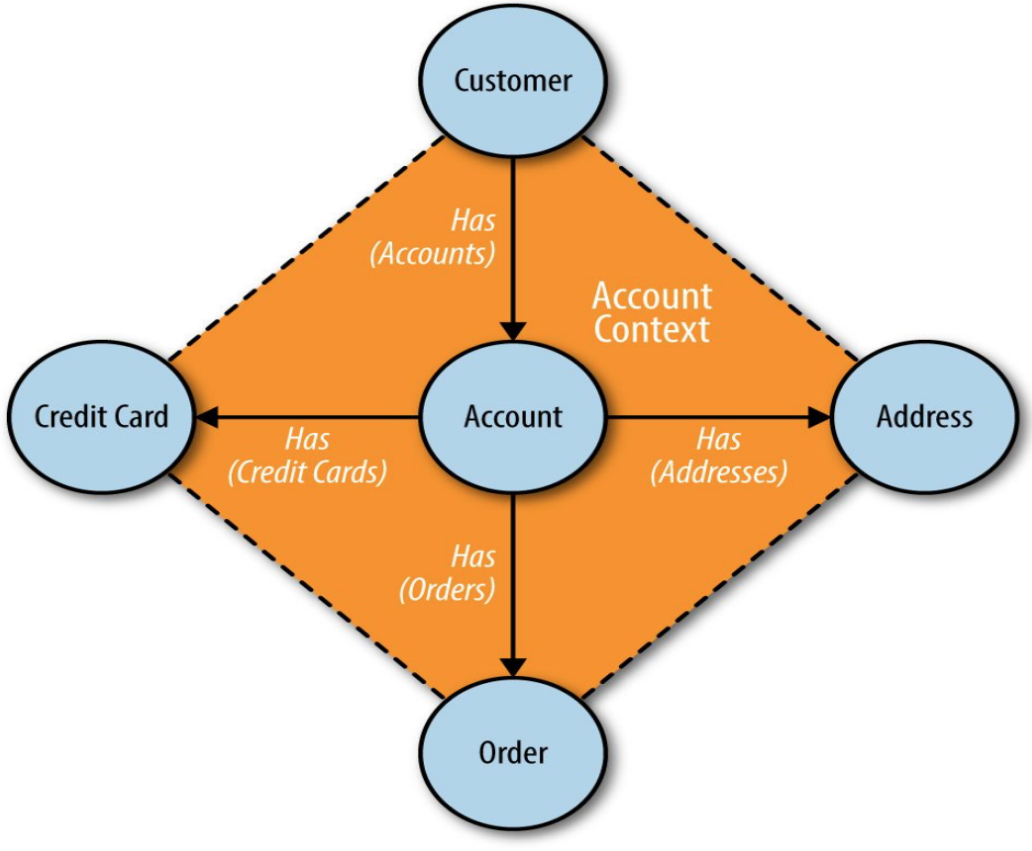
```yaml
spring:
  profiles:
    active: development
---
spring:
  profiles: development
  jpa:
    show_sql: false
    database: MYSQL
    generate-ddl: true
  datasource:
    url: jdbc:mysql://192.168.99.100:3306/dev
    username: root
    password: dbpass
---
spring:
  profiles: test
  jpa:
    show_sql: false
    database: H2
  datasource:
    url: jdbc:h2:mem:testdb;DB_CLOSE_ON_EXIT=FALSE
```

Activating test profile

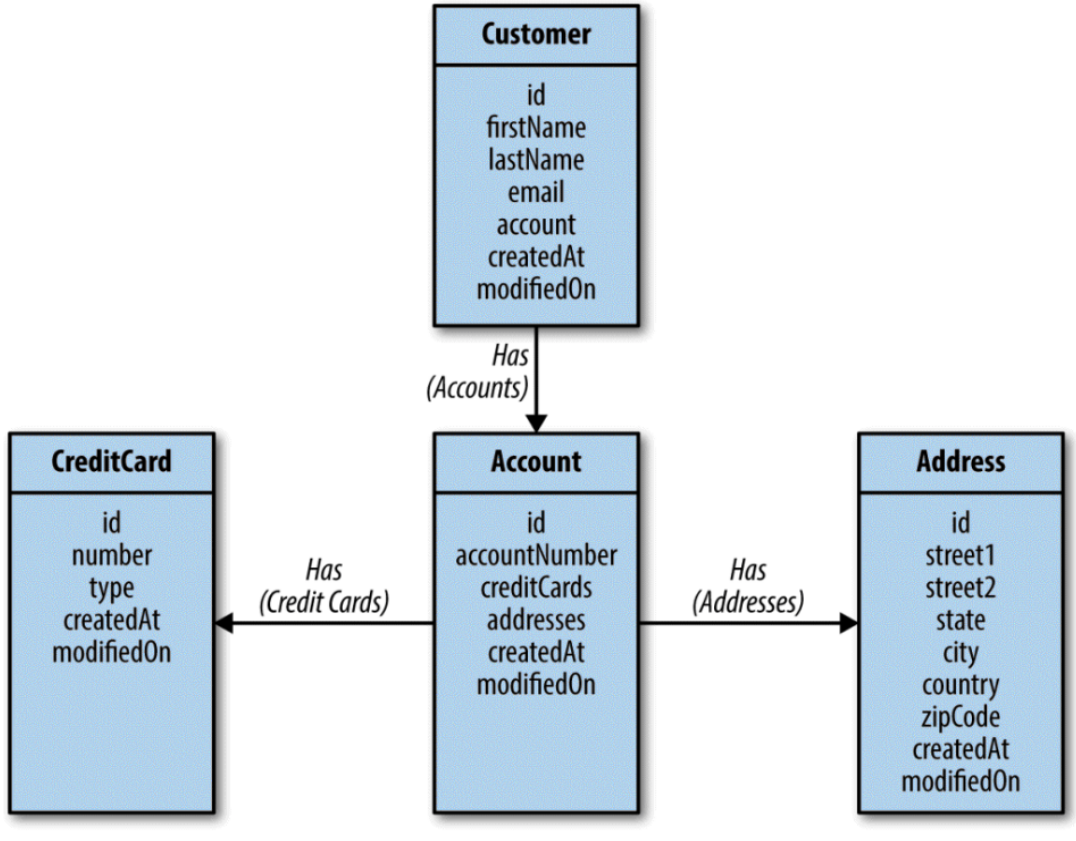
- To activate test profile override active profile:

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = AccountApplication.class)
@ActiveProfiles(profiles = "test")
public class AccountApplicationTests {
    ....
}
```

The AccountContext



The DomainModel



The Account Entity

```
@Entity
@Data
@NoArgsConstructor @AllArgsConstructor
public class Account extends BaseEntity {
 @Id @GeneratedValue(strategy = GenerationType.AUTO)
 private Long id;

 private String accountNumber;

 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
 private Set<CreditCard> creditCards = new HashSet<>();

 @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
 private Set<Address> addresses = new HashSet<>();
}
```

JPA Auditing

- record the creation and last-modified time with base Entity

```
@Data
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public class BaseEntity {

 @CreatedDate
 private Long createdAt;

 @LastModifiedDate
 private Long lastModified;

}
```

- Apply the @EnableJpaAuditing to the ApplicationClass to enable auditing.

Choosing Aggregates

- Account and Customer are the aggregate entities
- all other entities live and die when account, customer live and die
- Create repositories around Account and Customer

AccountRepository

```
package demo.account;

import org.springframework.data.repository.PagingAndSortingReposi

public interface AccountRepository extends
 PagingAndSortingRepository<Account, Long> {

}
```

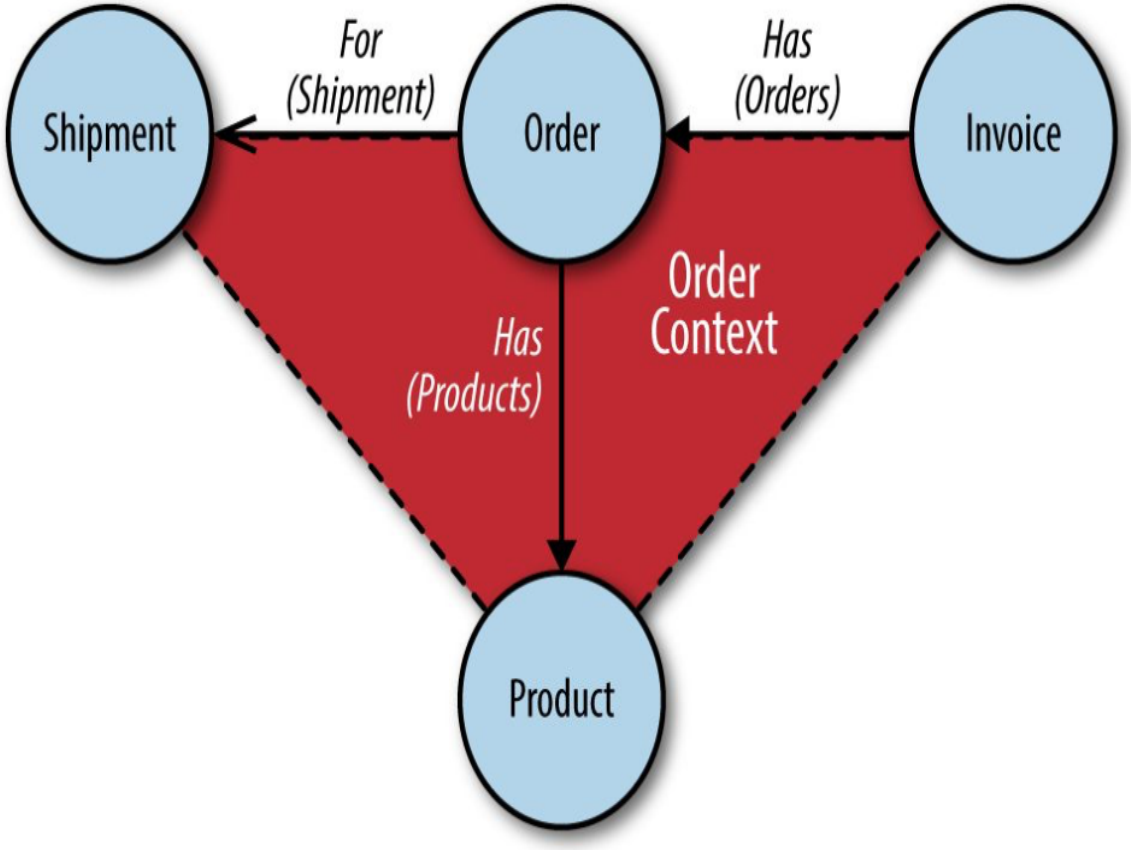
CustomerRepository

```
package demo.customer;

import org.springframework.data.repository.PagingAndSortingReposi

import java.util.Optional;

public interface CustomerRepository extends
 PagingAndSortingRepository<Customer, Long> {

 Optional<Customer> findByEmailContaining(String email);
}
```

Spring Data MongoDB

- documented oriented database
  - records stored as hierarchical json-like documents

The OrderContext



Inventory of behaviour 1

Behaviour	Domain classes
order is placed for a set of quantified products	Order, Product
shipment is delivered on address for an order	Shipment,Order, Address
invoice is created for set of orders for an account	Invoice, Order, Account

Inventory of behaviour 2

- In DDD -> important to describe behaviour of actor in terms of domain objects

@Document

- Spring Data MongoDB uses @Document to indicate that a class represents a document in MongoDB

The Invoice

- invoice is created for set of orders for an account
- when creating a new invoice in the OrderService
  - the invoice should be looked up by accountnumber
  - this number refers back to the Account domain class managed by the AccountService

Why document store

- reason for using document store:
  - lineItems in an order are not allowed to be modified after submission by user
  - the product should represent the product at the time of the order

The Invoice the code

```
@Data @AllArgsConstructor @NoArgsConstructor
@Document
public class Invoice extends BaseEntity {
 @Id private String invoiceId;
 private String customerId;
 private List<Order> orders = new ArrayList<Order>();
 private Address billingAddress;
 private InvoiceStatus invoiceStatus;

 public Invoice(String customerId, Address billingAddress) {...}

 public void addOrder(Order order) {
  order.setAccountNumber(this.customerId);
  orders.add(order);
 }
`
```

The Repository

```
public interface InvoiceRepository extends
 PagingAndSortingRepository<Invoice, String> {

 Invoice findByBillingAddress(Address address);
}
```

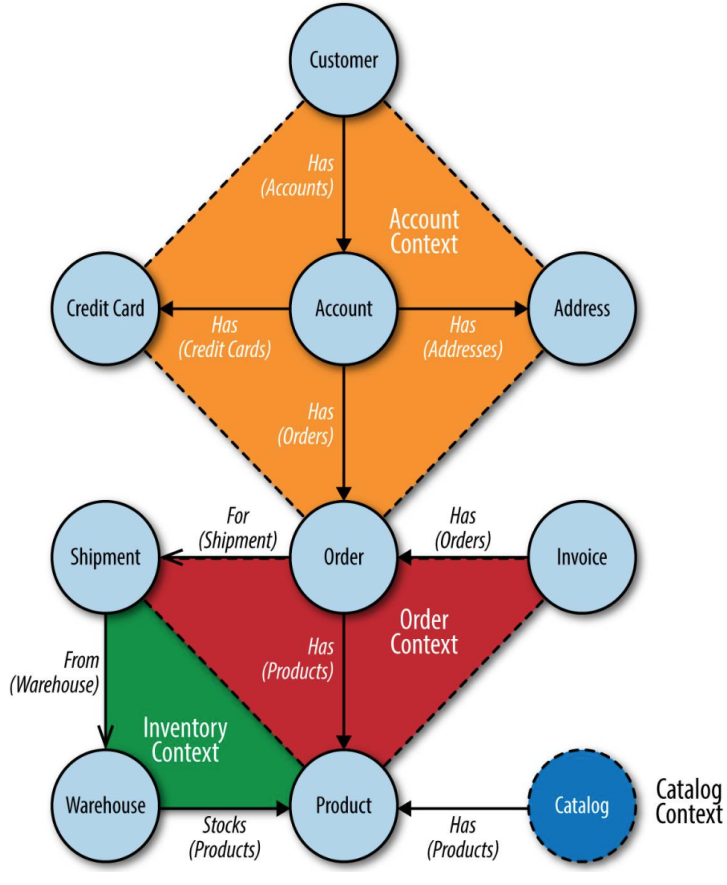
The Order Domain class details 1

- Invoice refers to Order

  1. An Order is placed for a number of quantified products
  2. An Order is delivered via a Shipment to a delivery address

- At 1: -> introduce a LineItem concept
- LineItem -> (product, quantity ordered)
- product will refer to remote reference of product living in other service

The domain


Customer
Has
(Accounts)
Account
Context
Credit Card
Account
Address
Has
(Credit Cards)
Has
(Addresses)
Has
(Orders)
Shipment
For
(Shipment)
Order
Has
(Orders)
Invoice
Order
Context
From
(Warehouse)
Has
(Products)
Inventory
Context
Warehouse
Product
Catalog
Catalog
Context
Stocks
(Products)
Has
(Products)

The Order Domain class details 2

The Order class

```
@Data @NoArgsConstructor @AllArgsConstructor
@Document
public class Order extends BaseEntity {
 @Id  private String orderId;
 private String accountNumber;
 private OrderStatus orderStatus;
 private List<LineItem> lineItems = new ArrayList<>();
 private Address shippingAddress;

 public Order(String accountNumber, Address shippingAddress) {
  //..setAddressType(AddressType.SHIPPING)..OrderStatus.PENDING;
 }
 public void addLineItem(LineItem lineItem) {
  this.lineItems.add(lineItem);
 }
`
```

LineItem

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class LineItem {

 private String name, productId;

 private Integer quantity;

 private Double price, tax;

}
```

Auditing with MongoDB

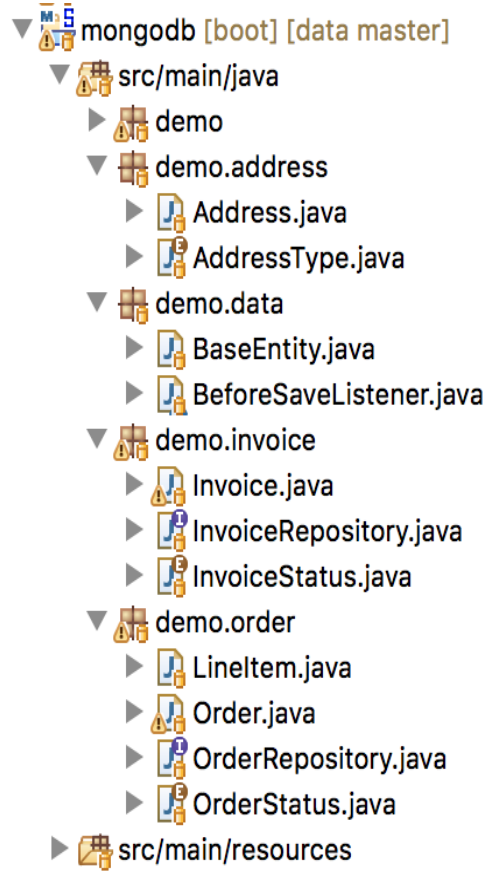
- Introduce a common baseclass

```
@Data
public class BaseEntity {

 private DateTime lastModified, createdAt;
}
```

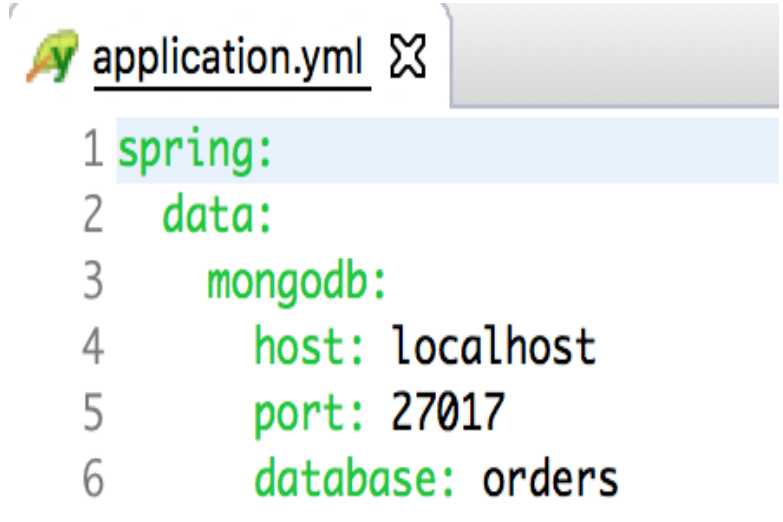
Spring Data MongoDB infrastructure

```
@Component
class BeforeSaveListener extends
                        AbstractMongoEventListener<BaseEntity> {
 @Override
 public void onBeforeSave(BeforeSaveEvent<BaseEntity> event) {

  DateTime timestamp = new DateTime();

  if (event.getSource().getCreatedAt() == null)
   event.getSource().setCreatedAt(timestamp);

  event.getSource().setLastModified(timestamp);

  super.onBeforeSave(event);
 }
`
```

package structure



application.yml



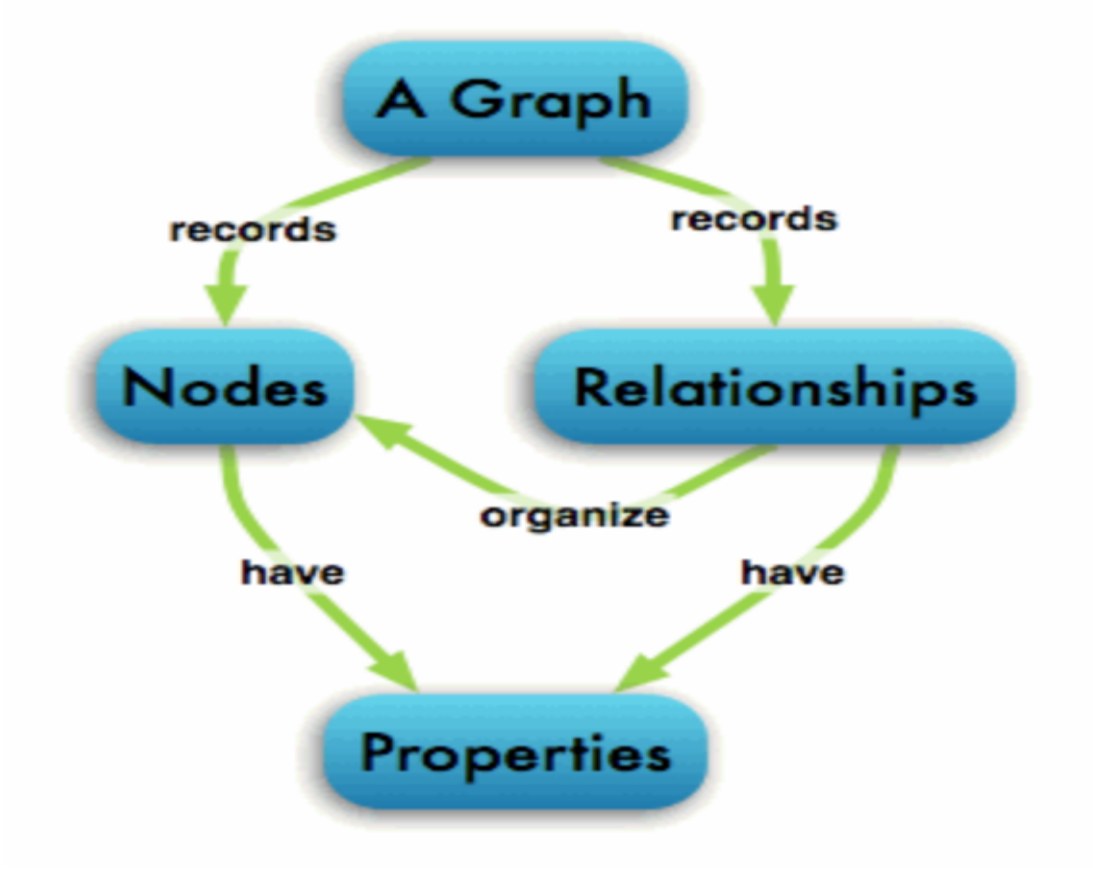
Spring Data Neo4j

- a popular graph database

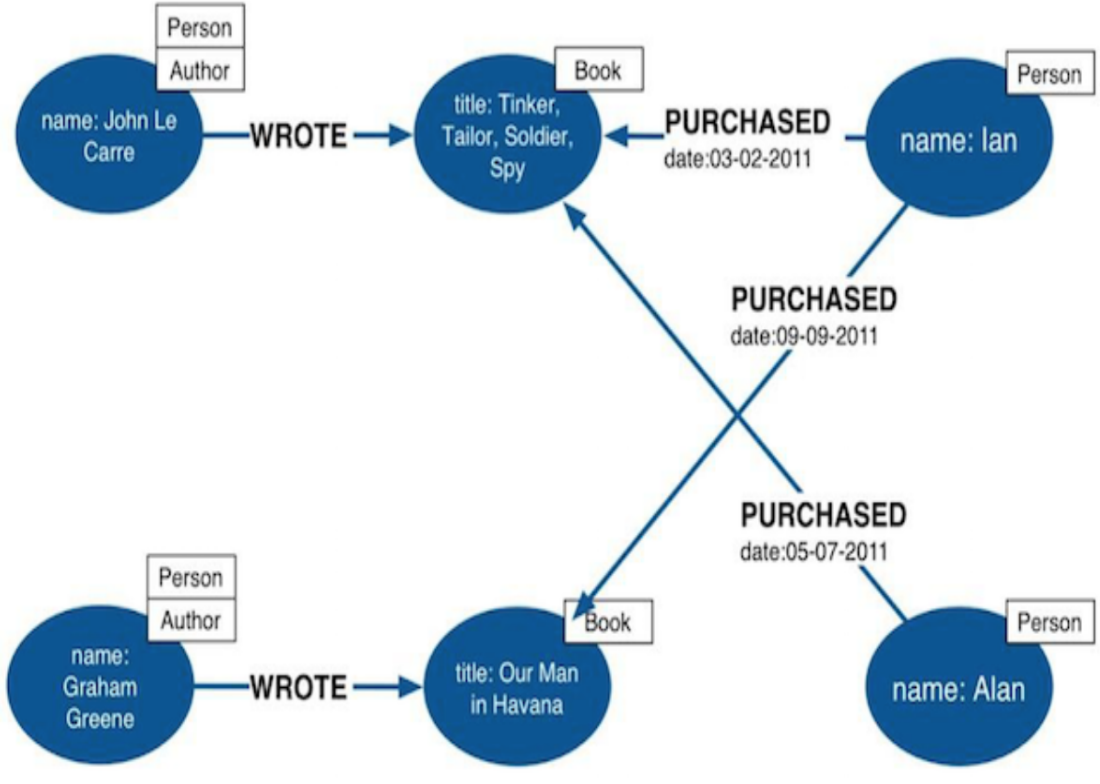
Graph database fundamentals

- can store any kind of data
- using a few simple concepts:
- Nodes - graph data records
- Relationships - connect nodes
- Properties - named data values

Fundamentals in a picture



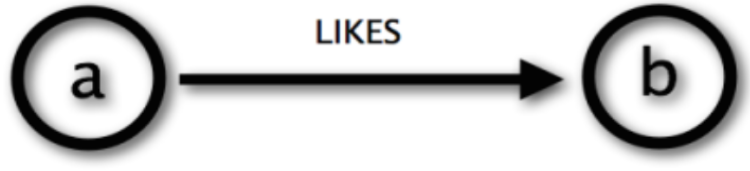
An example graph



Cypher

- Neo4j's query language

Cypher using relationship 'likes'



Cypher

```
(a) -[:LIKES]-> (b)
```

More about Cypher

- Usually, labels of the node are provided to distinguish between entities like (p:Person)
- use a pattern like (person:Person)-->(thing:Thing)
  - allows to access properties like person.name and thing.quality

A Cypher examples

```
MATCH (node:Label) RETURN node.property

MATCH (node1:Label1)-->(node2:Label2)
WHERE node1.propertyA = {value}
RETURN node2.propertyA, node2.propertyB
```

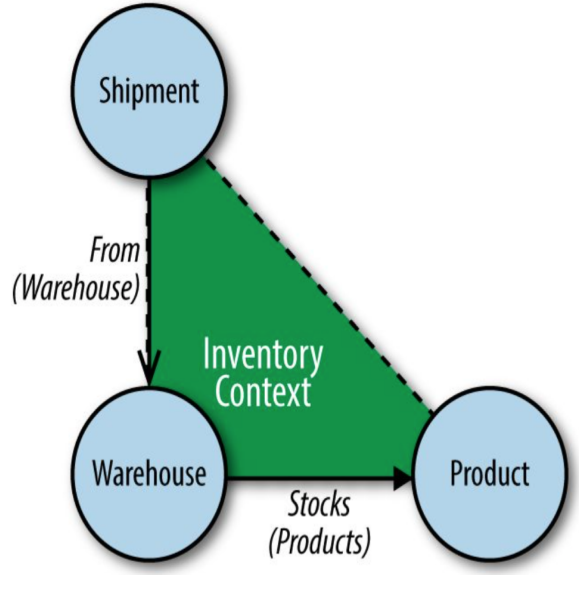
Create InventoryService with neo4j

- What do we want?
  1. design a graph model for the inventory
  2. implement model as neo4j nodes
  3. create repositories to manage nodes

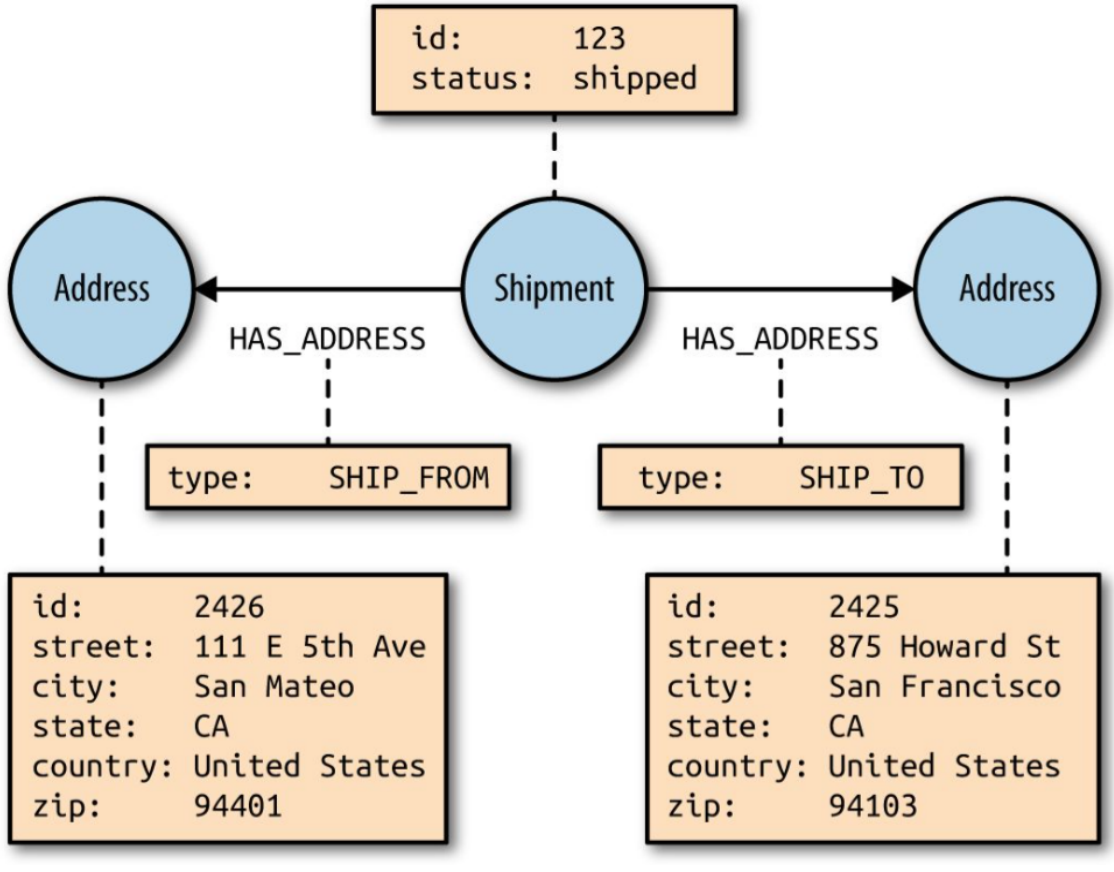
About the business Context

- The catalog contains product behaving differently
  - some products are sold the year round others are seasonal
- products need to be connected to warehouses that are located in dfferent countries
- create shipments from warehouses that are closest to the delivery location

The inventoryBoundedContext



The design


id: 123
status: shipped
Address
Shipment
Address
HAS_ADDRESS
HAS_ADDRESS
type: SHIP_FROM
type: SHIP_TO
id: 2426
street: 111 E 5th Ave
city: San Mateo
state: CA
country: United States
zip: 94401
id: 2425
street: 875 Howard St
city: San Francisco
state: CA
country: United States
zip: 94103

Cypher query1

- return the address of shipment wit id 11

```
MATCH (shipment:Shipment)-[r:HAS_ADDRESS]->(address)
WHERE shipment.id=11
RETURN address
```

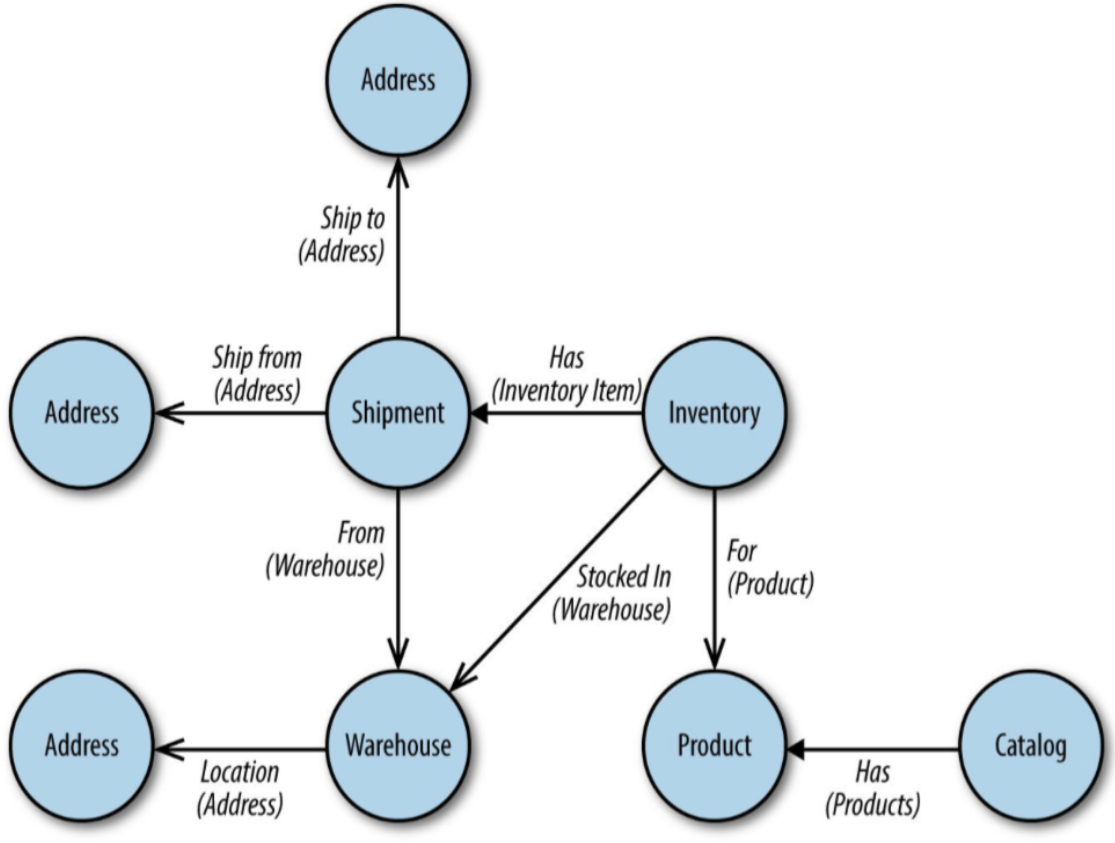
- returns 2 addresses -> add extra property to relation r

Cypher query2

- return the address of shipment wit id 11

```
MATCH (shipment:Shipment)-[r:HAS_ADDRESS{type:"SHIP_FROM"}]->(add
WHERE shipment.id=11
RETURN address
```

Comple model



The Address domain class

```
@Data @NoArgsConstructor @AllArgsConstructor
@NodeEntity
public class Address {
 @GraphId
 private Long id;
 private String street1, street2, state, city, country;
 private Integer zipCode;

 public Address(String street1, String street2, String state,
  String city, String country, Integer zipCode) {
  ...
 }
}
```

The Wharehous domain class

```
@Data @NoArgsConstructor @AllArgsConstructor
@NodeEntity
public class Warehouse {
 @GraphId
 private Long id;
 private String name;

 @Relationship(type = "HAS_ADDRESS")
 private Address address;

 public Warehouse(String n, Address a) { ...}
 public Warehouse(String name) {...}
}
```

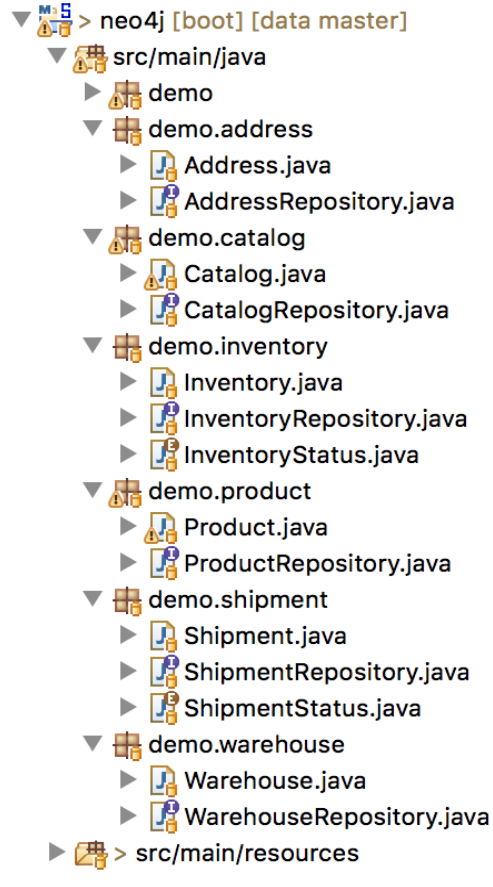
How much is in stock?

```
@Data @NoArgsConstructor @AllArgsConstructor
@NodeEntity
public class Inventory {
 @GraphId private Long id;
 private String inventoryNumber;

 @Relationship(type = "PRODUCT_TYPE", direction = "OUTGOING")
 private Product product;

 @Relationship(type = "STOCKED_IN", direction = "OUTGOING")
 private Warehouse warehouse;
 private InventoryStatus status;

 public Inventory(String inventoryNumber, Product product,
                  Warehouse warehouse,InventoryStatus status){..}
}
```

The Shipment Domain class

```
@Data @NoArgsConstructor @AllArgsConstructor
@NodeEntity
public class Shipment {
 @GraphId private Long id;
 @Relationship(type = "CONTAINS_PRODUCT")
 private Set<Inventory> inventories = new HashSet<>();
 @Relationship(type = "SHIP_TO")
 private Address deliveryAddress;
 @Relationship(type = "SHIP_FROM")
 private Warehouse fromWarehouse;
 private ShipmentStatus shipmentStatus;

 public Shipment(Set<Inventory> inventories, Address delAddress,
  Warehouse fromWarehouse, ShipmentStatus shipmentStatus) {...}
}
```

Package structure



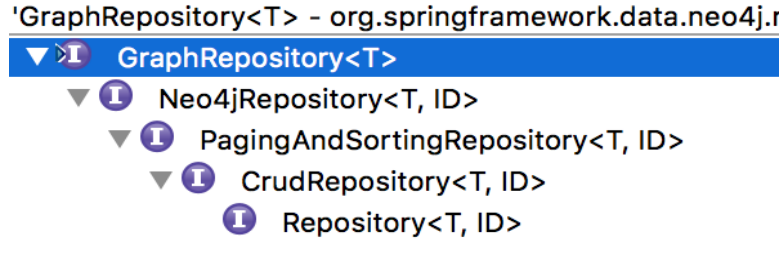
Package structure

*application.yml

```
#spring:
#  neo4j:
#    host: ${SPRING_NEO4J_HOST:192.168.99.100}
#    port: ${SPRING_NEO4J_PORT:7474}
spring:
  data:
    neo4j:
      password: geheim
      username: neo4j

```

A repository

```
import org.springframework.data.neo4j.repository.GraphRepository;

public interface InventoryRepository
                        extends GraphRepository<Inventory> {
}
```



Spring Data Redis

- key value store
- open source NoSQl database

About Redis

- categorized as a key value store
- explained as a in-memory key datastructure store
- allows to store different kinds of complex datastructures as value

Supported datastructures by Redis

- Strings
- Lists
- Sets
- Hashes
- Sorted Sets
- Bitmaps/HyperLogLogs

A distributed data store

- Redis can be distributed
- multiple applications can operate on values with the same key
- Redis provides an API to operate on the supported datastructures
  - operations can be applied atomically
  - results in greater transactional granularity

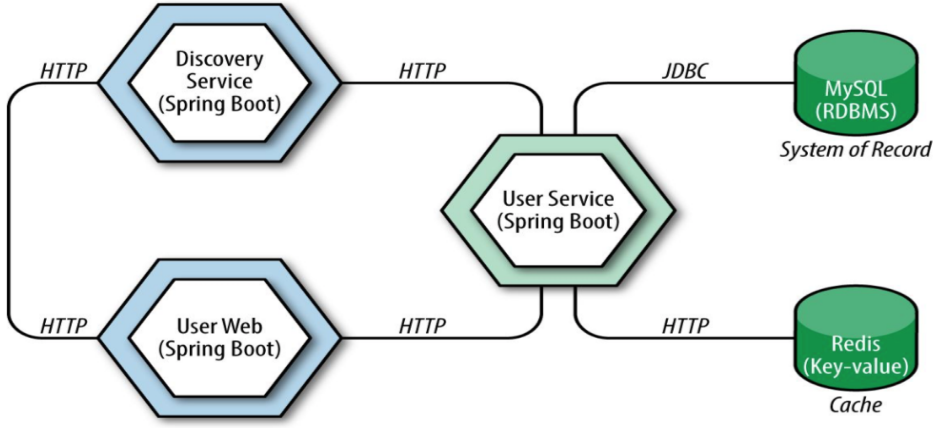
Other uses Redis

- also used for:
  - interprocess communication
  - and messaging
- can be a message broker
  - supports publish/subscribe
- caching

Redis for Caching

- most popular use case for Caching
- Spring Data Redis implements Spring CacheManager abstraction
  - eases centralized caching of records in Micro Service

The architecture



How it works

- records will be stored and persisted in the mysql db
- when a record is requested by other service the response is cached by replicating the record in Redis server
  - available -> in memory for fixed expiration time before being cached again

Some help from Spring boot

- will autoconfigure RedisTemplate
- dependency spring-boot-starter-data-redis enables CacheManager when @EnableCaching is used

Setup of UserService

```
@Service
public class UserService {

 private final UserRepository userRepository;

 @Autowired
 public UserService(UserRepository userRepository) {
  this.userRepository = userRepository;
 }
 // code continues in next slides...
}
```

@CacheEvict

- evicts the user record with the supplied id from the redis cache

```
@CacheEvict(value = "user", key = "#user.getId()")
public User createUser(User user) {

    User result = null;

    if (!userRepository.exists(user.getId())) {
      result = this.userRepository.save(user);
    }

    return result;
}
```

@Cacheable

- gets a user record from cache or put a user record from mysql in the redis cache

```
@Cacheable(value = "user")
public User getUser(String id) {
   return this.userRepository.findOne(id);
}
```

@CachePut

- evicts user record from cache and replace it with updated value

```
@CachePut(value = "user", key = "#id")
public User updateUser(String id, User user) {

    User result = null;

    if (userRepository.exists(user.getId())) {
        result = this.userRepository.save(user);
    }
    return result;
}
```

@CacheEvict

- evicts user record with the supplied id

```
@CacheEvict(value = "user", key = "#id")
public boolean deleteUser(String id) {

   boolean deleted = false;

   if (userRepository.exists(id)) {
      this.userRepository.delete(id);
      deleted = true;
   }

   return deleted;
}
```

Details

- the @Cache annotations use SpEl to access the key from parameter list of method

Warning

- When using Redis as a cache for multiple different kinds of services
  - the key is can't be suffixed with a namespace
  - when there is a name collision it is possible to evict an i.e. Account based on an id instead of the User