Quantum Satis

a model for building JakartaEE systems

"Neither too much nor too little"

"Without exaggeration"

"Suitable size"

"Enough"

Content

What is "Quantum Satis"?

QS is a model for how JakartaEE applications can be structured and how they communicate.

As the name suggests, everything is a bit "just right", there are influences from DDD (Domain Driven Design), Microservice, Repository Pattern, boilerplate code, CRUD and transaction management but QS compromises in several places with the "rules" that other models have. A typical example is that a deployed QS system has a database, not one for each microservice.

You can think of it as QS is a "backend-backend system", i.e. the back part of an ERP system in an application server. The code that is NOT covered by QS is the unique business logic. However, there are validation methods in QS that should be overridden in implementations of QS as well as the classes that control when updating the database, so all updates will be done in the same structured way.

QS is not just a model, but has a number of tools to automate the generation of the classes needed. In the description of the model, reference is sometimes made to these tools.

QS is developed and tested with Netbeans 20, Azul Java 17, Payara 6.2024.5, Eclipse as a JPA implementation and the database manager Informix 14.10

Current status

The model is documented (in this document), working java code for AmUserBase and its entities, class for rest communication, and façade generation for local and remote interfaces and packaging to client jarfiles. All java code is compilable and can be deployed and tested. However, the Java code needs to be analyzed and corrected since I haven't coded commercially since 2015, my code shouldn't go to production without review 😊

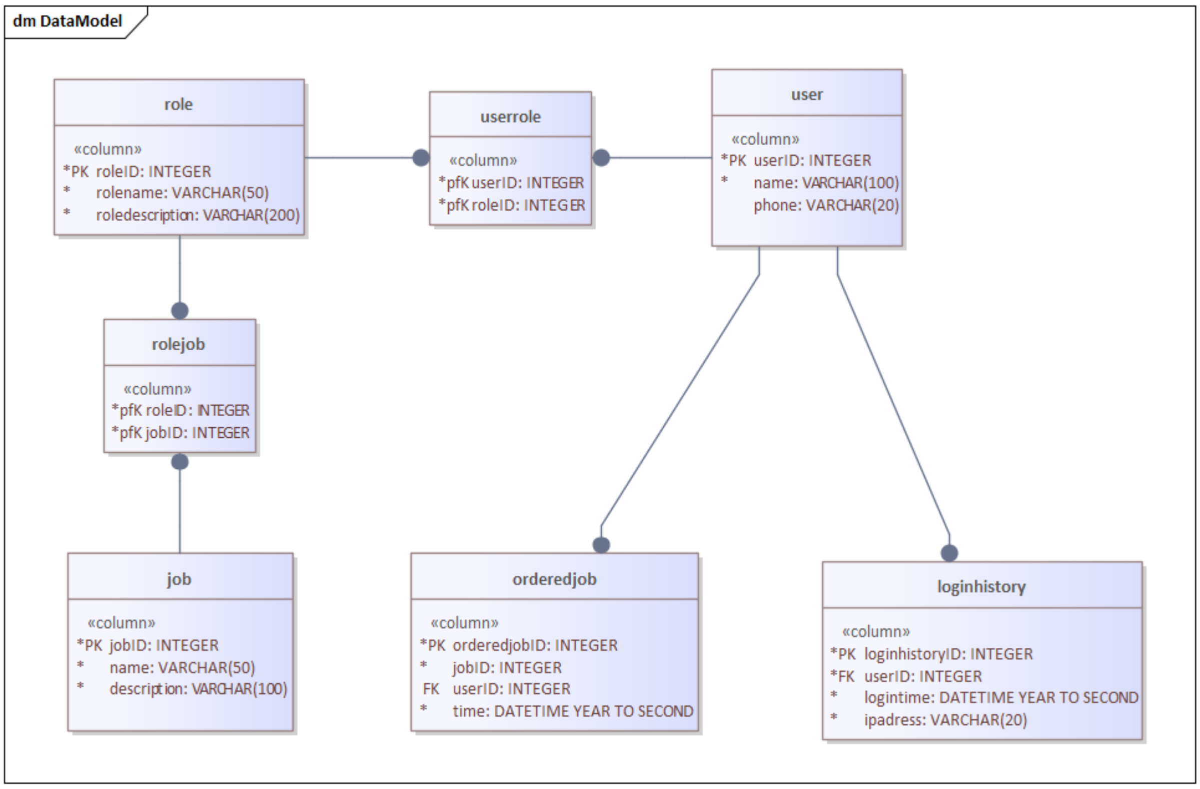
A Swing application to register the entity model is ready. The database schema for the entity model is available for Informix 14.10. (Only common database types are used, so it's easy to migrate to other databases.)

What remains is the code generation that will create all classes, package structure, etc. The code generation will use Apache Velocity.

From data model to services

QS is based on a data model that is implemented via rules to a number of stateless session beans.

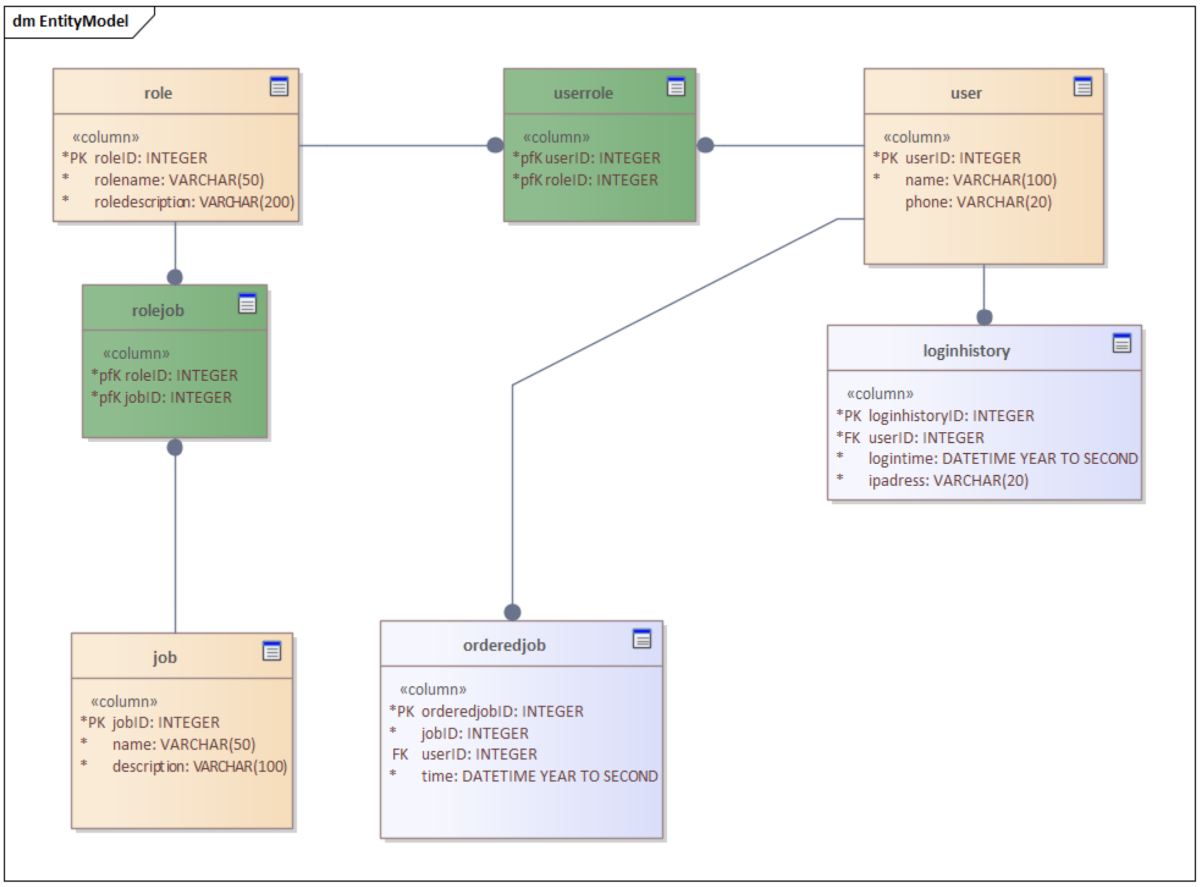
Data model



The content of this document and the description of QS are based on the data model above. The model shows part of a system where users and roles for permissions are registered. The user can have a number of roles. The user can order different jobs that are associated with a role the user has. For the jobs that have been ordered, the user can see the status, etc. Login history is also saved for the user.

It is not important which attributes are present in the different objects, what is important is that the objects exist and their relationships.

Entity model



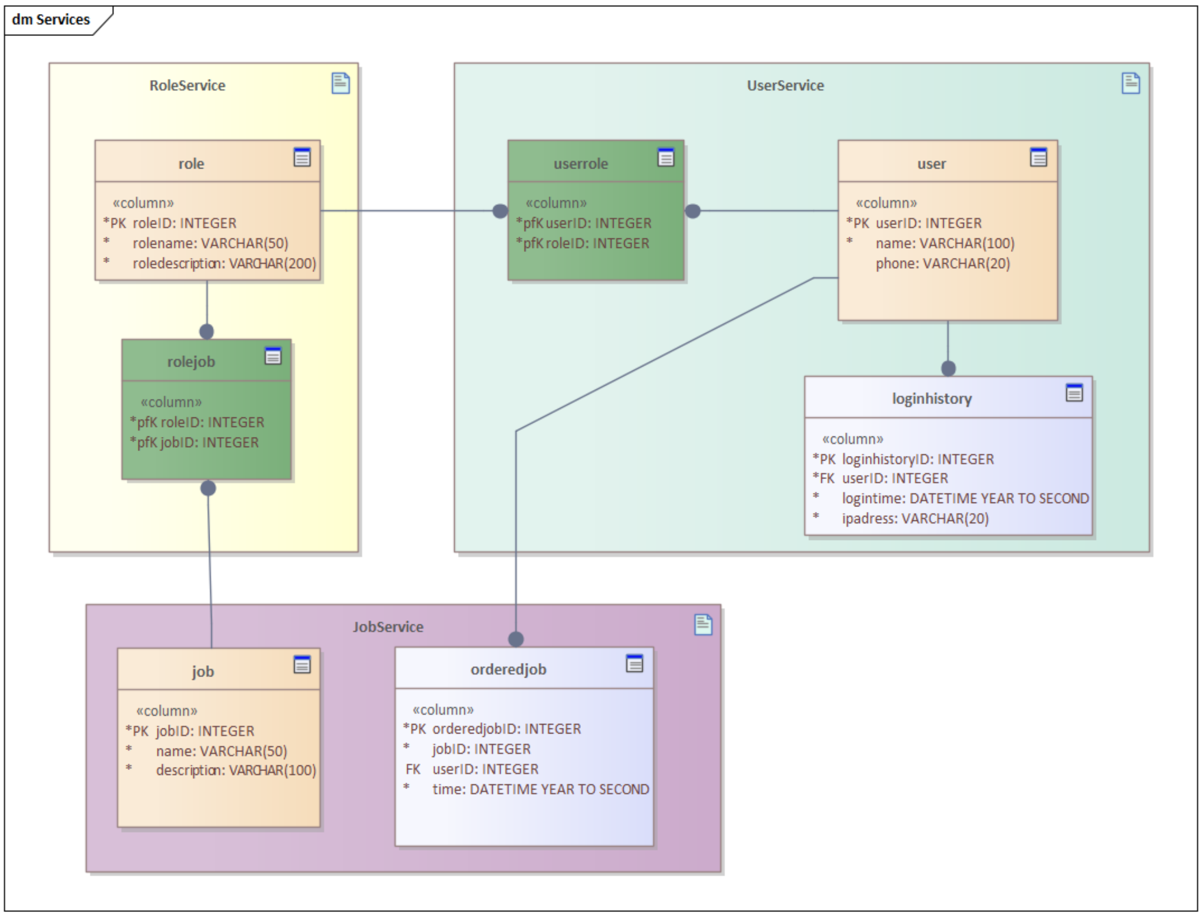
From the data model, an entity model is created.

The user, role, and job entities are selected to be primary entities. This means that those entities and their relationships will exist in class names that start with Am (access module). Logic for managing the user entity and its relationships will be in the AmUser class as well as the AmUserBase. An Am class is to some extent equivalent to an Aggregate in Domain Driven Design.

Rolejob and Userrole are relational tables and do not have a corresponding entity but are defined in user/role and role/job with @ManyToMany annotation.

Services

QS is based on the entity model being divided into logical parts, service, these services correspond to a Stateless Session Bean and are thus a deployable unit and the smallest unit that is managed.



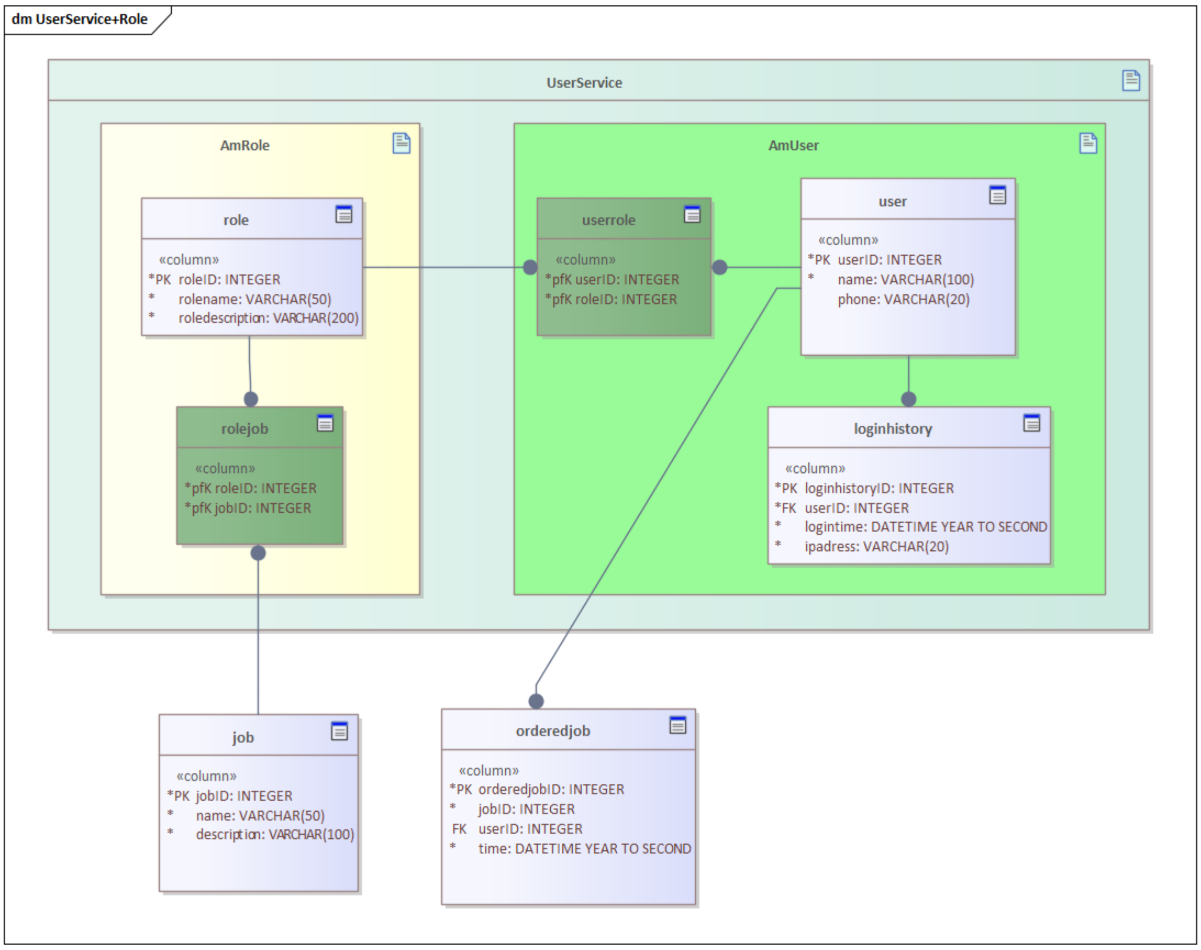
In the picture, the model has been divided into three services:

- UserService
- RoleService
- JobService

It is possible to register a service that contains more than one primary entity, see more under Packaging.

The userrole and rolejob relational entities need to be placed together with the primary entity that will contain validations and database joins for the entity. In this example, userrole has been placed together with user.

The examples in this document are based on UserService with the two primary entities user and role in the service to show a more complex packaging. Java code is available for UserService.


dm UserService+Role
UserService
AmRole
AmUser
role
«column»
*PK roleID: INTEGER
* rolename: VARCHAR(50)
* roledescription: VARCHAR(200)
rolejob
«column»
*pfK roleID: INTEGER
*pfK jobID: INTEGER
userrole
«column»
*pfK userID: INTEGER
*pfK roleID: INTEGER
user
«column»
*PK userID: INTEGER
* name: VARCHAR(100)
phone: VARCHAR(20)
loginhistory
«column»
*PK loginhistoryID: INTEGER
*FK userID: INTEGER
* logintime: DATETIME YEAR TO SECOND
* ipadress: VARCHAR(20)
job
«column»
*PK jobID: INTEGER
* name: VARCHAR(50)
* description: VARCHAR(100)
orderedjob
«column»
*PK orderedjobID: INTEGER
* jobID: INTEGER
FK userID: INTEGER
* time: DATETIME YEAR TO SECOND

Overarching principles

Entity Name

QS adds the prefix Eb before the table name as the name of the entity. The user table thus gets an entity called EbUser. Mapping to the correct table name is done with the annotation @Table(name = "user").

For guest entities, the Guest suffix is also added, so a table named role that is included as a guest entity in a service is named EbRoleGuest.

This increases the readability of the code.

Entities

The QS code generator generates code for all entities registered for a service based on the registered entity model. No code needs to be written by developers for entities. The entity that is considered to be the center/master of the service is marked as primary and becomes the entity that becomes the governing entity for the generation of classes for CRUD, etc.

Guest entities

When an entity needs to appear in a service where it should not be according to the entity model, Role in our case as seen from UserService, it can do so by being a "guest" in that service. A guest cannot be updated, only read, and will then cause the persistence handler in the UserService to see that the relationship between the UserRole and the Role is fulfilled correctly and will update the database. The number of attributes included in an entity when it is a guest should be limited as no business rules can be implemented in the entity to protect certain attributes. It is appropriate to include the attributes corresponding to the foreign-key as well as attributes to make it easier to understand what the occurrence represents, such as a description or a name.

A major advantage of this, instead of removing the foreign key by using two databases, is that the database is always consistent and that ID concepts between Role and UserRole are guaranteed to be correct and identical.

Guest entities don't have navigation to other guest entities. When the jobid column is included in the EbOrderedjobGuest guest entity, the getJobid method will return an Integer with the key, *not* the EbJobGuest entity! To retrieve the name of a job, the findJob (pID) class in AmJobBase needs to be called from, for example, AmUser. The code generator will create the method.

Databases

In Microservice and Domain Driven Design, for example, it is advocated that each module should have its own database in order to isolate problems to the same area and to achieve scalability. For QS, it is the requirements of the business that govern, but that a common database is preferable. There *may* be multiple application servers using the same database, and there *may* be application servers within the same larger system using their own databases. The advantage of having a database for all code executed in an application server is that it is possible to use the database's foreign-key to strengthen relationships between objects.

Primary keys

The only primary keys that QS supports are @SequenceGenerator the annotation. The corresponding column in the database must be of type Integer. Composite primary keys, UUIDs, and key values from tables in the database are NOT implemented in the code generation in QS version 1.

Persistence Manager

QS uses JPA (Jakarta Persistence API) to work against the database. By only including the entities that are directly processed by the main object in the service (e.g. UserService) as well as the related guest entities, the persistence manager will perceive it as if it sees the entire database, that the database consists only of these entities/tables. Persistence handlers can't manage a relationship with an entity that isn't among the covered entities.

In the example in our entity model, the service UserService can't update the userrole because the entity role doesn't exist, it belongs to the RoleService. The persistence handler will then set the roleid in table userrole attribute to NULL. QS solves this problem by allowing Role to be in the UserService service as a guest. This is possible because there is only one database that is used by both UserService and RoleService.

Transaction management

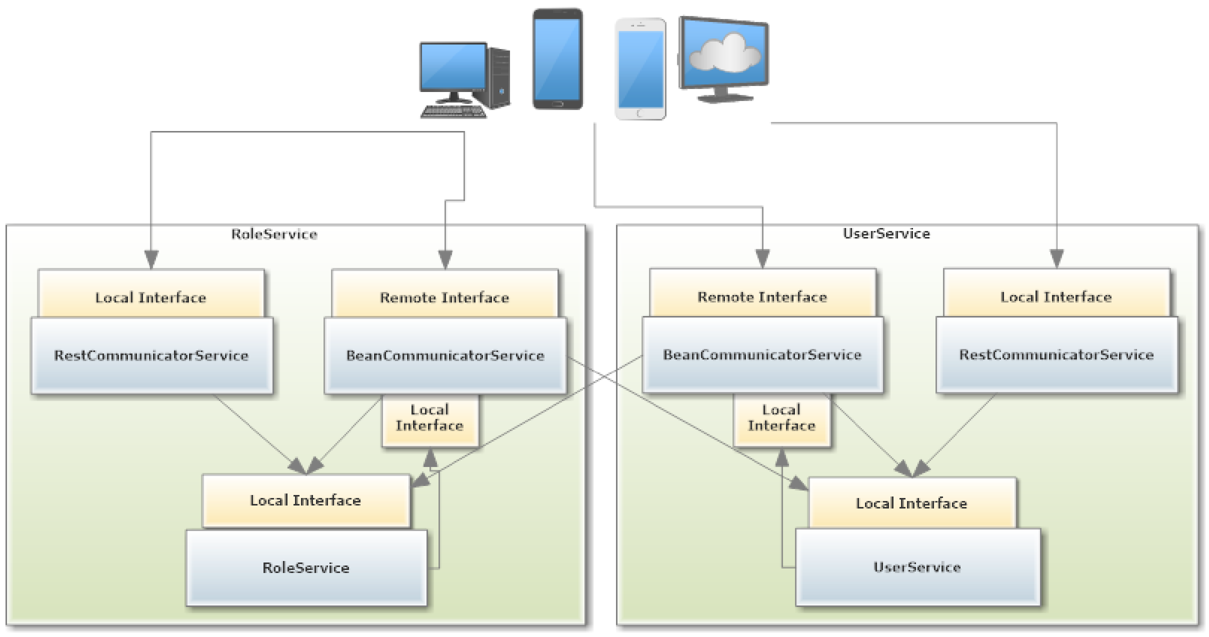
In the same way that the discussion about one or more databases should be used, there is a corresponding discussion about distributed transactions. QS advocates 2-phase commit and Jakarta Transactions with annotations. When the service providers call each other to perform a task, it is a great advantage to be able to trust that all updates are made, that the transaction is ACID (Atomicity, Consistency, Isolation, Durability). This satisfies 2-phase/XA transaction handlers which are the ones found in application servers. Messages and other features may also be included in the transactions. Also, no code needs to be written to compensate for a failed update (SAGA Pattern) and the database is always consistent, which provides high quality information.

Communication

Star Trek Communicator

All communication from the business logic in QS passes through a "Communicator" as the business logic should not know how a call is made. It could be sent as a message or with an RPC call or an internal REST call, it is determined by the implementation in the Communicator which also makes translations between the formats used externally and the formats, DTO, used by QS.

QS has rules for how communication should be done. The picture below shows how two servicemen can communicate with each other and with the outside world. Should an additional mode of communication be added, for example, the Jakarta RPC, the logic for this communication can be placed in a new Communicator session prayer.



As you can see in the picture, REST is implemented on the local interface even though it is for external communication. The reason is that the exposure to available methods of attack is decreasing. If REST had instead been implemented in the Remote interface, the methods would be callable with, for example, CORBA as common façade methods

In each service i.e., UserService, there is at least one stateless session bean for the entity that is primary and its related entities. Every communication bean is also a stateless session bean.

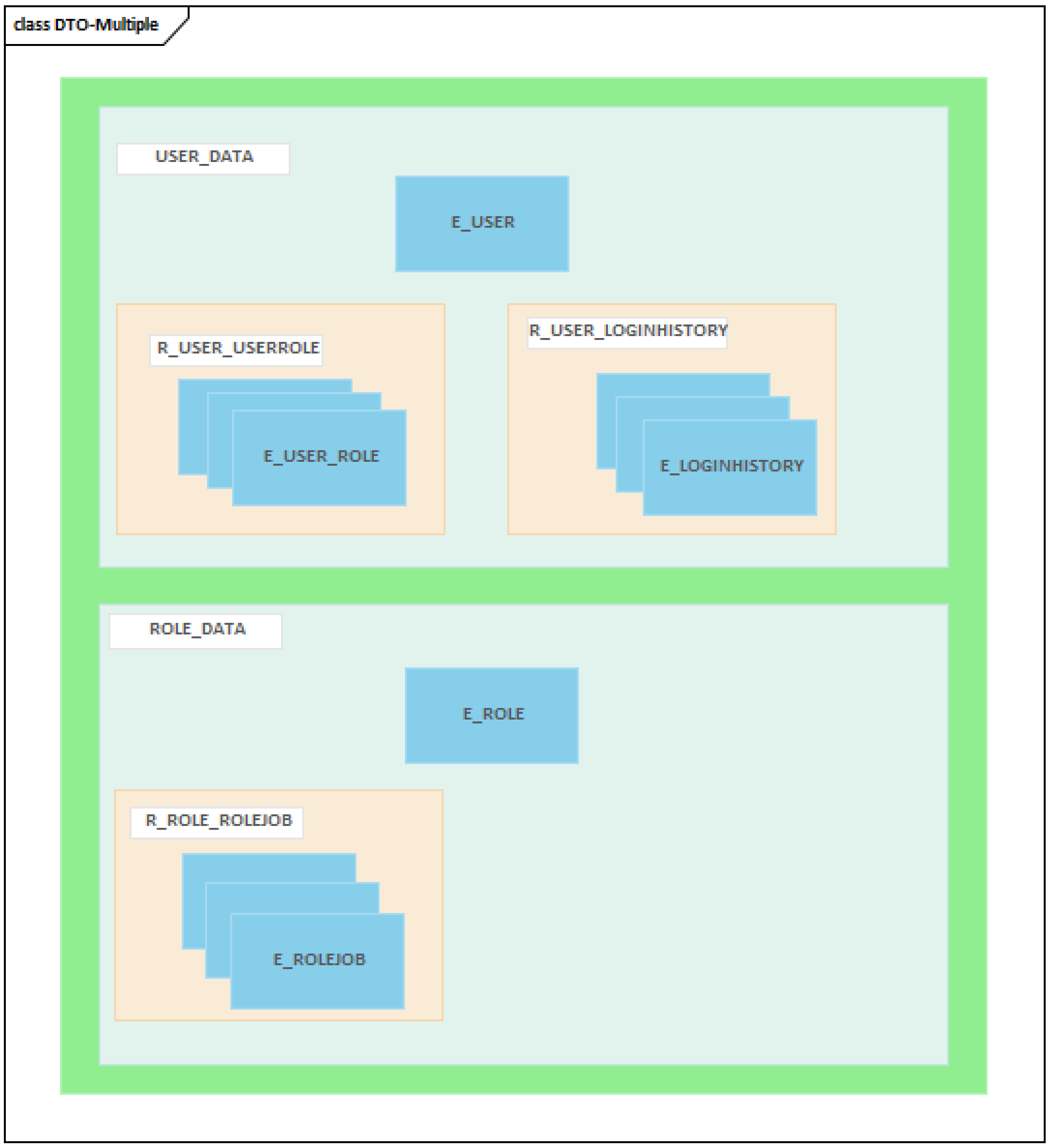
Data Transfer Objects

QS has a specific structure for how data to and from QS should be formatted. Data managed by QS should be in maps. The maps correspond to all the columns in the corresponding table, i.e. all the attributes in the entity. The key in the key/value pairs in the map is the code-generated constant EA_*ENTITY_ATTRIBUTE* and the value is an object. DTO means that no entities should be used by external resources such as clients or other services. These should use DTOs that contain only Java objects. QS does not detach entities. See the Metadata chapter for more information on the constants.

Below is an example of a DTO where a user is updated with name and phone, and a new login is created and a loginhistory is deleted. All of this is done in one call to AmUserBase.write(Map map).

```
// master = outer map
HashMap <String, Object> masterDTO = new HashMap<>() ;
// primary entity map , USER_DATA
HashMap <String, Object> userServiceDTO = new HashMap<>() ;
// Map for user
HashMap <String, Object> e_userDTO = new HashMap<>() ;
// Map for loginhistory
HashMap <String, Object> r_loginhistoryDTO = new HashMap<>() ;

// Populate entity user
e_userDTO.put(UserserviceMetadata.EA_USER_USERID,1);
e_userDTO.put (UserserviceMetadata.EA_USER_NAME, "Mr X");
e_userDTO.put (UserserviceMetadata.EA_USER_PHONE, "12345");

// Populate entity loginhistory
HashMap <String, Object> e_loginhistoryDTO = new HashMap<>() ;
e_loginhistoryDTO.put(UserserviceMetadata.EA_LOGINHISTORY_USERID, 31302);
e_loginhistoryDTO.put(UserserviceMetadata.EA_LOGINHISTORY_LOGINHISTORYID, 103);
e_loginhistoryDTO.put(UserserviceMetadata.EA_LOGINHISTORY_IPADRESS, "200.168.0.1");
e_loginhistoryDTO.put(UserserviceMetadata.EA_LOGINHISTORY_LOGINTIME, LocalDateTime.now());
e_loginhistoryDTO.put(UserserviceMetadata.DTO_STATUS, 0);
r_loginhistoryDTO.put("1", e_loginhistoryDTO);

HashMap <String, Object> e_loginhistoryDTO2 = new HashMap<>() ;
e_loginhistoryDTO2.put(UserserviceMetadata.EA_LOGINHISTORY_LOGINHISTORYID, 104);
e_loginhistoryDTO2.put(UserserviceMetadata.DTO_STATUS, 2);
r_loginhistoryDTO.put("2", e_loginhistoryDTO2);

userServiceDTO.put (UserserviceMetadata.E_USER, e_userDTO);
userServiceDTO.put (UserserviceMetadata.R_USER_LOGINHISTORY, r_loginhistoryDTO);

masterDTO.put (UserserviceMetadata.USER_DATA, userServiceDTO);
```

In our example with UserService, the maximum DTO (Data Transfer Object) that can be handled in a call has this structure:



The UserService session bean map contains maps with the keys from the metadata:

- USER_DATA
- E_USER
- R_USER_USERROLE
    - E_USER_ROLE
- R_USER_LOGINHISTORY
    - E_LOGINHISTORY
- ROLE_DATA
- E_ROLE
- R_ROLE_ROLEJOB
    - E_ROLEJOB

The maps that have a key that starts with R_ are HashSet with maps that contain key/value pairs corresponding to each attribute in the entity. This means that a 1:M relationship will be in a map with the name of the relationship, for example, R_User_Role and that every instance of Role will be in a HashMap in that map.

Based on the contents of the map, the code for CRUD can create entities to update the database.

QS understands how the maps are structured and when information is sent to QS, not all attributes in the DTO need to be present. QS scans the map for every known attribute and if it exists it is used, otherwise continue processing.

QS also has several help methods, which usually have an Integer with the key to an entity as a parameter, see under Classes.

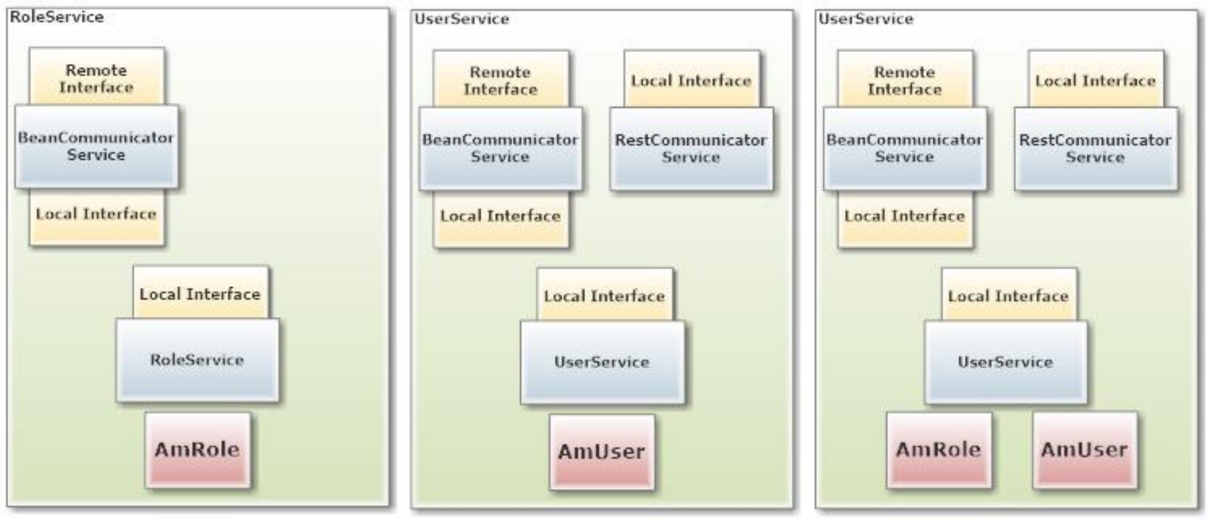
DTO_ STATUS

Each DTO that is a relationship to the primary entity *should have* an attribute DTO_STATUS that can have three different values:

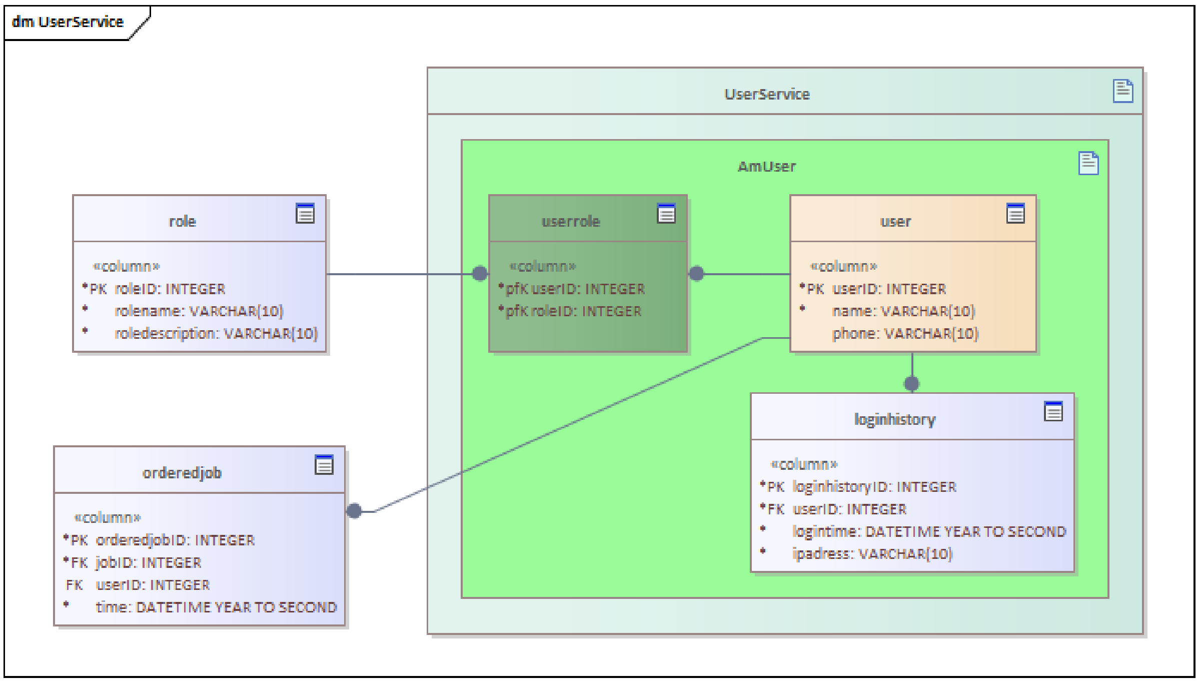
- DTO_STATUS = 0 -> Updated
- DTO_STATUS = 1 -> New
- DTO_STATUS = 2 -> Removed

The primary entity and its related entities are created or updated based on calls to the create and write methods, respectively.

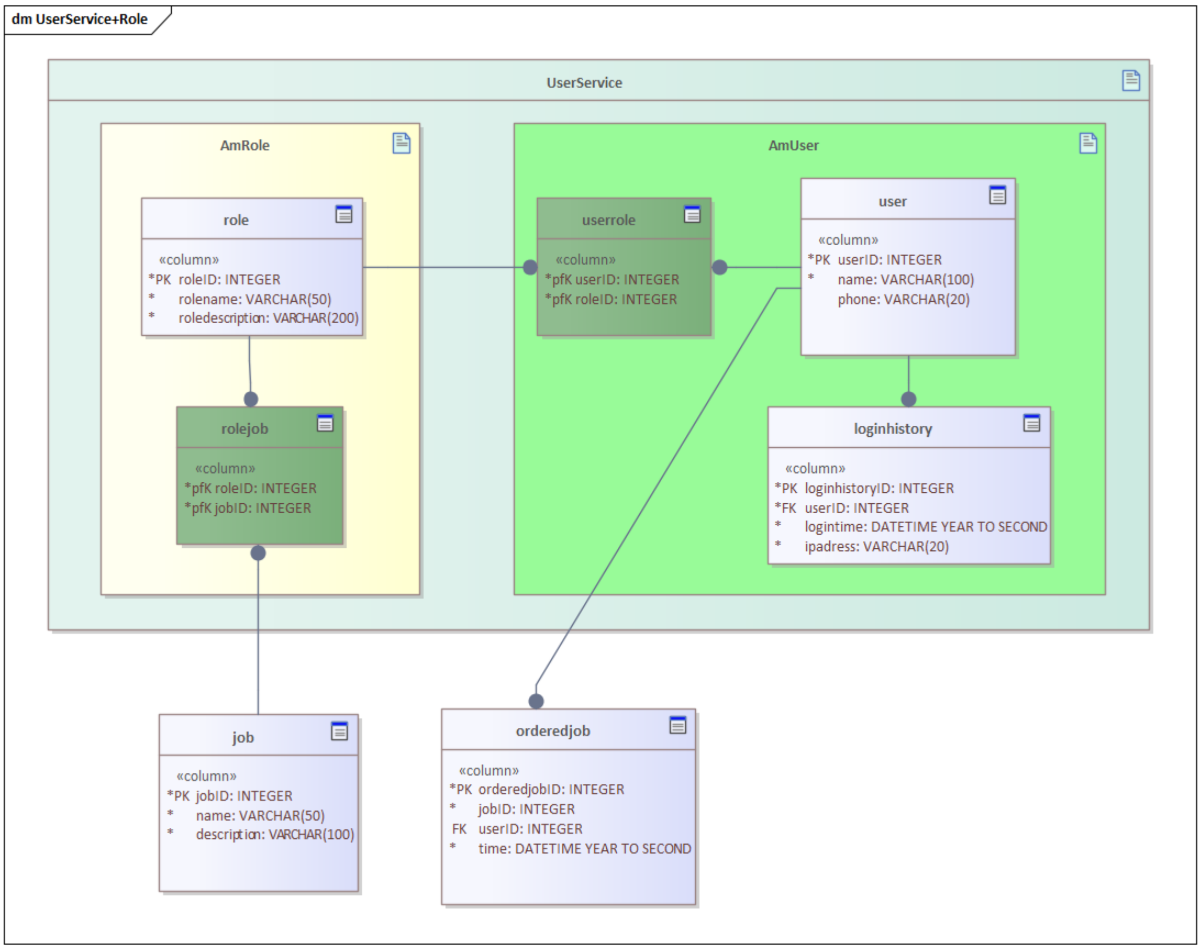
Packaging



There are several ways to package QS. What is most similar to microservicing is packaging a primary entity in a project and generating code for it.



If you choose to include both role and user entities in the same project, code will be generated for both primary entities in the same session bean. The advantage of this is that the call between AmRole and AmUser as well as from other classes in the service is made as pure Java calls and not as local or remote calls between service/session beans.

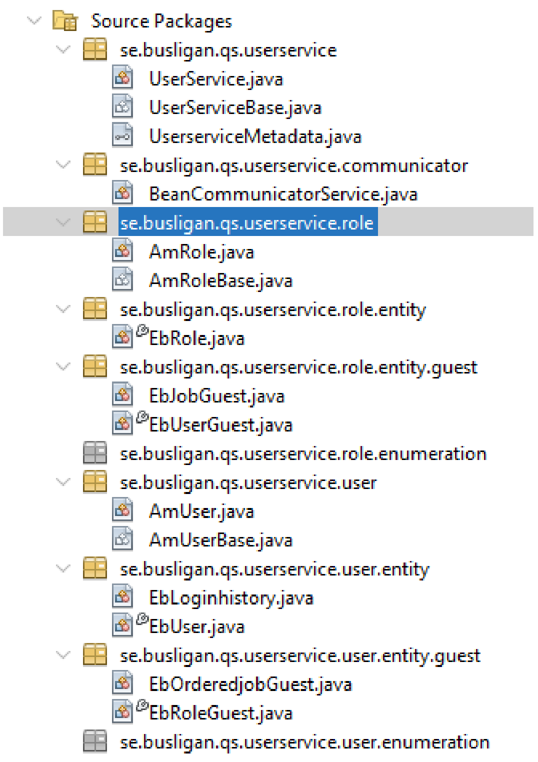


Updates to entities on the yellow area are made via the AmRole/AmRoleBase class and those on the green area via AmUser/AmUserBase. The Am classes are generated because role and user are primary entities. The entities that are in UserService can be updated, and the outside ones are guest entities. There is no relationship between job and orderedjob when these are guest entities. Navigation is not possible between guest entities.

In the image above, the roles and user entities will be included both as primary entities and guest entities. From the AmRole class, @ManytoMany the relationship to the user is perceived as the user being the guest entity to the role. Similarly, the AmUser class perceives that role is a guest entity to the user.

userrole is drawn as if it belonged to User, and this is to show that if there are validation rules for userrole, they are in the AmUser class. The naming convention also means that it would have been better to call the entity roleUser if it had belonged to the AmRole class.

The package structure and classes that QS uses for the image above are:



```
Source Packages
  se.busligan.qs.userservice
    UserService.java
    UserServiceBase.java
    UserserviceMetadata.java
  se.busligan.qs.userservice.communicator
    BeanCommunicatorService.java
  se.busligan.qs.userservice.role
    AmRole.java
    AmRoleBase.java
  se.busligan.qs.userservice.role.entity
    EbRole.java
  se.busligan.qs.userservice.role.entity.guest
    EbJobGuest.java
    EbUserGuest.java
  se.busligan.qs.userservice.role.enumeration
  se.busligan.qs.userservice.user
    AmUser.java
    AmUserBase.java
  se.busligan.qs.userservice.user.entity
    EbLoginhistory.java
    EbUser.java
  se.busligan.qs.userservice.user.entity.guest
    EbOrderedjobGuest.java
    EbRoleGuest.java
  se.busligan.qs.userservice.user.enumeration
```

se.busligan.qs is replaced with the unique domain name, and *userservice* is the name of the project specified in EntityModelBuilder.

Metadata

For each attribute in a constituent entity, a string constant is created with the value of the attribute

Ex: `public static final String A_USER_NAME = "Name";`

For each attribute in an entity, a string constant is created with the value of the entity.attribute

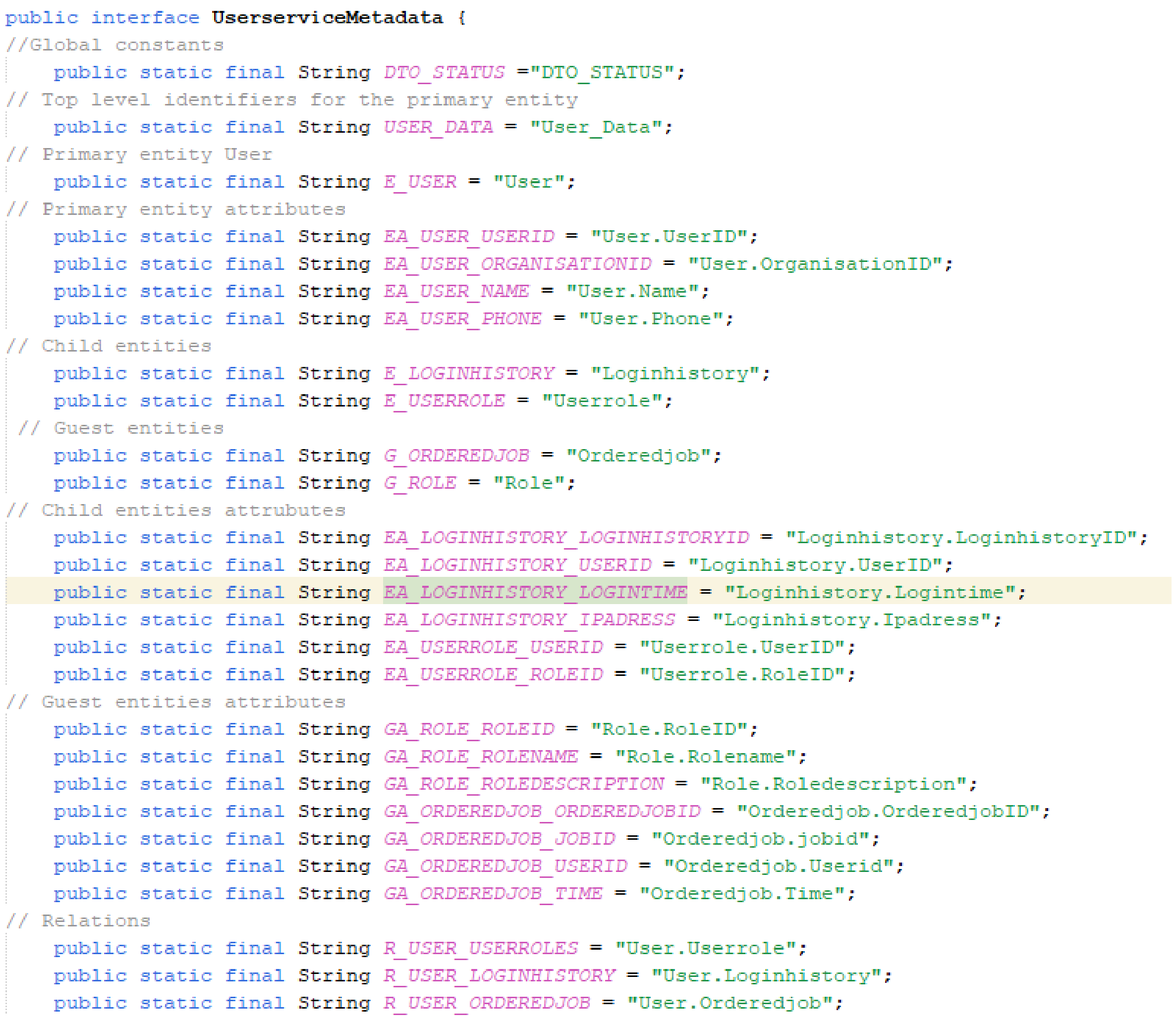
Ex: `public static final String EA_USER_NAME = "User.Name";`

For each relationship between two entities, a string constant is created with the value of the relationship name.

Ex: `public static final String R_USER_USERROLE = "User.Userrole";`

All string constants are written in a file with the name of the service with the Metadata Ex: UserserviceMetadata.java extension and placed in the same directory as the session prayer. Other Java-based clients of this service can import the class to facilitate the build-up of DTO to QS.

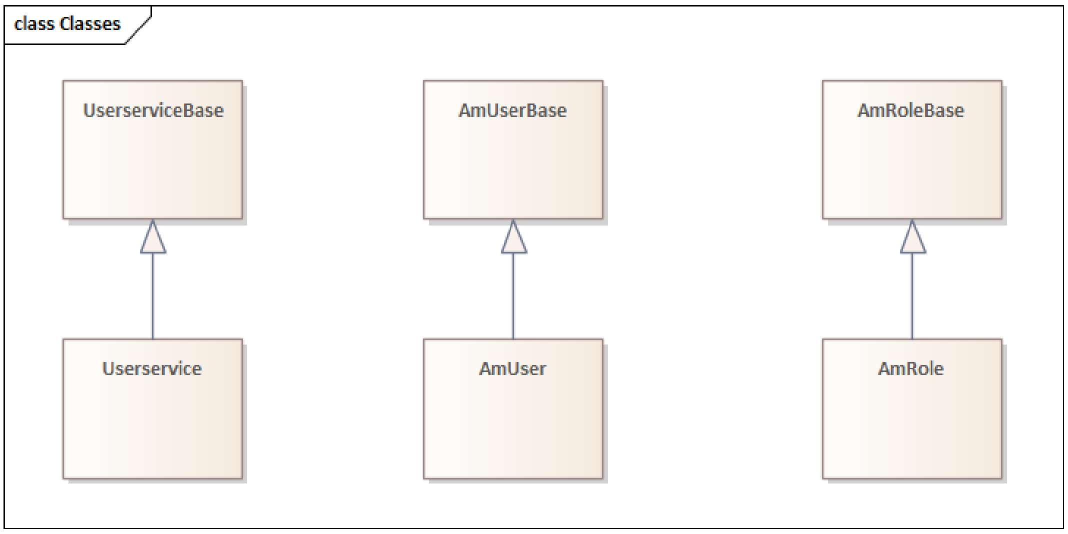
Example of generated metadata:



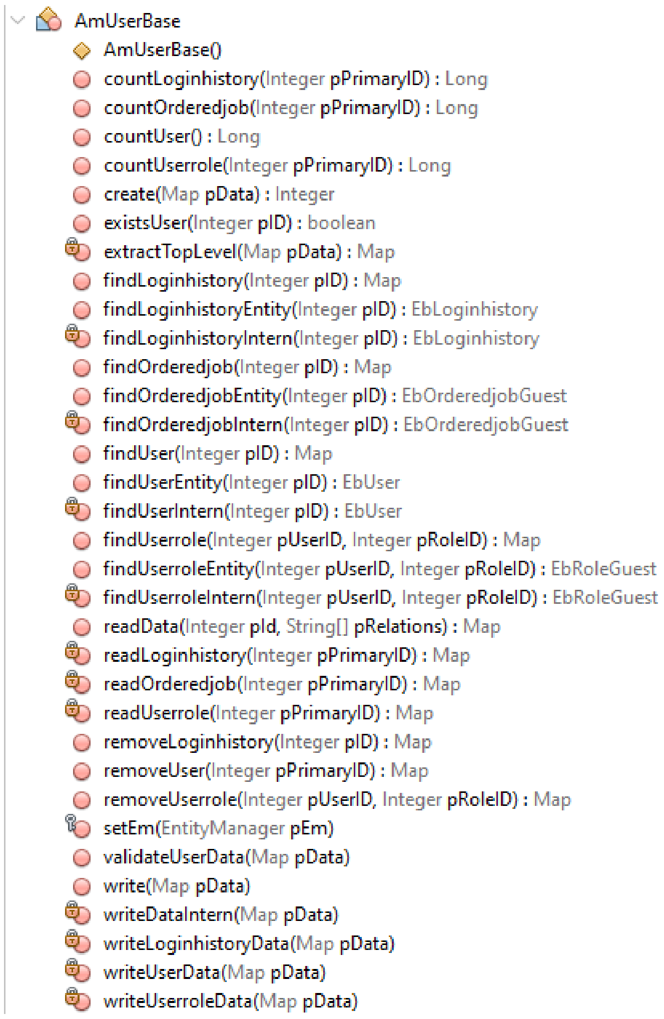
```java
public interface UserserviceMetadata {
//Global constants
    public static final String DTO_STATUS ="DTO_STATUS";
// Top level identifiers for the primary entity
    public static final String USER_DATA = "User_Data";
// Primary entity User
    public static final String E_USER = "User";
// Primary entity attributes
    public static final String EA_USER_USERID = "User.UserID";
    public static final String EA_USER_ORGANISATIONID = "User.OrganisationID";
    public static final String EA_USER_NAME = "User.Name";
    public static final String EA_USER_PHONE = "User.Phone";
// Child entities
    public static final String E_LOGINHISTORY = "Loginhistory";
    public static final String E_USERROLE = "Userrole";
 // Guest entities
    public static final String G_ORDEREDJOB = "Orderedjob";
    public static final String G_ROLE = "Role";
// Child entities attrubutes
    public static final String EA_LOGINHISTORY_LOGINHISTORYID = "Loginhistory.LoginhistoryID";
    public static final String EA_LOGINHISTORY_USERID = "Loginhistory.UserID";
    public static final String EA_LOGINHISTORY_LOGINTIME = "Loginhistory.Logintime";
    public static final String EA_LOGINHISTORY_IPADRESS = "Loginhistory.Ipadress";
    public static final String EA_USERROLE_USERID = "Userrole.UserID";
    public static final String EA_USERROLE_ROLEID = "Userrole.RoleID";
// Guest entities attributes
    public static final String GA_ROLE_ROLEID = "Role.RoleID";
    public static final String GA_ROLE_ROLENAME = "Role.Rolename";
    public static final String GA_ROLE_ROLEDESCRIPTION = "Role.Roledescription";
    public static final String GA_ORDEREDJOB_ORDEREDJOBID = "Orderedjob.OrderedjobID";
    public static final String GA_ORDEREDJOB_JOBID = "Orderedjob.jobid";
    public static final String GA_ORDEREDJOB_USERID = "Orderedjob.Userid";
    public static final String GA_ORDEREDJOB_TIME = "Orderedjob.Time";
// Relations
    public static final String R_USER_USERROLES = "User.Userrole";
    public static final String R_USER_LOGINHISTORY = "User.Loginhistory";
    public static final String R_USER_ORDEREDJOB = "User.Orderedjob";
```

Classes

QS is based on standardized code being in an abstract base class and the unique business code for the system is written in classes that extend the base class.



Base classes should not be edited by developers of the business system, but managed by developers who master the QS model.

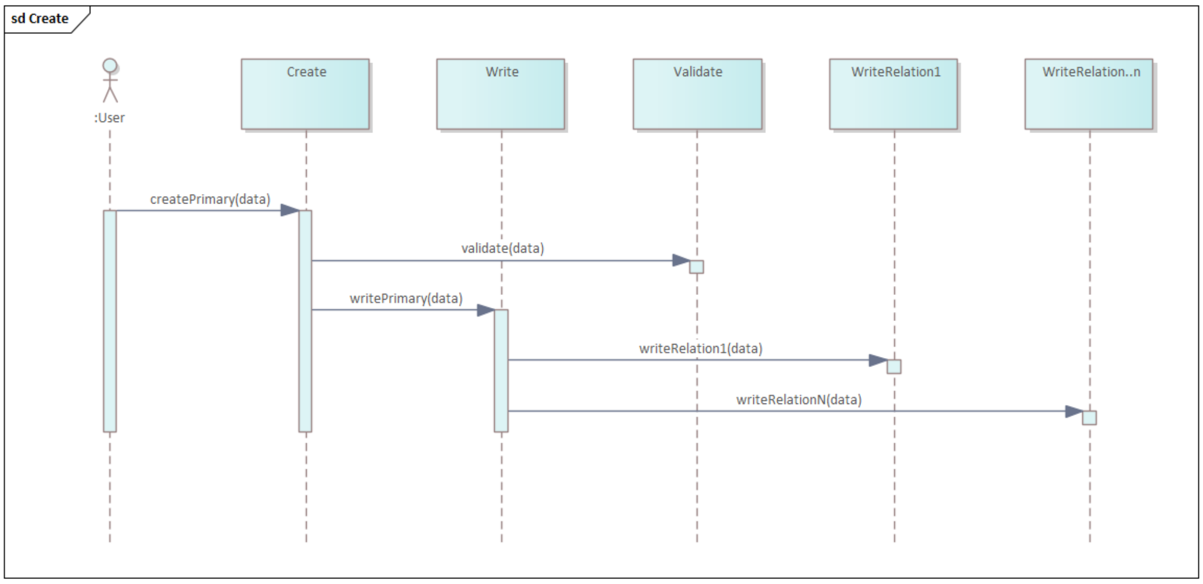


```
AmUserBase
    AmUserBase()
    countLoginhistory(Integer pPrimaryID) : Long
    countOrderedjob(Integer pPrimaryID) : Long
    countUser() : Long
    countUserrole(Integer pPrimaryID) : Long
    create(Map pData) : Integer
    existsUser(Integer pID) : boolean
    extractTopLevel(Map pData) : Map
    findLoginhistory(Integer pID) : Map
    findLoginhistoryEntity(Integer pID) : EbLoginhistory
    findLoginhistoryIntern(Integer pID) : EbLoginhistory
    findOrderedjob(Integer pID) : Map
    findOrderedjobEntity(Integer pID) : EbOrderedjobGuest
    findOrderedjobIntern(Integer pID) : EbOrderedjobGuest
    findUser(Integer pID) : Map
    findUserEntity(Integer pID) : EbUser
    findUserIntern(Integer pID) : EbUser
    findUserrole(Integer pUserID, Integer pRoleID) : Map
    findUserroleEntity(Integer pUserID, Integer pRoleID) : EbRoleGuest
    findUserroleIntern(Integer pUserID, Integer pRoleID) : EbRoleGuest
    readData(Integer pId, String[] pRelations) : Map
    readLoginhistory(Integer pPrimaryID) : Map
    readOrderedjob(Integer pPrimaryID) : Map
    readUserrole(Integer pPrimaryID) : Map
    removeLoginhistory(Integer pID) : Map
    removeUser(Integer pPrimaryID) : Map
    removeUserrole(Integer pUserID, Integer pRoleID) : Map
    setEm(EntityManager pEm)
    validateUserData(Map pData)
    write(Map pData)
    writeDataIntern(Map pData)
    writeLoginhistoryData(Map pData)
    writeUserData(Map pData)
    writeUserroleData(Map pData)
```

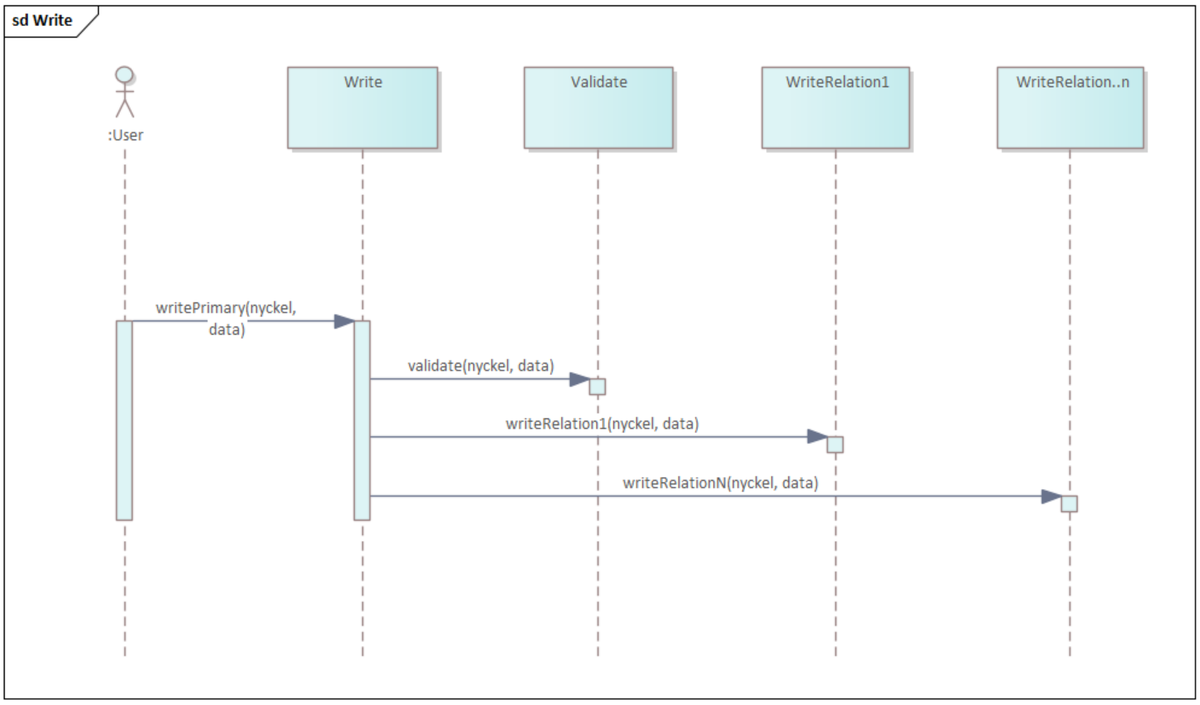
The image shows the methods that are available in AmUserBase. The methods handle all read and write to all entities associated with the primary User entity.

All public methods are also available in UserServiceBase with calls to the methods in AmUserBase.

To read and update the primary entity and its relationships, a pattern is used to automate the flow. To create a primary entity and relationships, the create method is used.



All columns that are not null in the primary entity must be in the folder that is submitted. After validating the folder, the primary entity is created and then write is called, which updates the primary entity's other columns and creates entities for the relationships.


sd Write
:User
Write
Validate
WriteRelation1
WriteRelation..n
writePrimary(nyckel, data)
validate(nyckel, data)
writeRelation1(nyckel, data)
writeRelationN(nyckel, data)

When the primary entity or any relationship is to be updated, write is called which does a validation of the map and then updates the primary entity and entities for the relationships

In other words, it is the same methods and flow for all management after the primary entity is created.

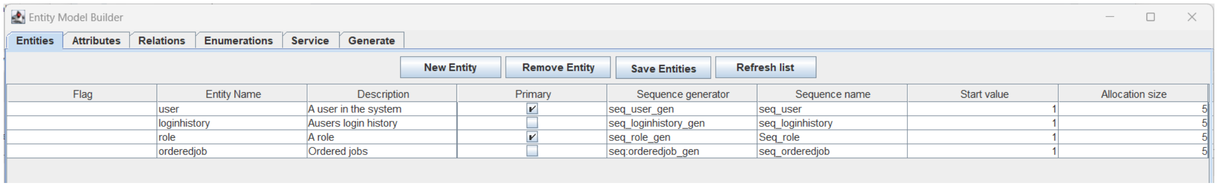
Tool

To facilitate development with QS, there are a number of tools. There are tools for:

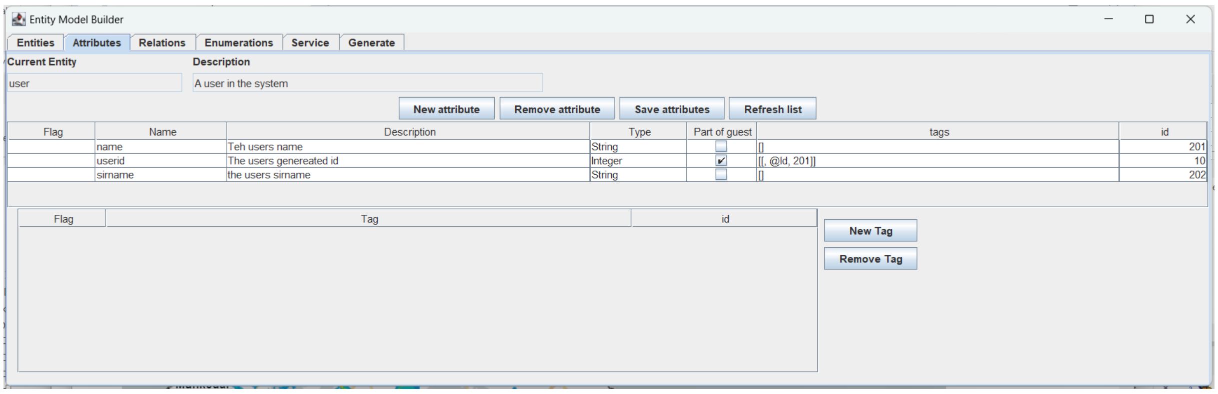
- Create an entity model
- Generate Java Code
- Generate package structure
- Generate local and remote interfaces
- Create client libraries

EntityModelBuilder

EntityModelBuilder is a Swing application where the entities are registered with attributes, tags, and relationships. It is also possible to register enumerations used in the business system. Via various tabs in the application, entities, attributes, annotation tags, enumerations, relationships, etc. are recorded.



Flag	Entity Name	Description	Primary	Sequence generator	Sequence name	Start value	Allocation size
	user	A user in the system	✓	seq_user_gen	seq_user	1	5
	loginhistory	Ausers login history		seq_loginhistory_gen	seq_loginhistory	1	5
	role	A role	✓	seq_role_gen	Seq_role	1	5
	orderedjob	Ordered jobs		seq:orderedjob_gen	seq_orderedjob	1	5



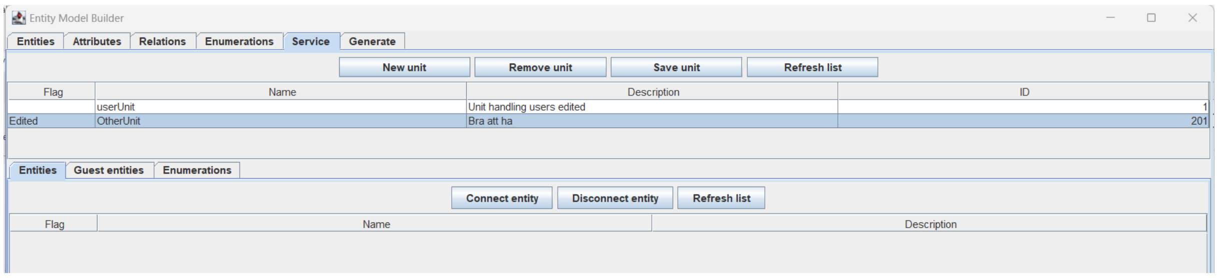
Current Entity: user

Description: A user in the system

Flag	Name	Description	Type	Part of guest	tags	id
	name	Teh users name	String		[]	201
	userid	The users genereated id	Integer	✓	[[, @Id, 201]]	10
	sirname	the users sirname	String		[]	202

Flag	Tag	id

The model can then be packaged into session beans in different ways, which affects code generation.

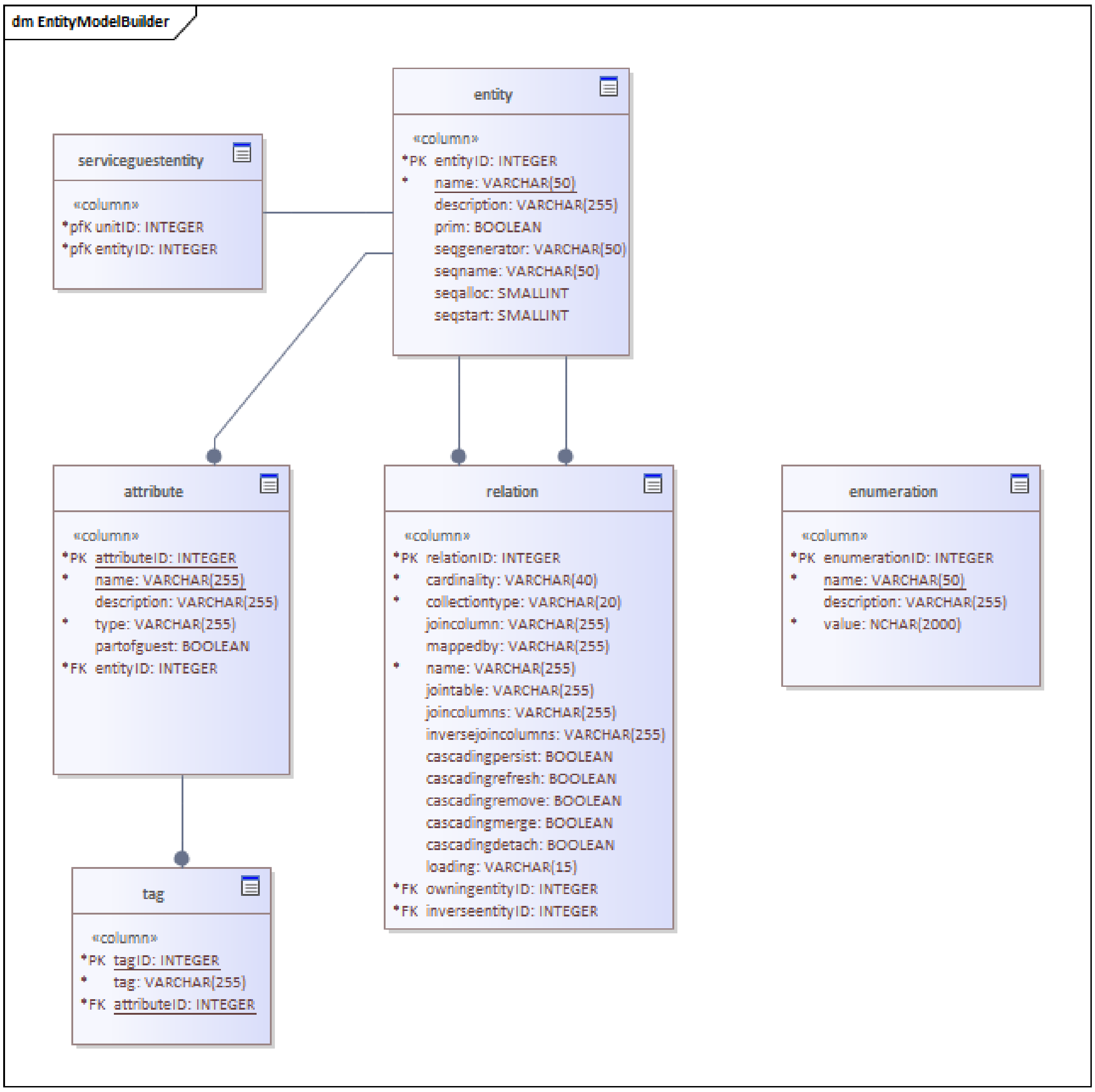


Flag	Name	Description	ID
	userUnit	Unit handling users edited	1
Edited	OtherUnit	Bra att ha	201

Flag	Name	Description

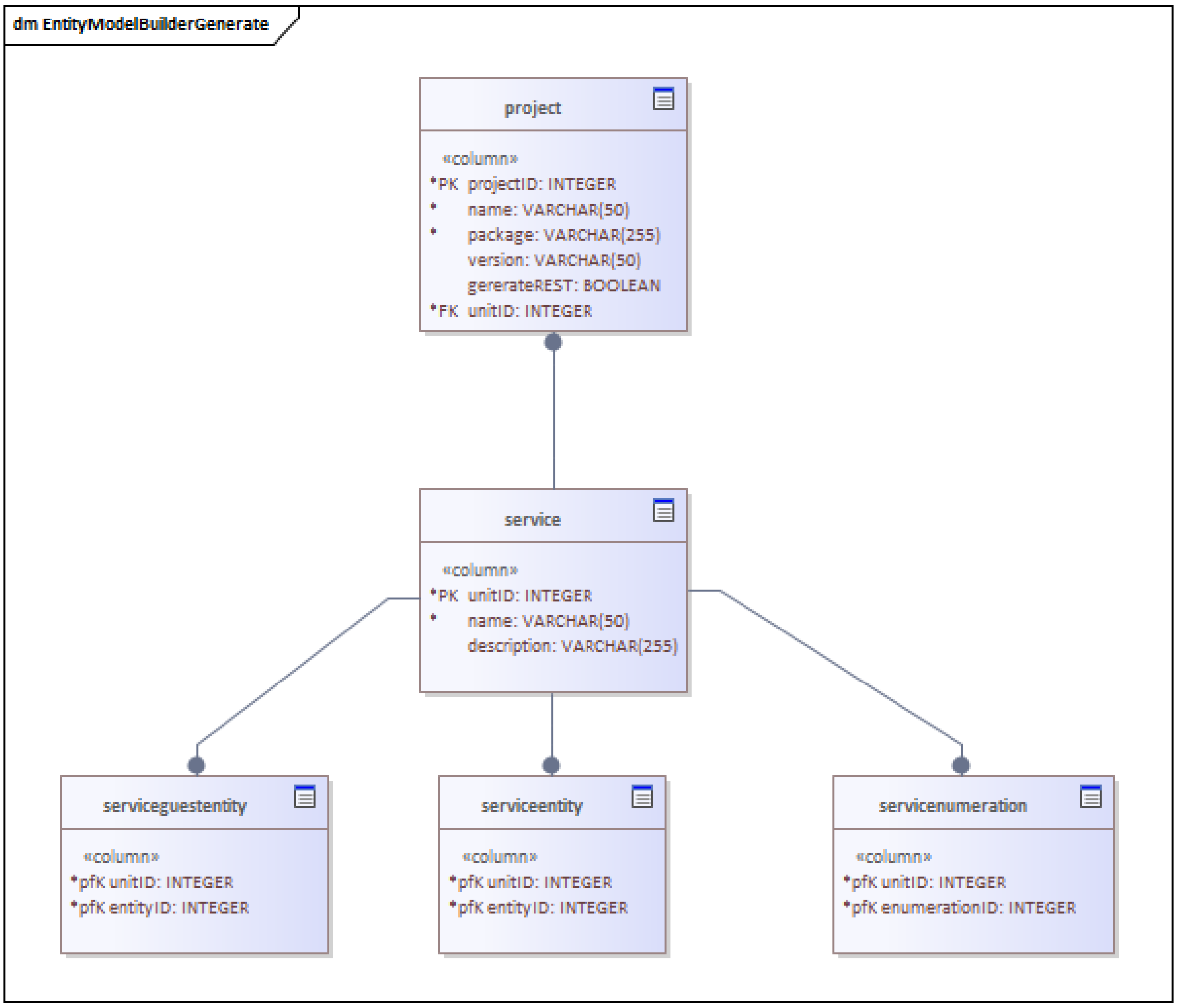
It is also from EntityModelBuilder that code generation is done.

EntityModelBuilder Data Model:



For packaging, you can choose which service an entity belongs to. It is also possible to version manage different variants. QS generates code based on a project where package names, domain names, etc. are registered.

Data model for packaging entities in EntityModelBuilder:

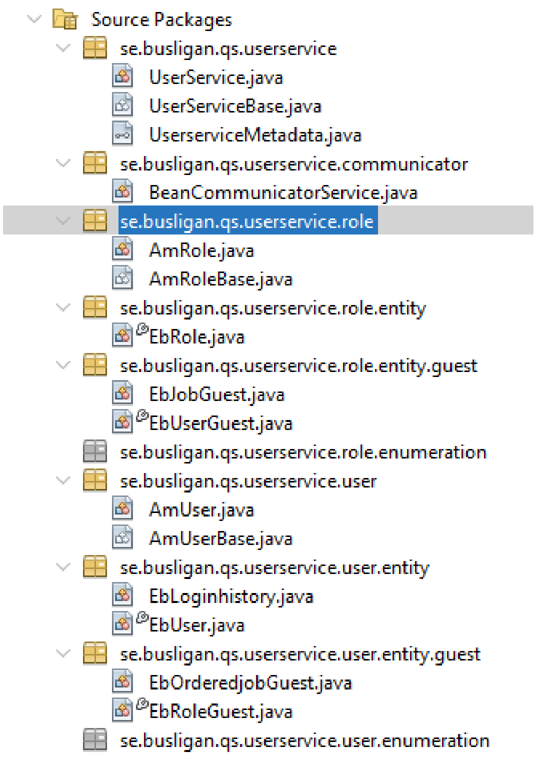


Code Generation

QS is based on the entity model registered in the database, which is then used to generate code for entity beans, and session beans so that all basic CRUD (Create, Read, Update, Delete) works for the entire model. The code is compilable and can perform CRUD operations in an application server immediately after compiling and deploying.

Packages and classes

All classes in the image below, including the package structure, are generated by QS.



```
Source Packages
  se.busligan.qs.userservice
    UserService.java
    UserServiceBase.java
    UserserviceMetadata.java
  se.busligan.qs.userservice.communicator
    BeanCommunicatorService.java
  se.busligan.qs.userservice.role
    AmRole.java
    AmRoleBase.java
  se.busligan.qs.userservice.role.entity
    EbRole.java
  se.busligan.qs.userservice.role.entity.guest
    EbJobGuest.java
    EbUserGuest.java
  se.busligan.qs.userservice.role.enumeration
  se.busligan.qs.userservice.user
    AmUser.java
    AmUserBase.java
  se.busligan.qs.userservice.user.entity
    EbLoginhistory.java
    EbUser.java
  se.busligan.qs.userservice.user.entity.guest
    EbOrderedjobGuest.java
    EbRoleGuest.java
  se.busligan.qs.userservice.user.enumeration
```

Metadata

The metadata code generation has created string constants for all entity attributes and all relationships in the entity model. See more under the chapter Metadata.

Remote and Local Facades

Facades for remote and local interfaces are generated by two processors. Annotate selected methods with @RemoteInterface respective @LocalInterface and register the annotation processor with your development tool and the interface will be generated during compilation. Annotations should be made in the sessionbean, in the example in UserService.java. That's what is exposed according to standards. It is possible to place the annotations in other classes, but the design pattern prefers that it is in the session bean that it is made.

All public methods that QS generates in AmUserBase are also generated to UserServiceBase.java with calls to AmUserBase.java.

No facades therefore need to be kept in sync with the business logic, QS generates these during compilation for all generated methods.

Client library

QS also has scripts to compile the interface as well as the metadata file (UserMetadata.java) and make a jar file that can be used by clients. Depending on the development tool, the script may need to be modified and moved. For Netbean's default EAR project, the command file is placed in the root of the project.