**Hexagonal Architecture**

The hexagonal architecture, or ports and adapters architecture, is an architectural pattern used in software design.

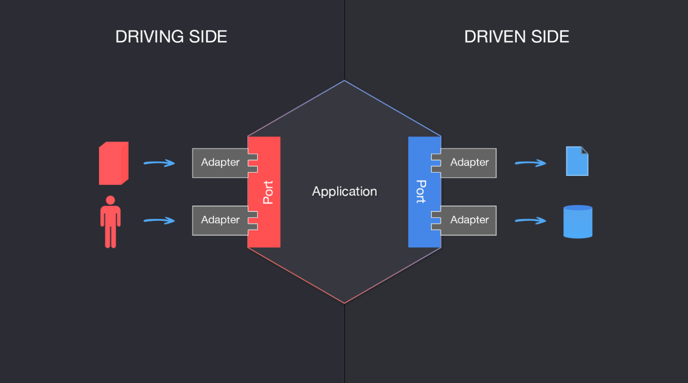
**What is Hexagonal Architecture?**

The hexagonal architecture divides a system into several loosely coupled interchangeable components, such as

* the application core
* the database
* the user interface
* test scripts
* the interfaces with other systems.

This approach is an alternative to the traditional layered architecture. The Hexagonal Architecture, also referred to as **Ports** and **Adapters**, is an **architectural pattern** that allows input by users or external systems to arrive into the Application at a Port via an Adapter, and allows output to be sent out from the Application through a Port to an Adapter.

This creates an abstraction layer that protects the core of an application and isolates it from external - and somehow irrelevant - tools and technologies.



**The basics: Ports and Adapters**

As discussed above 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.

**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.

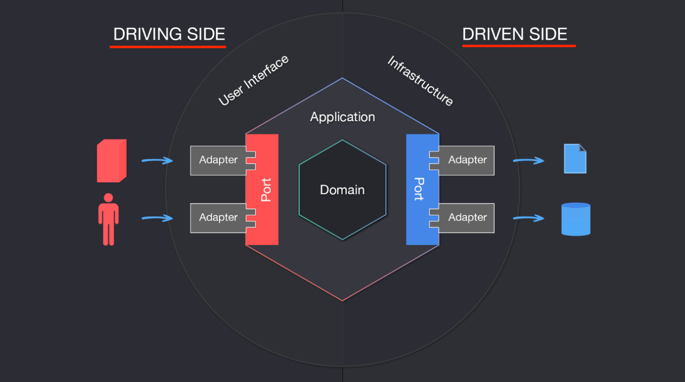
Here is an extremely important thing about the adapters architecture - ports/adapters work with both the external systems that start the communication (driving side) and the ones that receive it (driven side). But the exact mechanism slightly differs.

**Why it is called Hexagonal Architecture**

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 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.



**Driving Side vs. Driven Side**

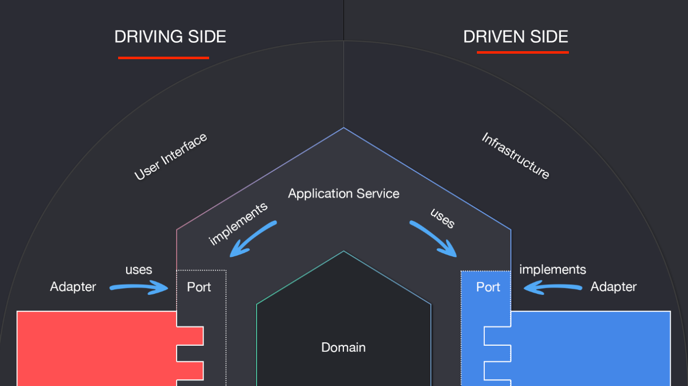
In the above image, we see Driving Side and Driven Side highlighted - so what are those? Let's demystify.

* **Driving (or primary)** actors are the ones that initiate the interaction, and are always depicted on the left side. For example, a driving adapter could be a controller, which is the one that takes the (user) input and passes it to the Application via a Port.
* **Driven (or secondary)** actors are the ones that are "kicked into behavior" by the Application. For example, a database Adapter is called by the Application so that it fetches a certain data set from persistence.

When it comes to implementation, there are a couple of important details that shouldn't be missed:

* Ports will be represented most of the time (depending on the language you choose) as interfaces in code.
* Driving Adapters will use a Port, and an Application Service will implement the Interface defined by the Port. In this case, both the Port's interface and implementation are inside the Hexagon.
* Driven adapters will implement the Port and an Application Service will use it. In this case, the Port is inside the Hexagon, but the implementation is in the Adapter, therefore outside of the Hexagon.

We can observe this in the following diagram.



**Hexagonal architecture: the benefits**

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

* When utilizing this organization correctly, the application and business logic can be isolated from external factors so that each can be **tested** easily and separately.
* At the same, their dependencies can be easily **mocked**.
* Designing the user interfaces by their purpose, rather than their technology, ensures that your application's **technology stack** can grow freely over time.
* This organization helps the implementation of **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 of 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 and ways to interact with the app do not require significant code changes.
* One can test the outside dependencies without any extra mocking tools, improving the overall **testability** of the application.

**Will Hexagonal Architecture solve all problems??**

In light of these benefits, using hexagonal architecture would seem to be a no-brainer. But is it really the case? Let's take it to the experts:

* We ask: *Should we use hexagonal architecture in our projects?* As always, the answer is: *It depends.* If the project is a fairly simple CRUD application, it is probably not worth it. However, the more complex the project is, the more sophisticated solutions are required.
* We build applications with a bunch of different tools. Frameworks are one of them. Frameworks have the tendency to dominate. With the hexagonal architecture, we can keep frameworks in line and keep the Core Application framework agnostic.

To sum it up: organizing your app in a way hexagonal architecture demands and keeping the separation of concern is not necessarily an easy task. It does require skill and time. But it will pay off in the long run - providing that your app is big and complex enough for the "long run" to be worth it! If it isn't, you may consider implementing only some aspects of the architecture to improve separation of concern. There are many ways to go about it, and it is something that you should discuss with your development

**Also read this**

* Alistair Cockburn's [original paper on Hexagonal Architecture](https://alistair.cockburn.us/hexagonal-architecture/)
* Awesome read on Domain-Driven Design: [Everything You Always Wanted to Know About it, But Were Afraid to Ask](https://medium.com/ssense-tech/domain-driven-design-everything-you-always-wanted-to-know-about-it-but-were-afraid-to-ask-a85e7b74497a)