UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Clase 8 - Buenas prácticas de desarrollo

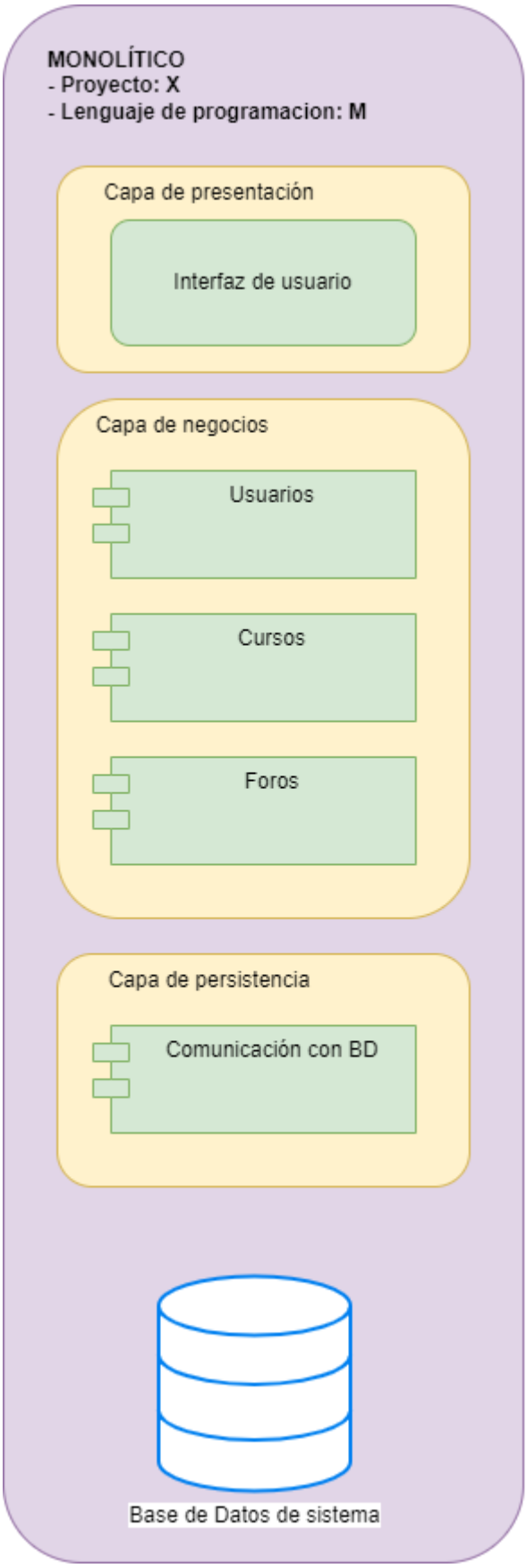
Tabla de Contenidos

Arquitecturas

Ninguna arquitectura es mejor que otra, todo depende de los requerimientos del sistema en cuestión.

Arquitectura monolítica

Hasta hace unos años, las aplicaciones usaban este tiempo de arquitectura.



¿Cuál es el problema que surgió con esta arquitectura?

Por ejemplo si surge el requerimiento de administrar exámenes esto implicaría un nuevo subsistema de exámenes, es decir, se agrega mas código a la capa de negocios, a la capa de presentación y a la capa de persistencia. Si este subsistema interactúa con alguno otro se estas interacciones a nivel código.

Esta interacción se llama acoplamiento y en el mejor de los casos el subsistema Usuarios le solicitara al subsistema Exámenes los exámenes del usuario logueado, en el peor de los casos el Subsistemas de usuarios hará la consulta de los exámenes logueados directamente a la capa de persistencia.

Si lo que se necesita es que el sistema sea escalable, se recomienda minimizar este acoplamiento ya que es inversamente proporcional al escalamiento.

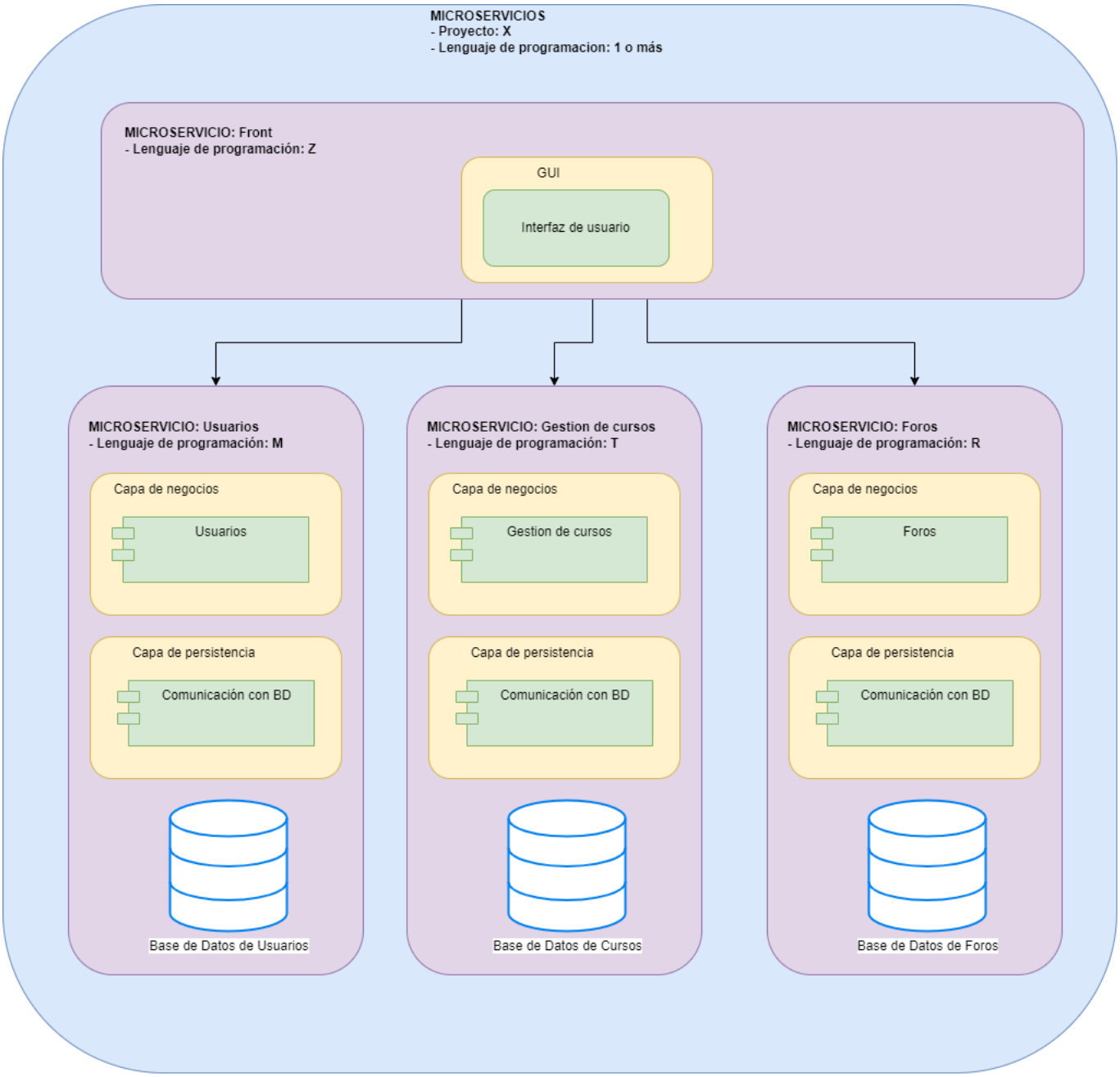
Afirmando que el pico de concurrencia de peticiones que responde este subsistema es en época de exámenes (algunos días al año) y suponiendo que debido a esta concurrencia de peticiones el sistema completo deja de responder y queda indisponibilizado para todos los usuarios. Esta situación sería caótica ya que el quizás el 25% de los usuarios son los que rinden pero el 100% no puede acceder.

Este ejemplo representa lo monolítico del sistema, ya que el sistema se maneja como uno. Si alguno de los subsistemas falla, toda la aplicación falla. La única forma de escalar es replicar todo el sistema, y quizás el requerimiento solo necesitaba escalar la administración de exámenes.

Entre las desventajas que tiene esta arquitectura en un sistema "grande" es el mantenimiento debido a la cantidad de código que maneja.

Arquitectura de microservicios

Esta arquitectura toma cada "subsistema" o un conjunto de subsistemas y generar una aplicación por cada uno de estos. Las buenas prácticas de microservicios indican que cada microservicio debe tener su propia bd de esta forma se logra desacoplar y en el caso de que algún microservicio deje de funcionar el sistema sigue levantado, obviamente sin la funcionalidad asociada al microservicio caído.



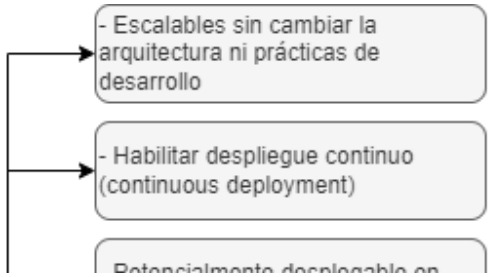
12 Factor App

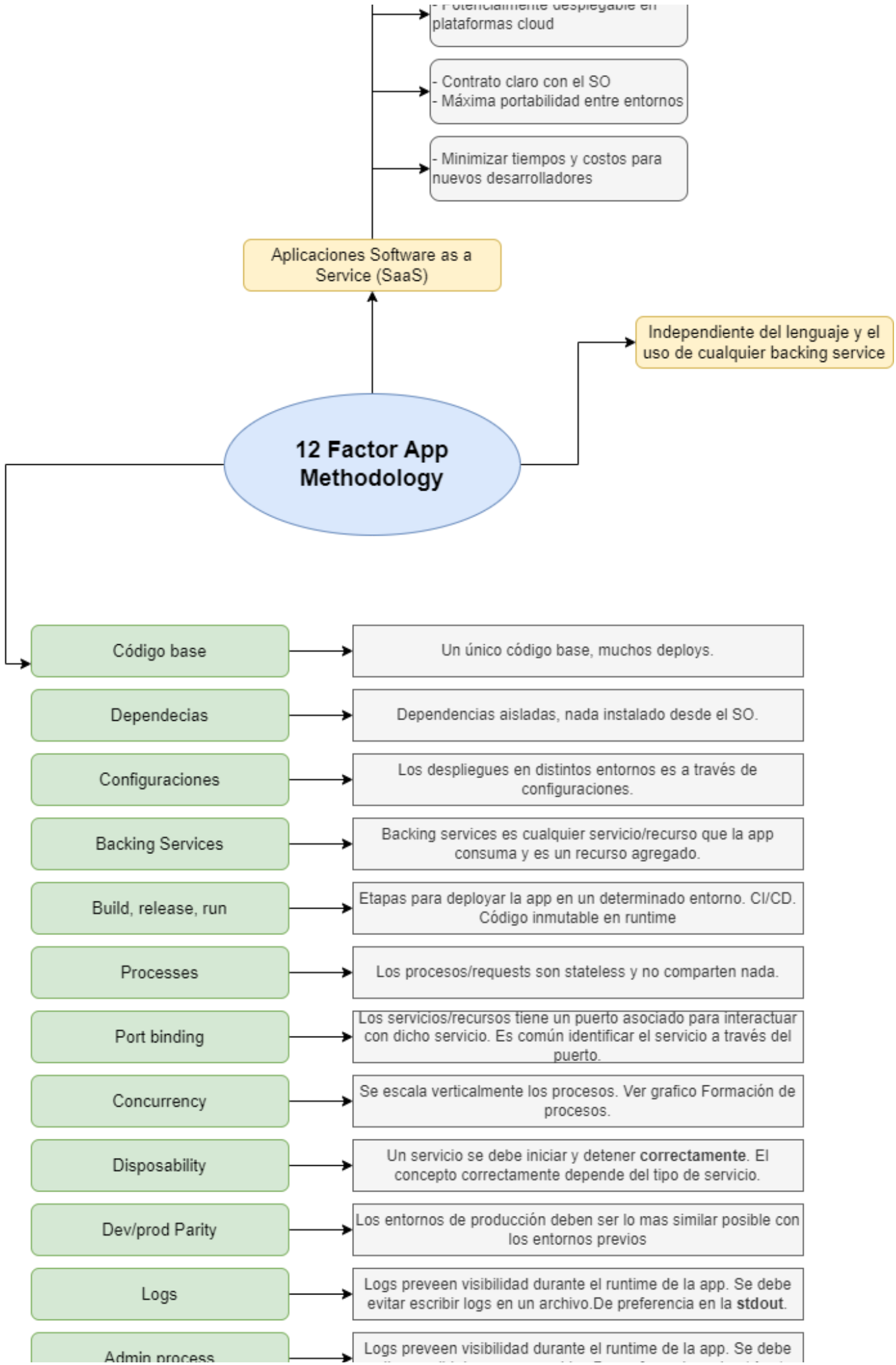
- Es aplicable a cualquier tipo de aplicaciones, escritas en cualquier lenguaje y que usen cualquier combinación de backing services (bases de datos, memorias cache, colas, etc).
- Tenerlo muy en cuenta cuando estamos hablando de microservicios.

Diagrama explicativo

Link a original aquí.

Link de solo lectura aquí




plataformas cloud
- Contrato claro con el SO
- Máxima portabilidad entre entornos
- Minimizar tiempos y costos para nuevos desarrolladores
Aplicaciones Software as a Service (SaaS)
Independiente del lenguaje y el uso de cualquier backing service
12 Factor App Methodology
Código base
Un único código base, muchos deploys.
Dependecias
Dependencias aisladas, nada instalado desde el SO.
Configuraciones
Los despliegues en distintos entornos es a través de configuraciones.
Backing Services
Backing services es cualquier servicio/recurso que la app consuma y es un recurso agregado.
Build, release, run
Etapas para deployar la app en un determinado entorno. CI/CD. Código inmutable en runtime
Processes
Los procesos/requests son stateless y no comparten nada.
Port binding
Los servicios/recursos tiene un puerto asociado para interactuar con dicho servicio. Es común identificar el servicio a través del puerto.
Concurrency
Se escala verticalmente los procesos. Ver grafico Formación de procesos.
Disposability
Un servicio se debe iniciar y detener correctamente. El concepto correctamente depende del tipo de servicio.
Dev/prod Parity
Los entornos de producción deben ser lo mas similar posible con los entornos previos
Logs
Logs preveen visibilidad durante el runtime de la app. Se debe evitar escribir logs en un archivo.De preferencia en la stdout.
Admin process
Logs preveen visibilidad durante el runtime de la app. Se debe

Bibliografía

- Twelve Factor App. Metodología de 12 factores.

Patrones de diseño

Los patrones de diseño (design patterns) son soluciones habituales a problemas comunes en el diseño de software

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución. El código del mismo patrón aplicado a dos programas distintos puede ser diferente.

Una analogía de un algoritmo sería una receta de cocina: ambos cuentan con pasos claros para alcanzar una meta. Por su parte, un patrón es más similar a un plano, ya que puedes observar cómo son su resultado y sus funciones, pero el orden exacto de la implementación depende de ti.

¿En qué consiste el patrón?

Las secciones que suelen estar presentes en la descripción de un patrón:

- El **propósito** del patrón explica brevemente el problema y la solución.
- La **motivación** explica en más detalle el problema y la solución que brinda el patrón.
- La **estructura** de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
- El **ejemplo de código** en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

Ventajas

- Son un juego de herramientas de **soluciones comprobadas**
- Definen un **lenguaje común** que puedes utilizar con tus compañeros de equipo para comunicarse de forma más eficiente.
- Los patrones intentan sistematizar soluciones cuyo uso ya es generalizado.
- Siempre hay que determinar cómo adaptarlo al contexto del proyecto en particular
- No es necesario intentar aplicar patrones en todas partes, incluso en algunas situaciones un código más simple funcionaría perfectamente bien.

Clasificación

- Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.

- Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
- Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

Catalogo

```
├── Patrones creacionales
│   ├── Factory Method
│   ├── Abstract Factory
│   ├── Builder
│   ├── Prototype
│   └── Singleton
├── Patrones estructurales
│   ├── Adapter
│   ├── Brigde
│   ├── Composite
│   ├── Decorator
│   ├── Facade
│   ├── Flyweight
│   └── Proxy
└── Patrones de comportamiento
    ├── Change of Responsability
    ├── Command
    ├── Iterator
    ├── Mediator
    ├── Memento
    ├── Observer
    ├── State
    ├── Strategy
    ├── Template Method
    └── Visitor
```

Bibliografía

https://refactoring.guru/es/design-patterns/

Patrones de diseño aplicados en ExpressJS

Código de express aquí

Aplicación de Factory Method

Expressjs crea el servidor (representado con la variable `app`) a través de la función `createApplication`, dicha función es expuesta por express.

```
/**
 * Expose `createApplication()`.
 */
```

```
exports = module.exports = createApplication;

function createApplication() {
    var app = function(req, res, next) {
        app.handle(req, res, next);
    };
...
    app.init();
    return app;
}
```

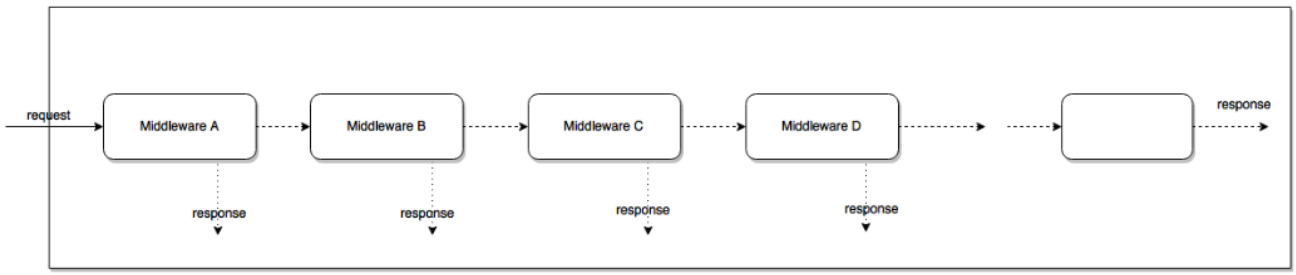
Cuando se necesita crear el servidor, se hace de la siguiente manera:

```
import express from 'express';
..
const app = express();
```

Donde `express()` invoca a la función exportada `createApplication`. De esta forma la implementación de la inicialización de servidor (variable `app`) esta abstraida para cualquier que cree una aplicación con `express()`.

Aplicación de Chain of Responsibility

Para entender es patro es necesario saber que es un middleware (software in the middle). Un middlware es código que se ejecuta antes de que el punto de ejecución llegué al controlador. Cada middle tiene un fin específico, una responsabilidad propia.



Por ejemplo, todas las peticiones HTTP viajan en texto plano. La siguientes petición es un POST que va a insertar un libro

```
POST /books HTTP/1.1
Host: localhost
Content-Type: application/json
Content-Length: 110

{
    "author": "Jonathan Lee Martin",
    "category": "learn-by-building",
    "language": "JavaScript"
}
```

Para indicarle a express que tiene que tiene que transformar el body de esta request a un JSON lo hacemos a través del siguiente middleware:

```
app.use(express.json()); // para poder leer json en el body
```

De esta forma antes de que el controlador accedamos al `req.body` este nos va a devolver el siguiente objeto y un string:

```
const book = req.body;
```

```
// Salida de console.log(book);
{
    author: "Jonathan Lee Martin",
    category: "learn-by-building",
    language: "JavaScript"
}
```

Ejemplos de middlewares

```
app.use(express.json());
```

```
app.use(errorHandler);
```

```
app.use(cors({
  origin: '*'
}));
```

Bibliografía

- https://dzone.com/articles/design-patterns-in-expressjs
- https://dzone.com/articles/understanding-middleware-pattern-in-expressjs
- https://jonathanleemartin.com/books/sample.pdf
- https://github.com/expressjs/express/blob/master/lib/express.js

Patrones de diseño aplicados en Sequelize

Código de Sequelize aquí

Singleton

Se instancia un objeto de tipo `Sequelize` (`new Sequelize ...`), el cual es almacenado en la variable `sequelize`. Cuando se quiera acceder a cualquier modelo o hacer alguna consulta se debe acceder a través de la varaible `sequelize`.

```
const { Sequelize, DataTypes, Model } = require('sequelize');

// Connect to the database
const sequelize = new Sequelize({
    dialect: 'sqlite',
    storage: 'path/to/database.sqlite'
});

// Create a model
class User extends Model {}

// Initialize the model
User.init({
    // Define your model's attributes
    name: {
        type: DataTypes.STRING,
        allowNull: false
    },
    age: {
        type: DataTypes.INTEGER
    },
    favoriteColor: {
        type: DataTypes.STRING,
        defaultValue: 'green'
    }
}, {
    // Set other options here
    sequelize, // we need to pass the connection instance
    modelName: 'User' // we need to choose the model name
});

// The defined model is the class itself
console.log(User === sequelize.models.User); // true
```

La conexión a la BD `sequelize` es un singleton, es importante remarcar que cualquier configuración que se aplique sobre este instancia `sequelize` aplica para cualquier conexión a la BD que se lleve cabo.

Bibliografía

- https://betterprogramming.pub/an-introduction-to-sequelize-a-node-js-object-relational-mapper-orm-267a51c2d978