Hexagonal Architecture

(Didn’t invest time in actual coding example!)

**Hexagonal architecture –  the genesis**

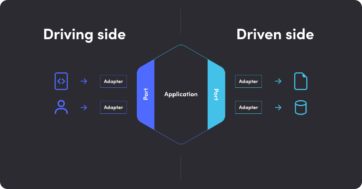
The concept of hexagonal architecture was first introduced in 2005 by an American computer scientist Alistair Cockburn, well known also for his contributions to the Agile movement.

Cockburn reached a conclusion that the core application interacts with the user interface or databases or automated test in a remarkably similar way. Therefore, **all of these external systems can be separated from the core app / business logic** and made to communicate with it in a technology-agnostic way.

Sounds confusing? It will all be clear when we take the concept of the hexagonal architecture under a microscope.

**Hexagonal architecture – let’s break it down!**

This is a basic overview of hexagonal architecture. In the following sections, we’re going to briefly talk about each high-level element of this architecture.



Hexagonal architecture – basic overview

**The basics – ports and adapters**

The hexagonal architecture follows a couple of basic principles, including adapters architecture, ports and adapters:

* Explicit separation of user-side, server-side, and business logic.
* The separation is achieved through the use of **Ports and Adapters** (adapters architecture).
* All dependencies move from the user-side / user interface and server-side towards the business logic.

In hexagonal architecture, the core application includes **all the business logic** as well as the services responsible for various functionalities and use cases. The core receives from and sends commands/queries to external systems using Ports and Adapters.

**Ports**

The term “ports” simply refers to entry points to the application core. They contain (typically technology-neutral) interfaces that make it possible for external entities to obtain a set of rules for communicating with the core. Since the ports are essentially just gateways, another agent is necessary to actually make the communication happen. These are adapters.

**Adapters**

The adapters actively initiate the communication between external entities and the core. Each port can serve many adapters. A common example of a controller could be a REST controller or any other API request handler.

Here is an extremely important thing about the adapters architecture – ports/adapters work with both the external systems that start the communication (driving side) and the ones that receive it (driven side). But the exact mechanism slightly differs.

**Driving side**

The driving actors are those that **start the interaction** with the application by initiating a query. For example, it can be the mobile application interface or user interface code of a web app. The user input passed into the UI is taken by the adapter and sent to the core through the port. Both the port (interface) and the implementation of the interface will be inside the core/hexagon.

**Driven side**

The driven actors are those that need the core application to interact with them. It could be databases or even other applications. In this case, the application calls the external (driven) entity. Then, the driven adapter implements the port for the core to use. This time, the implementation is within the driven adapter.

**Why a hexagon?**

The six *ends*, on their own, don’t really have any particular meaning as far as application code is concerned. So why a hexagon?

The hexagon shape (as in the above example) is simply a convenient way to depict that in this particular architecture:

* The core logic and services are inside,
* They communicate with various external actors (*ends*) using Ports and Adapters,
* Those can be divided into driving and driven actors, which again is easy to depict using a symmetrical shape.

Let’s see now see what hexagonal architecture can do for you.

**Hexagonal architecture – benefits**

Organizing your code in the manner prescribed by the hexagonal patterns has a lot of potential benefits:

* When done correctly, it makes it possible to isolate the application and business logic from external factors so that they can all be **tested** easily and separately.
* At the same, their dependencies can be easily **mocked**.
* Designing the user interfaces by their purpose rather than technology ensures that your application’s [**technology stack**](https://tsh.io/blog/web-development-stacks/) can freely grow over time.
* Helps implement the **Domain-Driven Design** by making sure that the domain logic does not leak out of the core.
* The ports and adapters are just as replaceable as all the external entities, further contributing to the **scalability**of the entire application.
* The advanced separation of concerns also makes the app **easier to maintain**, as changing the code in one place or adding new dependencies/ways to interact with the app, do not require significant code changes.
* Since one can test outside dependencies without any extra mocking tools, improving the overall **testability**of the application.

Code Example –

public class UserAdmin

{

private readonly IUserRepo \_userRepo;

public UserAdmin(IUserRepo userRepo)

{

\_userRepo = userRepo;

}

UserData \_userData;

public void Save()

{

Validate(\_userData);

\_userRepo.Save(\_userData);

}

...

}

class UserDatabaseRepository : IUserRepo

{

public void Save(UserData userData)

{

using(var db = GetDatabaseConnection())

{

db.ExecuteSave(userData);

}

}

}

class UserHttpRepository : IUserRepo

{

public void Save(UserData userData)

{

using(var http = GetHttpConnection(Connections.UserRepository))

{

http.Post(userData);

}

}

}

In the above three code snippets, the UserAdmin is least care about actual implementation of public void Save() method, this can be done by implementing http interface as well as implementing database interface!!!