Hexagonal Architecture in Java

Hexagonal Architecture in Java

Hexagonal architecture is a Domain-driven architecture. In other words, we can state it as a pattern for Application Design. It is also known as Ports and Adapter Architecture.

As we know every application has three parts:

Inbound Flow: Using this user/client connects to our application

Business Logic: This is the brain of our application, where all the business rules are implemented to produce the desired result

Outbound Flow: Using this application connects to the outside world, eg: Database

Like in Cricket we have Inner circle and Outer circle covering the whole ground, here we have inner Hexagon and Outer Hexagon layer.

Inner Hexagon contains core business logic and edges will act like ports.

And Outer Hexagon contains Adapters, which access application logic via Ports. Here we can have external frameworks, like Spring Boot, to help in connecting Database or giving Rest Call to other Microservices.

We will now try to understand this by an example, using HR Operation Application, where we will create an Employee module for HR operations. We will be using Spring Boot framework for creating it.

Don't worry about Spring Boot, because it doesn't matter if you don't know about it. On a high level you can think of it as a tool to develop Spring MVC application by providing a lot of optimized code modules ready to use.

```
@Entity
@Table(name = "employee")
public class Employee
{
   @Id
   @GeneratedValue(strategy= GenerationType.IDENTITY)
   private long id;
   private String name;
   private Long salary;

   //getters and setters
   //equals and hashcode implementation
   //toString for logging Object information
}
```

Domain: The first thing we will be creating is Employee Domain and define our core logic for basic operations.

Ports: Ports are actual contract defined for Adapters to use and access domain logic for data manipulation.

```
public interface IEmployeeService
{
   List<Employee> getAllEmployee();
   Employee save(Employee employee);
   Employee findEmployeeById(Long employeeId);
}
```

Now we have to create another Port to connect to DB for our outbound flows:

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee,Long>
{
   @Override
   Optional<Employee> findById(Long integer);
}
```

Adapters: Adapters are the implementation of Ports. Now don't get confused, because we have Primary and Secondary Adapters, where Primary Adapter helps the client to connect to our Application and Secondary Adapter gives us the implementation of Ports for business logic and also connects to external worlds like Database and other Applications.

```
@Service
public class EmployeeServiceImpl implements IEmployeeService
{
   @Autowired
   private EmployeeRepository repository;
```

```
    @Override
    public List<Employee> getAllEmployee() {
        List<Employee> employees = repository.findAll();
        return employees;
    }
    @Override
    public Employee save(Employee employee)
    {
        Employee emp=null;
        if(null != employee)
        {
            emp = repository.save(employee);
        }
        return emp;
    }
    @Override
    public Employee findEmployeeById(Long employeeId)
    {
        return repository.findById(employeeId).orElse(null);
    }
}
```

Here we have to create only one secondary Adapter by implementing EmployeeService Port for business logic. For EmployeeRepository Port Spring Boot provides ready to use Implementation. As we have already stated, it will be part of the outer hexagon layer.

```
@RestController
@RequestMapping("employee")
public class EmployeeResource
{
    @Autowired
    private IEmployeeService employeeService;

    @RequestMapping(value = "all", method = RequestMethod.GET)
    public List<Employee> getAllEmployee()
```

```
    {
        return employeeService.getAllEmployee();
    }

    @RequestMapping(value = "{id}", method = RequestMethod.GET)
    public Employee getEmployee(@PathVariable("id") Long id )
    {
        return employeeService.findEmployeeById(id);
    }

    @RequestMapping(value = "save", method = RequestMethod.POST)
    public Employee saveEmployee(@RequestBody Employee employee) {
        return employeeService.save(employee);
    }

}
```

Now we will create our last Primary Adapter by implementing Rest Endpoints.

```
[ {"id": 1, "name": "user1","salary": 50000},
  {"id": 2, "name": "user2","salary": 40000}
]
```

Sample Output:

Conclusion: Quick summary for our discussion in this tutorial:

This Architecture helps in layering your business logic, inbound, and outbound flow in such a way that your core business logic remains isolated from external components. Users can interact with Application using Rest Endpoints and Application uses ports to connect to outside components

It also enables us to use different adapters without changing core business logic

Testing Application logic also becomes easy

It makes your application more flexible and easier to integrate with new requirements

The full source code for the example can be found over on GitHub.