Upgrade Open in app

Published in Better Programming

Vlad Ostrenko Follow
Oct 27, 2021 · 10 min read · Listen

Save

Comparing Three-Layered and Clean Architecture for Web Development

The most popular approaches to web application development

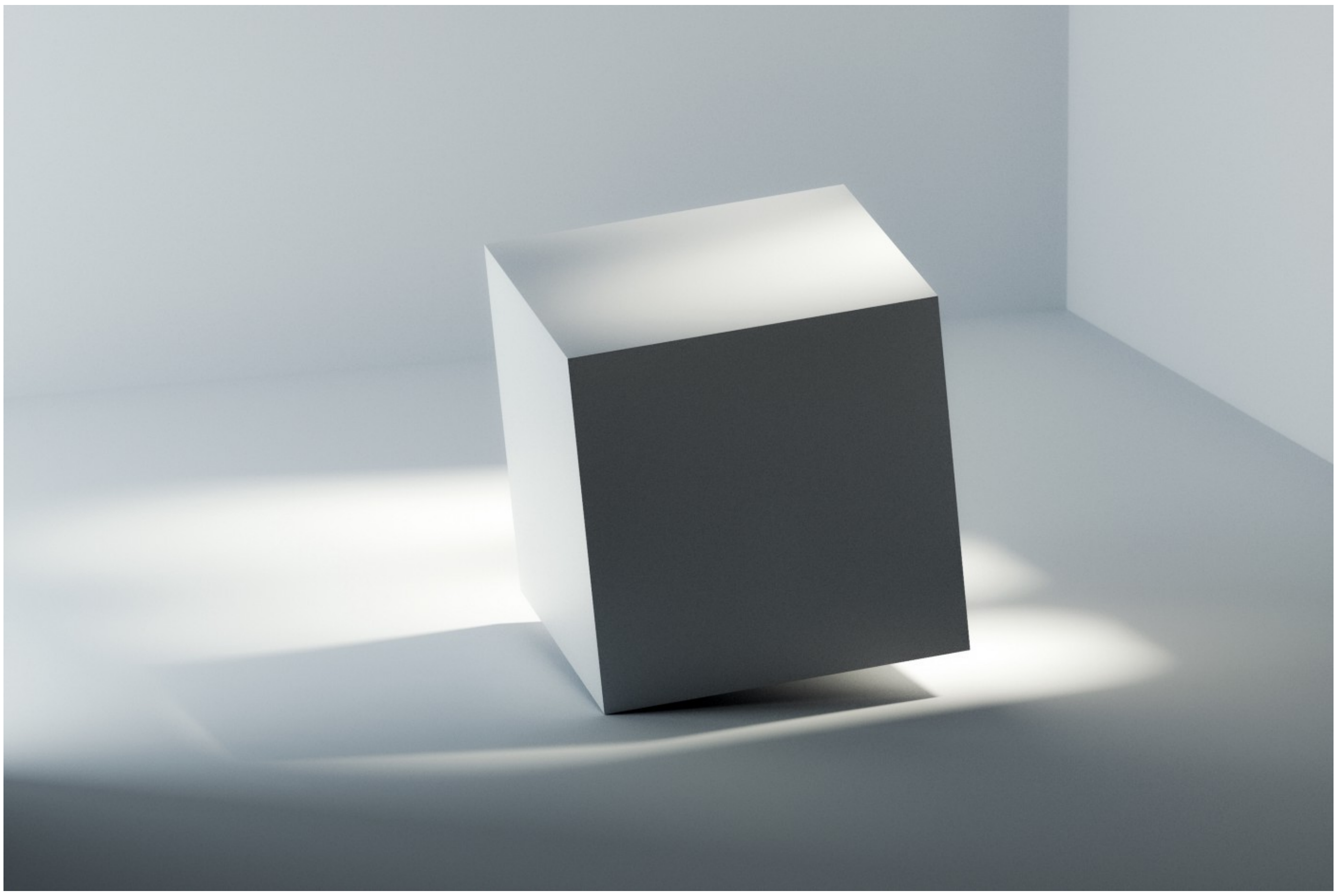


Photo by Fakurian Design on Unsplash

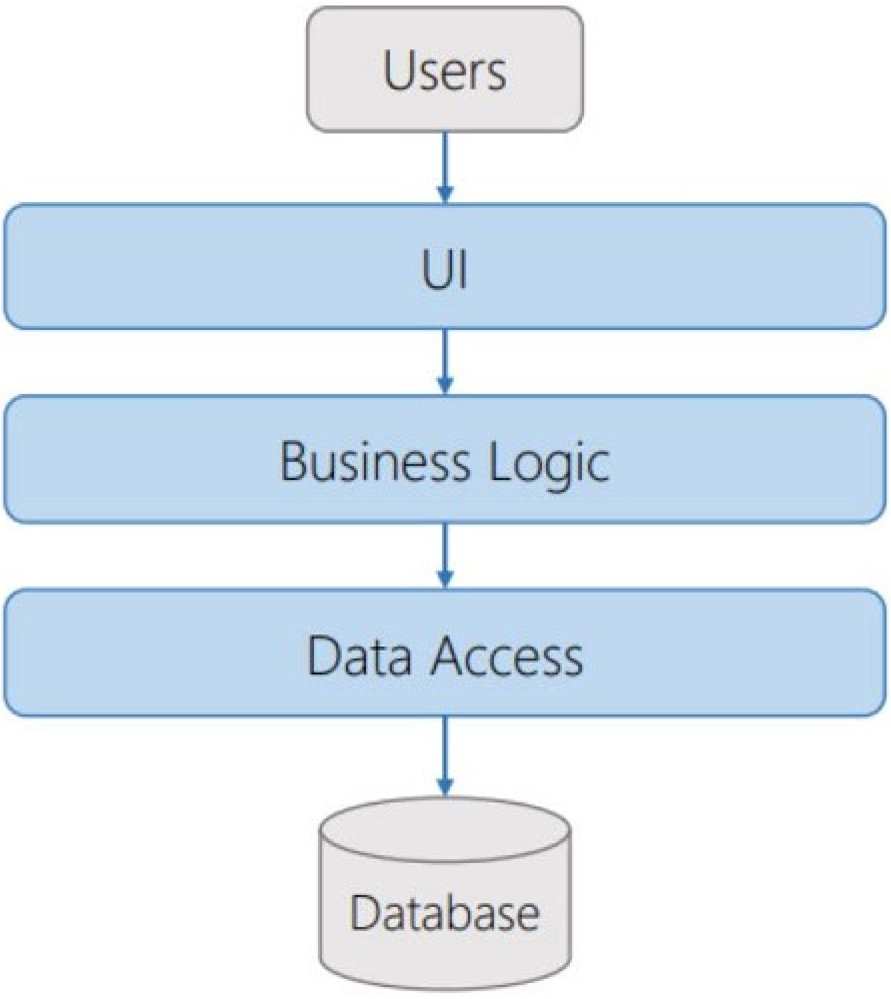
In this article, I would like to compare classic three-layer architecture and Clean Architecture. I will try to answer the following questions:

- What is the difference, and when to use each approach?

What Is Three-Layer Architecture?

I would like to point out at the outset that three-layer architecture is often confused with three-tier or n-tier architecture. The layer is a logical separation of the application code inside your monolith. The tier is physical separation. It means, if we are talking about a three-tier architecture, then the application could be divided into these levels: the database server, the web application on the web server, and the user's browser. With that being said, each tier would represent a distinct and separate physical process.

But in this article, we are going to break down three-layer architecture. This architecture logically separates your application code into three parts. Let's see what these layers are and what their responsibilities are.



contain controllers and view models, as well as views that make up the user interface (HTML static pages, JavaScript).

- Business logic layer. This layer contains a set of components that are responsible for processing the data received from the UI layer. It implements all the necessary application logic, all the calculations. It interacts with the database and passes the result of processing to the UI layer.
- Data access layer. It stores the data models. It also hosts specific classes for working with different data access technologies, like ORM.

The important thing here is how these layers depend on each other. The data access layer is independent of the other layers; the business logic layer depends on the data access layer, and the UI layer depends on the business logic layer. You can see it by the direction of the arrows in the diagram above.

It means that the data and the way it is persisted is the most crucial part of the application. That's because most of the application is dependent on the data access layer, and any changes to the technologies in this layer would require changes in other layers.

Apart from that, it should be noted that the presentation layer can not directly interact with the data access layer. It can be done only through the business logic layer.

Okay, let's see how we can implement our application with three layers.

Implementation of Three-Layer Architecture

Let's take a REST API as an example. In this case, our UI layer will be a controller that returns view models in JSON format. Here's the code:

```
@HttpController()
class OrderController {
  @Post('/orders')
  async createProduct(req: http.Request, res: http.Response) {
    try {
      if (!isValid(req)) {
        throw new Error('validation error')
      }
    }
  }
```

ProductController.ts hosted with ❤ by GitHub view raw

Here we have a simple validation of the incoming request along with business logic layer interaction. We are passing the request down to the next layer.

Next is our business logic layer. We abstract it into a `Service` class.

```
class OrderService {
  static async placeOrder(productId: string, clientId: string) {
    const client = await Client.find({ id: clientId })
    if (!client.isActive) {
      throw new Error('client is not allowed to make orders')
    }

    const product = await Product.find({ id: productId })
    if (product.quantity == 0) {
      throw new Error('product is out of stock')
    }

    const order = new Order({ product, client, quantity: 1 })
    await order.save()
  }
}
```

OrderService.ts hosted with ❤ by GitHub view raw

In the code example above, `OrderService` class is an implementation of the Transaction Script pattern. The Transaction Script is a module that organizes business logic by procedures where each operation handles instructions in a single request.

Here we verify if the client can place an order and check if the product is in stock. These checks are our business logic. Apart from that, we interact with the data access layer via our models. Here is the code:

```
@Entity()
class Order extends ORMEntity {

  @ManyToOne(Client)
  client: Client

  @Column('smallint')
  quantity: number
}
```

Order.ts hosted with ❤ by GitHub view raw

In models, you define data and methods. Saving, updating, and removing data is a responsibility of a model. In our example, these methods are inherited from `ORMEntity`.

This pattern of managing data is called the Active Record. Simply put, the Active Record is an approach to access the database within models.

I should also note that working with layered architecture involves implementing dependency injection to make layers loosely coupled. Dependency injection can be achieved via different libraries or using constructors.

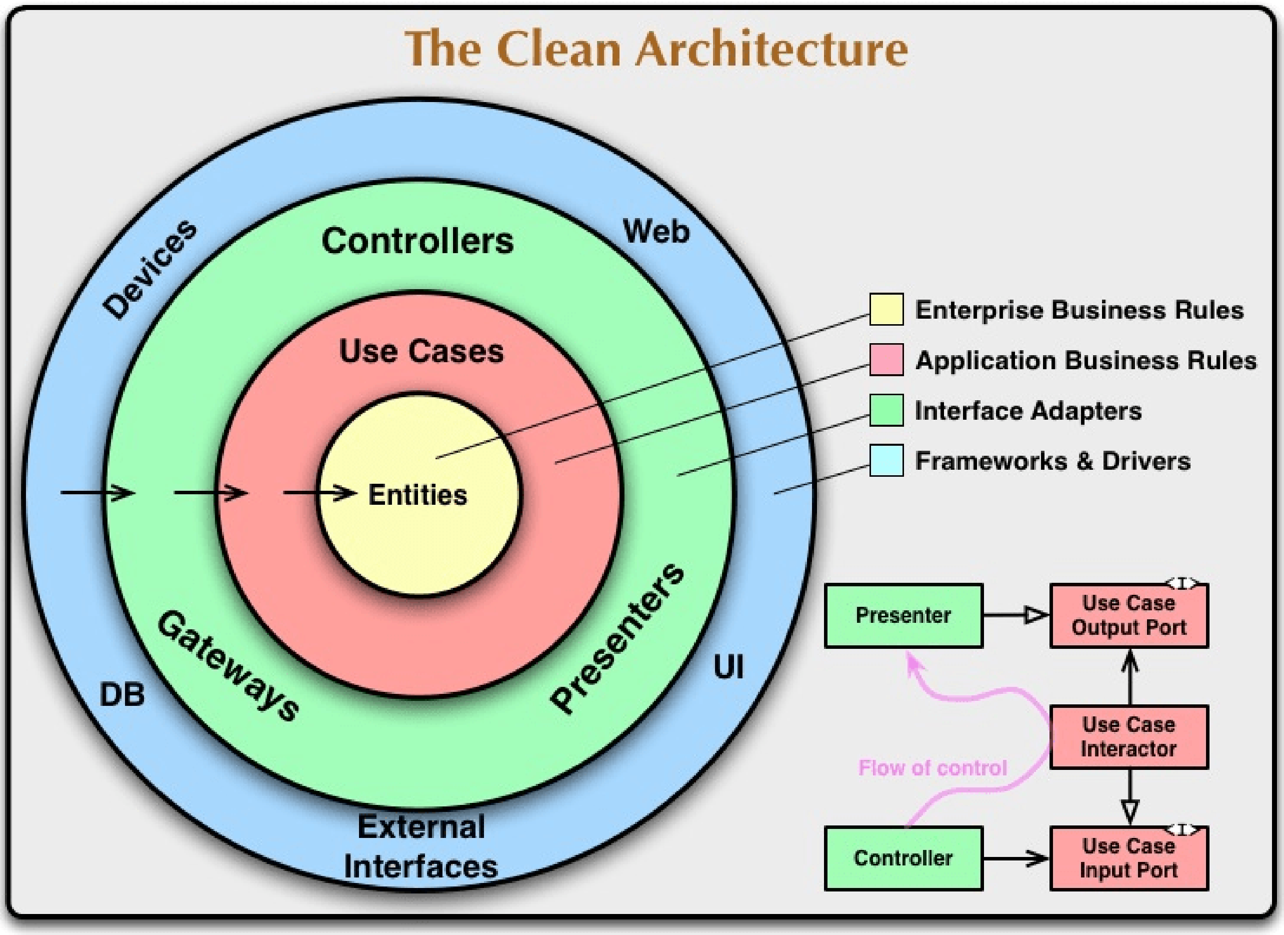
That is it. Now let us dig into Clean Architecture and then compare these two styles.

What is Clean Architecture?

Clean Architecture is also layered architecture. The layer domain (entities) is in the center surrounded by the application layer (use cases). The outer layer consists of ports and adapters that adapt the application to external systems (web, DB, UI) via controllers, repositories, presenters.

This architecture is domain-centric. It puts the domain model at the center of the application. The domain model incorporates both behavior and data but does not define the interaction with the database. The persistence and presentation of the domain model are just the details located as far away as possible.

Several popular styles implement domain-centric architecture — Hexagonal Architecture by Alistair Cockburn, Onion Architecture by Jeffery Palermo. But we are going to break down Clean Architecture by Robert Martin.



In the center, we have a **domain layer** that contains essential logic and data for business in the form of entities. For example, there is a business rule that the user can reject the order. The `Order` entity will have `status` property and `reject()` method.

Next is the **application layer**. It defines the application rules. In other words, it answers a question of how our entities can be used. For example, our business is selling products. So the use case can be place an order. We need to check if the product is in stock and then place an order. These are the rules of our application.

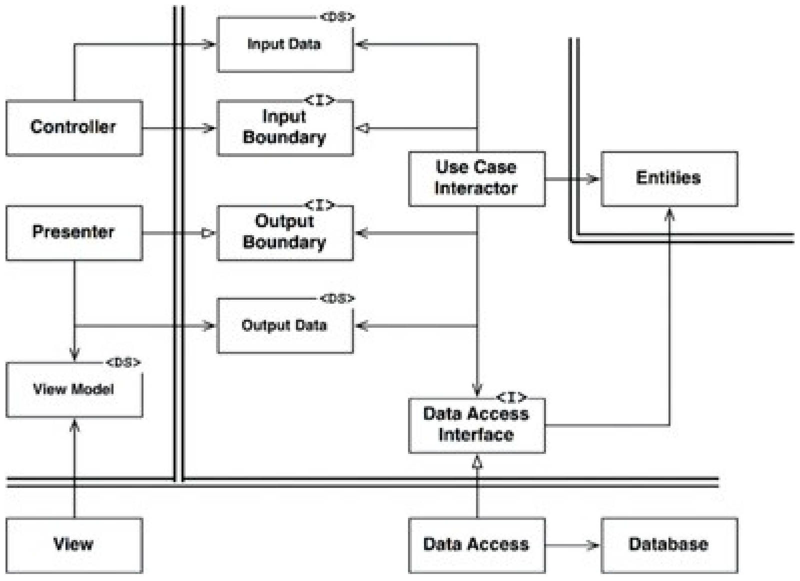
Below, we have the **adapters layer**. Controllers take input data from the infrastructure layer and transform it in a form required by use cases and entities. In our example, input data from the infrastructure can be an HTTP request. Our controller knows how to get request body or query params and pass them into the use case. After executing the use case, the result is passed into the presenter that transforms the model into the HTTP response.

Besides layer separation, we must remember the **dependency rule.**

1. Dependencies cannot be set bypassing the layers. It means that we can't use classes defined inside the application layer in the infrastructure layer, because we have adapters layer in between.

2. Our high-level layers must not depend on low-level layers. In other words, classes, functions, objects that are defined in low levels must not appear in high levels. This rule is shown in the diagram by black arrows. How can this be achieved? By using the Dependency Inversion Principle. We define ports (interfaces) at the higher level and implement them at the lower level.

The transition of data between adapters and application layers is carried out by input and output ports. The `Controller` calls a method from the `InputPort` that is implemented by the `UseCase` and passes `InputData` data structure in it. Then `UseCase` responds to the `OutputPort` with the `OutputData` data structure The `OutputPort` is implemented by `Presenter` .



Martin, R.C. Clean Architecture; page 196

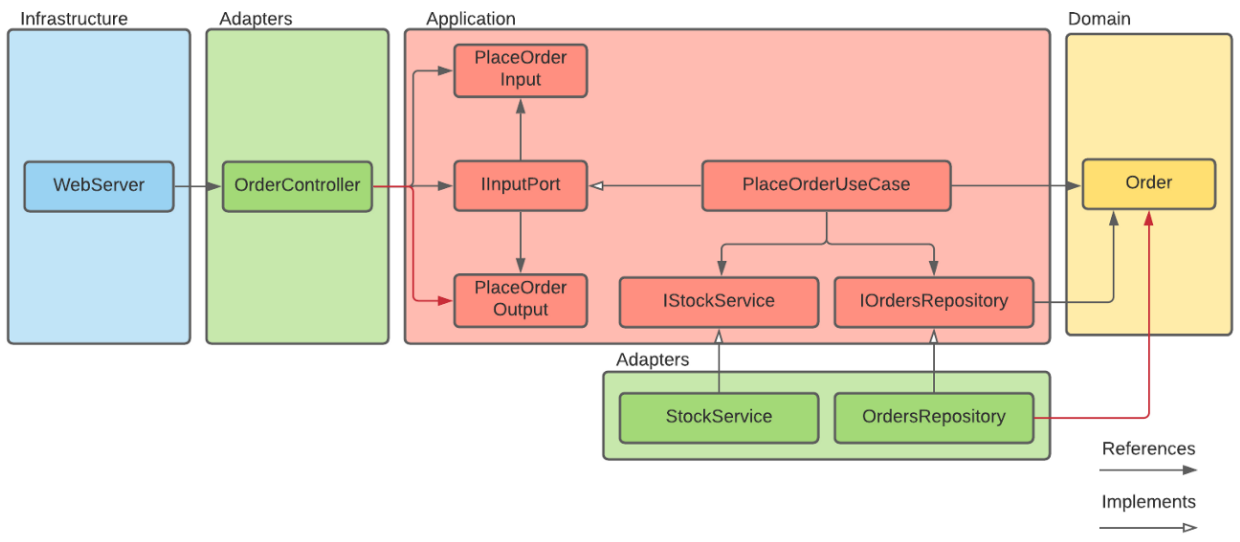
This is a diagram that Robert Martin gives in his book. We can see the dependency rule — all the arrows that are crossing layer boundaries point toward the Entities layer. Notice how dependencies are inverted for data access. Instead of using a direct implementation of `Data Access` in the `Use Case Interactor` we define an interface for

Implementation of Clean Architecture

Our example application will be a REST API. It will receive orders, check if the product is in stock, and store the order in the database.

I should point out that the implementation will be slightly different from that proposed by Robert Martin in the diagram above.

1. We won't use Presenters as we don't have UI in REST API.
2. `OrdersRepository` will refer `Order` entity. This violation is acceptable in the context of repositories because they construct entities from persisted data.



The interaction with the app starts from the user request that goes to the `HTTPServer`. `HTTPServer` class is located in the infrastructure layer. The server uses `OrderController` methods in its routes to pass HTTP requests into the adapters layer. Here is the code:

```
export class HttpServer {
  private readonly app: express.Express
  private readonly router: express.Router
  private server: http.Server

  constructor(controllers: IControllers) {
    this.app = express()
    this.router = express.Router()
  }

  private toHandler(fn) {
    return async (req, res, next) => {
      try {
        const result = await fn({ body: req.body, params: req.params })
        res.json(result)
      } catch (e) {
        next(e)
      }
    }
  }

  private static handleError(err, req, res, next) {
    res.status(500).json({ message: err.message })
  }
}
```

`OrderController` class is located in the adapters layer, and its responsibility is to validate and transform HTTP requests into an understandable format for the application layer. The format is defined inside the application layer by the `IPlaceOrderInputData` interface. Here is the code:

```
export class OrderController {
  constructor(
    private readonly placeOrderUseCase: IInputPort<IPlaceOrderInputData, IPlaceOrderOutputData
  ) {}

  public async placeOrder(req: IControllerRequest): Promise<IControllerResponse> {
    const placeOrderInputData = OrderValidator.validatePlaceOrder(req.body)
    const placeOrderOutputData = await this.placeOrderUseCase.execute(placeOrderInputData)

    return {
      status: 200,
      body: placeOrderOutputData
    }
  }
}
```

method. The request is passed into `PlaceOrderUseCase` after validation.

Next is our `PlaceOrderUseCase` . It interacts with the infrastructure layer via `IOrdersRepository` and `IStockService` interfaces. Here is the code:

```
export interface IPlaceOrderInputData {
  productId: string
}

export interface IPlaceOrderOutputData {
  orderId: string
}

export class PlaceOrderUseCase implements IInputPort<IPlaceOrderInputData, IPlaceOrderOutputDat
  constructor(
    private readonly ordersRepository: IOrdersRepository,
    private readonly stockService: IStockService
  ) {}

  public async execute(data: IPlaceOrderInputData): Promise<IPlaceOrderOutputData> {
    const isInStock = await this.stockService.checkInStock(data.productId)

    if (!isInStock) {
      throw new Error(`product with id=${data.productId} is not in stock`)
    }

    const order = new Order({
      status: OrderStatus.PENDING,
      productId: data.productId
    })

    await this.ordersRepository.save(order)

    return {
      orderId: order.id
    }
  }
}
```

implementation. The application layer remains the same. For example, we store product images in block storage, then decide to migrate to AWS S3.

After we make sure that the product is in stock, we create `Order` entity. The entity represents business-related data and methods. For example, the order status can't be changed if it's already set to `ACCEPTED` or `FAILED`. It's important to follow the single responsibility principle and not to mix the logic of one entity with another. Here is the code:

```
export class Order {
  constructor(
    private readonly props: IOrderProps,
    private readonly _id: string = Order.genId()
  ) {}

  get id(): string {
    return this._id
  }

  get productId(): string {
    return this.props.productId
  }

  get status(): OrderStatus {
    return this.props.status
  }

  public accept() {
    if (this.props.status !== OrderStatus.PENDING) {
      throw new Error('could not reject processed order')
    }
    this.props.status = OrderStatus.ACCEPTED
  }

  public reject() {
    if (this.props.status !== OrderStatus.PENDING) {
      throw new Error('could not reject processed order')
    }
    this.props.status = OrderStatus.REJECTED
  }
}
```

Afterward, the entity is persisted by `OrdersRepository` class.

Let's assume that we decided to use `pg` lib to manage data. `OrdersRepository` adapts `pg.Client` by implementing methods needed by the application layer. Here is the code:

```
export class OrdersRepository implements IOrdersRepository {
  constructor(private readonly db: pg.Client) {}

  public async save(entity: Order): Promise<void> {
    await this.db.query('insert ...')
  }

  public async delete(entity: Order): Promise<void> {
    await this.db.query('delete ...')
  }
}
```

OrdersRepository.ts hosted with ❤ by GitHub view raw

Here we provide `pg.Client` in the constructor because we need to set up the connection first. The database is our infrastructure so the code will be located inside the infrastructure layer. The code is shown below:

```
export class PGDatabase {
  public readonly client: pg.Client

  constructor() {
    this.client = new pg.Client({
      host: 'my.database-server.com',
      port: 5334,
      user: 'database-user',
      password: 'secretpassword!!',
    })
  }

  public async disconnect() {
    await this.client.end()
```

This is the elementary version of how REST API can be implemented using Clean Architecture. Complete code example you can find here.

Conclusion

The only significant advantage I can think of when comparing three-layer architecture to Clean Architecture is simplicity. Here are some of the advantages:

- There is no need to support that amount of abstractions.
- There is much easier to find and onboard developers on such projects.
- The overall development speed is higher compared to the Clean Architecture.

But advantages in the list above exist up to a certain point. When the business grows, its processes become more complex, so that it's challenging to implement them within the basic CRUD operations. Moreover, the lack of abstractions in the three-layer architecture allows you to be flexible in the choice of technologies.

On the other hand, Clean Architecture requires additional maintenance overhead but pays off significantly in the late stages. Here are some important points:

- Designing business logic independently of libraries and frameworks gives us the ability to leverage all the power of OOP. For example, we may use Domain-Driven Design for modeling our domain entities.
- If we are not sure what technology to use, we can replace it with less effort. For example, we can use memory for caching and then move to in-memory databases without modifying our domain logic. We just swap adapters.
- Clean Architecture also provides mechanisms to write modular code which can be easily mocked and tested.
- In general, Clean Architecture forces us to use SOLID principles. These are the tools that help bigger teams to maintain more complex applications.

What To Choose and When?

We can clearly see that the three-layer architecture is suitable for smaller projects. Let's

- A project doesn't involve a lot of business logic. It can be e-commerce sites, blogs, admin panels, simple APIs for CRUD.
- Development must be done fast. The project should be released in a short period of time.
- Projects that are not expected to change technologies and that have a small number of integrations.
- A small team of developers.

As for Clean Architecture, we can use it when we see that the product will be complex. Here are some examples:

- Enterprise projects like real estate or banking systems where you are going to have a lot of complex business rules.
- This architecture is suitable for pluggable applications like super apps. The apps that work in different environments like payment system that has sandbox and production APIs.
- A project that is supposed to grow and adapt technologies to its size.
- A large team of developers or a set of small teams.

But these two architectures are not mutually exclusive. I would say they can be used one after another.

Let's imagine we need to build a product from scratch, a startup. At the stage of business idea validation, we can use a three-layer architecture. It will give us development speed for some time, which should be used to find product/market fit.

After that, we can move to a Clean Architecture to be able to support functional enhancement. When our business becomes more complex, we need to tackle the complexity, and this architecture gives us tools for it.

Going further, we may use Domain-Driven Design to divide our domain into bounded contexts with their aggregates and entities. But that's a whole other story.

Upgrade Open in app

Sign up for programming bytes

By Better Programming

A monthly newsletter covering the best programming articles published across Medium. Code tutorials, advice, career opportunities, and more! Take a look.

Get this newsletter Emails will be sent to formationgeekjava@gmail.com. Not you?