To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

en in app Get started

Pasindu Hewagamage · Follow
May 11, 2020 · 4 min read

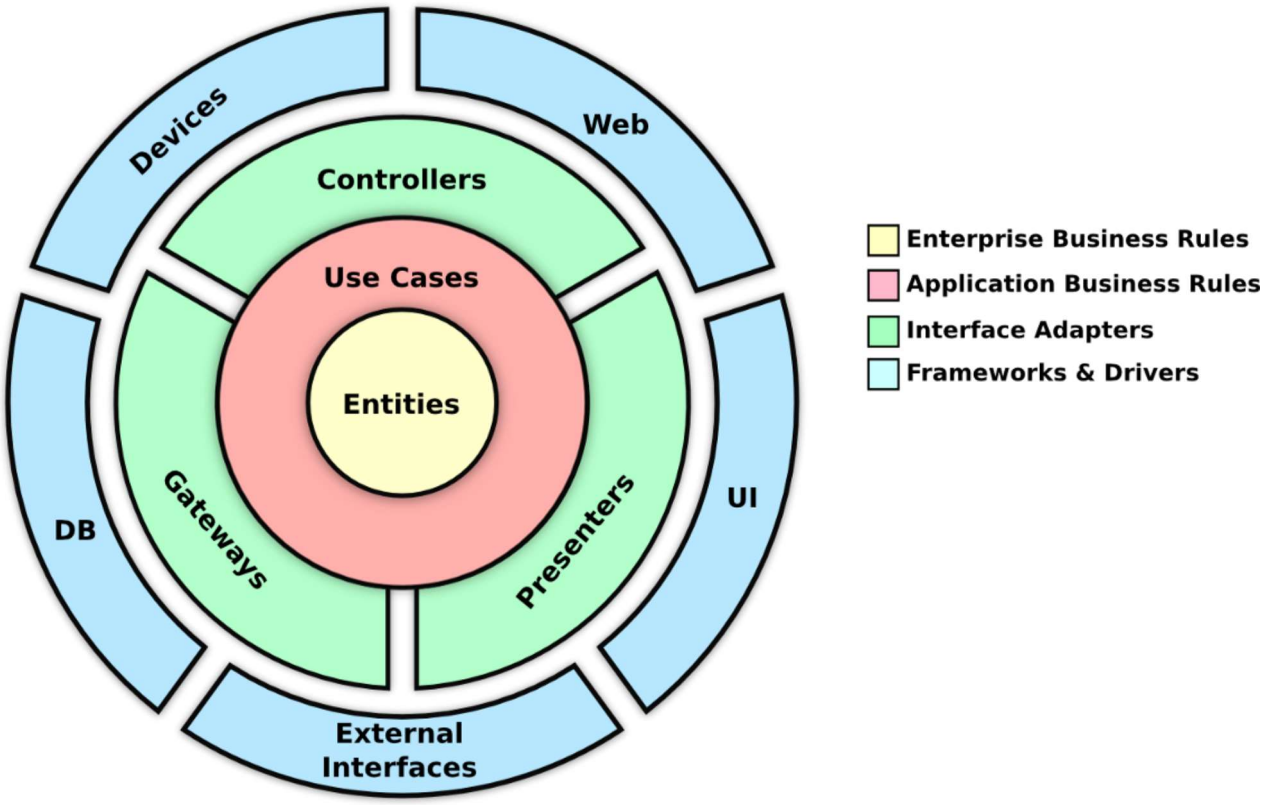
Clean Architecture on Spring Boot

Here I'm trying to give a way for a Spring Boot project with Clean Architecture based on Uncle Bob's Clean Architecture concept and my personal experiences. But there is a quiet difference in the way which I built this demo project with the Uncle Bob's guide.

I recommend you to get an idea about SOLID principles before following the Clean concept. Here I'm not going to give any explanation about SOLID Principles.

This will be a good read for SOLID principles: https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

Clean Architecture



To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.
en in app Get started

When you search in Goo cording to uncle Bobs's Clean concept your project can be separated for four layers.

1. Entities
2. Use Cases
3. Framework/Application
4. External

In my case I divide project in to three separate layers

1. App
2. Domain
3. External

Use Cases include business rules, Entities define for business entities and these entities are manipulated by use case. I'm using these two layers together in Domain layer as core layer of the project. However, entities are free to transfer between all layers. Application layer acts as a delivery layer. It is responsible for framework utilities such as handle request response, exception handling, alarm generating. Third party libraries and external things such as UI, DB, Web clients belongs to External. Domain belongs to most inner circle and external belongs to most outer circle as shown in the diagram 1.0. Domain components tightly couple with business rules and external component can be change eventually. However, Domain layer should not depend on external and application layers.

Key concepts in Clean Architecture

1. every component should include in its natural layer. eg: business rules should include in domain, database transactions belongs to external.
2. domain layer should be independent from each other layers. To satisfy this concept,need a clear understand about dependency inversion rule in SOLID principles. Any dependency doesn't exist in inner circle which belongs to any other

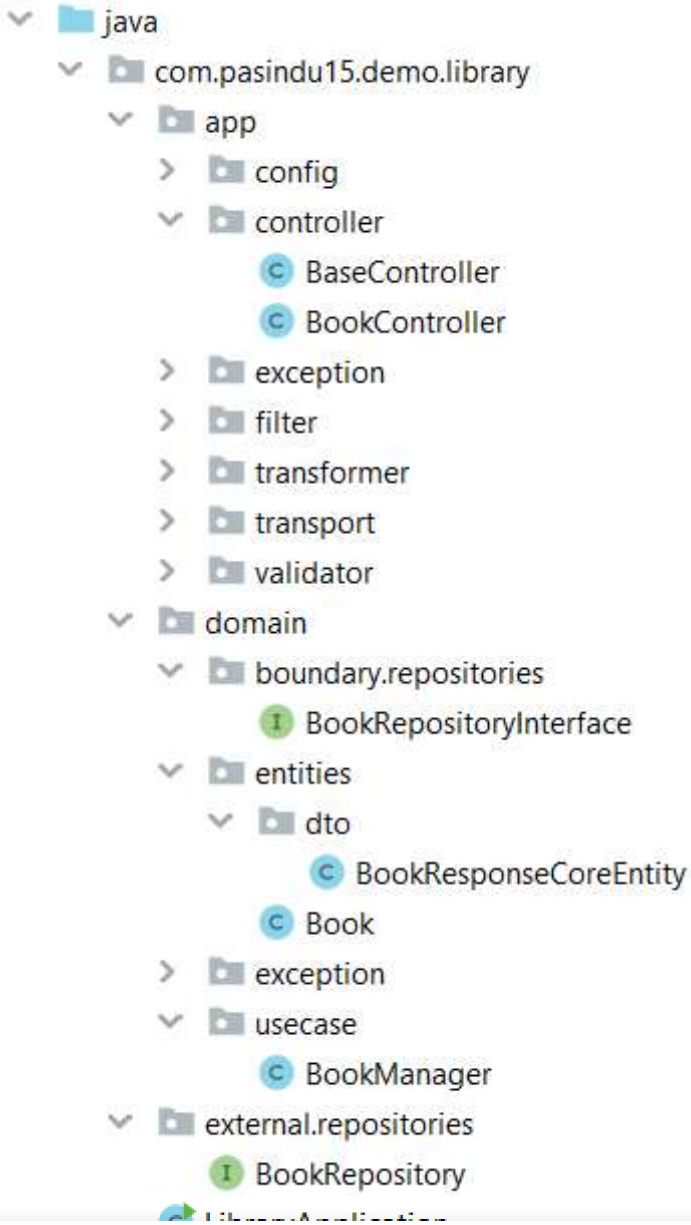
To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

en in app Get started

3. Unit Testing should perform in each layer separately. Then business rules can be tested without UI, DB, Web clients.

You can learn more about Clean Architecture in here https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

You can find demo project https://github.com/pasindu-15/libro which is a sample CRUD operation project.



To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.
en in app Get started

Application →Domain

```
1  public class BookController extends BaseController {
2      @Autowired
3      BookManager bookManager;
4      @PostMapping(value="/add", produces = MediaType.APPLICATION_JSON_VALUE)
5      public ResponseEntity<Map> integration(@RequestBody(required = true) BookRequestEntity book
6          Book book = new ModelMapper().map(bookRequestEntity, Book.class);
7          BookResponseCoreEntity bookResponseCoreEntity = bookManager.add(book);
8          Map trResponse = transformer.transform(bookResponseCoreEntity,new BookOperationResponse
9          return ResponseEntity.status(HttpStatus.OK).body(trResponse);
10     }
11 }
```

BookController.java hosted with ❤ by GitHub view raw

BookRequestEntity has been mapped to a domain Book entity

Before transferring request object from controller to use case it should map to a domain object. Otherwise use-case can not act independently.

Domain →External

```
1    @Autowired
2    BookRepositoryInterface bookRepository;
3    public BookResponseCoreEntity add(Book book) throws DomainException {
4          Book bookRes = bookRepository.save(book);
5          if(bookRes == null)
6              throw  createDomainError("0001","Add Book failure");
7          BookResponseCoreEntity bookResponse = new BookResponseCoreEntity("0000","Success");
8          return bookResponse;
9      }
```

domain/use-case depend on only domain components

```
1  package com.pasindu15.demo.library.domain.boundary.repositories;
2
3  import com.pasindu15.demo.library.external.repositories.BookRepository;
4
5  public interface BookRepositoryInterface extends BookRepository {
```

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

en in app Get started

Above example shows BookRepositoryInterface.java and it belongs to domain/boundary. Even domain/use-case can persist data without any visibility regarding external repository or DB. Only external repository is responsible for data persistence.

Testing

You are able to independently test each layer in a clean project. Testing and debugging both will be easy tasks under this concept.

In Spring Boot I'm using **Mockito** framework for mocking beans and **WireMock** for mocking external servers such as web clients.

we can perform unit tests for business rules without depending any other layers. Application and e*xternal layer components also can be tested* without existing any other layers.

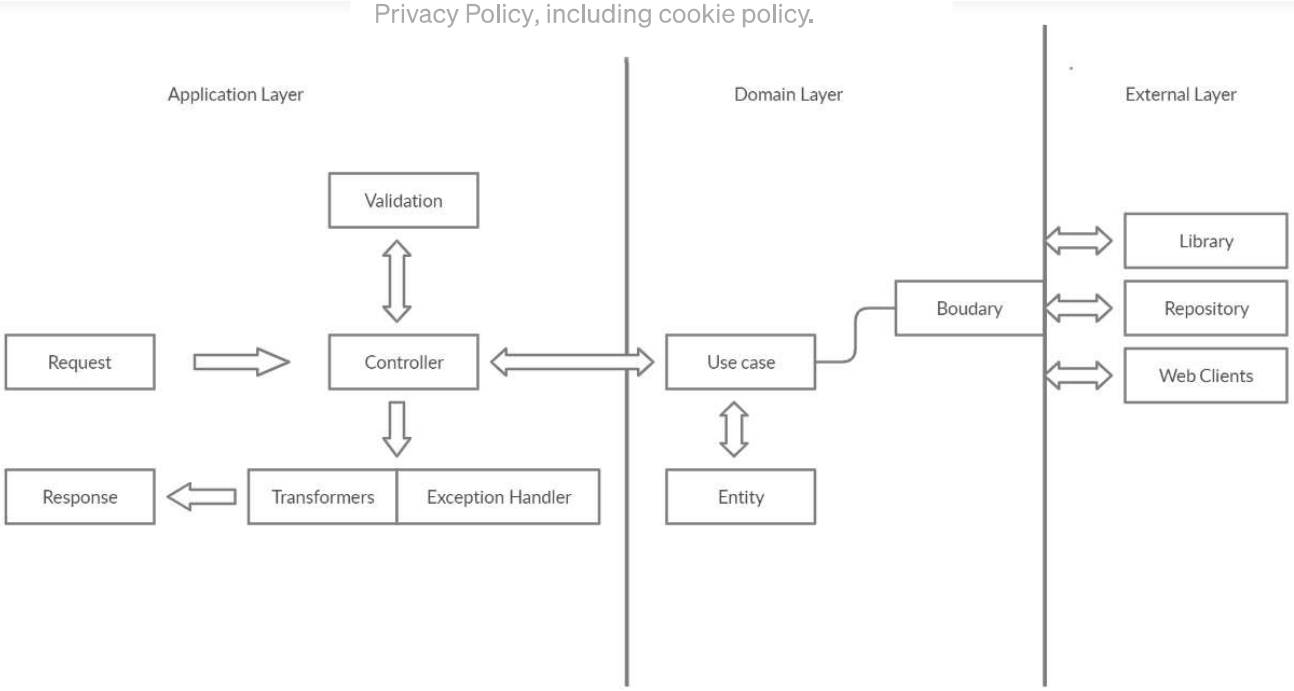
```
1   @MockBean
2   BookRepositoryInterface bookRepository;
3   @Autowired
4   BookManager bookManager;
5   @Test
6   void testAdd() throws DomainException {
7     Book book = new Book("Test Name","Test Type","Test Author");
8         when(bookRepository.save(book)).thenReturn(book);
9         BookResponseCoreEntity bookResponseCoreEntity = bookManager.add(book);
10        assertEquals(bookResponseCoreEntity.getCode(),"0000");
11        DomainException e = bookManager.createDomainError("0001","Add Book failure");
12        try {
13            when(bookRepository.save(book)).thenReturn(null);
14            bookManager.add(book);
15        }catch (DomainException ex){
16            assertEquals(ex,e);
17        }
18  }
19
```

BookManagerTest.java hosted with ❤ by GitHub view raw

BookManagerTest.java use-case test class

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

en in app Get started



control flow — diagram 1.2

Conclusion

Adding new feature to the project, removing or modifying features, validation, bug fixing, testing, error handling, project handover become more easier with clean architecture.

But clean Architecture is not recommended for small projects, because it naturally use more classes(components) and it's a reason for increasing the code complexity and decreasing the efficiency. In programming we should balance following factors in our code.

1. maintainability
2. complexity
3. efficiency
4. readability
5. best practices

In some cases, while We are following the clean concept, above factors can be deviated

To make Medium work, we log user data.
By using Medium, you agree to our
Privacy Policy, including cookie policy.

en in app Get started