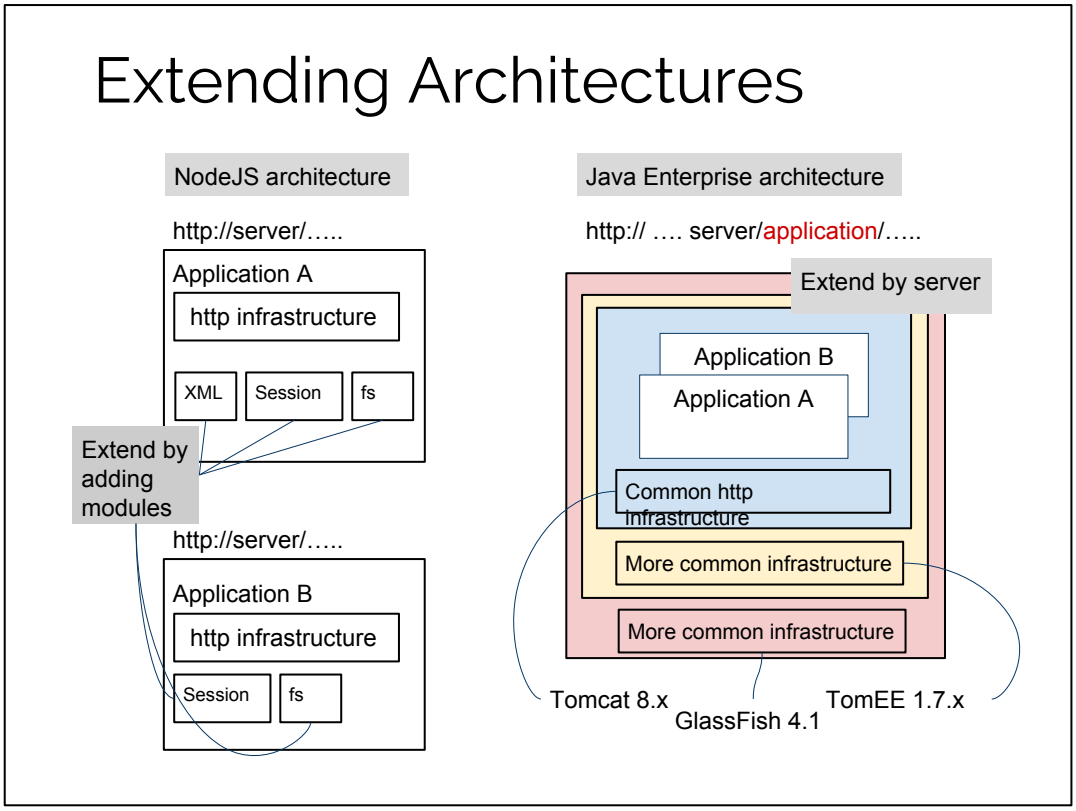
Web Applications 2017

Week 2, Slides 2

Content

- Extending Architectures
- Persistence
- Object Relational Mismatch
- Persistence Stack
- Application Persistence Layer
- Object Relational Mapping
- Java Persistence API
- Sequelize (Express ORM)
- Advanced ORM
- Enterprise Javabeans
- Query APIs



How to extend the capabilities of applications

- JEE: Also possible to extend by modules (add libraries). But tedious, simpler just to change application server.

Persistence

Persistent object: Object that outlives the execution of the program

- Have to store for later retrieval (next execution)
- We only use relational databases

Readings

- Persistence

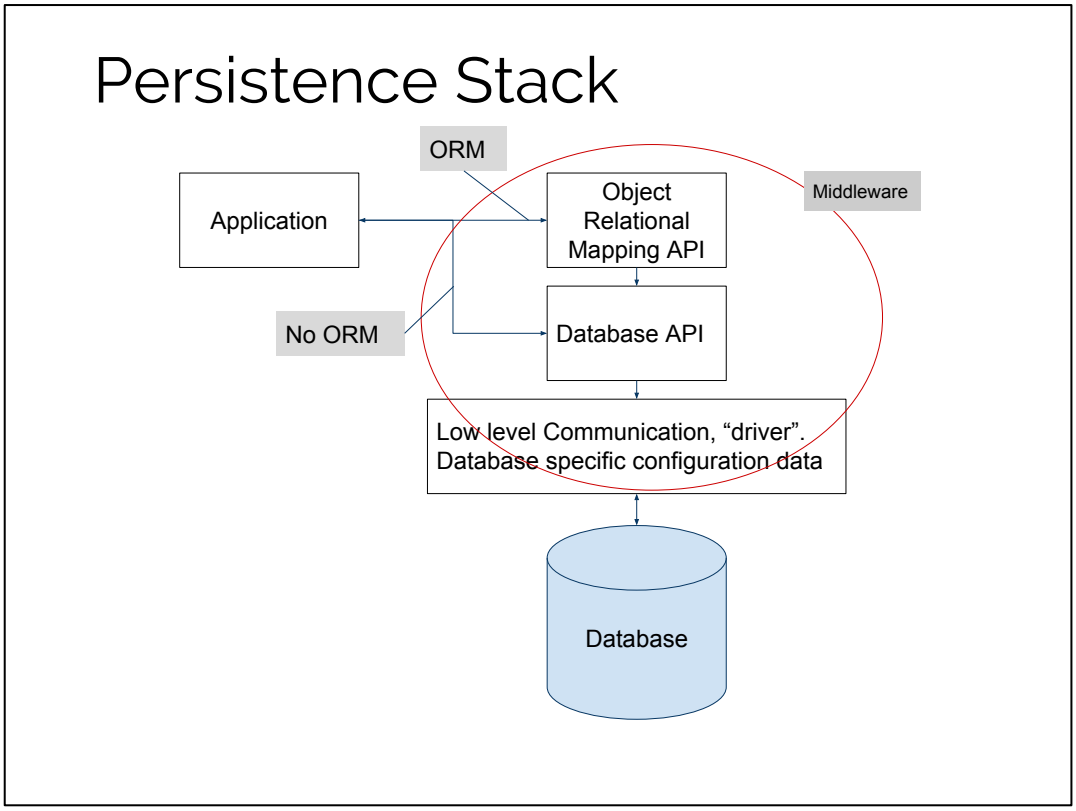
Object Relational Mismatch

Relational databases and object orientation doesn't fit!

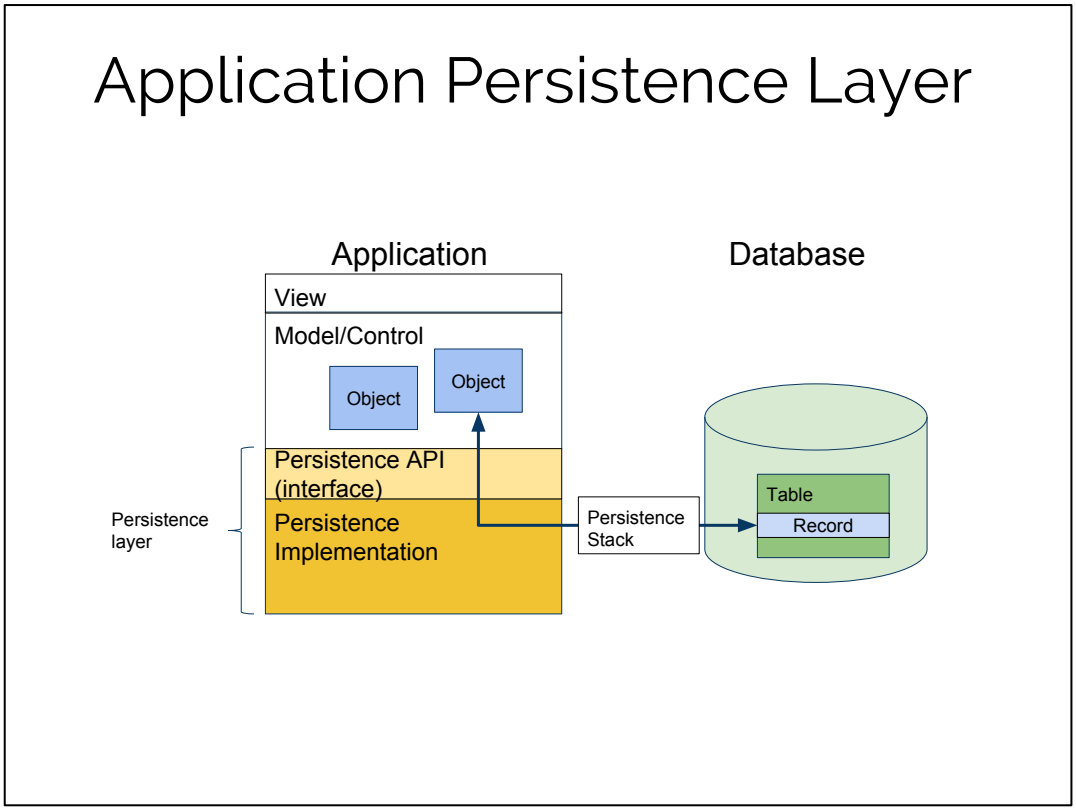
- Object orientation: Objects
- Relational databases: Records

Readings

- Object-relational impedance mismatch (links at bottom)



ORM = Object Relational Mapping



Persistence is a service, handled by a persistence layer

Readings

- Intro to JEE persistence layer
- Layered Application Guidelines (Microsoft)
- Transaction processing!

No ORM API

Data Access Object (DAO). JEE Design pattern

```
// Java database connectivity, JDBC
// Data format: "id;name"
public interface IDAO {
    public List<String> getPersons() throws SQLException;
    public String getById(Long id) throws SQLException;
    public void add(String person) throws SQLException;
    public void delete(Long id) throws SQLException;
    ...
}
```

Basic CRUD interfaces

```
// Node Persistence API NOTE: Asynchronous!
// Data format: {"id":2,"name":"olle"}
PersonList.prototype = (function() {
    return {
        getPersons: function(callback) { },
        getById: function(id, callback) { },
        add: function(name, callback) { },
        delete: function(id, callback) { },
        ...
    }
})();
```

Java using JDBC

- Get strings back …. Problematic…!

Express using mysql module

- Get JSON …

Readings

- CRUD

No ORM Implementation

Databases

- MySQL
- Derby/JavaDB (bundled with NetBeans)
- … use any you like

Middleware

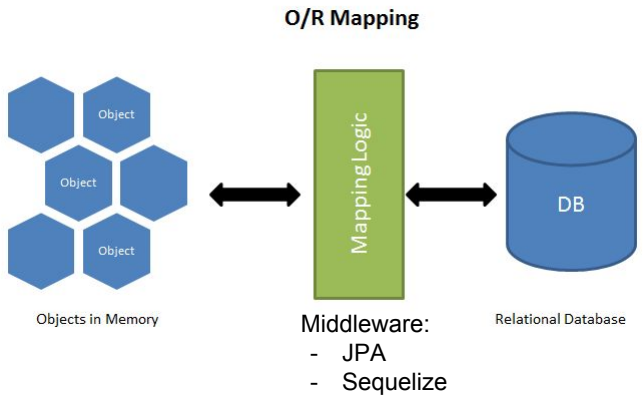
- JEE: A database specific driver (derby client driver) and JDBC API.
- Node: Database specific npm module (mysql) with API
- Configuration

Aside

Most of samples show are from JEE but the tasks, issues and problems are the same for any platform!

-

Handling the OO Mismatch



Add middleware to map between objects and records

Readings
- ORM
- ORM in detail

JEE ORM API

```
// JEE OO Persistence API (Data Access Object, DAO)
public interface IDAO {
    public List<Person> getPersons() throws SQLException;
    public String getById(Long id) throws SQLException;
    public void add(Person person) throws SQLException;
    public void delete(Long id) throws SQLException;
    ...
}
```

```
// Generic DAO API
public interface IDAO<T, K> {
    List<T> get();
    T getById(K id);
    void add(T data);
    void delete(K id);
    ...
}
```

Java Persistence API (JPA)

Standard Java API for doing ORM (any database)

- Entity classes, mapped to tables
- Entity Identity, mapped to keys
- Relationships, foreign keys
- Inheritance and Polymorphism
- Transactions
- Queries (SQL, Java Query Language)

Used internally by our DAO's

ORM specification for Java. Many implementations.

Readings

- Java Persistence API, JPA 2.x
- JPA Reference
- Getting started with JPA
- JPA Tutorial

Natural vs Surrogate Keys

Natural keys. A natural key is one or more existing data attributes that are unique to the business concept.

Surrogate key. Introduce a new column, called a surrogate key, which is a key that has no business meaning.

Use any or both

Readings

- Natural or Surrogate
- The great primary key debate

JPA Entity Class

Mandatory

```
@Entity
public class Book implements Serializable {

     @Id
     private Long id;      // No final!
     private String title;
     private Float price;
     private String description;

     protected Book() {  // Must have non-private
     }

     public Book(String title, Float price,
                                       String description) {
          this.title = title;
          this.price = price;
          this.description = description;
     }

     ...
}
```

An **entity class** is a Java class typically representing a database table

- Must be listed in config file persistence.xml (in a “persistence unit” more later…)

Readings

- Entity classes

JEE Keys

Database generated surrogate key

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

Using inner PK-class for natural key

```
@EmbeddedId
private PK id;

@Embeddable  // Must have default ctor, equals, hashCode
public static class PK implements Serializable {
    @Column(name = "BOOK_TITLE")
    private String title;
    @Getter
    private String author;

    public PK(String title, String author) {
    this.title = title;
    this.author = author;
    }
}
```

EntityManager

```
// javax.persistence.Persistence
EntityManagerFactory emf = Persistence
         .createEntityManagerFactory("person_pu");

EntityManager em = emf.createEntityManager();

// Use em for CRUD
em.find(Person.class, person.id);
em.persist(person);
em.merge(person);
em.remove(person.id);

// Many more ….
```

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" … >
   <persistence-unit name="person_pu"
            transaction-type="RESOURCE_LOCAL">
      <provider>org.eclipse.persistence.jpa….</provider>
      <class>orm.model.Person</class>
      <exclude-unlisted-classes>true</exclude-unlisted-classes>
      <properties>
         <property name="javax.persistence.jdbc.url"
               value="jdbc:derby://localhost:1527/persons"/>
         ...
      </properties>
   </persistence-unit>
</persistence>
```

API to handle Entity classes, persist, find, delete, etc.

- Used in conjunction with a persistence unit (config file)

Readings

- EntityManager

Transactions

JEE

- **JDBC**, When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed
- **JPA**, Write operation need explicit transaction demarcation (not if using EJB's more later)

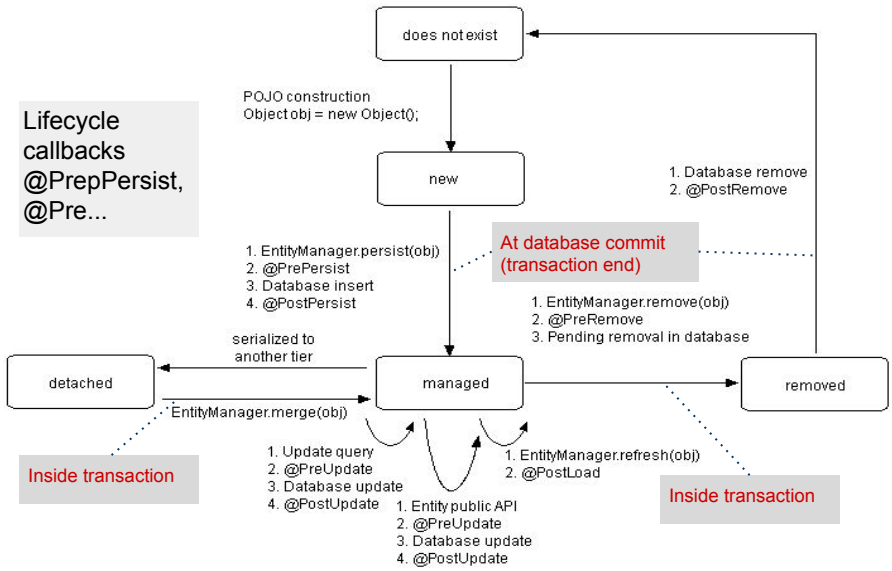
Node

- **mysql**, seems to be in auto commit mode(?) but manually possible
- **Sequelize**, auto commit or managed transactions

Readings

- JDBC Transactions
- JPA transactions (application managed)
- Sequelize transactions

Entity Class Instance Lifecycle



Reading

- [Working with JPA Entity Objects]

Customization

```
@Entity
@Table(
    name = "AvancedBook", // Default is BOOK
    uniqueConstraints={@UniqueConstraint(
        columnNames={"TITLE"})} // Just for demo,
                                // of course title may be same
)
public class Book implements Serializable {

    @Id
    private Long id;
    @Column(name = "book_title", nullable = false,
        updatable = false)
    private String title;
    private Float price;
    @Column(length = 50)
    private String description;
    ...
```

As usual JEE offers very many customization options

Readings

- Explanation of annotations

Collections and Enums

```
@Entity
public class Book implements Serializable {

    // Will create extra table Tag
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "Tag" )
    private List<String> tags;

    public enum Genre {
        HORROR,
        MISANTHROPY,
        ROMANCE
    };
    // Will end up in same table
    @Enumerated(EnumType.STRING)
    private Genre genre;

    …
```

Attributes as Collections or Enums no problems

- Collections in separate table (relationship created)
- Enums in same table

Embeddable

Any class

```
@Embeddable
public class Review implements Serializable {

    @Temporal(TemporalType.DATE) // Must have temporal for dates
    private Date reviewDate;
    private String reviewText = "No review yet";

    …
}
```

In entity class

```
@Embedded
private Review review;
```

Embeddable will end up in same table

Sequelize (Express ORM)

```
// A Sequelize model. Define mappings to table
// Promised based!
var Person = sequelize.define('Persons', {
    id: {
        type: Sequelize.BIGINT,
        autoIncrement: true,
        primaryKey: true
    },
    name: {
        type: Sequelize.STRING
    }
}, {
        timestamps: false
});

/*
    API for Sequelize models

    build, save, findAll, findById, findOne, update, delete, ...
*/
```

API works with model instances.

Readings

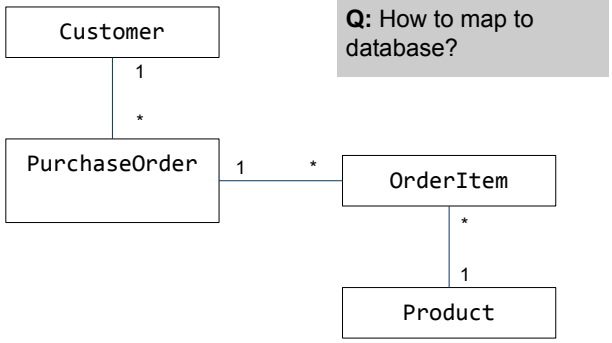
- Sequelize
- Sequelize in practice

Sequelize Customization

```
// Sample from Sequelize page
var Foo = sequelize.define('foo', {
 // instantiating will automatically set the flag to true if not set
 flag: { type: Sequelize.BOOLEAN, allowNull: false, defaultValue: true},

 // default values for dates => current time
 myDate: { type: Sequelize.DATE, defaultValue: Sequelize.NOW },

 // setting allowNull to false will add NOT NULL to the column, which means an
error will be
 // thrown from the DB when the query is executed if the column is null. If you
want to check that a value
 // is not null before querying the DB, look at the validations section below.
 title: { type: Sequelize.STRING, allowNull: false},

 // Creating two objects with the same value will throw an error. The unique
property can be either a
 // boolean, or a string. If you provide the same string for multiple columns,
they will form a
 // composite unique key.
 someUnique: {type: Sequelize.STRING, unique: true},
 uniqueOne: { type: Sequelize.STRING,  unique: 'compositeIndex'},
 uniqueTwo: { type: Sequelize.INTEGER, unique: 'compositeIndex'}

...
```

Many options … see definitions.

Advanced ORM



Q: How to map to database?

A: Must define mappings 1:1, 1:N, N:1, M:N on objects

Readings

- Sequelize Associations
- JPA Mappings

JEE Mappings

Mappings PurchaseOrder

```
@OneToMany
@JoinColumn(name = "PURCHASE_ORDER_ID")
private List<OrderItem> items;

@ManyToOne
@JoinColumn(name = "CUSTOMER_ID", nullable = false)
private Customer customer;
```

Also

```
@OneToOne
@ManyToMany
```

Reading

- JPA Wikibooks

JEE Cascade, Fetching and OrphanRemoval

Cascade and Fetch

```
public class Publication ... {

    // No cascade on remove!
    @ManyToOne(cascade = {CascadeType.PERSIST,
        CascadeType.MERGE}, fetch = FetchType.LAZY)
    private Book book;
    @ManyToOne(cascade = {CascadeType.PERSIST,
        CascadeType.MERGE}, fetch = FetchType.LAZY)
    private Author author;
    …
}
```

OrphanRemoval

```
@OneToMany(orphanRemoval = true)
private Set<AppUser> members = new HashSet<>();
```

Readings

- Cascade
- Fetch
- Orphan removal

JEE Inheritance and Generics

Inheritance

```
@MappedSuperclass
public class Person implements Serializable { … }

@Entity
public class Employee extends Person { … }
```

Generic Entity

```
@Entity
public class GenericEntity<T extends Serializable>
                                  implements Serializable {

    private T genericAttribute;
    …
}
```

JPA will handle generic classes and inheritance

Different strategies

- Single table for hierarchy (all super/sub-objects in same table)
- Joined strategy, many tables
- .. more. (see links i previous slides)..

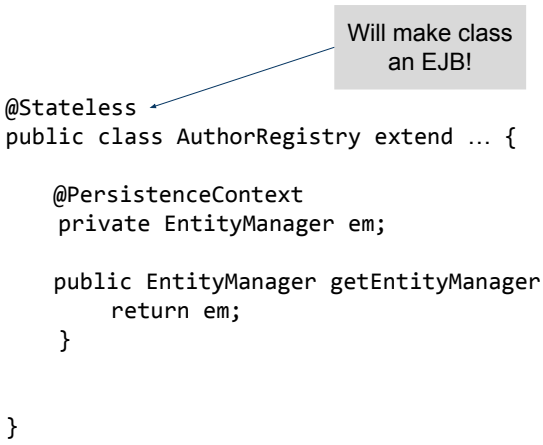
Enterprise JavaBeans EJB

This is more of enterprise application with lots of transactions.

Readings

- Enterprise JavaBeans EJBs

Stateless EJB Sample



Transactional aware objects.

Readings

- EJB tutorial

Context and Dependency Injection

```
@Inject
ProductCatalogue pc;

@Inject
CustomerRegistry cr;

@Inject
OrderBook ob;

@Inject
UserTransaction utx;

@PersistenceContext
private EntityManager em;
```

And much more ...

Objects created and delivered “for free” (not transactional aware)

Readings

- CDI
- CDI reference

Queries

JEE

- Standard Query API obtained from EntityManager
  - … use with raw SQL
  - … or Java Persistence Query Language (JPQL), objectified version of SQL
  - …or Criteria API, typesafe, but too ugly …
- Third party higher level query API's
  - Querydsl, typesafe query DSL
  - uaiCriteria

Prefer

JEE Query API

Standard API + native SQL

Single result, typed query

```
// em is EntityManager
Customer customer = em
    .createQuery("select * from Customer where id = 123",  Customer.class)
    .getSingleResult();
```

Collection result, typed query

```
String sql = "select * from Manufacturer where state = 'CA'";
List<Manufacturer> ms = em
               .createNativeQuery(sql, Manufacturer.class)
               .getResultList();
```

Bulk updates

```
int nRows = em.createNativeQuery("update Customer …. ").executeUpdate();
int nRows = em.createNativeQuery("delete from  …. ").executeUpdate();
```

JEE Query Parameters

Standard API + native SQL

Positional Parameters

```
String name = ...;
List<Customer> cs = em.createQuery(
    "select * from Customer where name = ?1", Customer.class)
    .setParameter(1, name)
    .setMaxResults(20)
    .getResultList();
```

Named Parameter

- Not supported for native SQL

Querydsl

```
// em is EntityManager
JPAQueryFactory cf = new JPAQueryFactory(em);
QManufacturer m = QManufacturer.manufacturer;
List<Manufacturer> result = new ArrayList<>();
for( String s : states ){
     result.addAll(cf.selectFrom(m)
                     .where(m.state.eq(s)).fetch());
}
```

Generated meta model classes

```
JPAQueryFactory cf = new JPAQueryFactory(em);
QProduct p = QProduct.product;
BigDecimal r =  cf.from(p).select(p.purchaseCost.sum()).fetchOne();
double d = (r != null) ? r.doubleValue(): 0;
```

Sequelize Queries

Sequelize

- sequelize.query(… raw SQL …)
- Object based API

Raw SQL

```
sequelize.query("SELECT * FROM `users`", { type:
sequelize.QueryTypes.SELECT}).then(function(users) {
     ...
})
```

Object based API

```
Post.findAll({
    where: sequelize.where(sequelize.fn('char_length',
                        sequelize.col('status')), 6)
});
```

Readings

- Object based API