Expertise
and insight
for the future

Ville Friman

Software Architectures for Reusable Cloud Components

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's thesis

4 December 2020

Tekijä(t) Otsikko Sivumäärä Aika	Ville Friman Software Architectures for Reusable Cloud Components 58 sivua 4.12.2020
Tutkinto	Insinööri (YAMK)
Tutkinto-ohjelma	Tieto–ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaaja	Ville Jääskeläinen, Principal Lecturer

Ohjelmistokehitys on nopeutunut huomattavasti viimeisen vuosikymmenen aikana teknologioiden kehittymisen ja uusien ohjelmistometodologien vuoksi. Samaan aikaan myös liiketoimintaympäristö on jatkuvan muutoksen alla, ja muutokseen vastaamiseen tarvitaan hyvin usein ohjelmistokehistystä joko olemassa olevaan sovellukseen tai täysin uuden sovelluksen kehittämisen. Nämä jatkuvat muutokset ja niiden nopeus vaativat ohjelmistoarkkitehtuurilta paljon, jotta arkkitehtuuri ei rapaudu.

Tässä opinnäytetyössä käydään aluksi läpi miksi ohjelmistoarkkitehtuuri on tärkeä, ja mitä tunnuspiirteitä voidaan joutua huomioimaan ohjelmistoarkkitehtuuri mietittäessä. Tunnuspiirteistä valittiin kolme tunnuspiirrettä, jotka tukevat ohjelmiston muutosta, laajennettavuutta ja ylläpidettävyyttä. Valittujen tunnuspiirteiden pohjalta esitellään kaksi ohjelmistoarkkitehtuuria.

Opinnäytetyön lopputuloksena on pieni sovellus, joka on toteutettu esitellyillä ohjelmistoarkkitehtuureilla. Molemmilla ohjelmistoarkkitehtuuritoteuksilla on vielä kaksi erillistä teknistä toteutusta pilvialuistoille, jotka molemmat käyttävät yhteistä liiketoimintalogiikkaa. Tehty sovellus osoittaa, että valitut ohjelmistoarkkitehtuurit tukevat sovelluksen muuttamista, laajennettavuutta ja ylläpidettävyyttä. Sovellusarkkitehtuuria mietittäessä on kuitenkin hyvä selvittää ovatko nämä valitut ominaisuudet sovelluksen kannalta tärkeitä vai ei. Sovellusarkkitehtuurissa on aina lopulta kyse erilaisista painotuksista ja vaihtoehdoista, jotka voivat olla toisensa poissulkevia.

Avainsanat	Ohjelmistoarkkitehtuuri, Pilvipalvelu, Clean Architecture, Ports and Adapters

Author(s) Title Number of Pages Date	Ville Friman Software Architectures for Reusable Cloud Components 58 pages 4 December 2020
Degree	Master of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructor	Ville Jääskeläinen, Principal Lecturer

Software development has become faster during last decade through technological advances and new software development methodologies. Same time business is in constant stage of change. To meet the requirements of this stage business requires changes in existing software applications and also new applications. These constant change requirements put lots of stress on the software architecture.

This thesis first goes explaining why software architecture matters and what kind of characteristics there are to consider when choosing software architecture. From those characteristics three are chosen that support change, extendability, and maintability. Based on those characteristics two software architectures are chosen.

A small application was implemented with the chosen architectures as a proof of concept. The application is a cloud application which supports more traditional way to serve software and newer more on demand way to serve software while sharing the business logic. The application shows that it is possible to implemented reusable, extendable and maintable application with the chosen architectures. The software characteristics should be always weighted against the needs of the project because different characteristics might be exclusive of each other.

Keywords	Software architecture, Cloud, Clean Architecture, Ports and Adapters

Contents

1 Introduction **1**

2 The Software Architecture **2**

2.1 Why the Software Architecture Matters 2
2.2 The Characteristics of the Software Architecture 4
2.2.1 The Operational Characteristics 5
2.2.2 The Structural Characteristics 5
2.2.3 The Cross-cutting Characteristics 6

3 The Architecture Styles for a Cloud Application **7**

3.1 The Microservice Architecture 7
3.2 The Serverless Cloud Computing 8

4 Software Architecture Styles for a Service **10**

4.1 The Ports and Adapters Pattern 10
4.1.1 The Core 12
4.1.2 The Ports 12
4.1.3 The Adapters 12
4.2 The Clean Architecture 13
4.2.1 The Entities 15
4.2.2 The Use Cases 15
4.2.3 The Interface Adapters 15
4.2.4 The Frameworks and Drivers 16

5 The Demo Application **17**

5.1 The Ports and adapters application 17
5.1.1 The Core Module 18
5.1.2 The Microservice Module 24
5.1.3 The Serverless Module 27
5.2 Clean architecture application 37
5.2.1 The Core module 38
5.2.2 The Microservice module 49
5.2.3 The Serverless module 51

6 Conclusions **56**

6.1 Future Research 56

Bibliography **57**

List of Abbreviations

API	Application Programmed Interface.
AWS	Amazon Web Services.
BBoM	Big Ball of Mud.
CD	Continuos Deployment / Delivery.
CI	Continuos Integration.
DDD	Domain Driven Design.
DTO	Data Transfer Object.
GDPR	General Data Protection Regulation.
GUI	Graphical User Interface.
IoC	Inversion of Control.
JSON	Javascript Object Notation.
REST	Representational state transfer.

1 Introduction

The software development cycle has grown shorter due rise of the cloud services which made possible to provision new applications and database servers at will. The cloud services enabled new practices such as Continuos Integration (CI), Continuos Deployment / Delivery (CD) and DevOps. These practices make possible to have multiple production deployments in a day.

Fast pace of the development puts lot of stress on the software architecture. How the software architecture can keep up if there are tens or hundreds production deployments a day and new ways to run the software, such as serverless computing, comes almost every year?

The business domain and its needs also does not evolve as fast as software development trends or technology. An application developed five or ten years ago can be still critical for business but application's technology stack can be outdated. Moving the application to cloud platform and updating the technology stack without software architecture which supports code leverageability might be a difficulty and costly task.

This thesis presents two software architecture models for a microservices which should help to tackle these problems and a proof of concept application with both the presented architectures. This thesis is not about the architecture of entire distribute application but instead of a one component in the distribute application. The presented architectures give blueprints to an application which should be able to withstand fast pace of modern application development and the ever changing business requirements.

2 The Software Architecture

The software has become ubiquitous. It is all around us and there is basically no business area left which is not at least partially affect by it. Because of this more software systems are build for the needs of business than ever. The business and its goals are often abstract ideas but the developed software system is concrete. The software architecture can be seen as a bridge between the abstract business goals and the concrete system.

The software architecture can defined multiple ways. One way to define it was introduced by Philippe Kruchten in his paper ”Architectural Blueprints—The “4+1” ViewModel of Software Architecture” [1].

> The fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.

Ralph Johnson has given less formal definition for the software architecture

> The important stuff (whatever that is).

Both quotes has same the idea behind them. The software architecture has something to do with the core values and concepts of the domain where the software system should work. The architecture can and should evolve along with the fundamental concepts or the properties. There are too many software architectures and architectural styles to go through. Therefore this thesis is about represents two styles where the business domain is middle of the architecture.

2.1 Why the Software Architecture Matters

The Software architecture is not something that user sees directly like user interface or new features. Usually users notice the poor software architecture indirectly. The application does not get new features as fast as it used to or new version of the application has new defects in old features. There are of course other reasons why these things may happen but the software architecture is one of them.

Neglecting the application's architecture usually leads to Big Ball of Mud (BBoM) [2] architecture. In their article Footer and Yoter defined the Big ball of mud as following

> A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined.

Adding new features or extending the existing features in that kind of code base is very hard. The application developers don't know what kind of side effects a code change may have in the other parts of the system without extensive testing. This usually leads to a copy-paste style of coding where the developer just copies the existing functionality and pastes it to a new function as is or with little modifications. The copy-paste style programming leads to a situation where the application and the business logic are duplicated in multiple places in the application. This makes finding and changing them difficult. Missing one the functions usually leads to the application's inconsistent behavior.

There are ways to mitigate this problem like a code refactoring [3]. The code refactoring means changing the code's internal structure without changing its external behavior. Unfortunately developers usually use the code refactoring in a very limited scope, like renaming variables or splitting up a large function into smaller private functions in a class scope. Local refactoring is not futile but does not help in situation described by Footer and Yoter where developers need to break up direct connections between the different parts of the system or push down information from the global state.

These kind of refactorings require lots of resources like money, time, skilled developers and domain experts. If the application is under active development a large scale refactoring comes more challenging because then the situation reminds a race where the teams and people who are doing the refactoring must keep up with the teams and people who are adding or fixing the features.

One way to solve the problem is to define the architecture before the implementation of the application starts like in the waterfall process Figure 1.

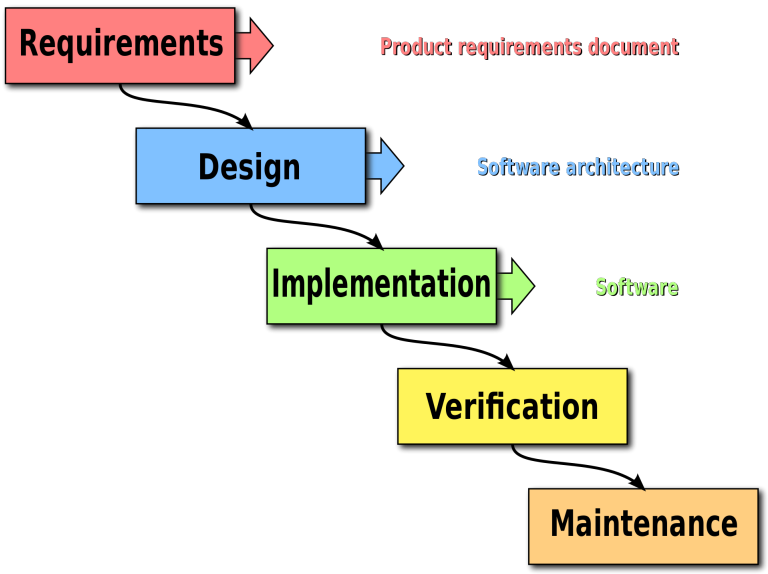

Figure 1: The waterfall process [4].

In the waterfall process the application's architecture is defined before the implementation, usually in very detailed manner. This is possible because the software requirements are defined and locked in in first phase of the process. In the agile development process this won't work because the requirements are not known or they can change during the implementation phase. So predefined and very detailed architecture can be too rigid to response changes in the software requirements.

2.2 The Characteristics of the Software Architecture

The software architecture is always more or less about the trade-offs between different architectural characteristics. Choosing what characteristics the application should support depends on multiple factors e.g. business domain, legislation, chosen platform, expected user count. A medical application which runs in hospital has very different requirements for the architecture characteristics than a social media application. The application should not try to support all the characteristics because it may lead to a situation where the application does not support any of those characteristics.

In the "Fundamentals of Software Architecture: An Engineering Approach" book [5] Mark Richards and Neil Ford define three categories for these characteristics with incomplete lists different kinds of abilities. The three categories are operational, structural and cross-

cutting.

2.2.1 The Operational Characteristics

The operational characteristics are heavily related to the operations, DevOps, and the application's performance. A partial list of the operational characteristics:

- Availability
- Continuity
- Performance
- Recoverability
- Reliability / safety
- Robustness
- Scalability

The performance characteristic is mostly like the best known characteristics in this category. The Availability and scalability characteristics are also well known especially in the cloud platforms.

2.2.2 The Structural Characteristics

The structural characteristics are related the application's code structure, modularity, and coupling between modules. A partial list of the structural characteristics:

- Compatibility
- Configurability
- Extensibility
- Installability
- Interoperability
- Leverageability / reuse
- Localization
- Maintability
- Portability
- Supportability

- Upgradeability

This category is most affected by the domain where the application is running. For example an embedded application rarely needs to support localization but a web application which supports users all over the world needs it.

2.2.3 The Cross-cutting Characteristics

The cross-cutting characteristics are characteristics which are needed to run an application but don't fall easily under any other category. A partial list of the cross-cutting characteristics:

- Accessability
- Archivability
- Authentication
- Authorization
- Legal
- Privacy
- Security
- Supportability
- Usability / achievability

This category has lots of characteristics which are not directly related to the application's business domain but are required so that the application can operate properly or by the third party like legislation such as the General Data Protection Regulation (GDPR).

3 The Architecture Styles for a Cloud Application

A cloud based application can be deployed into a cloud infrastructure as a monolith application but usually so called cloud native applications are deployed as multiple services or function based services. Following chapters will introduce two most common architecture for that kind of deployment.

3.1 The Microservice Architecture

The microservice architecture has risen to meet the needs of small autonomous teams and faster release cycle. The microservice architecture was made possible when the cloud providers made possible to create new services at will and automate building of the infrastructure needed by the software system.

Generally the microservices can be described with following principles [6]

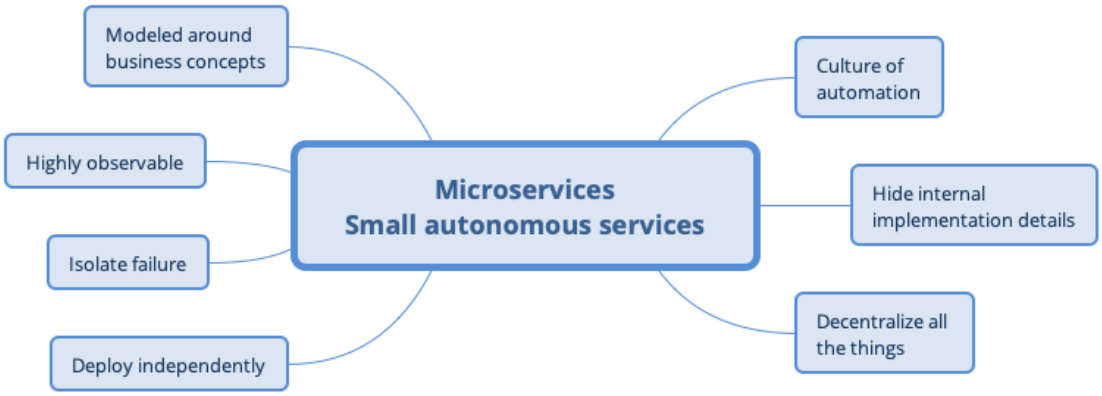

Figure 2: Microservice principles.

The central principle is that the microservices are small autonomous services but there is no exact definition how small a microservice should be. In upper right corner is the culture of automation principle which is required because an application can have tens or hundreds of microservices running simultaneously. Maintaining manually the microservices is usually unfeasible even impossible. Hiding the internal implementation principle

is required so that the clients of a microservice don't start depend how the microservice is implemented. If the clients become to depend the implementation details it makes replacing the microservice difficult. Decentralize all the things principle is required because the microservice should be autonomous. The team which owns the microservice should be able to release the service without coordination. Of course this may not be always possible but coordination with other teams or stakeholders should be exception not norm.

In the lower left corner is the deploy independently principle which is related to decentralize all things but more on a technical level. A microservice should be deployable without deploying other microservices and clients. Isolate failure principle is required so that the application does not suffer a cascading failure when one or more microservices suffer errors. This means that the failure scenarios should be part of the microservice design and implementation. Highly observable principle is required by the microservice architecture's distributed nature. In a distributed system it is not enough to observe a state of a single instance. The system should be observed holistically to have whole picture of the system's state. Modeled around the business concepts principle is required so that the microservice architecture can respond the business' needs more easily. When the microservice is modeled around the business concepts change in a related business should require change only in that microservice and possible in related the microservices. The Domain Driven Design (DDD) is commonly used technique to map business concepts into the microservices.

3.2 The Serverless Cloud Computing

The Serverless cloud computing means that the platform provider is responsible for everything else except the application code itself. The platform provider is responsible underlying infrastructure, execution of the code, and scaling the code. The serverless cloud computing model gained popularity since the Amazon Web Services (AWS) introduced the AWS Lambdas in 2014. The lambdas or functions as they are called on other cloud platforms can be compared to a function or a method in programming languages. They are usually simple, message driven, and focused to execute a single task.

Compared to a microservice a single serverless function is simpler and more task oriented.

Where a microservice was defined as a small autonomous service, a serverless function could be defined as a single autonomous task executor.

Because of focusing executing a single task the serverless functions require more refined model of the business domain. E.g. in the microservice architecture there could be a single microservice build around the order processing in a webshop application, the matching single serverless function would be considered too big. More serverless way would be finding what tasks an order should support and implement each of those tasks as a serverless function.

4 Software Architecture Styles for a Service

Following sections introduce two architecture styles. These styles were chosen because they represent the idea introduced in the chapter 2. The styles are focused on structural characteristics listed in the subsection 2.2.2, especially in the extensibility, the leverageability, and the maintability characteristics. There are of course other similar architectures like the DCI architecture [7], the EBC architecture [8] or the Screaming architecture [9]. Order of the following architectures is that the Ports adn adapters style is the simpler and the more light weighted and the Clean architecture is more complex.

4.1 The Ports and Adapters Pattern

The ports and adapters pattern was proposed by Alistair Cockburn in 2005. The ports and adapters pattern is also known as a Hexagonal architecture which was a working name for the pattern. Cockburn has given the architecture following description and motivation [10]:

> Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

The figure Figure 3 shows a high level picture of the pattern.

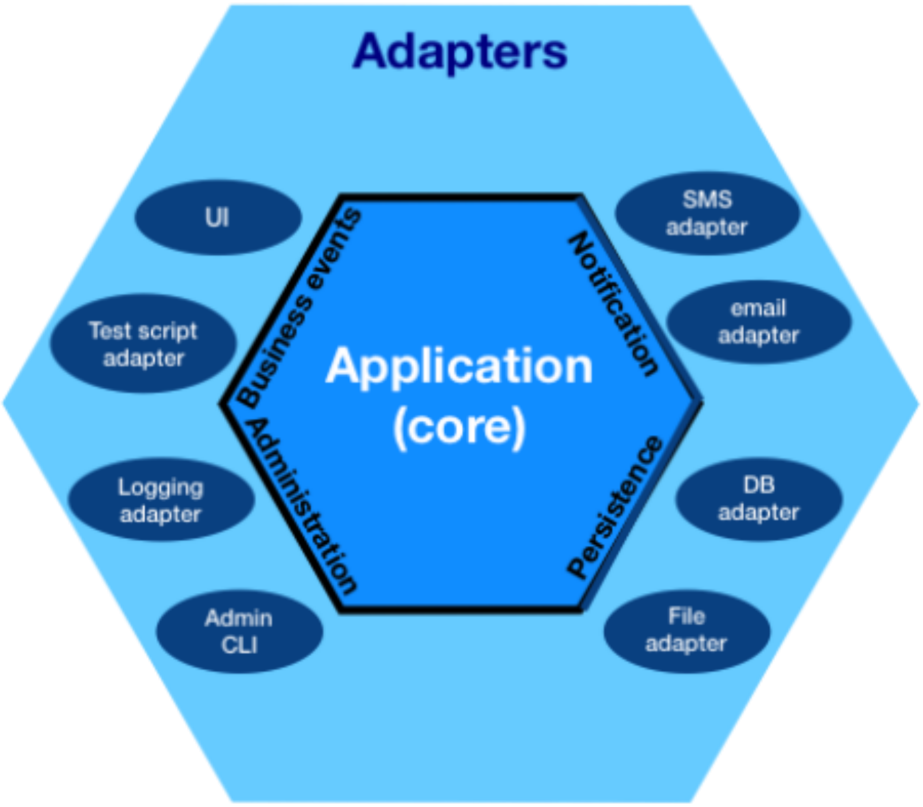

Figure 3: The Ports and Adapters pattern [11]

On the left side of the picture there are primary actors or drivers, and on the right side secondary actors or those who are driven. Difference between these two actors is that primary actors are the ones who initiate actions in the application or ”drive” the application. Secondary actors are services that the application itself commands or ”drives”.

There are two types of secondary actors, repositories and recipients. The repository actors are data storages where the application can save or read data. The recipient actors are either a fire-and-forget type, where the application just sends a notification to an actor and does not care response, or a request-response type of actor where application expects or requires a response from the actor. An example of a fire-forget actor could be sending an order confirmation email to a customer. An example of a request-response actor could be a request to a payment gate where the application waits that the payment has gone through so that a order process can continue.

4.1.1 The Core

On the dark blue center there is a application core which contains the business logic. The core shouldn't have any references to outer section. The core defines protocols or Application Programmed Interface (API)'s which are then used to communicate with a outside world. This architecture model does not say anything about how the core itself is structured, so it can be even BBoM. The core should be technology agnostic and should not have any outside dependencies such as frameworks. Of course this is not always feasible and there are cross-cutting concerns like logging where well know framework is better than implementing your own logging framework. The outside dependencies should be carefully weighted against pros and cons.

4.1.2 The Ports

The ports work as a application boundary which hides the application core from both the primary and secondary actors. The ports also provide means to interact with the application core for the actors. The ToDo application, for example, could have a driver port ”create todo list”, and a driven port for persistence ”save todolist”. The notifications or recipients in the Figure 3 picture are also considered as a driven ports.

The ports belong to the core because the core defines what kind of actions it supports for primary the actors and what kind of actions it requires from the secondary actors. The ports can have very fine granularity where every supported action is a port or ports can contain multiple functions related to a same subdomain area. The port granularity can change over the software life cycle and the granularity of the ports should be assessed frequently during the development phase.

4.1.3 The Adapters

The adapters are related to the Adapters pattern defined in the ”Design Patterns: Elements of Reusable Object-Oriented Software” [12]. In the book the pattern is given following intent:

> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

In the ports and adapters pattern adapters are technology specific software components which convert a technology specific request to a core's port specific request. Here is a list of some possible adapters on the driver side.

- Command line interface adapter
- Graphical user interface adapter
- MVC application adapter
- Representational state transfer (REST) controller adapter
- Robotic process automation adapter
- Test automation framework adapter

On driven side adapters are technology specific implementations of port defined by the core. This way core does not know how technology specific details from e.g. data storage or how notifications are sent. Here is list of possible adapters for driven side.

- Email adapter
- Event publishing adapter
- Mock data storage adapter
- NoSQL database adapter
- Relational database adapter
- Service to service adapter

4.2 The Clean Architecture

The clean architecture was first mentioned in blog post [13] by Robert C. Martin and more in detail in the book "Clean Architecture: A Craftsman's Guide to Software Structure and Design" [14].

The Figure 4 shows the components of the architecture on high level.

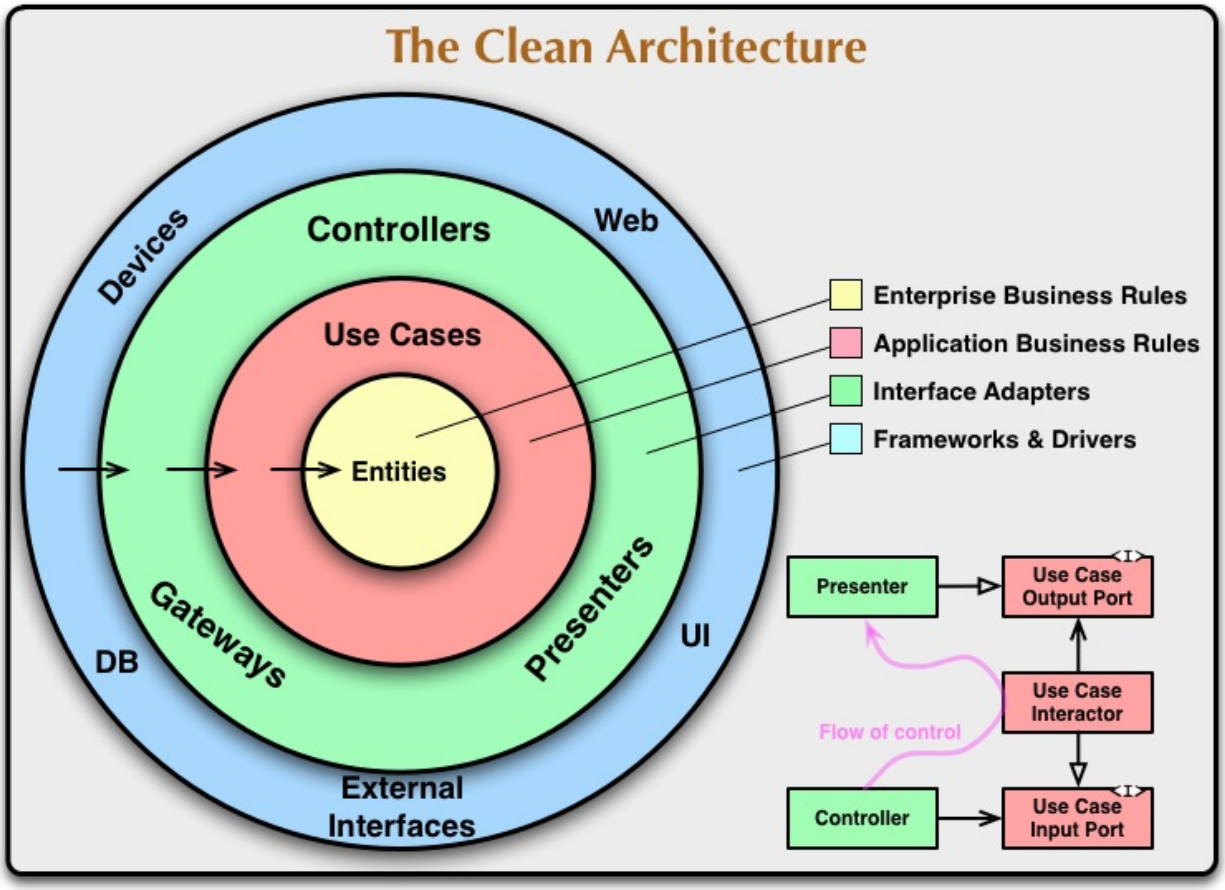

Figure 4: The Clean Architecture [14]

The architecture resembles the Port and Adapters architecture but the inner structure of the application defined in more detail. Martin gives following rule in his book for this architecture

> Source code dependencies must point only inward, toward higher-level policies.

The arrows on the left side of the picture passing through different circles pointing at the center shows this rule. An inner circle shouldn't have any knowledge of an outer circle. The higher-level policies in this context mean business or application logic and domain logic. Closer the center circle is more abstract and general the related logic is. This pattern is related to the Closed-Open Principle, which is part of the SOLID principles [15] . The Closed-Open Principle states [16]

> A module should be open for extension but closed for modification.

In the Clean Architecture the principle is used to protect components in the inner circles

from changes in components in the outer circles. Changes in a user interface should not require changes in the Entities circle.

On the lower right corner is a diagram which shows how communication between different parts flow. The diagram also shows source code dependencies. An open arrow means using and a closed arrow means implements or inheritance. The presenter uses Use Case Input Port which is implemented by the Use Case Interactor. The Use Case Interactor then uses Use Case Output Port which is implemented by the Presenter.

4.2.1 The Entities

The entities are at the center of the architecture and represent application's domain objects. The entities contain the most generic and high-level rules and should be most stable part of the application. Changes on any other circle should not require any changes on this circle. An example of a entity and related domain logic is an order object in an e-commerce application. Usually the order object has a state which tells a status of the order. The logic which is responsible for order's state transfers is a domain logic. The state transfers logic is general, high-level, and does not change often.

4.2.2 The Use Cases

The use cases contain the application's business logic. The use cases control how and which order the entities and their domain logic are called. This circle is changed when there are operational or business changes. An example of a use case in e-commerce application is a order creation. Usually order creator is responsible for creating the order based on customer's shopping basket, storing the order and calling directly or creating an event for services which are interested in newly created order.

4.2.3 The Interface Adapters

The Interface Adapters circle is responsible for converting the data from external actors to format used by the use cases and vice versa. This circle contains application's Graphical

User Interface (GUI), SQL or REST code. This circle work as a anti-corruption layer and protects the inner circles from external actors' data structures.

4.2.4 The Frameworks and Drivers

The Frameworks and Drivers circle is the most outer circle and contains the application implementation details. These details include but are not limited to: web frameworks, Inversion of Control (IoC) frameworks, dependency injection frameworks, databases, messaging frameworks.

5 The Demo Application

The demo application is a simple Todo application which is implemented in both architectures. The implementations are written with Java programming languages and both examples consists three modules; the core, the microservice, and the serverless module. The core module contains the business logic implemented with given architecture style. The microservice module contains a REST style implementation of the application. The serverless module contains AWS Lambda implementation of the application.

The demo application is a simple ToDo application which supports following user actions:

- User can create a new todo list
- User can list todo lists and related todo tasks
- User can add a new todo task to a todo list
- User can mark a todo task as completed

The actions define basic functionality where user can create and read todo lists. User can also add new tasks to a list and mark tasks as done. Both demo applications will implement these features.

The code examples are available on public repository https://bitbucket.org/vfriman/thesis_public/src/master/ .

5.1 The Ports and adapters application

The Ports and adapters application has three separate modules: the core, the microservice and the serverless module. The figure Figure 5 shows the public classes and interfaces.

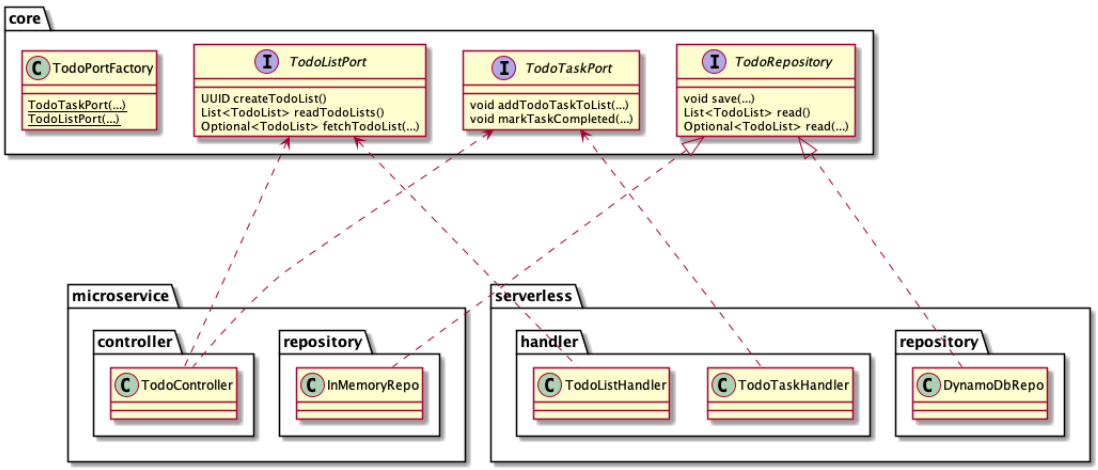

Figure 5: The Ports and adapters class diagram

The class diagram shows that the core module does not have any dependencies so it is considered as independent. The other modules either use the public interfaces of the core module or implement the interface defined by the core module.

5.1.1 The Core Module

The core module contains the application's business logic. The module defines three ports; the TodoListPort and the TodoTaskPort are driver ports and The TodoRepository is a driven port. The TodoListPort contains methods related to a todo list. The Listing 1 has listing of the supported methods. The TodoTaskPort contains methods related to a todo tasks which are listed in the Listing 2 . In this example the ports are quite granular.

```
public interface TodoListPort {

  /**
   * Creates new Todo list
   *
   * @return Returns UUID of created todo list
   */
  UUID createTodoList();

  /**
   * Returns all todo lists
   *
   * @return
   */
  List<TodoList> readTodoLists();

```

```
17    /**
18     * Returns todo list with given id.
19     *
20     * @param todoListiId
21     * @return
22     */
23    Optional<TodoList> fetchTodoList(final UUID todoListiId);
24  }
```

Listing 1: The TodoListPort interface

The TodoListPort interface defines methods for creating a todo list (line 8), reading all the todo lists (line 15), and fetching a single todo list with identifier (line 23). These methods are only methods which can be used for todo list object. The TodoService class implements this interface.

```
1   public interface TodoTaskPort {
2
3     /**
4      * Adds new todo task into list with completed status false.
5      *
6      * @param todolistId      Id of the todo list where the task is created
7      * @param taskDescription Task description
8      * @return Returns UUID of created task
9      */
10    UUID addTodoTaskToList(final UUID todolistId, final String
        taskDescription);
11
12    /**
13     * Marks task as done.
14     *
15     * @param todolistId
16     * @param taskid
17     */
18    void markTaskCompleted(final UUID todolistId, final UUID taskid);
19
20  }
```

Listing 2: The TodoTaskPort interface

The TodoTaskPort interface defines two methods: adding a task into a list (line 10), and marking a task as completed (line 18). There are no methods for reading tasks for a list because tasks are defined as part of a list. A task cannot exist without related todo list. The TodoService class implements this interface.

The TodoRepository port does not have concrete implementation in the core module be-

cause it is a driven port. A implementation of this port depends on how the application, which uses the core modules business rules, implements a data storage.

```
1  public interface TodoRepository {
2
3    /**
4     * Saves todo list
5     *
6     * @param todoList
7     */
8    void save(final TodoList todoList);
9
10   /**
11    * Returns all the lists. Returns empty list if none is found.
12    *
13    * @return
14    */
15   List<TodoList> read();
16
17   /**
18    * Returns todo list with given id.
19    *
20    * @param listId
21    * @return
22    */
23   Optional<TodoList> read(final UUID listId);
24 }
```

Listing 3: The TodoRepository interface

The TodoRepository interface defines three methods, a save method (line 8) for storing the todo list object, a read method (line 15) for reading all the todo lists from a data storage, and reading a single todo list (line 23) from data storage with identifier.

There is a only one concrete implementation of both driver ports called the TodoService which is shown in the Listing 4.

```
1  class TodoService implements TodoTaskPort, TodoListPort {
2
3    private final TodoRepository repository;
4
5    TodoService(final TodoRepository repository) {
6      this.repository = repository;
7    }
8
9    @Override
10   public UUID createTodoList() {
11     TodoList todoList =
       TodoList.builder().listId(UUID.randomUUID()).todoTasks(Collections.emptyList()).
```

```
12      this.repository.save(todoList);
13
14      return todoList.getListId();
15    }
16
17    @Override
18    public List<TodoList> readTodoLists() {
19      return this.repository.read();
20    }
21
22    @Override
23    public Optional<TodoList> fetchTodoList(final UUID todoListiId) {
24      return this.repository.read(todoListiId);
25    }
26
27    @Override
28    public UUID addTodoTaskToList(final UUID todolistId, final String
        taskDescription) {
29      TodoList todoList = this.repository.read(todolistId).get();
30
31      final TodoTask todoTask = TodoTask.builder().taskId(UUID.randomUUID())
32
        .listId(todolistId).completed(FALSE).description(taskDescription).build();
33
34      List<TodoTask> updatedTaskList = new
        ArrayList<>(todoList.getTodoTasks());
35      updatedTaskList.add(todoTask);
36
37
        this.repository.save(TodoList.builder().listId(todoList.getListId()).todoTasks(u
38
39      return todoTask.getTaskId();
40    }
41
42    @Override
43    public void markTaskCompleted(final UUID todolistId, final UUID taskid) {
44      Optional<TodoList> todoListOptional = this.repository.read(todolistId);
45      if (todoListOptional.isPresent()) {
46        TodoList todoList = todoListOptional.get();
47        Optional<TodoTask> possibleTodoTask =
        todoList.getTodoTasks().stream()
48            .filter(task -> Objects.equals(task.getTaskId(), taskid))
49            .findFirst();
50        if (possibleTodoTask.isPresent()) {
51          List<TodoTask> updatedTaskList = todoList.getTodoTasks().stream()
52            .filter(task -> !Objects.equals(task.getTaskId(), taskid))
53            .collect(Collectors.toList());
54
        updatedTaskList.add(TodoTask.builder().taskId(taskid).listId(todolistId).complet
55            .description(possibleTodoTask.get().getDescription()).build());
```

```
56
      this.repository.save(TodoList.builder().listId(todoList.getListId()).todoTasks(u
57        }
58      }
59
60    }
61  }
```

Listing 4: The Concrete implementation of ports

The TodoService class contains actual business logic code which implements the requirements defined in chapter 5. The class has one class member TodoRepository which is a technology specific implementation of a data storage. TodoService itself does not know how the data is stored and where. Generation of the todo list identifier (line 11) is considered as part of the business logic. This way the identifier is not tied into the data storage solution and the identifier can be freely shared to outside world.

Testability is one the benefits of this pattern. Creating unit tests for the business logic should be easy and the tests are not littered by platform specific details. The Listing 5 shows two test cases for the two most complicated methods in the TodoService class.

```
1   void setUp() {
2     this.repository = new StubTodoRepository();
3     this.todoService =
      TodoPortFactory.createTodoTaskService(this.repository);
4     this.listService =
      TodoPortFactory.createTodoListService(this.repository);
5   }
6
7   @Test
8   void shouldAddTodoTaskToList() {
9     var description = "Task_description";
10
11    UUID listId = this.createNewTodoList();
12    this.todoService.addTodoTaskToList(listId, description);
13
14   List<TodoTask> taskList = this.repository.read().get(0).getTodoTasks();
15    assertEquals(1, taskList.size(), "Todo list should have one task");
16    TodoTask task = taskList.get(0);
17    assertNotNull(task.getTaskId(), "Task should have identifier");
18    assertEquals(description, task.getDescription(), "Task should have
      given description");
19    assertFalse(task.getCompleted(), "Task should be created completed set
      to false");
20  }
21
22  @Test
```

```
23    void shouldMarkTaskAsCompleted() {
24
25      UUID listId = this.createNewTodoList();
26      UUID taskId = this.todoService.addTodoTaskToList(listId,
        "description");
27
28      this.todoService.markTaskCompleted(listId, taskId);
29
30      TodoList todoList = this.repository.read(listId).get();
31      assertTrue(todoList.getTodoTasks().get(0).getCompleted(), "Task should
        be marked as done");
32    }
33
34    private UUID createNewTodoList() {
35      return this.listService.createTodoList();
36    }
```

Listing 5: Two test cases for todo list methods

The setUp method creates necessary objects for the tests. Tests are using a stub implementation of the TodoRepository interface. The set up uses the factory class to create the classes under the test (line 3 and 4). The shouldAddTodoTaskToList test tests that the task can be added to a list with correct values. The shouldMarkTaskAsCompleted test tests that the created task can be marked as done. The shouldMarkTaskAsCompleted does not test that the other values are set correctly because the shouldAddTodoTaskToList is responsible for testing those things.

The TodoPortFactory (Listing 6) class in the core module is a factory class [5] and is responsible for wiring up the concrete implementation of the port interfaces.

```
1  public class TodoPortFactory {
2
3    public static TodoTaskPort createTodoTaskService(final TodoRepository
       todoRepository) {
4      Objects.requireNonNull(todoRepository, "Repository implementation
       cannot be null");
5
6      return new TodoService(todoRepository);
7    }
8
9    public static TodoListPort createTodoListService(final TodoRepository
       todoRepository) {
10     Objects.requireNonNull(todoRepository, "Repository implementation
       cannot be null");
11
12     return new TodoService(todoRepository);
13   }
```

```
}
```

Listing 6: The Todo factory class

There are two methods which both take an object, which implements the TodoRepository interface, as a parameter. Both methods return same concrete class but return values of the methods tell which port is returned. This way the TodoPortFactory hides the core's internal structure and internal dependencies from the outside users.

5.1.2 The Microservice Module

The microservice module contains a REST implementation of the application. It uses the TodoPortFactory class to create the TodoListPort and the TodoTaskPort. The InMemoryRepo class implements the TodoRepository interface.

The TodoControllerListing 7 is an adapter class which translates REST requests into a data structure supported by the core's ports. The adapter also converts core's response objects to its own response data structures.

```
@RestController
public class TodoController {
  private final TodoTaskPort todoTaskService;
  private final TodoListPort todoListService;

  public TodoController(final TodoTaskPort todoTaskService, final
    TodoListPort todoListService) {
    this.todoTaskService = todoTaskService;
    this.todoListService = todoListService;
  }

  @GetMapping("/list")
  public ResponseEntity<List<TodoListDTO>> readLists() {
    return ResponseEntity.ok(this.todoListService.readTodoLists()
      .stream().map(TodoListDTO::from).collect(toList()));
  }

  @PostMapping("/list")
  public ResponseEntity<TodoListCreatedDTO> createTodoList() throws
    URISyntaxException {
    String listId = this.todoListService.createNewTodoList().toString();
    return ResponseEntity.created(new URI("/list/"+listId)).body(new
    TodoListCreatedDTO(listId));
  }

```

```
23    @GetMapping("/list/{listId}")
24    public ResponseEntity<TodoListDTO> readList(@PathVariable final String
        listId) {
25      Optional<TodoList> list =
        this.todoListService.fetchTodoList(UUID.fromString(listId));
26      if (list.isPresent()) {
27        return ResponseEntity.ok(TodoListDTO.from(list.get()));
28      }
29      return ResponseEntity.notFound().build();
30    }
31
32    @PostMapping("/list/{listId}")
33    public ResponseEntity<Void> createTask(@PathVariable final String
        listId, @RequestBody final CreateTodoTaskDTO todoTaskDTO) throws
        URISyntaxException {
34      this.todoTaskService.addTodoTaskToList(UUID.fromString(listId),
        todoTaskDTO.content);
35      return ResponseEntity.created(new URI("/list/"+listId)).build();
36    }
37
38    @PutMapping("/list/{listId}/task/{taskId}")
39    public ResponseEntity<Void> markTaskDone(@PathVariable final String
        listId, @PathVariable final String taskId) {
40      this.todoTaskService.markTaskCompleted(UUID.fromString(listId),
        UUID.fromString(taskId));
41
42      return ResponseEntity.noContent().build();
43    }
44  }
```

Listing 7: The REST adapter

The TodoController uses Spring framework to implement REST specific details such as converting the data sent in HTTP POST request into a Data Transfer Object (DTO), and defining resource URLs with annotations. Spring framework also offers a dependency injection where the framework is responsible of creating required objects and setting up the required dependencies.

The Listing 8 lists few of the test cases for REST adapter

```
1     @Test
2     void createTodoList() throws Exception {
3
4         this.mockMvc.perform(post("/list"))
5         .andExpect(status().isCreated())
6         .andExpect(header().exists("Location"))
7         .andExpect(jsonPath("$.listId").isString());
8     }
```

```
    @Test
    void shouldReturn404WhenListNotFound() throws Exception {
        this.mockMvc.perform(get("/list/"+UUID.randomUUID()))
        .andExpect(status().isNotFound());
    }

    @Test
    void createTaskToList() throws Exception {

        UUID listId = createTodolist();

        this.mockMvc.perform(post("/list/"+listId.toString())
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectToString(new
    TodoController.CreateTodoTaskDTO("test_task"))))
        .andExpect(status().isCreated())
        .andExpect(header().exists("Location"));
    }
```

Listing 8: Test cases for REST adapter

The adapter's test cases are more focused on technology specific characteristics instead of testing the business logic. For example test case shouldReturn404WhenListNotFound checks that the adapter returns correct HTTP status code 404 when a resource is not found.

The Listing 9 lists microservice module's implementation of the TodoRepository port.

```
public class InMemoryRepo implements TodoRepository {

  private static final Map<UUID, TodoList> todoListMap = new
    ConcurrentHashMap<>();

  @Override
  public void save(final TodoList todoList) {
    this.todoListMap.put(todoList.getListId(), todoList);
  }

  @Override
  public List<TodoList> read() {
    if(this.todoListMap.isEmpty()){
      return Collections.emptyList();
    }
    return new ArrayList<>(this.todoListMap.values());
  }

  @Override
  public Optional<TodoList> read(final UUID listId) {
    return Optional.ofNullable(this.todoListMap.get(listId));
```

```
  }
}
```

Listing 9: The concrete implementation of TodoRepository

The InMemoryRepo does not use persistent storage for storing the todo lists and related tasks. Instead storage is implmented as a map structure where the data is saved on runtime. The stored data is lost when the application is shutdown or rebooted. This solution is not usable in actual production system but is good enough solution for this example.

5.1.3 The Serverless Module

The serverless module contains implementation for AWS cloud platform. There are two platform specific adapters which both implement the lambda function interface offered by the AWS sdk. The serverless application is deployed with Serverless framework **??** which take care of setting up request routes, API gateway and DynamoDB database.

The Listing 12 contains code for a lambda function which handles todo task related requests.

```
public class TodoListHandler implements
  RequestHandler<APIGatewayProxyRequestEvent,
    APIGatewayProxyResponseEvent> {
  private final Logger logger =
    LoggerFactory.getLogger(TodoListHandler.class);
  private final TodoListPort todoListService;

  public TodoListHandler() {
   this(DynamoDbRepo.getInstance());
  }

  TodoListHandler(final TodoRepository todoRepository) {
    this.todoListService =
    TodoPortFactory.createTodoListService(todoRepository);
  }

  @Override
  public APIGatewayProxyResponseEvent handleRequest(final
    APIGatewayProxyRequestEvent requestEvent, final Context context) {
    logger.info("request_event {}", requestEvent);

    if (isPost(requestEvent)) {
      UUID listId = this.todoListService.createTodoList();
```

```
20        Map<String, String> headers = new HashMap<>();
21        headers.put("Location", "/list/" + listId.toString());
22
23        return new APIGatewayProxyResponseEvent()
24          .withHeaders(headers)
25          .withStatusCode(HttpStatus.SC_CREATED);
26      } else if (isGet(requestEvent)) {
27        List<TodoListDTO> listdto = new ArrayList<>();
28        for (TodoList todoList : this.todoListService.readTodoLists()) {
29          listdto.add(TodoListDTO.from(todoList));
30        }
31        Gson gson = new GsonBuilder().setPrettyPrinting().create();
32        return new
      APIGatewayProxyResponseEvent().withStatusCode(200).withBody(gson.toJson(listdto)
33      } else {
34        return new APIGatewayProxyResponseEvent()
35          .withStatusCode(HttpStatus.SC_BAD_REQUEST)
36          .withBody(String.format("{\"error_message\": \" Method %s not
      supported\" }",
37           requestEvent.getHttpMethod()));
38      }
39    }
40  }
```

Listing 10: The Todo list lambda adapter

The TodoListHandler implements RequestHandler interface and its one method handleRequest. The method is responsible for handling all the requests routed to it and converting the request data into core's data structures. The method is also responsible for converting the core's response object into data structure used by the calling client.

The Listing 11 lists the test cases for list adapter.

```
1  class TodoListHandlerTest {
2
3    private static final Context NULL_CONTEXT = null;
4    private TodoRepository repository;
5    private TodoListHandler handler;
6
7    @BeforeEach
8    void setUp() {
9      this.repository = new StubTodoRepository();
10     this.handler = new TodoListHandler(this.repository);
11   }
12
13   @Test
14   void shouldCreateNewTodoList(){
15     var responseEvent = this.handler.handleRequest(createPostEvent(),
       NULL_CONTEXT);
```

```
16
17      assertEquals(responseEvent.getStatusCode().intValue(),
        HttpStatus.SC_CREATED);
18      assertNotNull(responseEvent.getHeaders().get("Location"), "Event
        should have location header");
19
20    }
21    @Test
22    void shouldReturnTodoLists(){
23
        this.repository.save(TodoList.builder().listId(UUID.randomUUID()).build());
24
25      var responseEvent = this.handler.handleRequest(createGetEvent(),
        NULL_CONTEXT);
26
27      assertEquals(responseEvent.getStatusCode().intValue(),
        HttpStatus.SC_OK);
28      assertFalse(responseEvent.getBody().isEmpty());
29    }
30
31    @Test
32    void unsupportedMethodsReturn400() {
33      var unsupportedMethods = Arrays.stream(HttpMethod.values())
34        .filter(method -> !(method.equals(HttpMethod.GET) ||
        method.equals(HttpMethod.POST)))
35        .collect(toList());
36
37      unsupportedMethods.stream().forEach(method ->{
38        var responseEvent = this.handler.handleRequest(createEvent(method),
        NULL_CONTEXT);
39        assertEquals(responseEvent.getStatusCode().intValue(),
        HttpStatus.SC_BAD_REQUEST,
40            "Expected method "+method.name()+" to return status 400");
41      } );
42    }
43
44  }
```

Listing 11: The Todo list adapter test

The AWS does not offer same kind of tools for testing like Spring framework did so these test cases are closer to unit tests rather than the integration tests. Tests are using stub implementation of the repository interface same like the core tests. Tests are more focused on asserting a correct RESTul functionality. For example test shouldCreateNewTodoList (line 14) checks that the response has correct HTTP status and the response's headers contain location attribute.

The Listing 12 lists code which handles the requests related to todo tasks. Like the TodoListHandler it implements the RequestHandler interface.

```
1  public class TodoTaskHandler implements
2    RequestHandler<APIGatewayProxyRequestEvent,
       APIGatewayProxyResponseEvent> {
3    private final TodoTaskPort todoTaskService;
4
5    public TodoTaskHandler() {
6      this(DynamoDbRepo.getInstance());
7
8    }
9    TodoTaskHandler(final TodoRepository repository) {
10     this.todoTaskService  =
       TodoPortFactory.createTodoTaskService(repository);
11
12   }
13
14   @Override
15   public APIGatewayProxyResponseEvent handleRequest(final
       APIGatewayProxyRequestEvent requestEvent, final Context context) {
16     String listId = requestEvent.getPathParameters().get("list_id");
17     String taskId = requestEvent.getPathParameters().get("task_id");
18
19     if (isPost(requestEvent)) {
20       if (listId != null && payloadIsValid(requestEvent.getBody())) { //
       create new
21         JsonObject payload =
       JsonParser.parseString(requestEvent.getBody()).getAsJsonObject();
22         this.todoTaskService.addTodoTaskToList(UUID.fromString(listId),
       payload.getAsJsonPrimitive("description").getAsString());
23         return createResponseWithStatus(SC_CREATED);
24       }
25       else{
26         return createResponseWithStatus(SC_BAD_REQUEST);
27       }
28     } else if (isPut(requestEvent)) {
29       if(listId != null && taskId != null) {
30         this.todoTaskService.markTaskCompleted(UUID.fromString(listId),
       UUID.fromString(taskId));
31         return new
       APIGatewayProxyResponseEvent().withStatusCode(SC_NO_CONTENT);
32       }
33       else{
34         return createResponseWithStatus(SC_BAD_REQUEST);
35       }
36     }
37     return createResponseWithStatus(SC_BAD_REQUEST)
38       .withBody(String.format("{\"error_message\": \" Method %s not
       supported\" }", requestEvent.getHttpMethod()));
```

```
39    }
40  }
```

Listing 12: The Todo task lambda adapter

The handleRequest supports two HTTP methods: POST and PUT. The POST method (line 21) is meant for creating a new task and the PUT method (line 30) is meant for updating the task status to done.

The Listing 11 lists the test cases for task adapter.

```
1  class TodoTaskHandlerTest {
2
3    private static final Context NULL_CONTEXT = null;
4    private TodoRepository repository;
5    private TodoTaskHandler handler;
6
7    @BeforeEach
8    void setUp() {
9      this.repository = new StubTodoRepository();
10     this.handler = new TodoTaskHandler(this.repository);
11   }
12
13   @Test
14   void shouldCreateNewTask() {
15     UUID listId = createEmptyList();
16     var event = createPostEvent();
17     var pathParams = new HashMap<String, String>();
18     pathParams.put("list_id", listId.toString());
19     event.setPathParameters(pathParams);
20     event.setBody("{\"description\":\"test\"}");
21
22     var response = handler.handleRequest(event, NULL_CONTEXT);
23
24     assertEquals(response.getStatusCode().intValue(),
       HttpStatus.SC_CREATED);
25   }
26
27   @Test
28   void shouldMarkTaskDone() {
29     UUID listId = createEmptyList();
30     UUID taskId = addTaskToList(listId);
31
32     var event = createPutEvent();
33     var pathParams = new HashMap<String, String>();
34     pathParams.put("list_id", listId.toString());
35     pathParams.put("task_id", taskId.toString());
36     event.setPathParameters(pathParams);
37
```

```
38      var response = handler.handleRequest(event, NULL_CONTEXT);
39
40      assertEquals(response.getStatusCode().intValue(),
        HttpStatus.SC_NO_CONTENT);
41    }
42
43    @Test
44    void unsupportedMethodsReturn400() {
45      final var unsupportedMethods = Arrays.stream(HttpMethod.values())
46        .filter(method -> !(method.equals(HttpMethod.PUT) ||
        method.equals(HttpMethod.POST)))
47        .collect(toList());
48      final var params = new HashMap<String, String>();
49
50      unsupportedMethods.stream().forEach(method ->{
51        var event = createEvent(method);
52        event.setPathParameters(params);
53        var responseEvent = this.handler.handleRequest(event, NULL_CONTEXT);
54        assertEquals(responseEvent.getStatusCode().intValue(),
        HttpStatus.SC_BAD_REQUEST,
55          "Expected method "+method.name()+" to return status 400");
56      } );
57    }
58  }
```

Listing 13: The cTodo task adapter test

Like tests for the todo task handler these test cases are also closer to a unit test than a integration test. Tests are using the stub implementation of the repository interface like the TodoListHandlerTest and core module's tests. The tests are more focused on asserting that the lambda function works RESTul way, just like the tests in TodoListHandlerTest.

The Listing 14 lists the AWS platform's implementation of the TodoRepository port. The DynamoDb is a NoSQL database offered by the AWS cloud platform.

```
1   public class DynamoDbRepo implements TodoRepository {
2
3     private Logger logger = LoggerFactory.getLogger(DynamoDbRepo.class);
4     private final static DynamoDbRepo adapter = new DynamoDbRepo();
5
6     private final AmazonDynamoDB client;
7
8     private DynamoDbRepo() {
9       client = AmazonDynamoDBClientBuilder.standard()
10        .withRegion(Regions.EU_CENTRAL_1)
11        .build();
12      logger.info("Created DynamoDB client");
13    }
```

```
14
15    DynamoDbRepo(final AmazonDynamoDB dynamoDB) {
16      this.client = dynamoDB;
17    }
18
19    public static TodoRepository getInstance() {
20      return adapter;
21    }
22
23    @Override
24    public void save(final TodoList todoList) {
25      DynamoDBMapper mapper = new DynamoDBMapper(client);
26      mapper.save(this.toTodoListDynamo(todoList));
27    }
28
29    @Override
30    public List<TodoList> read() {
31
32      DynamoDBMapper mapper = new DynamoDBMapper(client);
33      DynamoDBScanExpression scanExpression = new DynamoDBScanExpression();
34      scanExpression.setLimit(100);
35      List<TodoListDynamo> todoListDynamo =
        mapper.scan(TodoListDynamo.class, scanExpression);
36      return
        todoListDynamo.stream().map(this::fromDynamoList).collect(Collectors.toList());
37    }
38
39    @Override
40    public Optional<TodoList> read(final UUID listId) {
41      DynamoDBMapper mapper = new DynamoDBMapper(client);
42      Map<String, AttributeValue> vals = new HashMap<>();
43      vals.put(":todolist_id", new
        AttributeValue().withS(listId.toString()));
44
45      DynamoDBQueryExpression<TodoListDynamo> queryExpression = new
        DynamoDBQueryExpression<TodoListDynamo>()
46        .withKeyConditionExpression("todolist_id = :todolist_id ")
47        .withExpressionAttributeValues(vals);
48
49      List<TodoListDynamo> todoListDynamo =
        mapper.query(TodoListDynamo.class, queryExpression);
50      if (todoListDynamo.isEmpty()) {
51        return Optional.empty();
52      }
53      return Optional.of(this.fromDynamoList(todoListDynamo.get(0)));
54    }
55
56    private TodoList fromDynamoList(final TodoListDynamo todoListDynamo) {
57      final UUID listId = UUID.fromString(todoListDynamo.todolist_id);
58      List<TodoTask> todoTaskList = todoListDynamo.tasks.stream()
```

```
59        .map(task -> fromDynamoTask(listId, task))
60        .collect(Collectors.toList());
61      return
        TodoList.builder().listId(listId).todoTasks(todoTaskList).build();
62    }
63
64    private TodoTask fromDynamoTask(final UUID listId, final TodoTaskDynamo
        todoTaskDynamo) {
65      return
        TodoTask.builder().taskId(UUID.fromString(todoTaskDynamo.task_id))
66
        .listId(listId).completed(todoTaskDynamo.completed).description(todoTaskDynamo.d
67        .build();
68    }
69
70    private TodoListDynamo toTodoListDynamo(final TodoList todoList) {
71
72      List<TodoTaskDynamo> tasks = todoList.getTodoTasks().stream()
73        .map(this::toTodoTaskDynamo).collect(Collectors.toList());
74      TodoListDynamo todoListDynamo = new TodoListDynamo();
75      todoListDynamo.todolist_id = todoList.getListId().toString();
76      todoListDynamo.tasks = tasks;
77
78      return todoListDynamo;
79    }
80
81    private TodoTaskDynamo toTodoTaskDynamo(final TodoTask todoTask) {
82      TodoTaskDynamo todoTaskDynamo = new TodoTaskDynamo();
83      todoTaskDynamo.task_id = todoTask.getTaskId().toString();
84      todoTaskDynamo.completed = todoTask.getCompleted();
85      todoTaskDynamo.description = todoTask.getDescription();
86      return todoTaskDynamo;
87    }
88  }
```

Listing 14: The DynamoDB implementation of TodoRepository interface

The DynamoDB repository is more complicated than the in-memory repository used in the microservice module but this shows how driven ports can easily change between different implementations.

For testing DynamoDB related code AWS offers a local instance of the DynamoDB. The Listing 15 lists tests for the DynamoDB repository.

```
1  @ExtendWith(LocalDynamoDbRule.class)
2  class DynamoDbRepoTest {
3
4    private AmazonDynamoDB amazonDynamoDB;
5    private DynamoDbRepo repository;
```

```
6
7   DynamoDbRepoTest(final AmazonDynamoDB amazonDynamoDB) {
8     this.amazonDynamoDB = amazonDynamoDB;
9   }
10
11  @BeforeEach
12  public void setUp() {
13    this.repository = new DynamoDbRepo(amazonDynamoDB);
14    var dynamoDBMapper = new DynamoDBMapper(amazonDynamoDB);
15    try {
16      CreateTableRequest tableRequest =
    dynamoDBMapper.generateCreateTableRequest(TodoListDynamo.class);
17      tableRequest.setProvisionedThroughput(new ProvisionedThroughput(1L,
    1L));
18      amazonDynamoDB.createTable(tableRequest);
19    } catch (ResourceInUseException e) {
20      // Do nothing, table already created
21    }
22  }
23
24  @AfterEach
25  public void tearDown(){
26    DynamoDBMapper mapper = new DynamoDBMapper(amazonDynamoDB);
27    DynamoDBScanExpression scanExpression = new DynamoDBScanExpression();
28    mapper.batchDelete( mapper.scan(TodoListDynamo.class,
    scanExpression));
29  }
30
31  @Test
32  public void shouldSaveAndRetrieveTodoList() {
33    var task = TodoTask.builder()
34      .taskId(UUID.randomUUID()).description("test
    task").completed(false).build();
35    var todoList =
    TodoList.builder().listId(UUID.randomUUID()).todoTasks(List.of(task)).build();
36    this.repository.save(todoList);
37
38    var savedLists = this.repository.read();
39
40    assertEquals(1, savedLists.size(), "There should be only one list");
41
42    var savedList = savedLists.get(0);
43    assertEquals(1, savedList.getTodoTasks().size(), "Saved list should
    have one task");
44
45    var savedTask = savedList.getTodoTasks().get(0);
46    assertEquals(task.getTaskId(), savedTask.getTaskId(), "Task id should
    match");
47    assertEquals(task.getDescription(), savedTask.getDescription(), "Task
    description should match");
```

```
    assertFalse(task.getCompleted(), "Task should be marked as non completed");
  }

  @Test
  public void shouldSaveDoneTaskCorrectly() {
    var task = TodoTask.builder()
      .taskId(UUID.randomUUID()).description("test task").completed(false).build();
    var todoList = TodoList.builder().listId(UUID.randomUUID()).todoTasks(List.of(task)).build();
    this.repository.save(todoList);

    var savedList = this.repository.read(todoList.getListId()).get();
    var completedTask = TodoTask.builder(savedList.getTodoTasks().get(0)).completed(true).build();
    var updatedList = TodoList.builder().listId(savedList.getListId())
      .todoTasks(List.of(completedTask)).build();
    this.repository.save(updatedList);

    var completedList = this.repository.read(todoList.getListId()).get();
    assertTrue(completedList.getTodoTasks().get(0).getCompleted(), "Task should be marked as done");

  }

}
```

Listing 15: The tests for DynamoDB repo

The tests are written with JUNIT5 and the ExtendWith annotation at the begin of the file tells the test executor that the there is a LocalDynamoDbRule class which should be executed first. The LocalDynamoDbRule class is responsible for starting the local instance of the DynamoDB and set up the connection for the test class. The setUP and tearDown methods are run after every test and are responsible for creating the database schema at the begin of the test and clearing after the test has been executed.

The test suite contains two test cases. The shouldSaveAndRetrieveTodoList test case tests that the repository can save given todo list correctly and all the todo list in the database can be retrieved. The shouldSaveDoneTaskCorrectly test case tests that the updated todo list can be saved correctly and the a single todo list can be read from the database with identifier.

5.2 Clean architecture application

Like the Ports and Adapters application the Clean architecture also has three different modules: the core, the Microservice, and the serverless module. The figure Figure 6 shows class dependencies for a single use case because otherwise figure is quite hard to read.

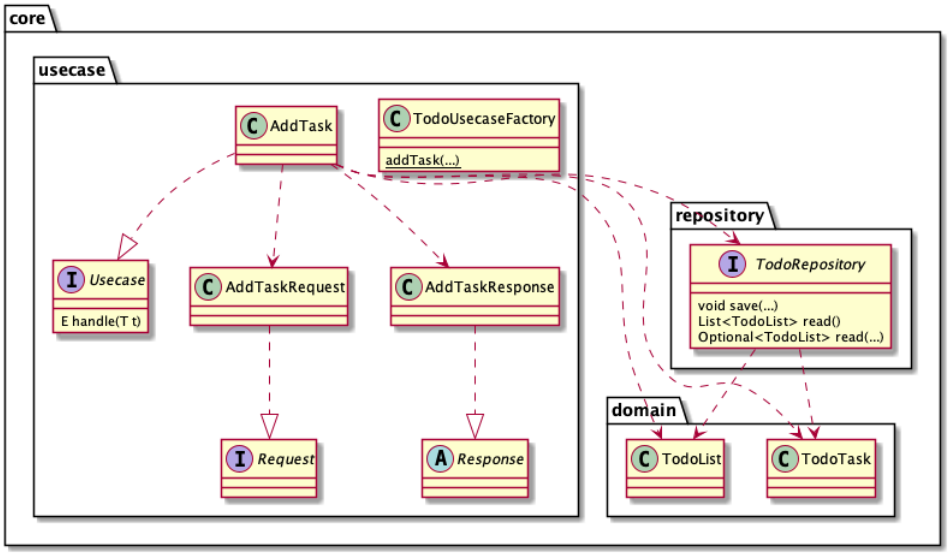

Figure 6: The Clean Architecture class diagram

The core module contains three packages: the usecase package, the repository package and the domain package. The domain package contains the TodoList and the TodoTask classes which are the domain objects. The domain package is the entities circle in the Figure 4 and it does not have any dependencies. The usecase package contains public interfaces UseCase and Request. The package also contains an abstract class Response. The usecase package is the use cases circle in the Figure 4 and it uses classes and interface from domain and repository packages. The usecase package also contains factory class which is responsible for building the concrete use classes. The repository package contains the TodoRepository interface. The repository interface is defined by the use cases and implemented by the interface adapters circle.

The figure Figure 7 shows how different modules depend on the core module.

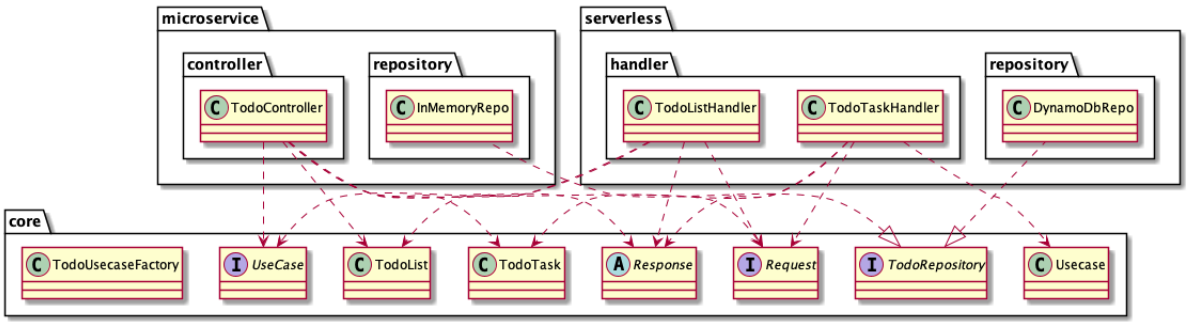

Figure 7: The Clean Architecture class diagram

The figure does not show classes which implement the Request or UseCase interface or classes which extend the Response class because the figure would become quite hard to read. But on high level the other modules depend only on public interface or classes and the core module does not depend other modules.

5.2.1 The Core module

The core module contains the two most inner circles from Figure 4. The domain package contains domain or entity objects and the usecase package supported use cases.

The Usecase interface is listed in Listing 16.

```
/**
 * Interface implemented by all use cases
 *
 * @param <T> Type of request object
 * @param <E> Type of response object
 */
public interface Usecase<T extends Request, E extends Response> {

  E handle(T t);
}
```

Listing 16: Use Case interface

The interface uses Java Generics to define a single method. The methods input parameter implements the Request interface Listing 17 and the return parameter extends the Response abstract class Listing 18. All the use cases implements this interface with their own request and response classes.

The Listing 17 lists the Request interface.

```
/**
 * Marker inteface for use case request object
 *
 */
public interface Request {}
```

Listing 17: Request interface

The Request interface is a marker interface. It has no methods or fields and is only used to identify the request objects for the use case.

The Listing 18 lists the Response abstract class.

```
public abstract class Response<T> {

  private final Throwable throwable;

  Response(final Throwable throwable) {
    this.throwable = throwable;
  }

  public abstract Optional<T> result();

  public final Optional<? extends Throwable> failure() {
    return Optional.ofNullable(throwable);
  }
}
```

Listing 18: Response abstract class

All the response objects extend this class. It has one abstract method result which is implemented by the extending class. The return value of the method is an Optional class which wraps the actual return value. This allows caller of the use case to use more fluent coding style without null checks. The other method failure is used to return an exception to the caller of the use case so it can handle it anyway it wants.

The listing Listing 19 lists the AddTask use case. This use case is responsible for handling new todo task.

```
final class AddTask implements Usecase<AddTaskRequest, AddTaskResponse>{


  private final TodoRepository repository;
  private final Usecase<FindTodoListRequest, FindTodoListResponse>
    findTodoList;

  AddTask(final TodoRepository repository, final
    Usecase<FindTodoListRequest, FindTodoListResponse> findTodoList) {
```

```
    this.repository = repository;
    this.findTodoList = findTodoList;
  }

  @Override
  public AddTaskResponse handle(final AddTaskRequest request) {
    var response = this.findTodoList.handle(new
    FindTodoListRequest(request.getListId()));
    if (response.result().isEmpty()) {
      return new AddTaskResponse(null);
    }
    TodoList todoList = response.result().get();
    var todoTask = TodoTask.builder()
      .listId(request.getListId())
      .taskId(UUID.randomUUID())
      .completed(FALSE)
      .description(request.getDescription())
      .build();

    var updatedTasks = new ArrayList<>(todoList.getTodoTasks());
    updatedTasks.add(todoTask);
    var updatedTodoList = TodoList.builder()
      .listId(todoList.getListId()).todoTasks(updatedTasks).build();

    this.repository.save(updatedTodoList);

    return new AddTaskResponse(todoTask.getTaskId());
  }
}
```

Listing 19: AddTask use case

This use case uses FindTodoList use case (Listing 25) to find the correct todo list where to add the new task. This shows how use cases can call other use cases and use their business logic.

The handle method which contains the business logic to add a new task into to list is quite straightforward. It uses the FindTodoList use case to find the correct todo list. If the list is found a new task is created and added to list. The updated list then stored and saved. A response is returned with newly created task id.

The Listing 20 lists the AddTask request class.

```
public final class AddTaskRequest implements Request{
  private final UUID listId;
  private final String description;

  public AddTaskRequest(final UUID listId, final String description) {
```

```
    this.listId = Objects.requireNonNull(listId, "list id cannot be null");
    this.description = Objects.requireNonNull(description, "Description
    cannot be null");
  }

  UUID getListId() {
    return listId;
  }

  String getDescription() {
    return description;
  }

}
```

Listing 20: AddTasRequest class

The class implements the Request interface (Listing 17). Its constructor takes to parameters, a todo list identifier and the task's description. The constructor also validates the inputs. The getter methods are only package visible and only acceptable to use cases in same package. This limits the access to data.

The **??** lists the AddTask response class.

```
public final class AddTaskResponse  extends Response<UUID>{

  private final Optional<UUID> taskId;

  AddTaskResponse(final UUID taskId) {
    super(null);
    this.taskId = Optional.ofNullable(taskId);
  }

  @Override
  public Optional<UUID> result() {
    return this.taskId;
  }
}
```

Listing 21: AddTaskResponse class

The class extends the Response class (Listing 18). The class has package visible constructor so it can be only constructed inside the use case package without tricks. The class implements the generic result method from super class and the the method's return value is Optional with UUID object.

The Listing 22 lists the CreateTodoList use case. This use case is responsible for creating

a new todo list.

```
final class CreateTodoList implements Usecase<VoidRequest,
    CreateTodoListResponse>{

  private final TodoRepository repository;

  CreateTodoList(final TodoRepository repository) {
    this.repository = repository;
  }

  @Override
  public CreateTodoListResponse handle(final VoidRequest voidRequest) {
    var todoList = TodoList.builder().listId(UUID.randomUUID()).build();
    this.repository.save(todoList);

    return new CreateTodoListResponse(todoList.getListId());
  }
}
```

Listing 22: CreateTodoList use case

The CreateTodoList use case is straightforward. It creates a new todo list instance and saves it. The id of the newly created list is generated in this class so it won't depend on used persistence solution.

The Listing 23 lists the VoidRequest class.

```
public final class VoidRequest implements Request{

  private static final VoidRequest INSTANCE = new VoidRequest();
  private VoidRequest(){};

  public static VoidRequest getInstance(){return INSTANCE;}
}
```

Listing 23: VoidRequest class

The VoidRequest class is a placeholder class to fulfill the Usecase interface's contract. It is used when the use case's does not require any input data to execute its business logic.

The Listing 24 lists the CreateTodoListResponse class.

```
public final class CreateTodoListResponse extends Response<UUID> {

  private final UUID listId;

  CreateTodoListResponse(final UUID listId) {
    super(null);
    this.listId = listId;
  }
```

```java

  public Optional<UUID> result() {
    return Optional.ofNullable(listId);
  }
}
```

Listing 24: CreateTodoList response

The response class contains the identifier of the newly created todo list. The class has package visible constructor because it should be instantiated only from usecase package.

The listing Listing 25 lists the FindTodoList class.

```java
final class FindTodoList implements Usecase<FindTodoListRequest,
    FindTodoListResponse>{

  private final TodoRepository repository;

  FindTodoList(final TodoRepository repository) {
    this.repository = repository;
  }

  @Override
  public FindTodoListResponse  handle(final FindTodoListRequest request) {
    return new
    FindTodoListResponse(this.repository.read(request.getListId()).orElse(null));
  }
}
```

Listing 25: FindTodoList use case

This use case is used to find a single todo list. The handle method uses the repository to find the todo list with given identifier. If there is no todo list with the given identifier a null value is set to response object.

The Listing 26 lists the FindTodoListRequest class.

```java
public final class FindTodoListRequest implements Request {

  private final UUID listId;

  public FindTodoListRequest(final UUID listId) {
    this.listId = Objects.requireNonNull(listId, "List id cannot be null");
  }

  UUID getListId() {
    return listId;
  }

}
```

Listing 26: FindTodoListRequest class

The FindTodoListRequest class holds the identifier of the wanted todo list. In the constructor (line 6) the identifier is checked for a null value. This way the use case does not have to check if the parameter is null or not.

The Listing 27 lists the FindTodoListResponse class.

```
1  public class FindTodoListResponse extends Response<TodoList>{
2
3    private final TodoList todoList;
4
5    FindTodoListResponse(final TodoList todoList) {
6      super(null);
7      this.todoList = todoList;
8    }
9
10   @Override
11   public Optional<TodoList> result() {
12     return Optional.ofNullable(todoList);
13   }
14
15 }
```

Listing 27: FindTodoListResponse class

The FindTodoListResponse class contains the found todo list object. The result method (line 11) returns either Optional object with todo list or an empty Optional object if no todo list was found with the identifier.

The Listing 28 lists the FindTodoLists class.

```
1  final class FindTodoLists implements Usecase<VoidRequest,
       FindTodoListsResponse>{
2
3    private final TodoRepository repository;
4
5    FindTodoLists(final TodoRepository repository) {
6      this.repository = repository;
7    }
8
9    @Override
10   public FindTodoListsResponse handle(final VoidRequest voidRequest) {
11     return new FindTodoListsResponse(this.repository.read());
12   }
13 }
```

Listing 28: FindTodoLists use case

The FindTodoLists use case is used to find all the todo lists. Like the CreateTodoList use case (Listing 22) the FindTodoList use case does not require any input parameters from the use case caller so it's request object is the VoidRequest (Listing 23).

The Listing 29 lists the FindTodoListResponse class.

```
public class FindTodoListsResponse extends Response<List<TodoList>>{

  private final List<TodoList> list;

  FindTodoListsResponse(final List<TodoList> list) {
    super(null);
    this.list = list == null ? Collections.emptyList() : List.copyOf(list);
  }

  @Override
  public Optional<List<TodoList>> result() {
    return Optional.of(list);
  }

}
```

Listing 29: FindTodoListsResponse class

The FindTodoListsResponse class contains the list of all the todo lists. The constructor validates the input parameter against a null value. If the input value is null an empty list is set to return value. The semantics of the null value or undefined and an empty list are the same so we can return an empty list. If the input list is not null a copy is created from the input list for the response object.

The Listing 30 lists the MarkTaskDone class.

```
final class MarkTaskDone implements Usecase<MarkTaskDoneRequest,
    MarkTaskDoneResponse>{
  private final TodoRepository todoRepository;

  MarkTaskDone(final TodoRepository todoRepository) {
    this.todoRepository = todoRepository;
  }

  @Override
  public MarkTaskDoneResponse handle(final MarkTaskDoneRequest request) {
    var optionalList = this.todoRepository.read(request.getListId());

    if(optionalList.isEmpty()) {
      return new MarkTaskDoneResponse(false);
    }
    var todoList = optionalList.get();
```

```
    var markedToBeDoneTask = todoList.getTodoTasks().stream()
      .filter(task ->
    task.getTaskId().equals(request.getTaskId())).findFirst().get();
    var updatedTask =
    TodoTask.builder(markedToBeDoneTask).completed(TRUE).build();
    var updatedTasks = todoList.getTodoTasks().stream()
      .filter(task -> !task.getTaskId().equals(request.getTaskId()))
      .collect(toList());
    updatedTasks.add(updatedTask);

    var updatedList =
    TodoList.builder().listId(todoList.getListId()).todoTasks(updatedTasks).build();

    this.todoRepository.save(updatedList);
    return new MarkTaskDoneResponse(true);
  }
}
```

Listing 30: MarkTaskDone use case

The MarkTaskDone use case is responsible for marking the given task as done. Its business logic is somewhat complex due the immutable nature of the domain objects. The immutable means basically that the object's state does not change after its creation. Instead a new object is created with updated state.

The Listing 31 lists the MarkTaskDoneRequest class.

```
public final class MarkTaskDoneRequest implements Request{

  private final UUID listId;
  private final UUID taskId;

  private MarkTaskDoneRequest(final UUID listId, final UUID taskId) {
    this.listId = Objects.requireNonNull(listId, "List id cannot be null");
    this.taskId = Objects.requireNonNull(taskId, "Task id cannot be null");
  }

  UUID getListId() {
    return listId;
  }

  UUID getTaskId() {
    return taskId;
  }

  public static MarkTaskDoneRequestBuilder builder() {
    return new MarkTaskDoneRequestBuilder();
  }

```

```
  public static class MarkTaskDoneRequestBuilder{
    private UUID listId;
    private UUID taskId;

    private MarkTaskDoneRequestBuilder() {}

    public MarkTaskDoneRequestBuilder listId(final UUID listId) {
      this.listId = listId;
      return this;

    }
    public MarkTaskDoneRequestBuilder taskId(final UUID taskId) {
      this.taskId = taskId;
      return this;
    }

    public MarkTaskDoneRequest build() {
      return  new MarkTaskDoneRequest(listId, taskId);
    }
  }
}
```

Listing 31: MarkTaskDoneRequest class

The MarkTaskDoneRequest is a request object with required data to marking a task as done. The class uses the builder pattern [12] to construct an MarkTaskDoneRequest object. The build object is not a complex object anyway but pattern offers more fluent and human readable API to build an object.

The Listing 32 lists the MarkTaskDoneRequest class.

```
public final class MarkTaskDoneResponse extends Response<Boolean>{

  private final boolean success;

  MarkTaskDoneResponse(final boolean success) {
    super(null);
    this.success = success;
  }

  @Override
  public Optional<Boolean> result() {
    return Optional.ofNullable(this.success);
  }
}
```

Listing 32: MarkTaskDoneResponse class

The MarkTaskDoneResponse class simple DTO class. The object holds a boolean value

which tells the use case caller if the task was marked done successfully or not.

The Listing 33 lists the TodoUsecaseFactory class.

```
/**
 * A factory class which knows how to assemble use cases
 */
public class TodoUsecaseFactory {

  public static Usecase<AddTaskRequest, AddTaskResponse> addTask(final
    TodoRepository todoRepository) {
    checkForNull(todoRepository);
    return new AddTask(todoRepository, findTodoList(todoRepository));
  }

  public static Usecase<VoidRequest, CreateTodoListResponse>
    createTodoList(final TodoRepository todoRepository) {
    checkForNull(todoRepository);
    return new CreateTodoList(todoRepository);
  }

  public static Usecase<FindTodoListRequest, FindTodoListResponse>
    findTodoList(final TodoRepository todoRepository) {
    checkForNull(todoRepository);
    return new FindTodoList(todoRepository);
  }

  public static Usecase<VoidRequest, FindTodoListsResponse>
    findTodoLists(final TodoRepository todoRepository) {
    checkForNull(todoRepository);
    return new FindTodoLists(todoRepository);
  }

  public static Usecase<MarkTaskDoneRequest, MarkTaskDoneResponse>
    markTaskDone(final TodoRepository todoRepository) {
    checkForNull(todoRepository);
    return new MarkTaskDone(todoRepository);
  }

  private static void checkForNull(final TodoRepository todoRepository) {
    Objects.requireNonNull(todoRepository, "Repository implementation
    cannot be null");
  }

}
```

Listing 33: TodoUsecaseFactory class

The TodoUsecaseFactory class is a factory class responsible of constructing the use case objects. Like its counter part in the Ports and Adapters example (Listing 6) all its methods

take an object implementing the TodoRepository interface as a input parameter. The methods then return a concrete implementation of the request use case.

5.2.2 The Microservice module

The Microservice module contains a REST style implementation of the todo application. The implementation consists a single REST-controller which is responsible for handling both todo list and todo task related HTTP-requests. The Microservice service module uses Spring framework to handle REST related code.

The tests for the controller and InMemoryRepo implementation are the same as in the Ports and Adapters implementation (subsection 5.1.2) so they are not listed in here again.

The Listing 34 lists TodoController class.

```
1  @RestController
2  public class TodoController {
3
4    private final Usecase<AddTaskRequest, AddTaskResponse> addTask;
5    private final Usecase<VoidRequest, CreateTodoListResponse>
       createTodoList;
6    private final Usecase<FindTodoListRequest, FindTodoListResponse>
       findTodoList;
7    private final Usecase<VoidRequest, FindTodoListsResponse>findTodoLists;
8    private final Usecase<MarkTaskDoneRequest,MarkTaskDoneResponse>
       markTaskDone;
9
10   TodoController(final Usecase<AddTaskRequest, AddTaskResponse> addTask,
11                  final Usecase<VoidRequest, CreateTodoListResponse>
       createTodoList,
12                  final Usecase<FindTodoListRequest, FindTodoListResponse>
       findTodoList,
13                  final Usecase<VoidRequest, FindTodoListsResponse>
       findTodoLists,
14                  final Usecase<MarkTaskDoneRequest, MarkTaskDoneResponse>
       markTaskDone) {
15     this.addTask = addTask;
16     this.createTodoList = createTodoList;
17     this.findTodoList = findTodoList;
18     this.findTodoLists = findTodoLists;
19     this.markTaskDone = markTaskDone;
20   }
21
22   @GetMapping("/list")
23   public ResponseEntity<List<TodoListDTO>> readLists() {
```

```
24        return
          ResponseEntity.ok(this.findTodoLists.handle(VoidRequest.getInstance()).result().
25          .stream().map(TodoListDTO::from).collect(toList()));
26      }
27
28      @PostMapping("/list")
29      public ResponseEntity<TodoListCreatedDTO> createTodoList() throws
          URISyntaxException {
30        String listId =
          this.createTodoList.handle(VoidRequest.getInstance()).result().get().toString();
31        return ResponseEntity.created(new URI("/list/" + listId)).body(new
          TodoListCreatedDTO(listId));
32      }
33
34      @GetMapping("/list/{listId}")
35      public ResponseEntity<TodoListDTO> readList(@PathVariable final String
          listId) {
36
37        var response = this.findTodoList.handle(new
          FindTodoListRequest(UUID.fromString(listId)));
38
39        if (response.result().isPresent()) {
40          return ResponseEntity.ok(TodoListDTO.from(response.result().get()));
41        }
42        return ResponseEntity.notFound().build();
43      }
44
45      @PostMapping("/list/{listId}")
46      public ResponseEntity<Void> createTask(@PathVariable final String listId,
47                                             @RequestBody final CreateTodoTaskDTO
          todoTaskDTO) throws URISyntaxException {
48
49        var addRequest = new AddTaskRequest(UUID.fromString(listId),
          todoTaskDTO.content);
50
51        var response = this.addTask.handle(addRequest);
52
53        if(response.result().isPresent()) {
54          return ResponseEntity.created(new URI("/list/" + listId)).build();
55        }
56        return ResponseEntity.unprocessableEntity().build();
57      }
58
59      @PutMapping("/list/{listId}/task/{taskId}")
60      public ResponseEntity<Void> markTaskDone(@PathVariable final String
          listId, @PathVariable final String taskId) {
61        var markAsDoneRequest = MarkTaskDoneRequest
62          .builder().listId(UUID.fromString(listId))
63          .taskId(UUID.fromString(taskId))
64          .build();
```

```
65
66      var response = this.markTaskDone.handle(markAsDoneRequest);
67
68      if(response.isSuccess()) {
69        return ResponseEntity.noContent().build();
70      }
71      return ResponseEntity.notFound().build();
72    }
73    ...
74  }
```

Listing 34: TodoController

The TodoController has the supported use cases as class variables (lines four through eight). As the code shows the use cases are only separated through their request and response objects. The controller does not know anything about the concrete implementations of the use cases.

The actual REST-methods are marked with different annotations with Mapping suffix. Basically all the methods have same flow:

1. Create an use case request object from the method's input parameter if needed.
2. Call the use case with the newly created request object.
3. Check the response object if necessary and return the REST-style response back to the caller.

On the lines from 61 to 64 is shown how the builder pattern makes building an instance more readable. Especially in this case where both input parameters have same type. The builder pattern makes it harder to caller to put the input parameters in wrong places.

5.2.3 The Serverless module

The serverless module contains implementation for the AWS cloud platform using Lambda functions and the Serverless framework. The module contains two handler classes, one for todo list related requests and for todo task related requests.

The tests for the handlers and DynamoDbRepo implementation and its tests are the same as in the Ports and Adapters implementation (subsection 5.1.3) so they are not listed in here again.

The Listing 35 lists the todo list related handler.

```
public class TodoListHandler implements
  RequestHandler<APIGatewayProxyRequestEvent,
    APIGatewayProxyResponseEvent> {

  private final Logger logger =
    LoggerFactory.getLogger(TodoListHandler.class);

  private final Usecase<VoidRequest, CreateTodoListResponse>
    createTodoList;
  private final Usecase<VoidRequest, FindTodoListsResponse> findTodoLists;

  public TodoListHandler() {
   this(DynamoDbRepo.getInstance());
  }

  TodoListHandler(final TodoRepository todoRepository) {
    this.createTodoList =
    TodoUsecaseFactory.createTodoList(todoRepository);
    this.findTodoLists = TodoUsecaseFactory.findTodoLists(todoRepository);
  }

  @Override
  public APIGatewayProxyResponseEvent handleRequest(final
    APIGatewayProxyRequestEvent requestEvent, final Context context) {
    logger.info("request_event {}", requestEvent);

    if (isPost(requestEvent)) {
      UUID listId =
    this.createTodoList.handle(VoidRequest.getInstance()).result().get();
      Map<String, String> headers = new HashMap<>();
      headers.put("Location", "/list/" + listId.toString());

      return new APIGatewayProxyResponseEvent()
        .withHeaders(headers)
        .withStatusCode(HttpStatus.SC_CREATED);
    } else if (isGet(requestEvent)) {
      var listdto =
    this.findTodoLists.handle(VoidRequest.getInstance()).result().get()
        .stream().map(TodoListDTO::from).collect(toList());

      Gson gson = new GsonBuilder().setPrettyPrinting().create();
      return new
    APIGatewayProxyResponseEvent().withStatusCode(200).withBody(gson.toJson(listdto)
    } else {
      return new APIGatewayProxyResponseEvent()
        .withStatusCode(HttpStatus.SC_BAD_REQUEST)
        .withBody(String.format("{\"error_message\": \" Method %s not
    supported\" }", requestEvent.getHttpMethod()));
    }
```

```
41    }
42
43    private boolean isPost(final APIGatewayProxyRequestEvent requestEvent) {
44      return POST.name().equals(requestEvent.getHttpMethod());
45    }
46
47    private boolean isGet(final APIGatewayProxyRequestEvent requestEvent) {
48      return GET.name().equals(requestEvent.getHttpMethod());
49    }
50  }
```

Listing 35: TodoListHandler class

The TodoListHandler class implements the RequestHandler interface offered by the AWS Lambda framework. The class has the needed use cases as class variables (lines six and seven) and it has two constructors. The public constructor is used by the AWS cloud platform and the constructor with default visibility by the tests.

The RequestHandler interface has a single method which implementation starts at the line 19. The method is quite simple. It checks if the request event was a HTTP POST (line 22) it then calls the creteTodoList use case and then creates appropriate response event for the caller.

If the request event wasn't a HTTP POST event the code checks on line 30 if the event was a HTTP GET. If the request was a GET event the uses findTodoLists use case to to read all the todo lists and converts them into DTO objects. The list of DTO objects are then serialized into Javascript Object Notation (JSON) format and returned to the caller. From lines 36 to 39 the method handles the request events which were not supported by this handler by returning an response event with error code and error message.

The Listing 36 lists the todo task related handler.

```
1  public class TodoTaskHandler implements
2    RequestHandler<APIGatewayProxyRequestEvent,
       APIGatewayProxyResponseEvent> {
3
4    private final Usecase<AddTaskRequest, AddTaskResponse> addTask;
5    private final Usecase<MarkTaskDoneRequest, MarkTaskDoneResponse>
       markTaskDone;
6
7    public TodoTaskHandler() {
8      this(DynamoDbRepo.getInstance());
9
10   }
```

```
11    TodoTaskHandler(final TodoRepository repository) {
12      this.addTask = TodoUsecaseFactory.addTask(repository);
13      this.markTaskDone = TodoUsecaseFactory.markTaskDone(repository);
14    }
15
16    @Override
17    public APIGatewayProxyResponseEvent handleRequest(final
        APIGatewayProxyRequestEvent requestEvent, final Context context) {
18      String listId = requestEvent.getPathParameters().get("list_id");
19      String taskId = requestEvent.getPathParameters().get("task_id");
20
21      if(listId == null){
22        return createResponseWithStatus(SC_BAD_REQUEST)
23          .withBody(String.format("{\"error_message\": \"list id cannot be
        null\" }"));
24      }
25
26      if (isPost(requestEvent)) {
27        if (payloadIsValid(requestEvent.getBody())) { // create new
28          JsonObject payload =
        JsonParser.parseString(requestEvent.getBody()).getAsJsonObject();
29          String description =
        payload.getAsJsonPrimitive("description").getAsString();
30          var request = new
        AddTaskRequest(UUID.fromString(listId),description);
31          var response = this.addTask.handle(request);
32          if(response.result().isPresent()) {
33            return createResponseWithStatus(SC_CREATED);
34          }
35          return createResponseWithStatus(SC_INTERNAL_SERVER_ERROR);
36        }
37        else{
38          return createResponseWithStatus(SC_BAD_REQUEST);
39        }
40      } else if (isPut(requestEvent)) {
41        if(taskId != null) {
42          var request = MarkTaskDoneRequest.builder()
43            .listId(UUID.fromString(listId))
44            .taskId(UUID.fromString(taskId))
45            .build();
46          var response = this.markTaskDone.handle(request);
47          if(response.isSuccess()) {
48            return createResponseWithStatus(SC_NO_CONTENT);
49          }
50          return createResponseWithStatus(SC_INTERNAL_SERVER_ERROR);
51        }
52        else{
53          return createResponseWithStatus(SC_BAD_REQUEST);
54        }
55
```

```
56      }
57      return createResponseWithStatus(SC_BAD_REQUEST)
58        .withBody(String.format("{\"error_message\": \" Method %s not
        supported\" }", requestEvent.getHttpMethod()));
59    }
60
61  ..
62  }
```

Listing 36: TodoTaskHandler

Like the TodoListHandler this handler also implements the same RequestHandler interface. The handler has related use cases as class variables which are instantiated in the package visible constructor (line 11). The handleRequest method's implementation supports HTTP POST event (line 21) and HTTP PUT event (line 35). On line 21 the method checks that the parameter required by supported events is not a null value. If the value is null the method fails fast and returns an error event.

Both supported methods have same structured: check that the required parameters exists and call correct use case. The correct use case is in a POST event the addTask use case (line 31) and in a PUT event correct use case is the markTaskDone use case (line 46). If the parameter checks fail an error event is returned and if the event is unsupported an error event with a message is returned.

6 Conclusions

This section covers what the two PoC projects achieved. In thesis' Introduction1 says that this thesis represents two architectures which should be able to withstand fast pace of modern application development and the ever changing business requirements. The two PoC implementations show that it is possible to separate the business logic implementation from the details of the underlying platforms. Both of the implementations were build to support HTTP-protocol when handling requests from the clients but it should not be any different to create an implementation which support command line interface for inputs and flat text files for data storage.

The chosen architectures focused on extensibility, the leverageability, and the maintability characteristics and these characteristics might not be important for a software project. The characteristics should be always weighted against the needs of the project.

6.1 Future Research

The chosen architectures in this thesis were used to implement a project from the start. The implementations did not have any historical weight from the previous business requirement changes or technology changes and upgrades. It would feasible to research how these architectures would help existing projects. It would most likely a long project to refactor an old software project but at least with the Ports and Adapters architecture should be doable.

Bibliography

1 Kruchten P. Architectural Blueprints—The “4+1” ViewModel of Software Architecture. IEEE Software 12. 1995;Available from: https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf.

2 Footer B , Yoder J. Big ball of mud. Pattern Languages of Program Design 4 (Software Patterns Series). 1999;Available from: http://laputan.org/pub/foote/mud.pdf.

3 Fowler M , Beck K. Refactoring : improving the design of existing code. Addison-Wesley; 1999.

4 Kemp P SP. File:Waterfall model.svg; 2010. [Online; accessed 2020-01-10]. Available from: https://commons.wikimedia.org/wiki/File:Waterfall_model.svg.

5 Richards M , Ford N. Fundamentals of Software Architecture: An Engineering Approach. O’Reilly Media; 2020.

6 Newman S. Building Microservices. O’Reilly; 2015.

7 Reenskaug, Trygve and Coplien, James O . The DCI Architecture: A New Vision of Object-Oriented Programming [website]; 2009 [cited 2020-02-25]. Available from: https://klevas.mif.vu.lt/~donatas/Vadovavimas/Temos/DCI/2009%20The%20DCI%20Architecture%20-%20A%20New%20Vision%20of%20OOP.pdf.

8 Jacobson I. Object Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley; 1992.

9 Screaming architecture [website]; 2011 [cited 2020-02-25]. Available from: https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html.

10 Hexagonal architecture [website]; 2009 [cited 2019-09-23]. Available from: https://web.archive.org/web/20090122225311/http://alistair.cockburn.us/Hexagonal+Architecture.

11 Cth027. Example of hexagonal architecture; 2019. [Online; accessed 2020-02-15]. Available from: https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)#/media/File:Hexagonal_Architecture.svg.

12 Gamma E , Helm R , Johnson R , Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley; 1994.

13 The Clean Architecture [website]; 2012 [cited 2020-10-31]. Available from: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

14 Martin R C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall; 2017.

15 Design Principles and Design Patterns [Article]; 2000 [cited 2020-11-02]. Available from: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

16 Meyer B . Object-Oriented Software Construction. Prentice Hall; 1997.