

Abstract:

The software ezScrum is a legacy web application that has been developed and maintained for more than ten years. With web technology constantly evolving, the ezScrum development team has found it increasingly difficult to keep up and many older technologies are still used. A consequence is the slowdown of release. Examining the issues, we have found the main cause to be the coupling of ezScrum modules since ezScrum is a monolith. Thus, effort has been taken to convert ezScrum into a set of microservices so that coupling is reduced. This thesis reports our work on extracting the product backlog from ezScrum. As demonstrated, the extracted product backlog microservice operates as an independent web application that collaborates with other extracted microservices from ezScrum including account management.

Keyword : Monolithic, Microservice, Migration, ezScrum, Product Backlog.

1.1. Introduction

ezScrum since last year was in the phase of architectural transition from the old style of the monolithic to the microservice, this transition is due to the monolithic architecture that currently used by ezScrum is very dependent on specific technologies that have been used more than ten years since the first release ezScrum and the development of some of these technologies have even discontinued. In the implementation of the microservice architecture, large monolithic applications such as ezScrum are split into several modules each of which would become independent service and each service has its own technology coverage with its functionality and clear boundary[1].

ezScrum is a web application that facilitates system development through a scrum framework with several key functions such as product backlog, sprint planning, sprint backlog, taskboard, burndown chart, task management and et cetera. some of the above features have become the primary candidates that need to be decomposed into small bounded functionality. The product backlog is a list of everything to be accomplished in a project that is currently running. This list can be the addition of new features, enhancement running feature, or even fixing the issues. Usually, product backlog sorted by priority level. These priority levels are generated from several factors such as value added, cost and risk based on these criteria the product owner would be able to determine which user story that needs to be finished in the sprint backlog for the next sprint[2]

Scrum is one of the most popular method among the agile methods[3] , it has a few simple rules designed to help a team to organize, achieve high quality, high customer satisfaction and good developer experience. The open-source application ezScrum [4] is maintained around mature practices such as automated tests including user acceptance test cases for verifying the functional requirements and unit testing to test the individual classes and methods. The application is developed in Java, more specifically in Struts framework [5]. The 1.2.7 version of the Struts framework that is used to develop ezScrum became old and it is no longer supported by the community [6]. There are numbers of research studies and literatures that address issues on identifying and investigating possible methods and models to describe microservice, one of such is a Domain-Driven Design notion of Bounded Context [7]. It helps the development team gain a clear and shared understanding of what has to be consistent and what can be developed independently. Basically, it defines explicit boundaries of the service, which is essential in

developing microservice. Figure 1 is a sample diagram of a bounded context, as it shows how two unrelated concepts are separated into two services where they only share the common concepts Customer and Product [4-7].

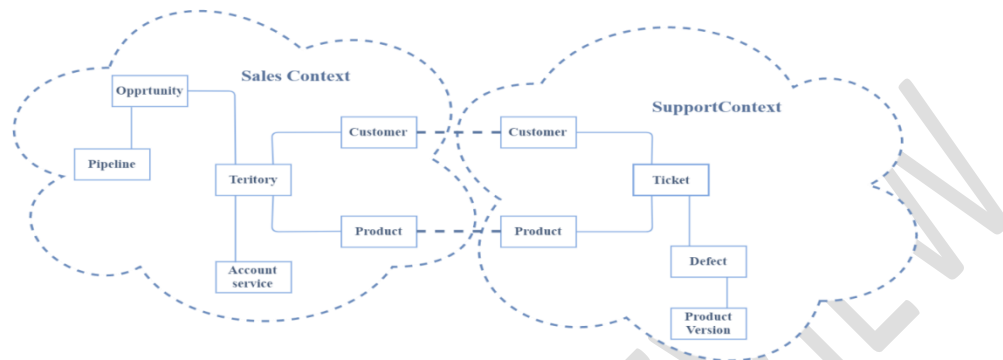


Figure 1 Bounded Context [8]

However, getting service boundaries wrong would be costly and risky in the long run. Hence, the team has to be cautious when defining and modeling the loosely coupled and high cohesive services. Once the bounded context are determined and have the explicit public interface defined, it is then up to developers to develop the micro services around the business capabilities [9].

METHODOLOGY STARTING WITH MICROSERVICE

1.2. Preliminary Preparation:

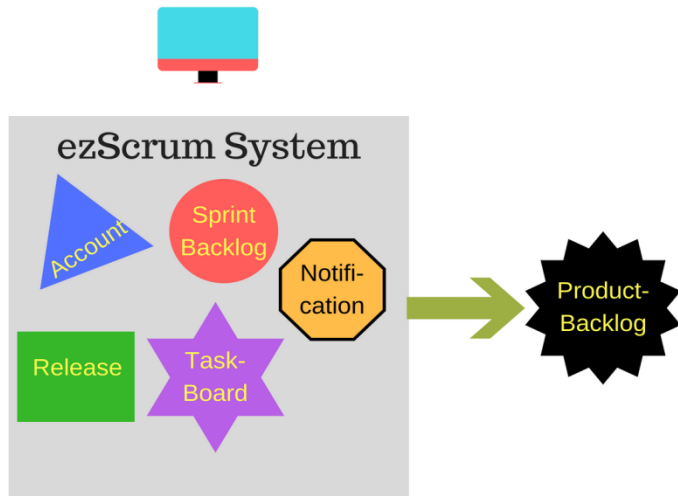


Figure 1.a: Splitting Illustration

The first step that needs to be done is to identify the logical architecture of the ezScrum and find out in which cluster that business logic and domain model reside. This process is called isolating the domain [1]. This task including reading and organizing the existing source code and draw it into context diagram which is necessary to get the more profound understanding of the overall business process of ezScrum. The context diagram is also essential for mediating with domain professional to categorize the related domain into the defined business context and filter out the domains which not related to the product backlog microservice.

1.2.1. ezScrum Package Structure:

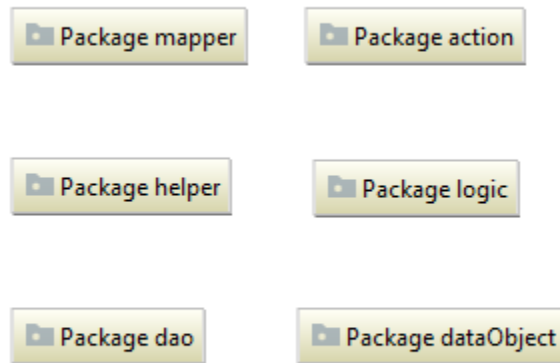


Figure 1.b : ezScrum Packages

In the legacy ezScrum java classes that have the similar functionality are grouped in the single package which is named into a meaningful name that is reflecting the concept and functionality for each class member. With this package naming and structure, providing the convenient way for locating the class that should be delegated to complete some specific task. For instance, AjaxAddStoryAction class and AjaxAddTaskAction class are providing the similar functionality, both classes primarily handling the client request and prepare the response data back to the client and these classes are grouped in the Package action. In above figures each package has its specific functionality that respectively describes as follows, *Package action* is responsible for coordinating the task that came from client and delegates work to *Package helper* for specific process with help from *Package logic* if there is any calculation involves then the process continues to *Package Mapper* which provides the higher level functionality to do a database query and maps this query to the underlying database query using *Package Dao* and return the output by creating an object from *Package dataObject*[6].

1.2.2. Analyzing ezScrum Architecture

ezScrum package naming structure above provides us with a convenient way to presume which package that contains the domain concept and also provides the excellent foundation for defining the specific architecture that is currently applied in ezScrum. This task is necessary to define the specific concern of each ezScrum package. This process also provides a reference for isolating the domain concept and decoupling it from the framework technology or other unrelated concepts that might be mixed with domain concept [3].

Follows is the sequence diagram that taken when new product backlog request occurs, which is notated using product backlog class and story class in ezScrum:

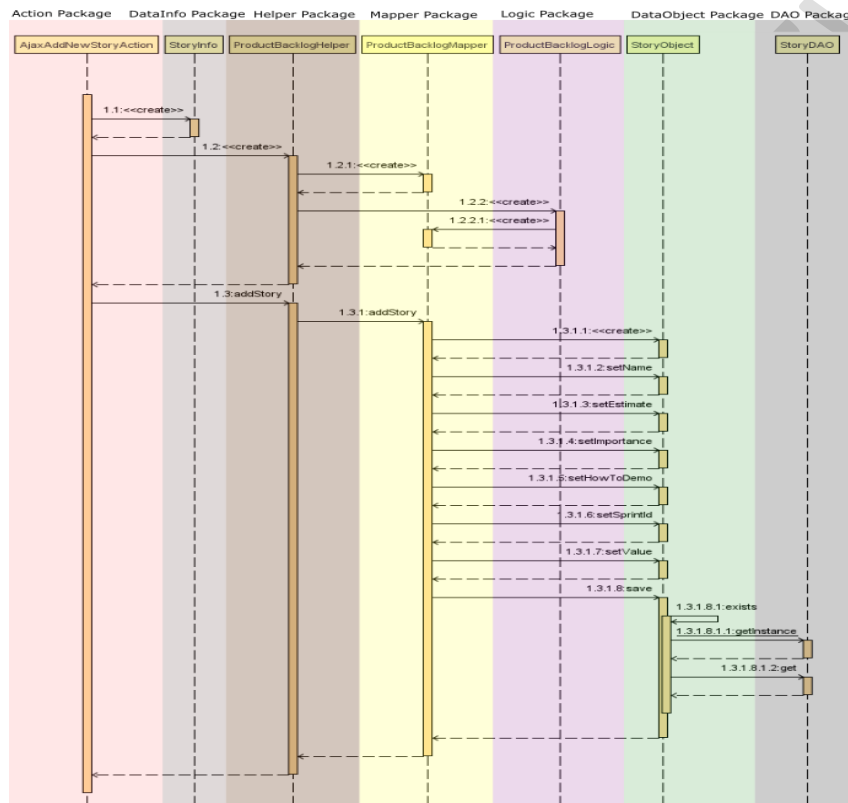


Figure 1.c: sequence diagram of class call-stack in ezScrum

Above figure mildly illustrates the request action for creating a new story in ezScrum, the process that comes from HTTP request with body parameter firstly flows from *AjaxAddnewStoryAction* class converts the HTTP request body to *Story Info* object and delegates the creation process to the *Product Backlog Helper* class which relies on *ProductBacklogLogic* class for handling the conditional checking and *StoryObject* for storing the story data to persistence database that facilitated by *Story Dao* class and when the process has been finished the delegated class notifies the caller class package by package.

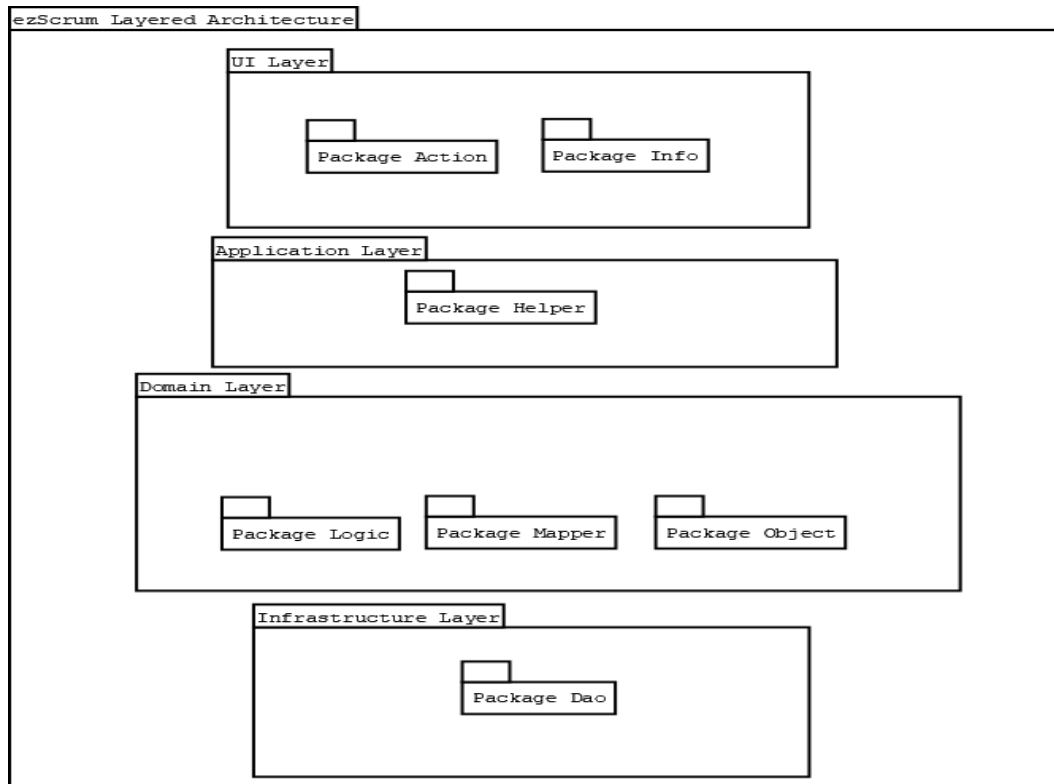


Figure 1.d: ezScrum layered architecture

From above analysis it is straightforward to denote that ezScrum architecture is reflected with above figure which states that each layer is specialized in specific aspect of ezScrum process:

Table 1 ezScrum layer description

| | |
|----------------------------|---|
| ezScrum presentation layer | In ezScrum, this layer is responsible for handling the user request that comes from a browser and returns the response whenever the related task is done. the classes that responsible for handling this task are grouped together in the Package Action. |
| ezScrum application layer | In ezScrum, this layer which consisted with helper packages that are responsible for mediating a task comes from the action class for package naming, although the member classes are named with a related specific business process for example "ProductBacklogHelper" this package is not involved with any business situation, only for delegating the task to the package in domain layer and tracks the progress of the currently running process to notify the package in an upper layer. |
| ezScrum domain layer | In ezScrum this layer is consisted with the package that manages the business definition, for an instance the StoryObject class member of the package object contains with variables that reflecting the states and attributes of the user |

| | |
|---------------------------|---|
| | story. |
| ezScrum persistence layer | When the specific task that demands for storing data to persistence storage the domain layer invokes the mapper that maps the java storing procedure using the dao object to specific sql query which would be processed by underlying database system. |

1.2.3. ezScrum domain layer entities

Class diagram is used to express the situation and followed with the description for each class respectively. Two diagrams would be created the first diagram exhibits the list of the classes that were used by ezScrum and all the attributes that belongs to it respectively and the other one exhibits the interaction between these two classes.

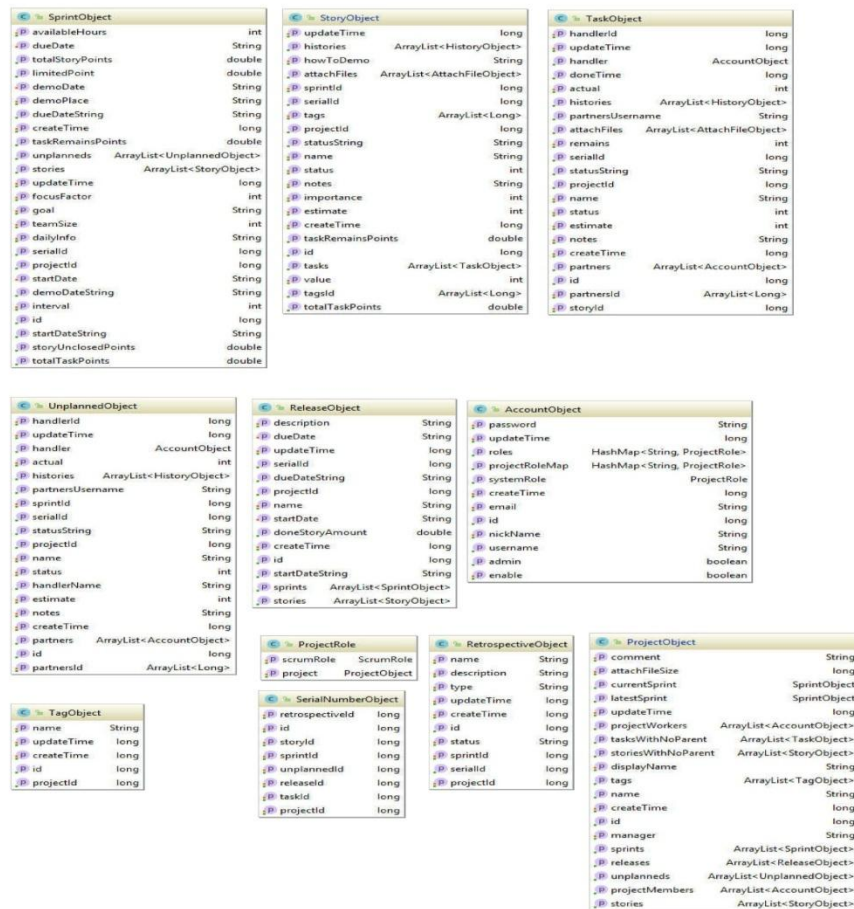


Figure 1.e: ezScrum object entities

1.2.4. ezScrum database entities

Having all the database entities listed in the report would help us to understand which table is currently being mapped from the domain entities to underlying database. Back to ezScrum database, we have identified 15 tables each of these tables representing the object model in ezScrum for storing the data in MySQL database, then in the splitting work. We will identify which entity that part of Product Backlog context and declare the relationship for each table based on the table below.

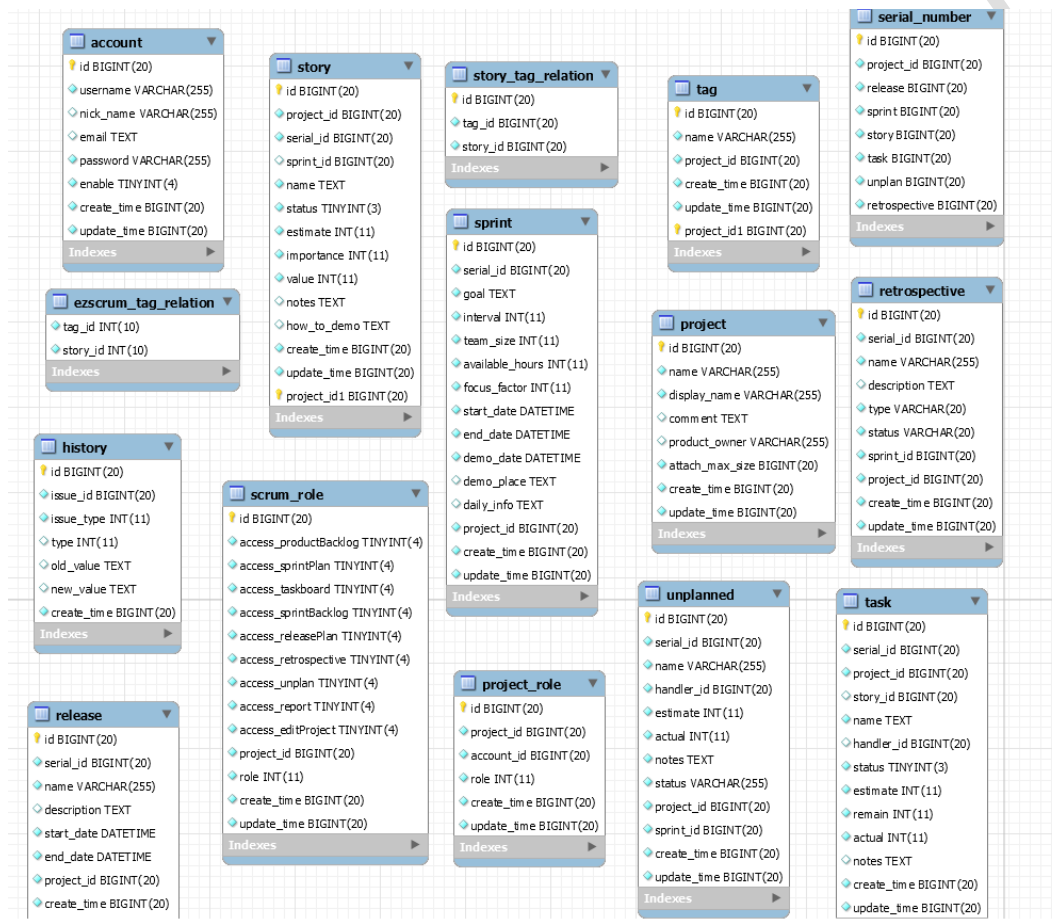


Figure 1.f: ezScrum database entities

1.3. Splitting Work

1.3.1 Product backlog context boundary

To ensure our splitting processes would work in right path we implement one of DDD technique that called bounded context this approach states that each component and these context models are only used within their bounded scope and data aren't shared across the bounded contexts. We can explicitly defined the bounded context of product backlog with some general concept of product backlog:

“The product backlog is a list of product requirements to be accomplished in a project that is currently running. This list can be categorized into several category like the addition of new features, enhancement running feature, or even fixing the issues. Usually, product backlog sorted by priority level. These priority levels are generated from several factors such as value added, cost and risk based on these criteria.”[8].

From above product backlog concept we sorted out several important terms that would be important to be the list of sub-domain and explicitly compare these sub-domain into ezScrum related terms through a discussion session with domain expert.

Table 2 Product Backlog sub-domain list

| Subdomain | Description | In ezScrum |
|----------------------|--|----------------|
| Product requirements | The items that has to be done. | Story |
| Category | A predefined categories that allow the PO to categorize the story. | Tagging system |
| Project | Aplanned work that need to be finished in the range of time. | Project |

Noticed that we ignored the product owner term because in ezScrum product owner is expressed by the user object that part of Account management context[5].

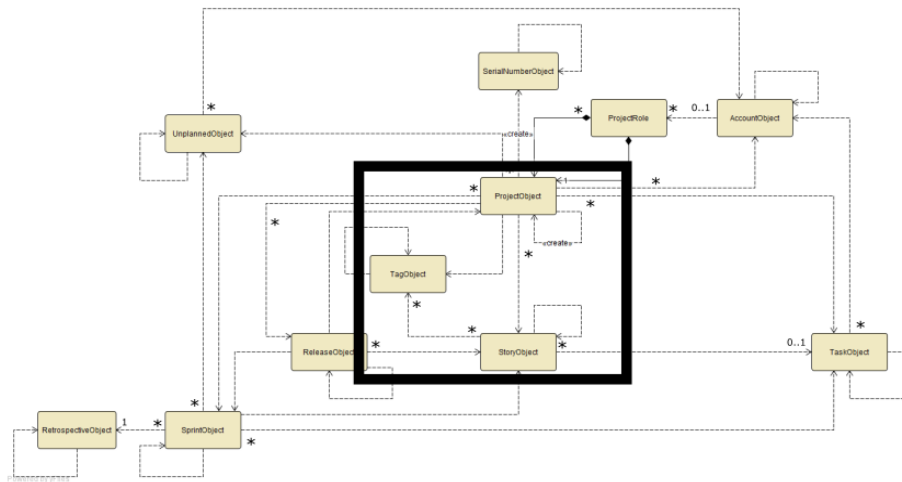


Figure 1.g: Product Backlog context boundary

In the scrum, product requirement is a single item that needs to be accomplished to deliver a viable product. This item contains several attributes that defined with ranged value such as risk, business value, dependencies, size, and date needed. Similar with ezScrum product requirements and its attributes terms represented explicitly with story item that has the attributes with ranged value like a status, important, value and related tags that would categorize a story into the predefined category.

From above boundary we extracted several scenarios which involving the project, tags system, and story. Further we would analyze these scenario to confirm that we solve this boundary effectively.

1.3.1. Product backlog scenarios

Based on above analysis we extracted several product backlog related scenarios by running these scenarios in legacy ezScrum and track every class that appears in callstack and express this scenario into call sequence using this phases intended to confirm that we solved the analysis effectively this phase would also help for structuring the microservice design and channeling for implementation as a reference.

Scenario 1: the story and project scenario

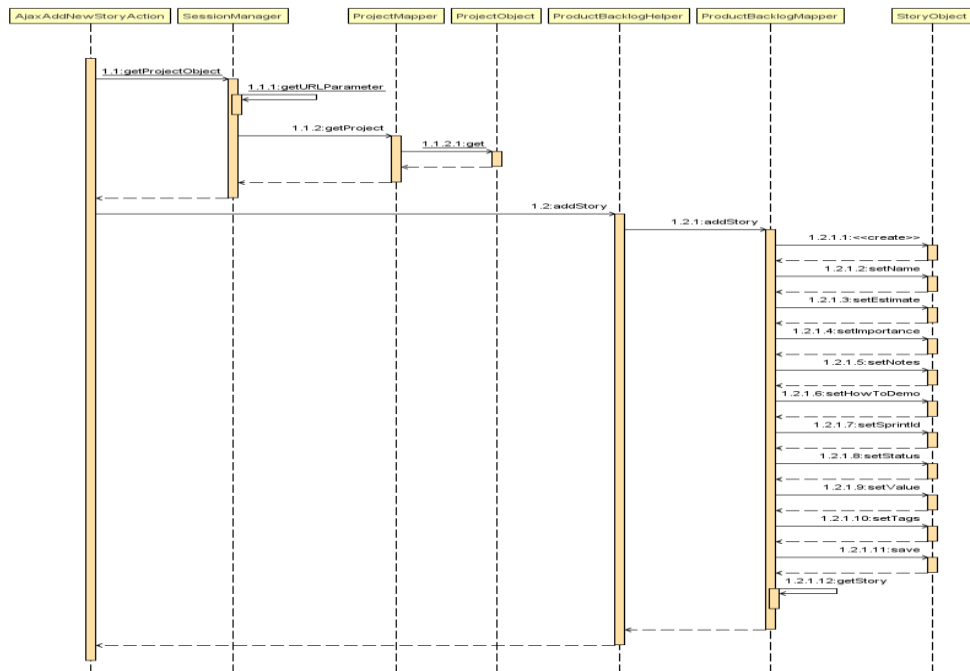
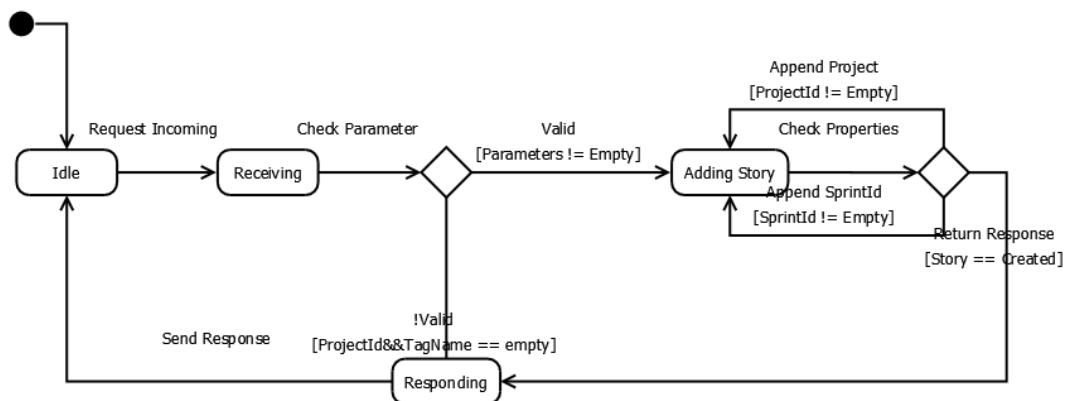


Figure 1.h: Sequence diagram for story creation scenario

Whenever the request comes from the user, the process would first pass through the action class AjaxAddNewStoryAction which stand in the user interface layer, the raw requested parameters that contain the Story Info are mapped into the StoryObject in the domain layer data. Product Backlog Helper in the application layer that responsible for tracking the progress status.

The specific work here occurs in Product Backlog Logic which responsible for maintaining the new story creation state and initializes the ProductBacklogMapper for mapping the Story Object to StoryDao for storing the data in the underlying database. Noticed that the id of Project Object is needed for basic parameter to instantiate Story Object, the relationship detail would be covered later.

Flow Chart 1 : The state of ezScrum and story creation process



The state diagram above exhibits the state of ezScrum and story creation process, state starting from the request that comes until appending the response back. From above analysis we noticed that StoryObject and ProjectObject are classes that residing in domain layer.

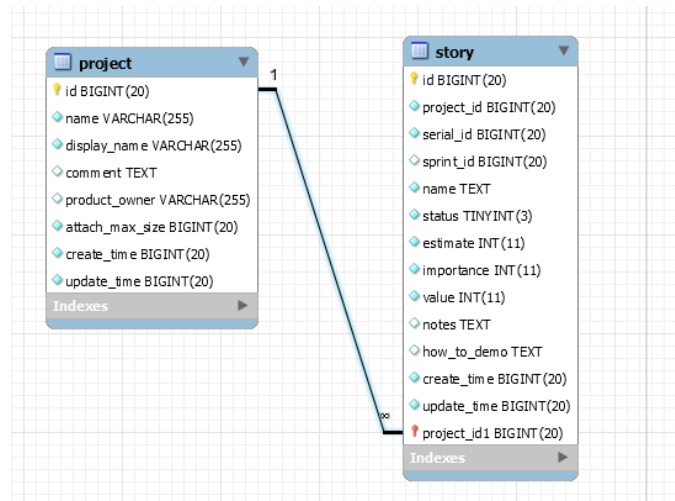


Figure 2.i: Project and Story relationship

Above figure describing the relationship between project and story which has one-to-many cardinality type. This type states that one project can contains with many story and many story can only belongs to one project and based on above analysis we extract these requirements as noted in the following table.

Table 3 : Project based analysis

| | |
|-----------|--|
| R1 | As a developer I want to develop api that serve for story creation, so that the client can store the story data in the microservice. |
| R2 | As a developer I want to develop api for project creation, so that the project data can be appended in my new story data. |
| R3 | As a developer I want to develop api for updating and removing the story, so that client can alter their story data. |
| R4 | As a developer I want to develop api for modifying the project attributes, so that client can alter their project data. |
| R5 | As a developer I want to develop api for retrieving a list of story based on parent project, so that the client can manage the detailed story in current running project |

Scenario 2: The tag scenario

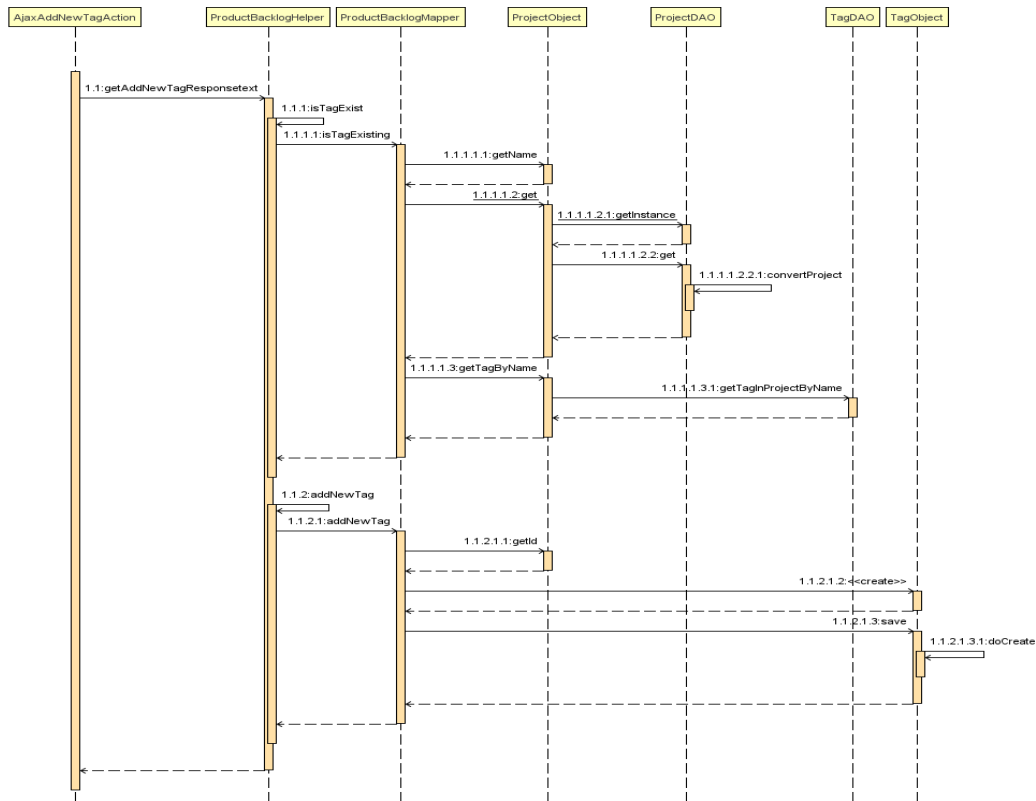


Fig.2 a : the process for tag creation in legacy ezScrum

The above figure describes the process for tag creation in legacy ezScrum, the request that forwarded to AjaxAddNewTagAction class, before the creation process, the system ensures that only one tag data with the similar name that exist in the current running project.

With the ProductBacklogHelper class delegates the checking task to ProductBacklogMapper class for looking up the database entry, if the new tag does not exist, the system can leap to creation process by delegating the process to TagDao class for storing it with defined SQL-query.

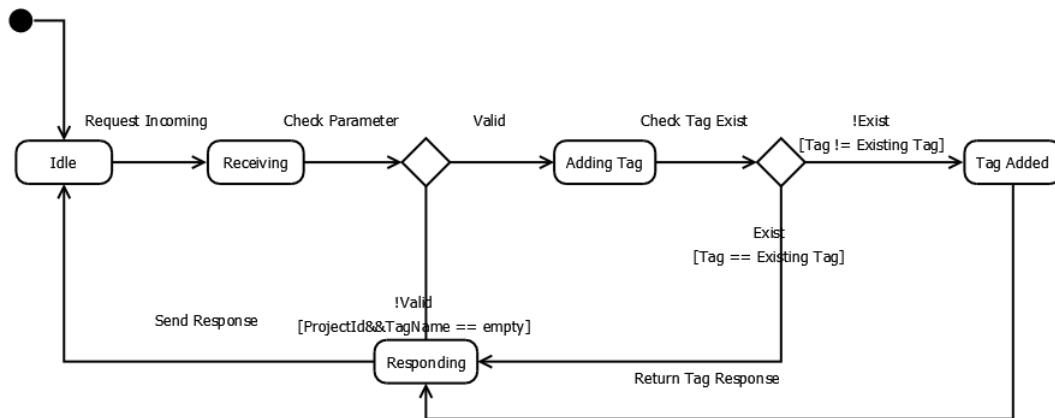
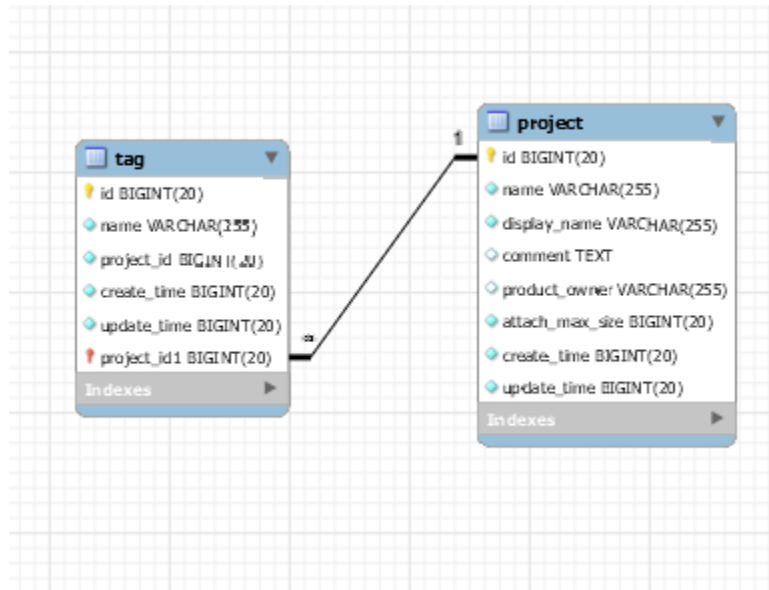


Fig.2b : State of ezScrum and for tag creation process

The state diagram above exhibits the state of ezScrum and for tag creation process state, the request must come with valid parameters that contains the tag data which would be stored in database if the tag does not exist and return the status back.

From the tag sequence diagram we noticed a sentence for ensuring that only one tag with similar tag_name exists in current running project, this sentence expressing the relationship between tag and project which one tag can only belongs to one project, whereas one project can have many tags as expressed in the following image.

Fig. 3: project tags



From the tag creation scenario analysis, it demands two object that should be involved in from domain layer TagObject and ProjectObject and extracts two following requirements:

Table 4 : Domain layer Tag Object

| | |
|----|---|
| R1 | As a developer I want to develop api tag for a tag creation, so that the client can manage their story with defined tag. |
| R2 | As a developer I want to develop api for tag checking, so that the client can ensure only one tag exist in current running project. |
| R3 | As a developer I want to develop api for updating and removing the tag, so that client can alter their tag. |

Scenario 3: Story and tag relationship

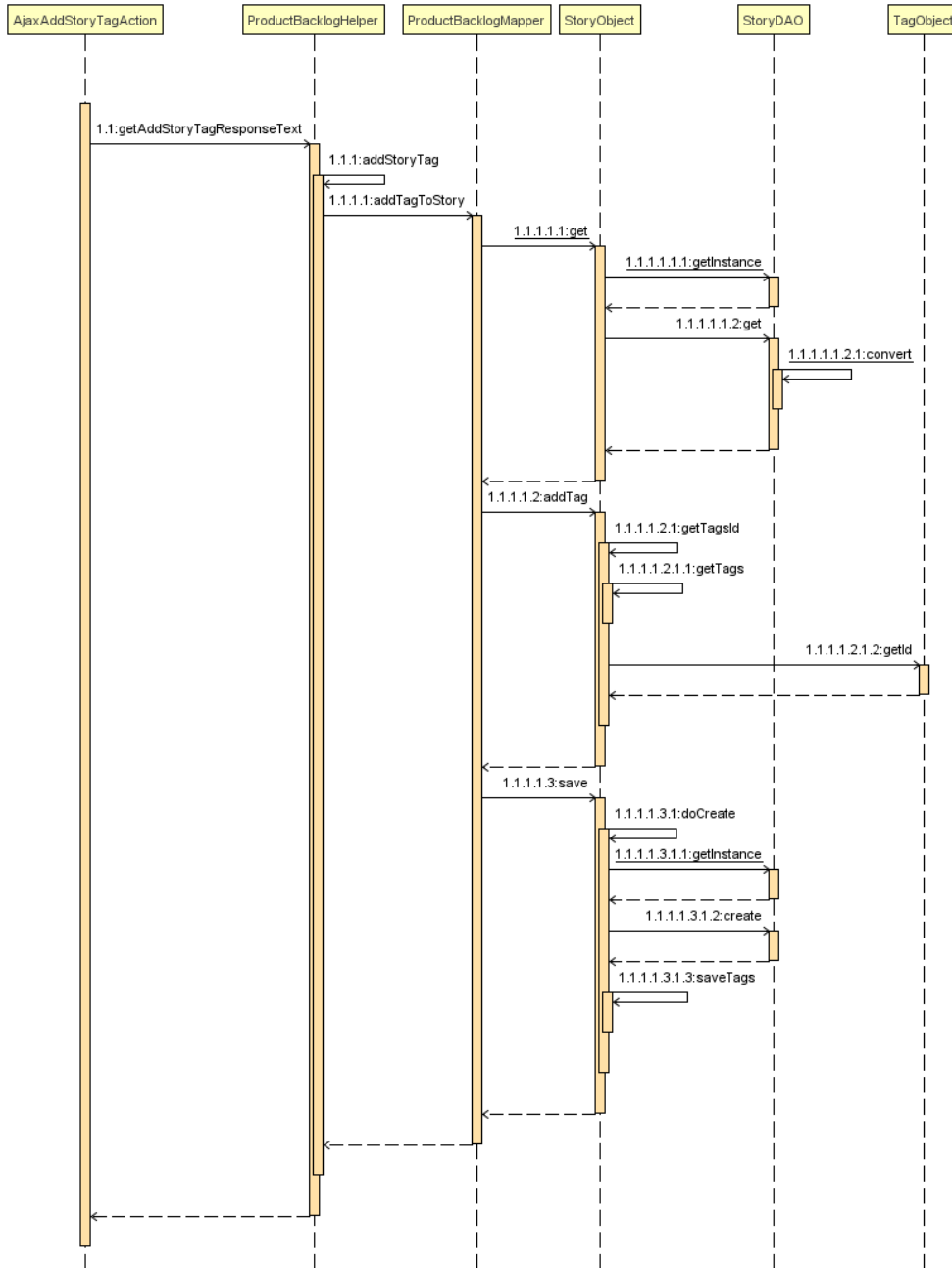


Fig. 4: Story and tag relationship

The valid requests for attaching story to tag job are firstly forwarded through AjaxAddStoryTagAction class, the ProductBacklogHelper class then delegating the job to the ProductBacklogMapper for retrieving the story and tag data from database and attach these two data by creating the new record in pivot table in mysql database.

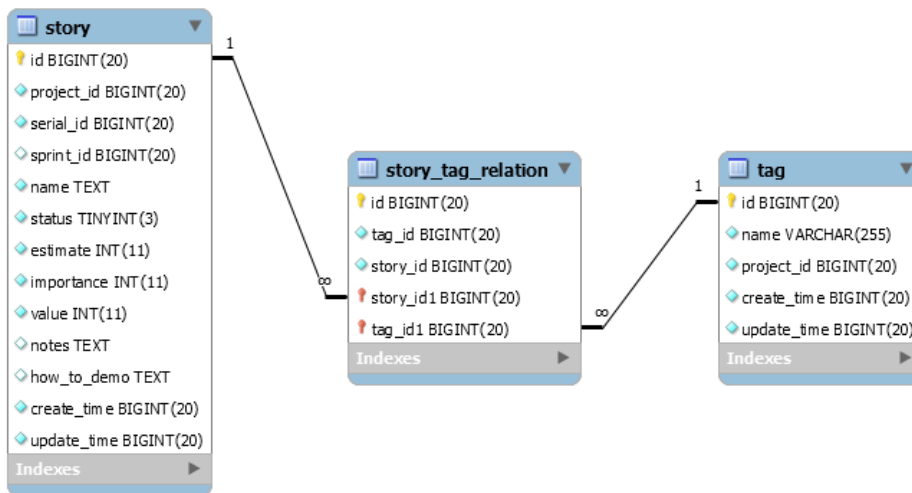


Fig. 5: Story and tag data from database

In ezScrum story and tag can be attached together by storing the story_id value and tag_id value in story_tag_relation table, this table acts as a pivot table for mediating the many-to-many relationship between tag and story which has the bidirectional relationship type, states that either story or tag can be the aggregate point for retrieving the relationship between the story and tag job which would be detailed in the requirement list in this section.

Table 5 : From above analysis we extracted the requirements as following:

| | |
|----|--|
| R1 | As a developer I want to develop api for attaching defined tags to story, so that the client can organize the user story based on tag |
| R2 | As a developer I want to develop api for removing attached tags from story, so that the client can change it whenever incorrect tagging. |
| R3 | As a developer I want to develop api retrieving a list of story based on certain tag, so that the client can organize the story with tag |

| | |
|------------|--|
| | that the client can change it whenever incorrect tagging. |
| R10 | As a developer I want to develop api retrieving a list of story based on certain tag, so that the client can organize the story with tag |
| R11 | As a developer I want to develop api for retrieving a list of story based on parent project, so that the client can manage the detailed story in current running project |

1.5. Product Backlog Microservice

Our product backlog microservice basically a small instance of service that should serve functionality which we have defined before in splitting works, however in the implementation phase microservice should also have a clear data exchange format, and communication channel, which will be consumed regardless of underlying technology is being applied by the client. Which we would discuss in this chapter including the architecture, communication mechanism, and microservice framework.

1.5.1. The architecture

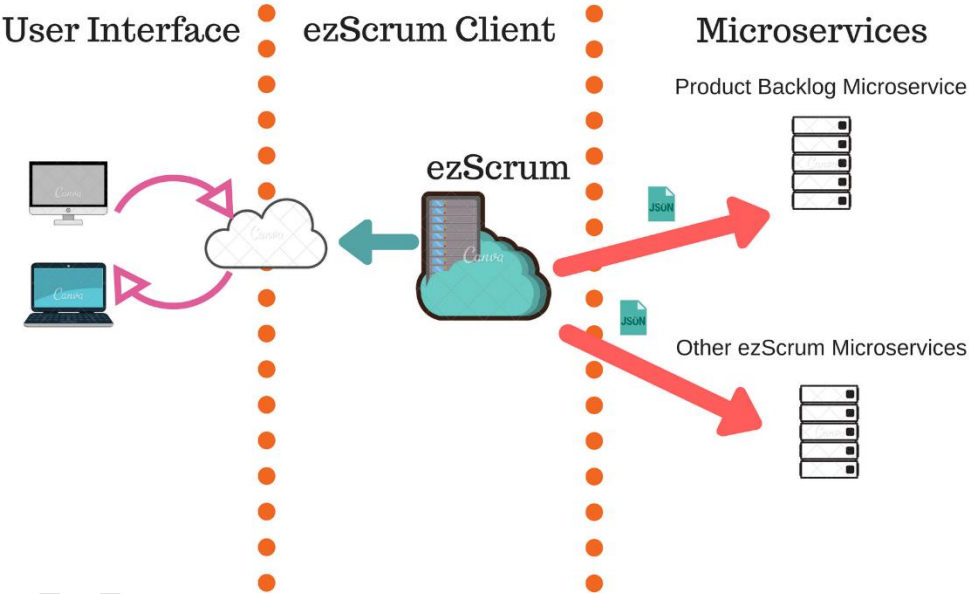


Fig. 7: Architecture of our microservice

The figure above describes the purposed architecture of our microservice. We split the purposed architecture into three tiers, the first tier is called user interface tier and the second tier is a legacy ezScrum which mainly communicate via web browser client through HTTP protocol, but in our project we did not touch the first tier since there are residing legacy services which are still not decomposed yet and need to be orchestrated by the legacy ezScrum in order to work, such as Sprint Backlog service, Taskboard service, and et cetera. Ideally when all these services have been established the API gateway

will take over the orchestration job so that the client in the first tier would leap to the third tier without needing to communicate to legacy ezScrum anymore.

In the third tier, the legacy ezScrum would communicate with Product Backlog Microservice using the API endpoints that are publicly published. This endpoint and detail will be discussed later.

1.5.2. Connect with product backlog microservice

When our product backlog microservice has been deployed properly, we need to expose the service to the client with common communication mechanism. Restful API with HTTP protocol is chosen to expose our microservice mainly because this concept encourages the developer to address the resource inside the service using the URL pattern which has been used widely over the world for exposing the resource. Our microservice resource would have an endpoint interface using the URL pattern as follows:

http://host:port/product_backlog/{resource_name_path}/{identifier}/{child_resource}

Since we are implementing the HTTP protocol the URL pattern would be started with http:// term follows by our microservice hostname and port. The product_backlog path is representing the context of our microservice followed by resource_name path in our service it could be a story, tag, project with identifier path that is required when the client need specific single resource and child_resource path for retrieving the related resource that belongs to the parent.

1.5.3. Microservice Framework

Spring boot is chosen as a microservice framework in implementation part, since it provides a production ready code with minimum configuration thus make the application up and running as fast as possible, spring also comes with embedded application server (Tomcat by default) so we don't need to build and deploy our application into WAR file. These two advantages align with our project goal, since we need to deliver our application for every sprint or iteration.

4. PRODUCT BACKLOG MICROSERVICE

So far we have extracted the ezScrum parts that related to product backlog context, started from isolating the layer contains the business domain, analyzing the database schema that which belonged to product backlog and capturing the product backlog business requirements. Further, we would develop the product backlog microservice by using the spring boot framework as an environment and properly up and running the microservice to ensure the extraction will not affect the operation of legacy ezScrum.

4.1. Product Backlog DB Entity Details

4.1.1. Entity Relationship Diagram

From the product backlog sub-domain analysis phase we have extracted the following entities in our development database, noted that some of the field are required by spring framework and not related to the product backlog context, which will be explained later in field description part.



Figure 8 Entity relationship diagram of product backlog microservice table

We have noticed between from above figure each entity is circularly connected, states that each entity could be an aggregate point for retrieving the entity which belongs to it. In our case for an instance we could retrieve the list of story or list of tag by picking up one single project and traverse the relationship to point out the story or tag that belongs to it; thus the relation between project and story or project and tag would be a one-to-many with condition that each project has 0 or more story and tag. Or we could also say each story and tag should only exist in one single project.

From the scenario analysis part, state that we could attach the story to tag or attach the tag to the story. Hence the relation between these entities would be many-to-many which states one tag could have zero or more story and one story could have zero or more tag. Because of the complexity of the relationship, we could not connect these two tables directly; thus we need the story_tag table as a pivot table that stores the relationship between them.

4.1.2. Entity Field Descriptions

Table 7 Project table field descriptions

| Field name | Data type | Description |
|------------------------|-----------|--|
| id | BIGINT | The unique identified for project, maximum 20 character |
| comment | VARCHAR | Additional details for the project, maximum 255 character, can be null |
| display_name | VARCHAR | The displayed project name in client, maximum 255, not null |
| name | VARCHAR | The project name, maximum 255 character and not null |
| product_owner | VARCHAR | The product owner name, maximum 255 character, not null |
| attach_max_size | BIGINT | A configurable maximum size of the attached file in MB, default 2 |
| created_time | DATETIME | Storing the date and time for the row creation in datetime format |
| updated_time | DATETIME | Storing the date and time for the last update row, in datetime format |

Table 8 : Story table field descriptions

| Field name | Data type | Description |
|-------------------|-----------|--|
| id | BIGINT | The unique identified for story, maximum 20 character |
| notes | TEXT | Additional details for the story |
| status | INTEGER | Story status, new : 1, assigned : 2, closed : 3 |
| name | VARCHAR | The story name, maximum 255 character and not null |
| importance | INTEGER | Importance level for the story, maximum 11 character, required |
| value | INTEGER | A value level for the story, maximum 11 character, |

| | | |
|---------------------|----------|---|
| | | required |
| estimate | INTEGER | An estimation value for the story, maximum 11 character, required |
| how_to_demo | TEXT | Demo description |
| project_id | INTEGER | Reference to the project table, not null |
| serial_id | INTEGER | Reference to the serial, not null |
| sprint_id | INTEGER | Reference to the sprint id |
| created_time | DATETIME | Storing the date and time for the row creation in datetime format |
| updated_time | DATETIME | Storing the date and time for the last update row, in datetime format |

Table 9: Tag table field descriptions

| Field name | Data type | Description |
|---------------------|-----------|---|
| id | BIGINT | The unique identified for tag, maximum 20 character |
| name | VARCHAR | The tag name, maximum 255 character and not null |
| project_id | BIGINT | A reference value to the project |
| created_time | DATETIME | Storing the date and time for the row creation in datetime format |
| updated_time | DATETIME | Storing the date and time for the last update row, in datetime format |

Table 10 : Story tag pivot table, field descriptions

| Field name | Data type | Description |
|-----------------|-----------|---------------------------------|
| story_id | BIGINT | A reference value for the story |
| tag_id | BIGINT | A reference value for the tag |

4.2. Product backlog API contracts

Our API endpoint pattern would be like the following with the orange colored text path represent the context prefix and black colored text path represent the resource:

http://host:port/product_backlog/{resource_name_path}/{identifier}/{child_resource}

From the product backlog requirements, we have developed a list of API endpoint contracts with the pattern like as the following table:

Table 11 Product Backlog API endpoint list

| RESOURCE | METHOD | Description |
|---|--------|--|
| /project/projects | GET | Retrieving all the project in microservice. |
| /project/get/{id} | GET | Retrieving single project data with appended id {id}. |
| /project/get_by_name/{name} | GET | Retrieving single project data with appended project name {name}. |
| /project/check_by_name/{name} | GET | Provides the Boolean data for checking the project existence. |
| /project/create | POST | Create single project with appended project data in request body. |
| /project/update/{id} | PUT | Update the project with id {id} and appended project data in request body. |
| /project/delete/{id} | DELETE | Remove single project data with id {id}. |
| /project/{id}/stories/{filterType} | GET | Retrieving the stories that belongs to project id {id} with optional filter type {filter type} ALL, BACKLOG, DONE, DETAIL. |
| /project/{id}/tags | GET | Retrieving the tags that belongs to project with id {id} . |

| | | |
|--|--------|---|
| /story/stories/{filterType} | GET | Retrieving the stories with optional filter type {filter type} ALL, BACKLOG, DONE, DETAIL. |
| /story/get/{id} | GET | Retrieving single story data with appended id {id}. |
| /story/get_by_project_serial/{project_id}/{serial_id} | GET | Retrieving single story data with appended project id {project_id} and serial id {serial_id}. |
| /story/{id}/project | GET | Retrieving project that belongs to story with id {id} |
| /story/{id}/tags | GET | Retrieving tags that attached to story with id {id} |
| /story/{id}/attach_tag/{tag_id} | GET | Attach story with id {id} to the tag with id {tag_id} |
| /story/{id}/un_attach_tag/{tag_id} | GET | Unattached story with id {id} to the tag with id {tag_id} |
| /story/create | POST | Create single project with appended story data in request body. |
| /story/update/{id} | PUT | Update the story with id {id} and appended story data in request body. |
| /story/delete/{id} | DELETE | Remove single story data with id {id}. |
| /tag/create | POST | Create single tag with the appended tag data in request body |
| /tag/update/{id} | PUT | Update single tag data with id {id} and appended tag data in request body. |
| /tag/delete/{id} | DELETE | Delete single tag data |

| | | |
|--------------------------|-----|--|
| | | with id {id} |
| /tag/{id}/project | GET | Retrieving project data that has the tag id {id} |
| /tag/{id}/stories | GET | Retrieving the stories data that has the tag id {id} |

UNDER PEER REVIEW

4.3. Class Diagram

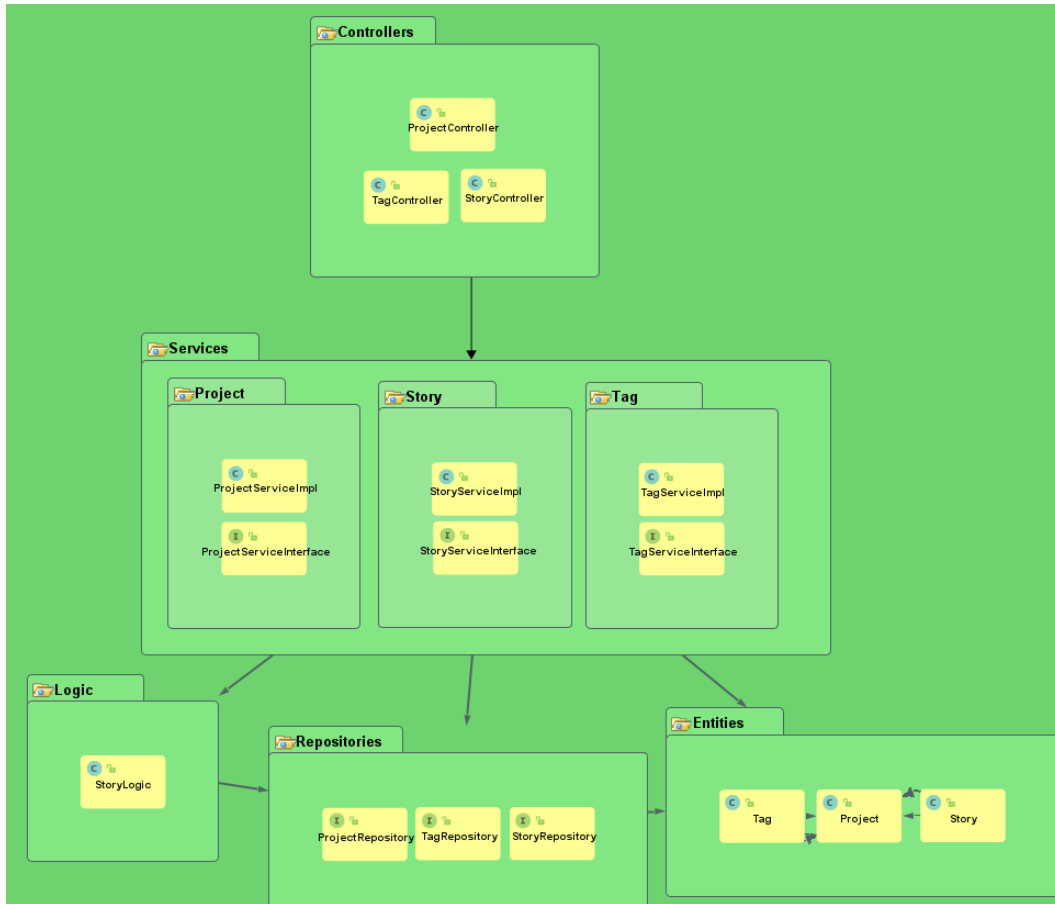


Figure 9 Class diagram of the product backlog microservice

The product backlog microservice has three controllers which contain all the microservice API endpoints that we have explained in chapter 9. The classes are the slim class with responsibility for handling the incoming HTTP request, and the request further delegated to service package.

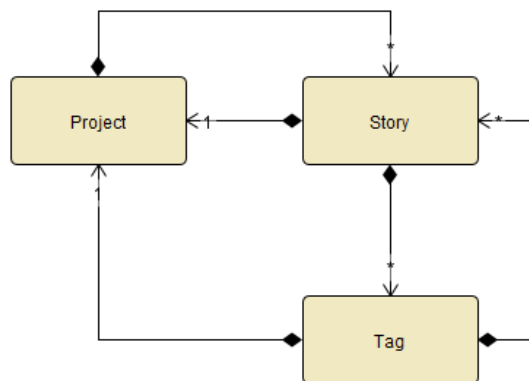


Figure 10. Product Backlog Microservice Entity Relationship Diagram

The service package has three class packages which are responsible for maintaining the state of the request progress and delegates more complicated job, business rules and computation in the logic layer which for now only has one class.

The Repositories is Java Persistence API the class implementing the repository enterprise architecture, and this implementation is needed for mediating the access between the domain and data mapping layer in spring boot technology[7].

4.4. The integration

We have picked up several AT test case scenario which related to the product backlog requirements with additional test case scenario that might potential having a share data with product backlog context. This phase intended to ensure our product backlog microservice has been appropriately implemented in the legacy ezScrum and covered all the requirements that are needed by the client to run the application without any defecting the existing functionality of our legacy ezScrum.

Test Statistics

| Total Statistics | | | | | | |
|------------------|-------|------|------|----------|-------------|--|
| | Total | Pass | Fail | Elapsed | Pass / Fail | |
| Critical Tests | 117 | 117 | 0 | 01:41:50 | | |
| All Tests | 117 | 117 | 0 | 01:41:50 | | |

| Statistics by Tag | | | | | | |
|-------------------|-------|------|------|----------|-------------|--|
| | Total | Pass | Fail | Elapsed | Pass / Fail | |
| AccountManagement | 13 | 13 | 0 | 00:11:18 | | |
| Bug | 4 | 4 | 0 | 00:03:48 | | |
| Concurrent | 1 | 1 | 0 | 00:03:35 | | |
| FilterDone | 1 | 1 | 0 | 00:01:09 | | |
| Import | 6 | 6 | 0 | 00:04:35 | | |
| Login | 1 | 1 | 0 | 00:00:10 | | |
| ProductBacklog | 18 | 18 | 0 | 00:15:33 | | |
| Project | 3 | 3 | 0 | 00:03:48 | | |
| ReleasePlan | 8 | 8 | 0 | 00:06:14 | | |
| Retrospective | 4 | 4 | 0 | 00:02:21 | | |
| ScrumReport | 1 | 1 | 0 | 00:00:24 | | |
| ScrumRole | 11 | 11 | 0 | 00:11:08 | | |
| Serial Number | 8 | 8 | 0 | 00:07:00 | | |
| SprintBacklog | 13 | 13 | 0 | 00:08:55 | | |
| SprintPlan | 8 | 8 | 0 | 00:07:41 | | |
| tag | 4 | 4 | 0 | 00:03:09 | | |
| TaskBoard | 19 | 19 | 0 | 00:16:58 | | |
| Unplan | 3 | 3 | 0 | 00:02:10 | | |
| Unstable | 9 | 9 | 0 | 00:11:30 | | |

| Statistics by Suite | | | | | | |
|---|-------|------|------|----------|-------------|--|
| | Total | Pass | Fail | Elapsed | Pass / Fail | |
| robotTesting | 117 | 117 | 0 | 01:42:27 | | |
| robotTesting_ezScrum 01 Login TestSuite | 1 | 1 | 0 | 00:00:10 | | |
| robotTesting_ezScrum 02 Project TestSuite | 3 | 3 | 0 | 00:03:48 | | |
| robotTesting_ezScrum 03 ProductBacklog Mark TestSuite | 2 | 2 | 0 | 00:02:08 | | |
| robotTesting_ezScrum 03 ProductBacklog Search TestSuite | 1 | 1 | 0 | 00:02:59 | | |
| robotTesting_ezScrum 03 ProductBacklog TestSuite | 15 | 15 | 0 | 00:10:27 | | |
| robotTesting_ezScrum 04 SprintPlan TestSuite | 8 | 8 | 0 | 00:07:41 | | |
| robotTesting_ezScrum 05 SprintBacklog TestSuite | 13 | 13 | 0 | 00:08:56 | | |
| robotTesting_ezScrum 06 AccountManagement TestSuite | 13 | 13 | 0 | 00:11:18 | | |
| robotTesting_ezScrum 07 ScrumRole TestSuite | 11 | 11 | 0 | 00:11:08 | | |
| robotTesting_ezScrum 08 ReleasePlan TestSuite | 8 | 8 | 0 | 00:06:14 | | |
| robotTesting_ezScrum 09 TaskBoard TestSuite | 19 | 19 | 0 | 00:16:58 | | |
| robotTesting_ezScrum 10 Retrospective TestSuite | 4 | 4 | 0 | 00:02:21 | | |
| robotTesting_ezScrum 11 Unplan TestSuite | 3 | 3 | 0 | 00:02:10 | | |
| robotTesting_ezScrum 12 Import TestSuite | 6 | 6 | 0 | 00:04:35 | | |
| robotTesting_ezScrum 13 Serial Number TestSuite | 8 | 8 | 0 | 00:07:01 | | |
| robotTesting_ezScrum 16 ScrumReport TestSuite | 1 | 1 | 0 | 00:00:24 | | |
| robotTesting_ezScrum 99 Concurrent TestSuite | 1 | 1 | 0 | 00:04:09 | | |

Figure 11 Automatic testing result

5. CONCLUSION

As we have accomplished so far, we have reached our primary goal to split the product backlog context from ezScrum application, our work progress has extracted 11 requirements which we have converted into single product backlog microservice that serves with 24 API endpoints, in the splitting process we first came through for understanding our problem which is the context of our product backlog, we have identified the boundary of this context by using the general concept of product backlog because we did believe that each component or sub-domain is firmly connected and would share the data in frequent time within this context. This DDD concept was became our firm guidelines for splitting the product backlog related from ezScrum.

Then based on the sub-domains which we obtained above, we extracted several business scenarios which aligned within the product backlog context and pointed out the java class and database entities that representing the data model by running these scenarios in the legacy ezScrum. Then we generated the requirement list based on the scenarios and its specification based on the data model.

Further, we developed the product backlog micro service which serving 24 API endpoint that satisfied the requirement list. Although we have established a well prepared a product backlog micro service but still there are some aspect that we would seize this opportunity in the future such as: how we coordinate and arrange since we will have more micro service eventually, how do we ensure our micro service will keep serving whenever exception appears so that our micro service will not be a single point of failure, and the last but not least how our micro service will be more tenacious from external interference since our micro service directly facing the public and we need to have a mechanism to hide the actual implementation to filter out the malicious request is addressing to it.

REFERENCES

1. Sedeño, J., et al. *Modelling agile requirements using context-based persona stories*. in *WEBIST 2017: 13th International Conference on Web Information Systems and Technologies (2017)*, p 196-203. 2017. ScitePress Digital Library.
2. Mereu, S., et al. *Top-down vs bottom-up approaches to user segmentation: The best of both worlds*. in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. 2017. SAGE Publications Sage CA: Los Angeles, CA.
3. K. Beck, M. Beedle, and a Van Bennekum, "Principles behind the agile manifesto," *Retrieved*, pp. 2–3, 2001.
4. M. Fowler, "Monolith First," 2015. [Online]. Available: <https://martinfowler.com/bliki/MonolithFirst.html>.
5. E. Evans, *Domain Driven Design*. 2006.
6. G. Bakyei, "Adding new Functionalities to a Legacy System with Microservices: A case study of ezScrum," 2017.
7. 鄭安發, "Implementing Account Management Service using the Microservice Architecture: A Case Study of ezScrum."
8. D. Alur, J. Crupi, and D. Malks, "Core J2EE Patterns," *Design*, p. 650, 2003.
9. M. Fowler, *Patterns of Enterprise Application Architecture*, vol. 23. 2003.
10. P. Deemer, G. Benefield, C. Larman, and B. Vodde, "The Scrum Primer," *InfoQ*, pp. 1–20, 2012.
11. Tryfonas, T. and I. Askoxylakis, *Human Aspects of Information Security, Privacy, and Trust*. 2013: Springer.
12. Losana, P., et al., *A Systematic Mapping Study on Integration Proposals of the Personas Technique in Agile Methodologies*. *Sensors (Basel)*, 2021. **21**(18).
13. Wolkerstorfer, P., et al., *Probing an agile usability process*, in *CHI'08 Extended Abstracts on Human Factors in Computing Systems*. 2008. p. 2151-2158.
14. Seffah, A., J. Gulliksen, and M.C. Desmarais, *Human-centered software engineering-integrating usability in the software development lifecycle*. Vol. 8. 2005: Springer Science & Business Media.
15. Felderer, M. and G.H. Travassos, *Contemporary Empirical Methods in Software Engineering*. 2020: Springer.