BAMA Backend Coding Guidelines

v 1.0

# BAMA backend framework

## Brief architecture



Layers responsibility:

* Controller layer: publish restful web services
  + Dos
    - Restful web services definition
    - Data binding
    - Data validation
    - Invoke service layer to accomplish business logics
    - Call one service API one time to do task
    - Give UI friendly message to user
  + Don’ts
    - Don’t write business codes here
    - Don’t bypass service layer to invoke DAL (data access layer) objects here
* Service layer: business logic must go here.

It has two layers:

Atom service layer

* + Dos
    - Basic/atom level service (non-business associated) of one domain entity (like CRUD) fall into this layer
    - In order to use declarative transaction management, please follow API naming prefix convention if API need to be in transaction scope:
      * Create: create\*, store\*, save\*, add\*, insert\*
      * Update: update\*, edit\*, modify\*
      * Delete: remove\*, delete\*
      * Query: find\*
      * Other business process: do\*, proceed\*, process\*, set\*
    - Need review to add APIs in this layer
    - Can be injected into controller for invocation
  + Don’ts
    - Complex/business associated codes MUST not present in this layer

Business service layer

* + Dos
    - All business logics MUST write here
    - Invoke atom service layer to accomplish basic entity operations, like CRUD
    - Can inject DAL classes to achieve special goals
  + Don’ts
    - Don’t invoke DAL class to do tasks that already provided by Atom service layer
    - Don’t write FE scripts/tags
    - Don’t write DAL functions/scripts
    - Don’t access DAL functions in loop
* Data access layer: access database codes go here
  + Dos
    - Only data access codes can write here
    - SQL scripts/DSL write here as a must
    - DAL APIs can be invoked multiple times to accomplish one goal, but try to do data access in one database connection
  + Don’ts
    - No business logic allowed in this layer. Business logics should move to service layer
    - Don’t loop to access database

## Implantation HOWTO

### Project directory structure

See attached file:



### Domain Object

Dos and Don’ts

* Business entity that DOES require auditable properties should extend from BusinessEntity

*/\*\*  
 \* Patient entity.  
 \*/*@Entity  
@Table(name = "t\_identification\_patient")  
public class Patient extends BusinessEntity

* Non-business entity that DOESN’T require auditable properties can extend from GenericEntity

*/\*\*  
 \* Dictionary entity.  
 \*/*@Entity  
@Table(name = "t\_core\_dictionary")  
public class DictionaryEntity extends GenericEntity

* Specify entity property type and length if any. For most commonly used data type String, the length options: 2, 4, 8, 16, 32, 64, 128, 256, 512…

@Column(name = "name", length = 64)  
private String name;  
@Column(name = "gender\_code", length = 32)  
private String genderCode;

* New entity introduces new db script to initialize db tables/updates. About DB script guideline, please refer bama\_db\_script\_guideline (TBD)
* Use domain models as DTO to exchange data between frontend and backend (controller/service layers)

### Data Access Layer

Dos and Don’ts

* Assumption is that common uses data access scenarios are simple CRUD operations which can get handy help from spring data jpa. So recommend to use spring data jpa to do jobs first, although the performance is not the top
* Entity repository should extend from BaseRepository. BaseRepository offers softly save/delete APIs (aka. logically save/delete) against target entity
* With spring data jpa, entity repository APIs are very straight forward, no need method comments
* If need customized API, create PatientRepositoryCustom interface and implement it by your own. About how to use this, please refer spring data jpa document

refer repository sample for patient entity:

*/\*\*  
 \* Patient repository.  
 \*/*public interface PatientRepository extends BaseRepository<Patient, Long>, PatientRepositoryCustom {  
  
 Patient findByName(String name);  
  
 List<Patient> findByGenderCode(String genderCode);  
  
 Page<Patient> findByEthnicCodeAndActiveOrderByGenderCodeDesc(String ethnicCode, Boolean active, Pageable pageable);

* When encounters complex scenario (complex query, performance issue…etc) that cannot be done via spring data jpa, also feel free to use hibernate native APIs in DAO package, even native SQL scripts
* Recommend to use spring data jpa APIs, then QueryDsl, then HQL, then native SQL, then native JDBC. Store procedure is no allowed to use.

### Business Service Layer

Dos and Don’ts

* Interface first, then implement it in impl package
* Since unified declarative transaction management is configured on service layer, if the method needs to be in transaction scope, please be aware of the suffix of API name should follow the convention as mentioned above

*<!-- Create -->*<tx:method name="create\*" propagation="REQUIRED"/>  
<tx:method name="store\*" propagation="REQUIRED"/>  
<tx:method name="save\*" propagation="REQUIRED"/>  
<tx:method name="add\*" propagation="REQUIRED"/>  
<tx:method name="insert\*" propagation="REQUIRED"/>  
*<!-- Update -->*<tx:method name="edit\*" propagation="REQUIRED"/>  
<tx:method name="update\*" propagation="REQUIRED"/>  
<tx:method name="modify\*" propagation="REQUIRED"/>  
*<!-- Delete -->*<tx:method name="remove\*" propagation="REQUIRED"/>  
<tx:method name="delete\*" propagation="REQUIRED"/>  
*<!-- BUSINESS PROCESSING -->*<tx:method name="do\*" propagation="REQUIRED"/>  
<tx:method name="proceed\*" propagation="REQUIRED"/>  
<tx:method name="process\*" propagation="REQUIRED"/>  
<tx:method name="set\*" propagation="REQUIRED"/>  
  
*<!--  
 Query for enabling lazy loading.  
  
 VERY VERY VERY IMPORTANT!  
 Enable find/query method in tx scope will lead to entity property updates auto-commit  
 to database that not intend to.  
 So DO NOT update entity in service layer or change once we found better approach.  
-->*<tx:method name="find\*" propagation="REQUIRED"/>

* Generally speaking, atom service returns entity considering reuse in more widely cases; business service returns model (aka. DTO) to rest controller or other client to hide entity design details (encapsulation in OO)

PatientService.java

*/\*\*  
 \* Find unique patient by id, associations are loaded lazily.  
 \*  
 \** ***@param*** *patientOid target oid of patient entity to load  
 \** ***@return*** *target patient if exist otherwise null  
 \*/*Optional<Patient> findPatientById(@NotNull Long patientOid);

PatientBizService.java

*/\*\*  
 \* Load patient according to oid.  
 \*  
 \** ***@param*** *patientOid target entity oid to find  
 \** ***@return*** *target entitfy if exist otherwise null  
 \*/*PatientComprehensiveModel findPatient(@NotNull Long patientOid);

*Note. Entity/DTO is a long debatable issue that it adds effort to do Entity/DTO conversion, to get the encapsulation/decoupling.*

* In order to get rid of the pains in entity/model conversion, use entity/dto mapping library (Dozer) to convert entity/model, then return model to client

@Autowired  
private Mapper beanMapper;  
  
return patientOptional.isPresent() ? beanMapper.map(patientOptional.get(), PatientCompactModel.class) : null;

* Pagination support
  + Native spring data jpa pagination sample:

*/\*\*  
 \** ***@see*** *PatientQueryService#findAllPageable(Pageable)  
 \*/*@Override  
public Page<Patient> findAllPageable(@NotNull Pageable pageable) {  
 return patientRepository.findAll(pageable);  
}

* + Customized query pagination
    - Define customized paging interface in extend repository, like PatientRepositoryCustom
    - Implement this interface with paging APIs, sample as below:

*/\*\*  
 \** ***@see*** *PatientRepositoryCustom#findBySearchCriteriaPageable(PatientSearchCriteriaModel, Pageable)  
 \*/*@Override  
public Page<Patient> findBySearchCriteriaPageable(PatientSearchCriteriaModel searchCriteria, Pageable pageable) {  
 QPatient patient = QPatient.*patient*;  
 JPQLQuery<Patient> query = getJPQLQueryFactory()  
 .selectFrom(patient)  
 .where(assembleSearchCriteria(searchCriteria));  
  
 long count = query.fetchCount();  
  
 List<Patient> patientList = query  
 .offset(pageable.getOffset())  
 .limit(pageable.getPageSize())  
 .fetch();  
  
 if (CollectionUtils.*isEmpty*(patientList)) patientList = new ArrayList<>();  
  
 return new PageImpl<>(patientList, pageable, count);  
}

### Rest Controller Layer

Dos and Don’ts

* Implement REST controller around resource:

*/\*\*  
 \* REST controller for patient services.  
 \*/*@RestController  
@RequestMapping("/patients")  
public class PatientController

* The functions defined in rest controller just return POJO as usual, the framework will wrap the POJO as payload in a rest response envelope to rest client

*function signature:*

*/\*\*  
 \* patient.show action. Show all full attributes one specific patient.  
 \*  
 \** ***@param*** *patientOid target entity oid  
 \** ***@return*** *entity model with details  
 \*/*@RequestMapping("/{patientOid}")  
public PatientComprehensiveModel getShow(@PathVariable Long patientOid) {  
 PatientComprehensiveModel patient = patientBizService.findPatient(patientOid);  
 return patient;  
}

*restful response sample, query for one patient:*

[*http://127.0.0.1:8080/ccs/service/patients/3*](http://127.0.0.1:8080/ccs/service/patients/3)

{  
 "success": **true**,  
 "status": 200,  
 "data": {  
 "oid": 3,  
 "name": "张三",  
 "genderCode": "002",  
 "inpatientId": "",  
 "ethnicCode": "003",  
 "idNo": "450328198606230056",  
 "patientId": "PID0092002"  
 }  
}

*restful response sample, query for one patient:*

*[PUT]<http://127.0.0.1:8080/ccs/service/patients/9>*

{

"name":"李四",

"ethnicCode":"004"

}

{  
 "success": **false**,  
 "status": 500,  
 "data": **null**,  
 "error": {  
 "status": 500,  
 "errorCode": "0x010211001",  
 "message": "病人未找到",  
 "developerMessage": "target patient is not found",  
 "moreInfoUrl": "mailto:admin@mycompany.com"  
 }  
}

*restful response sample, query for paging patients:*

[*http://127.0.0.1:8080/ccs/service/query/patients/criteria?page=1&size=1&sort=name,desc&genderCode=002&ethnicCode=003*](http://127.0.0.1:8080/ccs/service/query/patients/criteria?page=1&size=1&sort=name,desc&genderCode=002&ethnicCode=003)

{  
 "success": **true**,  
 "status": 200,  
 "data": {  
 "content": [  
 {  
 "oid": 4,  
 "name": "李四",  
 "genderCode": "002",  
 "inpatientId": ""  
 }  
 ],  
 "last": **true**,  
 "totalPages": 2,  
 "totalElements": 2,  
 "size": 1,  
 "number": 1,  
 "sort": [  
 {  
 "direction": "DESC",  
 "property": "name",  
 "ignoreCase": **false**,  
 "nullHandling": "NATIVE",  
 "ascending": **false** }  
 ],  
 "numberOfElements": 1,  
 "first": **false** }  
}

* Do following RESTful services with HTTP methods to define functions/uri/operations
  + Don’t do data write operation in GET method

Actions Handled By Resource Controller

| **Verb** | **Path** | **Action** | **Route Name** |
| --- | --- | --- | --- |
| GET | /patient | index | patient.index |
| POST | /patient | create | patient.create |
| GET | /patient/{oid} | show | patient.show |
| PUT/PATCH | /patient/{oid} | update | patient.update |
| DELETE | /patient/{oid} | delete | patient.delete |

## Common HOWTO

Common coding guidelines for general parts, like logging, exception handling, and error code definition, constants, configuration…etc.

### Logging

Dos and Don’ts

* Use slf4j logger to do logging but not raw log4j APIs

*log exception*

*logger*.warn("array index out of bounds in processing patient updates: {}", e);

*log message with parameters*

*logger*.debug("no patient with patientOid can be found: patientOid is {} and model is {}.", patientOid, model);

* Don’t append strings in log message
* In most of the cases, developer doesn’t need to do logging except very important status/data/situation. Framework will help to log errors/exceptions for you by default. Be aware of high log levels (info+) to use
* @see PatientBizServiceImpl#updatePatient for detail cases

### Exception handling

Dos and Don’ts

* Recommend use runtime exception in most of the cases; if it’s a business exception, extends from BusinessException; if it’s a unexpected system exception, extends from SystemException

*/\*\*  
 \* Exceptions occur when target patient not found.  
 \*/*public class PatientNotFoundException extends BusinessException

*/\*\*  
 \* Exceptions occur when try to modify inactive patient.  
 \*/*public class InactivePatientModificationException extends SystemException

* If do need expected checked exception, extends from BaseCheckedException and go
* Exception handling cases:

case 1

// #case 1  
// if patient is inactive, violate update patient rule and throw system exception  
if (!patient.getActive()) {  
 // if no patient can be found, throw biz exception, don't log any msg if you don't want to  
 // logger.info("target patient is set as inactive and cannot be modified.");  
 throw new InactivePatientModificationException("target patient is inactive and cannot be modified any more.");  
}

case 2

// #case 2  
// un-caught exception and it throws out for platform to process  
ArrayList arrayList = null;  
arrayList.get(0);

client will receive response by framework

{  
 "success": **false**,  
 "status": 500,  
 "data": **null**,  
 "error": {  
 "status": 500,  
 "errorCode": "0x010201001",  
 "message": "server internal error",  
 "moreInfoUrl": "mailto:admin@mycompany.com"  
 }  
}

case 3

// #case 3  
// catch an exception manually and process it  
arrayList = new ArrayList();  
try {  
 arrayList.get(0);  
} catch (ArrayIndexOutOfBoundsException e) {  
 *logger*.warn("array index out of bounds in processing patient updates: {}", e);  
 throw new PatientUpdateException("update updates with internal error that patient code is out of range.", e);  
}

case 4

if (patientOptional.isPresent()) {  
 // do biz logic here   
} else {  
 // #case 4  
 // if no patient can be found, throw biz exception  
 *logger*.debug("no patient with patientOid can be found: patientOid is {} and model is {}.", patientOid, model);  
 throw new PatientNotFoundException("target patient is not found");  
}

### Error code

Error code definition as below:

# error code definition  
# 0x 00 00 00 000 / app-subsystem-module-code  
# e.g.0x010201001  
# 01 - bama  
  
# 01 - platform  
# 02 - app  
  
# 01 - system  
# 02 - admin  
# 03 - core  
# 04 - security  
# 11 - identification  
# 12 - workflow  
# 13 - clinic  
# 14 - vendor  
  
# 001 - patient not found  
# 002 - duplicate patient  
  
  
# exception mapping definitions  
  
# Sample:  
#com.philips.h2h.bama.app.exception.BrandNotExistException=504, \_exmsg  
#com.philips.h2h.bama.app.exception.DuplicateBrandsException=505, 0x010201001, duplicate patient  
#com.philips.h2h.bama.app.exception.SampleBusinessException=505, 0x010201001, error.patient.duplicate, error.patient.duplicate, more info please contact admin  
#com.philips.h2h.bama.app.exception.security.AccountAlreadyExistedException=500, 0x010204001, security.account.username.exist  
  
com.philips.h2h.bama.app.identification.exception.PatientNotFoundException=500, 0x010211001, error.patient.not.found  
com.philips.h2h.bama.app.identification.exception.InactivePatientModificationException=503, 0x010204003, error.patient.inactive.modified.message, patient is inactive that should not be modified according to rule, more info please contact admin  
  
Throwable=500, 0x010201001, server internal error

Usage:

#{full exception class name}={http status code}, {internal error code}, {message to client}, {message to dev}, {message to help}

Sample:

*#com.philips.h2h.bama.app.exception.SampleBusinessException=505, 0x010201001, error.patient.duplicate, error.patient.duplicate, more info please contact admin*

* message could be raw string or resource key define in I18N bundles, framework is smart to find the one that exist for use
* exception cannot find its configuration will try to find its parent exception class, then its parent’s parent class until to the default root of exception Throwable
* a switch to turn on/off to debug/release mode. Release mode will hide the {message to dev} in response
* see Throwable configuration. 3 parameters are required: {http status code}, {error code} and {message to client}. Others are optional.

### Configuration

All runtime changeable values MUST not hardcoded in source codes.

Dos and Don’ts

* all runtime changeable values should be move to external configurations files, like frequency, times, service hosting IP…etc. Don’t hardcode these values in source codes
* commonly used configuration files are \*.properties, \*.xml, \*.json and even \*.txt
* place system level configuration in classpath:appconfig/setting.properties; place app level configuration in classpath:appconfig/application.properties
* if wants to add more configuration files, use spring to load your new properties/xml, see spring-application-context.xml for a reference

### Constants

Dos and Don’ts

* Don’t use hard code string, magic number in source codes. Define them in internal class if private used or public constants class for reuse
* com.philips.h2h.bama.app.common.util.AppConstants is used to define system level constants; app level constants please define in its own package
* It’s allow to write raw string for exception message for dev purpose. If the error message is given to UI then define them in resource bundle, the resource key in constants and resource value in resource bundle file

### Audit

Key business events need to be audited as audit log in system.

There are two approaches to add audit logs for business events.

1. Audit events via annotation

@Audit (action = AuditConstants.ACTION\_PATIENT\_CREATE,

repository = AuditConstants.REPOSITORY\_PATIENT)  
public PatientComprehensiveModel createPatient(@NotNull PatientComprehensiveModel model) {

1. Audit events manually

// manually audit logging in order to capture username in obj param and ignore password  
AuditEvent auditEvent = new AuditEvent(auditMetaData.getActor(), AuditConstants.*ACTION\_LOGIN*,  
 auditMetaData.getOrigin(),  
 new Field(AuditConstants.*FIELD\_LOGIN\_USERNAME*, token.getPrincipal().toString()),  
 new Field(AuditConstants.*FIELD\_LOGIN\_SUCCESS*, String.*valueOf*(authenticated)));  
auditEvent.setRepository(AuditConstants.*REPOSITORY\_SECURITY*);  
  
AuditManager.getInstance().audit(auditEvent);

*P.S. create audit table with script first.*

### Cache

The places to enable cache should be discussed first.

Developer is not allowed to add cache in function implementation. If developer finds there is a case necessary to enable cache to save performance, appreciate to raise a discussion.

The sample to add cache as below:

1. Enable cache setting in configuration file: $classpath/appcofig/cache/ehcache.xml

<cache name="app.common.dictionary"  
 maxElementsInMemory="10000"  
 eternal="true"/>

1. Add cache keys in annotation on the function

*/\*\*  
 \** ***@see*** *DictionaryTermService#findDictionaryTermByGroup(String)  
 \*/*@Override  
@Cacheable(value = "app.common.dictionary", key = "'group.name.' + #groupName")  
public Optional<Map<String, DictionaryTerm>> findDictionaryTermByGroup(@NotNull String groupName) {  
}

# General Practice

## JDK8

Dos and Don’ts

* Use lambda expression as first option to write codes
* Use Optional<T> to avoid missing null checking

sample:

// set gender  
dictionaryTermService  
 .findDictionaryTerm(DictionaryConstants.*GROUP\_GENDER*, genderCode)  
 .ifPresent(e -> {  
 DictionaryModel model = new DictionaryModel();  
 model.setCode(e.getCode());  
 model.setLabel(e.getLabel());  
 externalModel.setGender(model);  
 });

if (entity.isPresent()) {  
 DictionaryModel model = new DictionaryModel();  
 model.setCode(entity.get().getCode());  
 model.setLabel(entity.get().getLabel());  
 return model;  
} else {  
 return null;  
}

* Use JDK8 date/time APIs to deal with date/time

## Loop

Dos and Don’ts

* Use lambda expression to loop collection/array
* Use foreach to loop collection/array
* Avoid to use index/iterator to loop collection/arrays any more in most of the cases

Samples:

list to String

return modelList.stream().map(DictionaryModel::getCode).collect(Collectors.*joining*(AppConstants.*COMMA*));

object list to object property list

if (CollectionUtils.*isNotEmpty*(patientList)) {  
 result = patientList.stream().map(Patient::getOid).collect(Collectors.*toList*());  
}

convert list to another list

List<DictionaryModel> modelList = optionList.stream().map(e -> {  
 DictionaryModel model = new DictionaryModel();  
 model.setCode(e.getCode());  
 model.setLabel(e.getLabel());  
 return model;  
}).collect(Collectors.*toList*());

list to set

if (CollectionUtils.*isNotEmpty*(patientList)) {  
 patients.addAll(patientList.stream().map(p -> p.getName()).collect(Collectors.*toSet*()));  
}

list to Map<String, T>

Map<String, DictionaryTerm> dictionaryTermMap = dictionaryTermList  
 .stream().collect(Collectors.*toMap*(DictionaryTerm::getCode, Function.*identity*()));

list to Map<Long, T>, T is model mapped from class in list

// convert to map  
Map<Long, TopRecommendationProductModel> modelMap = patientList.stream().map(e -> {  
 PatientModel model = new PatientModel();  
 model.setOid(e.getOid());  
 model.setName(e.getName());  
 return model;  
}).collect(Collectors.*toMap*(PatientModel::getOid, Function.<TopRecommendationProductModel>*identity*()));

list to Map<String, List<T>>

Map<String, List<PatientModel>> result = patientModelList.stream().collect(Collectors.*groupingBy*(PatientModel::getGenderCode));

List to Map<String, List<T>>

// convert to map  
Map<String, List<String>> patientModelMap =  
 patientModelList.stream()  
 .collect(Collectors.*groupingBy*(  
 PatientModel::getGender,  
 Collectors.*mapping*(PatientModel::getEthnic, Collectors.*toList*()))  
 );

map to list

// create patient models for each  
if (MapUtils.*isNotEmpty*(patientModelMap)) {  
 patientList = patientModelMap.entrySet().stream().map((e) -> {  
 Long patientOid = e.getKey();  
 List<PatientModel> modelList = e.getValue();  
 Patient patient = new Patient ();  
 return patient;  
 }).collect(Collectors.*toList*());

filter

Optional<PattientModel> result = Arrays.*stream*(PatientModelArray.*values*()).filter(e -> StringUtils.*equalsIgnoreCase*(e.getCode(), code)).findFirst();  
return result.orElse(null);

map/reduce

BigDecimal totalScore = scoreList.stream().map(s -> s.getScore().multiply(new BigDecimal(s.getQuantity()))).reduce((total, score) -> total.add(score)).get();

lambda in lambda

Map<String, List<patientModel>> patientModelMap = patientModelList.stream().collect(  
 Collectors.*groupingBy*(PatientModel::getGender));  
patientModelMap.forEach((k, v)  
 -> System.*out*.println(k + " "  
 + v.stream().map(PatientModel::getEthnic).collect(Collectors.*joining*(","))));

## Utils

JDK gives not rich util APIs for programming, so recommend handy 3rd party util libraries:

* apache common library
* google guava library

Samples:

*StringUtils*

if (StringUtils.*isNotBlank*(searchCriteria.getName())) {  
 whereHql += "and p.name = :name ";  
  
 paramNames.add("name");  
 paramValues.add(searchCriteria.getName());  
}

*CollectionUtils*

List result = find("from Patient p where p.oid = ?", patientOid);  
  
if (CollectionUtils.*isEmpty*(result)) {  
 result = new ArrayList<>();  
}

*NumberUtils*

if (NumberUtils.*compare*(patient.getOid(), patientModel.getPatientOid()) != 0) {  
 throw new PatientNotEqualException("not the same patient: " + id);  
}

int number = NumberUtils.*toInt*(this.codeExpiredMinutesString);

BigDecimal valueOfDecimal = NumberUtils.*createBigDecimal*(value);

*FileUils*

FileUtils.*forceDelete*(targetFile);

FileUtils.*moveFile*(sourceFile, targetFile);

# Design Principles

* Programming on interface, design against abstract
* MVC pattern will still working in long time
* Following Open-Close-Principle and Single-Responsibility-Principle
* Don’t over design
* Coding for read further than functional
* Refactory codes when you have duplicates
* Don’t tuning performance until it needs
* Do it right the first time