La BOGOTÁ que estamos construyendo

UNIVERSIDAD SERGIO ARBOLEDA

TODOS A LA U
Fórmate digital

MasterClass Unidad 4

Desarrollo de Software

Arquitecturas de aplicaciones web comunes

- La mayoría de aplicaciones web tradicionales se implementan como unidades únicas que se corresponden a un archivo ejecutable o una sola aplicación que se ejecuta dentro de un único dominio de aplicación de IIS.
- Este es el modelo de implementación más sencillo y sirve muy bien a muchas aplicaciones internas y públicas más pequeñas.
- Pero incluso con esta única unidad de implementación, la mayoría de las aplicaciones de negocio importantes se aprovechan de cierta separación lógica en varias capas.

¿Qué son las capas?

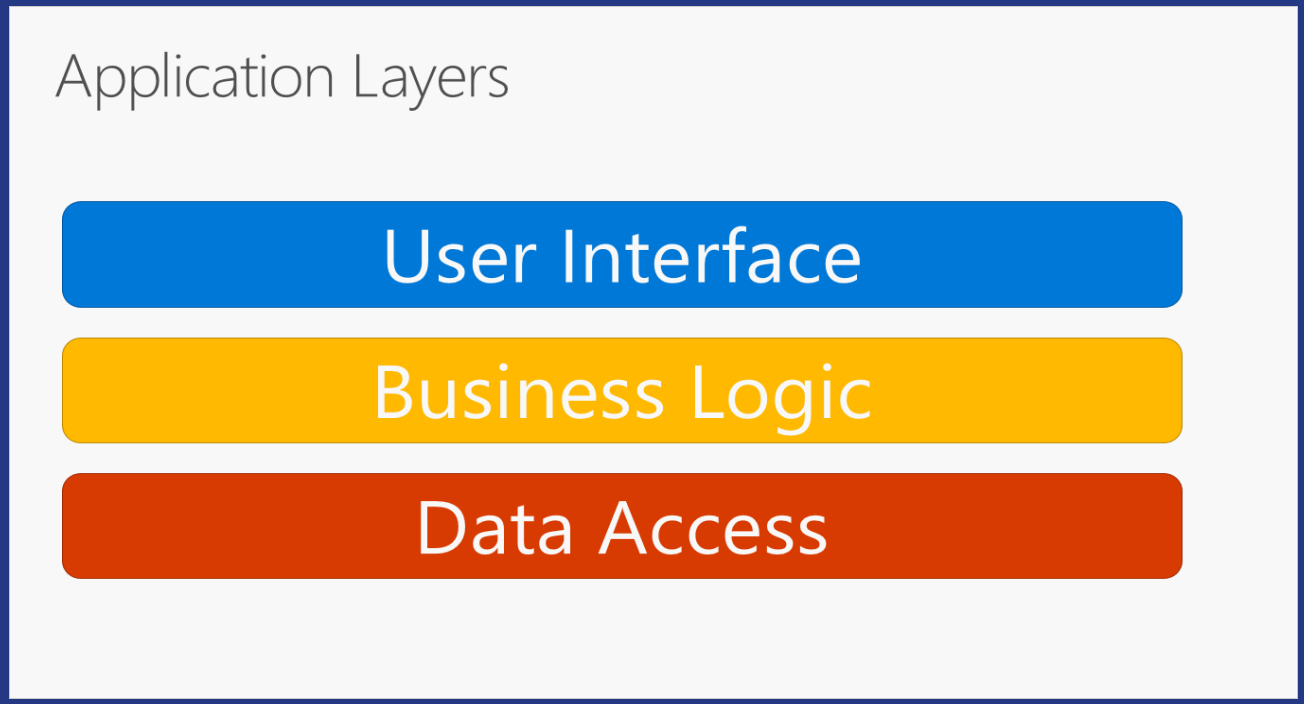
Cuando aumenta la complejidad de las aplicaciones, una manera de administrarla consiste en dividir la aplicación según sus responsabilidades o intereses. Esto sigue el principio de separación de intereses y puede ayudar a mantener organizado un código base que crece para que los desarrolladores puedan encontrar fácilmente donde se implementa una funcionalidad determinada. Pero la arquitectura en capas ofrece una serie de ventajas que van más allá de la simple organización del código.

¿Qué son las capas?

Con una arquitectura en capas, las aplicaciones pueden aplicar restricciones sobre qué capas se pueden comunicar con otras capas. Esto ayuda a lograr la encapsulación. Cuando se cambia o reemplaza una capa, solo deberían verse afectadas aquellas capas que funcionan con ella. Mediante la limitación de qué capas dependen de otras, se puede mitigar el impacto de los cambios para que un único cambio no afecte a toda la aplicación.

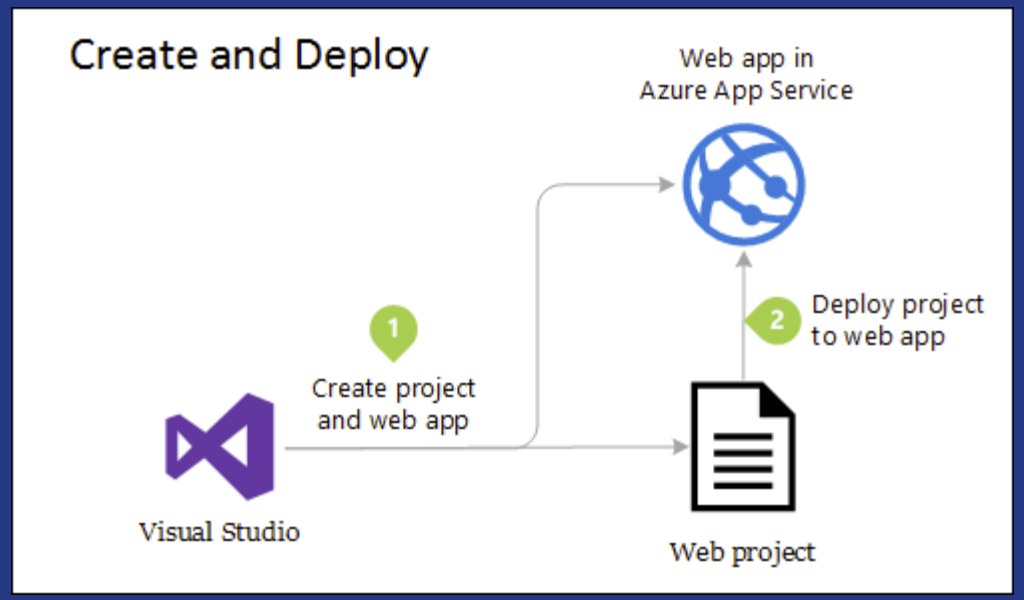
Las capas representan una separación lógica dentro de la aplicación. En caso de que la lógica de la aplicación se distribuya físicamente en servidores o procesos independientes, estos destinos de implementación físicos independientes se conocen como niveles. Es posible, y bastante habitual, tener una aplicación de N capas que se implemente en un solo nivel.

Aplicaciones tradicionales de arquitectura de "N capas"



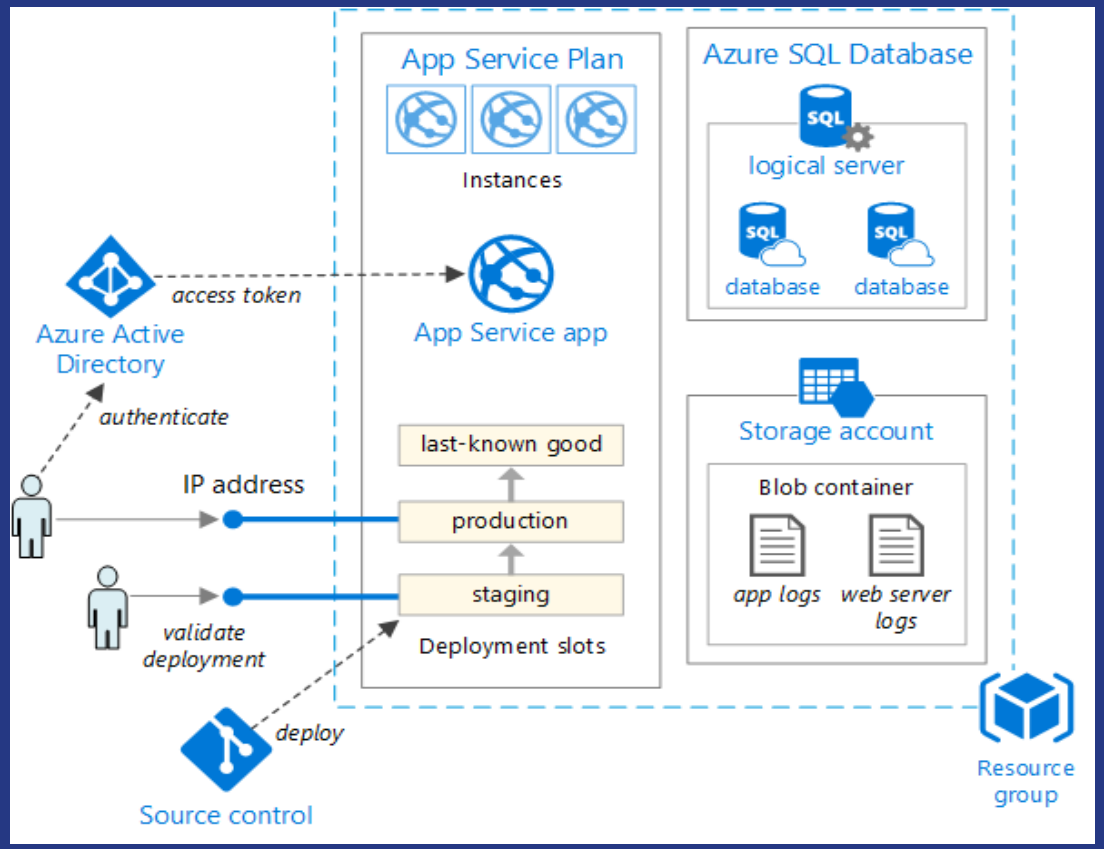
Aplicaciones tradicionales de arquitectura de "N capas"

Implementación simple de una aplicación web de Azure



Aplicaciones tradicionales de arquitectura de "N capas"

Implementación de una aplicación web en Azure App Service



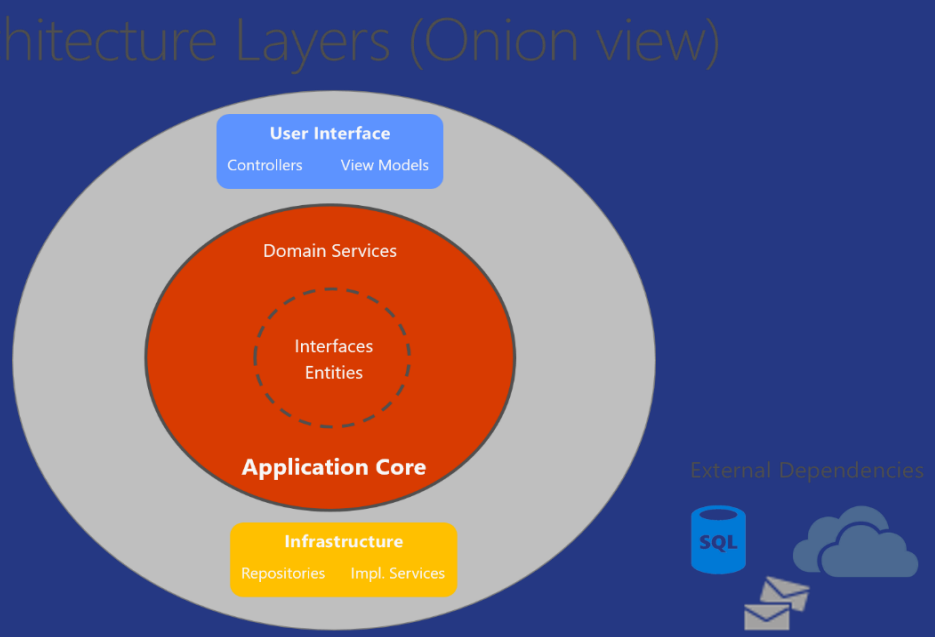
Arquitectura limpia

- Las aplicaciones que siguen el principio de inversión de dependencias, así como los principios de diseño controlado por dominios (DDD), tienden a llegar a una arquitectura similar. Esta arquitectura ha pasado por muchos nombres con los años. Uno de los primeros nombres fue Arquitectura hexagonal, seguido por Puertos y adaptadores

- El término Arquitectura limpia se puede aplicar tanto a las aplicaciones que se compilan mediante principios de DDD como a las que no. En el caso de las primeras, esta combinación se puede denominar "Arquitectura DDD limpia".

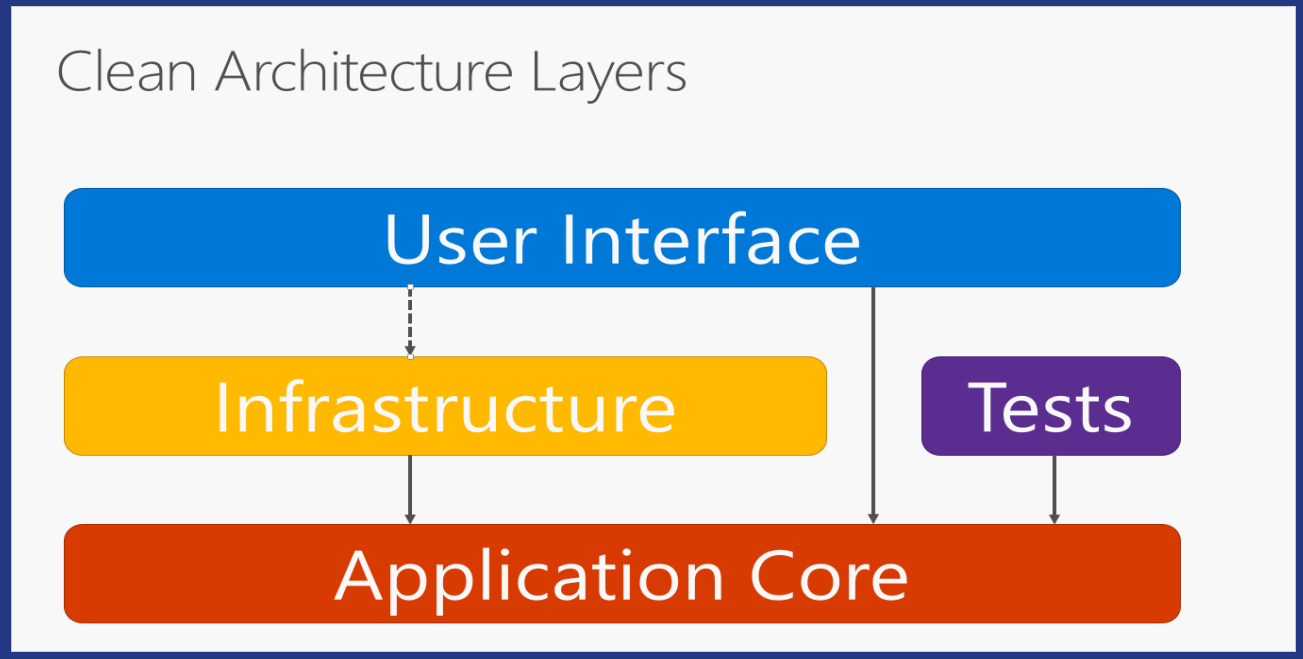
Arquitectura limpia

- La arquitectura limpia coloca el modelo de lógica de negocios y aplicación en el centro de la aplicación. En lugar de tener lógica de negocios que depende del acceso a datos o de otros aspectos de infraestructura, esta dependencia se invierte: los detalles de la infraestructura y la implementación dependen del núcleo de la aplicación



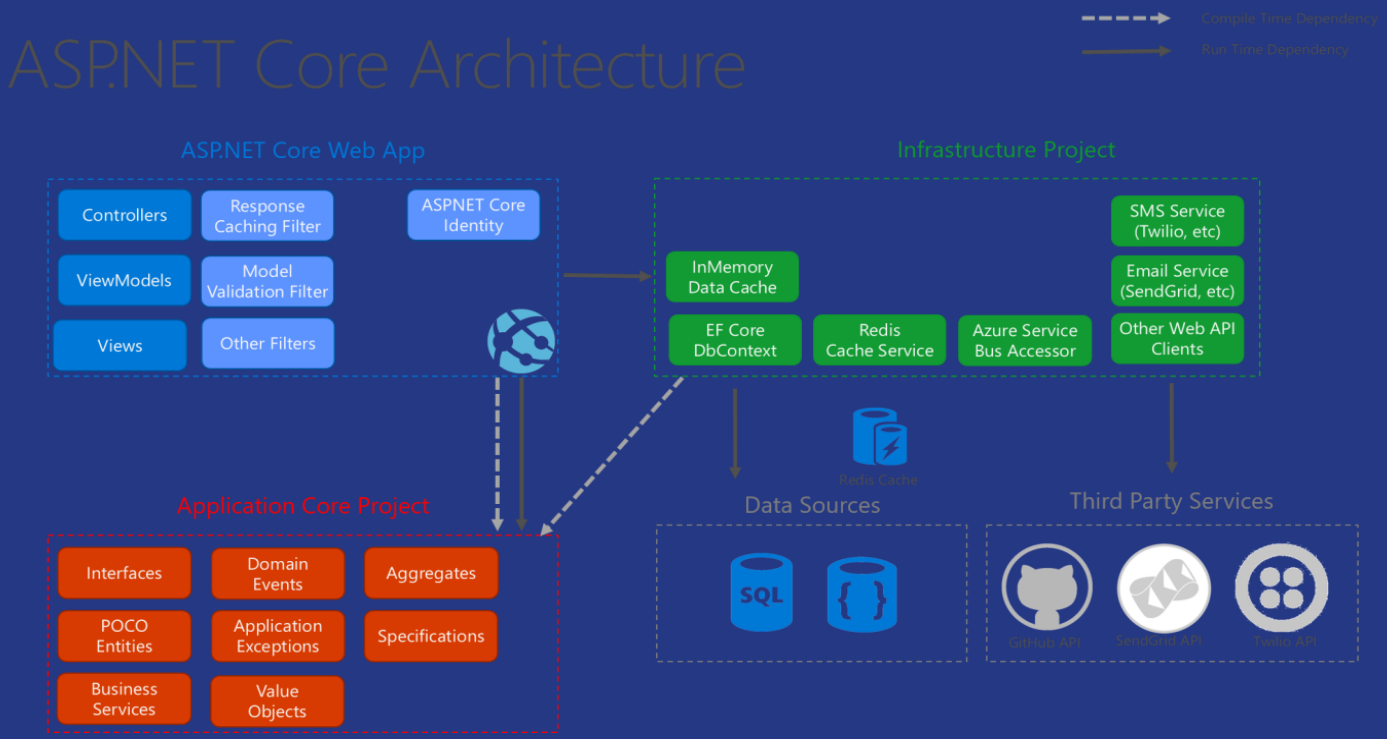
Arquitectura limpia

Arquitectura limpia; vista de capas horizontal



Arquitectura limpia

Diagrama de arquitectura de ASP.NET Core en el que se sigue la arquitectura limpia.



Organización del código en la arquitectura limpia

Tipos de núcleo de la aplicación

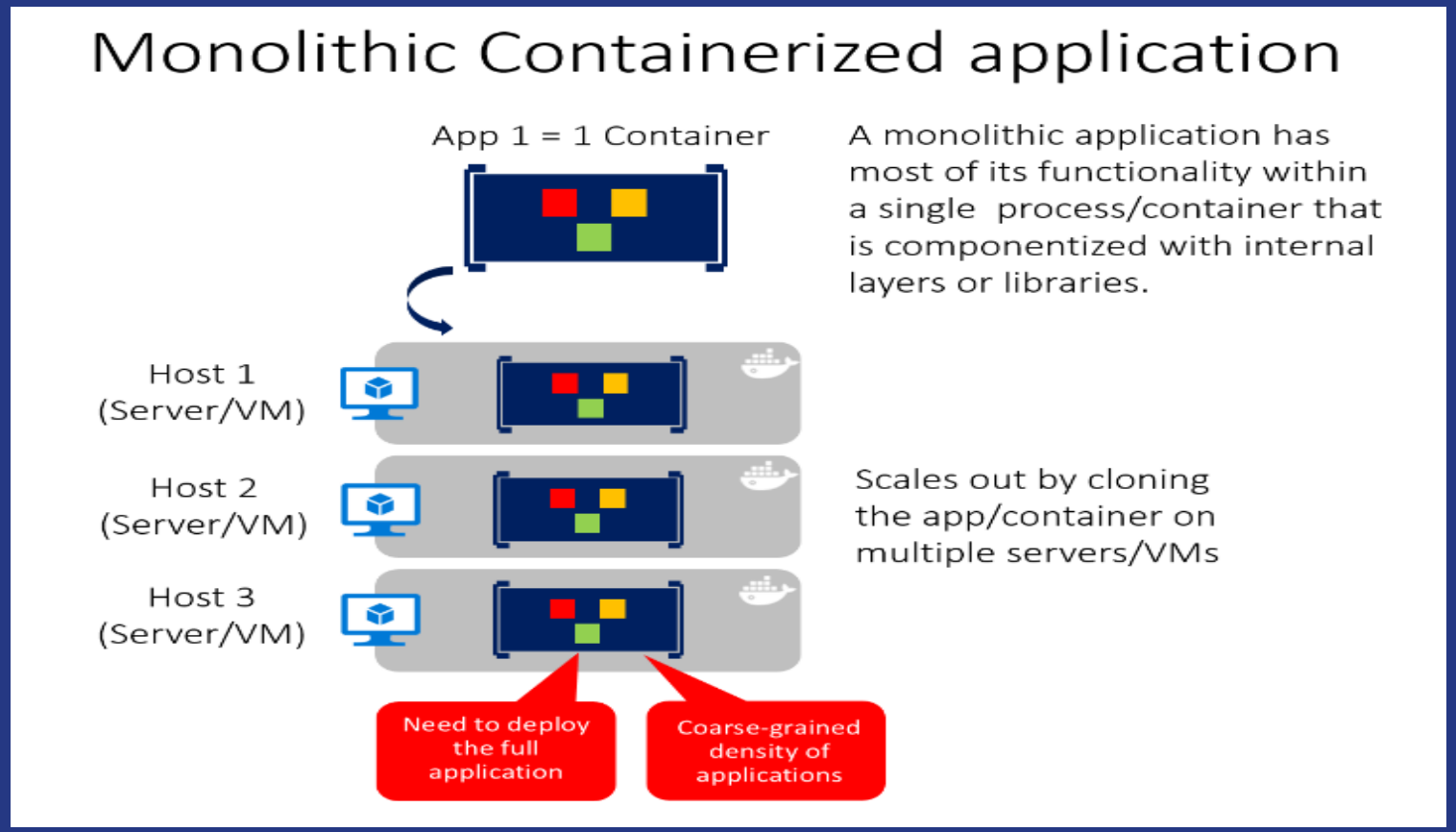
- Entidades (las clases de modelo de negocio que se conservan)
- Interfaces
- Servicios
- DTO

Organización del código en la arquitectura limpia

Tipos de capa de interfaz de usuario

- Controladores
- Filtros
- Vistas
- ViewModels
- Inicio

Aplicaciones monolíticas y contenedores

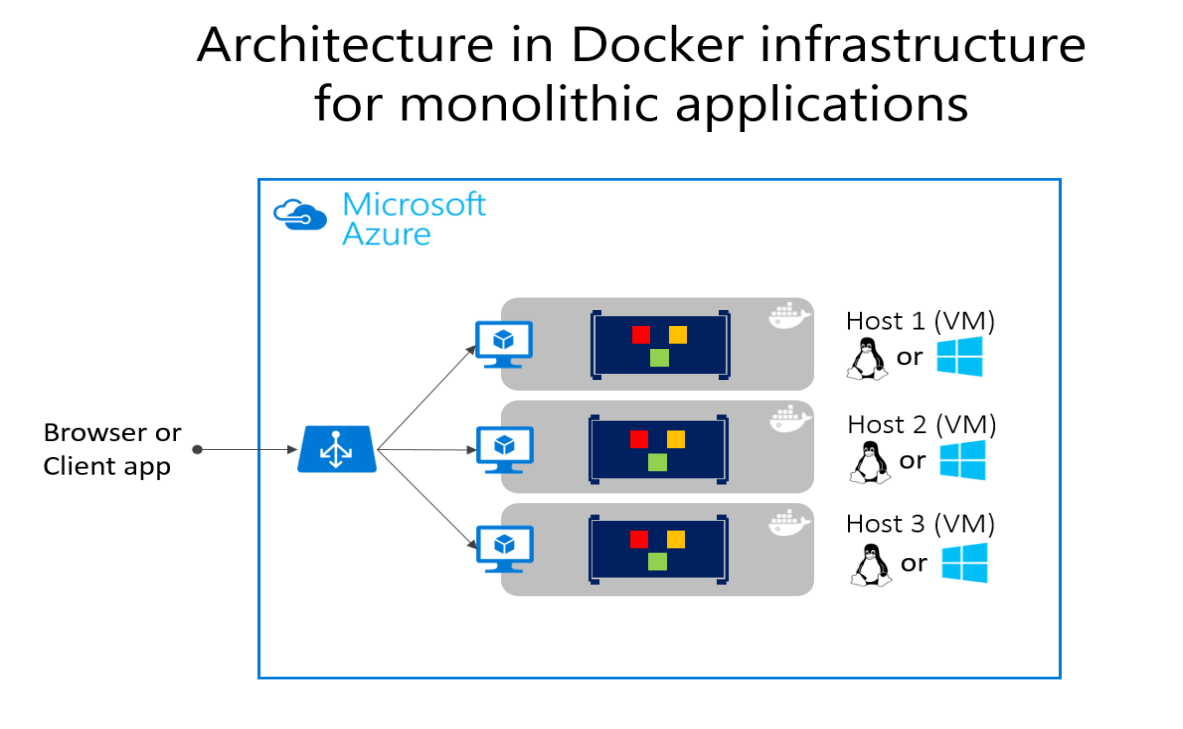


Aplicaciones monolíticas y contenedores

El enfoque monolítico es habitual y muchas organizaciones realizan el desarrollo con este enfoque de diseño. Muchas obtienen resultados bastante positivos, mientras que otras alcanzan los límites. Muchas diseñaron sus aplicaciones con este modelo, ya que crear arquitecturas orientadas a servicios (SOA) con infraestructura y herramientas resultaba demasiado difícil, y no vieron la necesidad hasta que la aplicación creció. Si comprueba que está alcanzando los límites del enfoque monolítico, dividir la aplicación para que pueda aprovechar mejor los contenedores y microservicios puede ser el siguiente paso lógico.

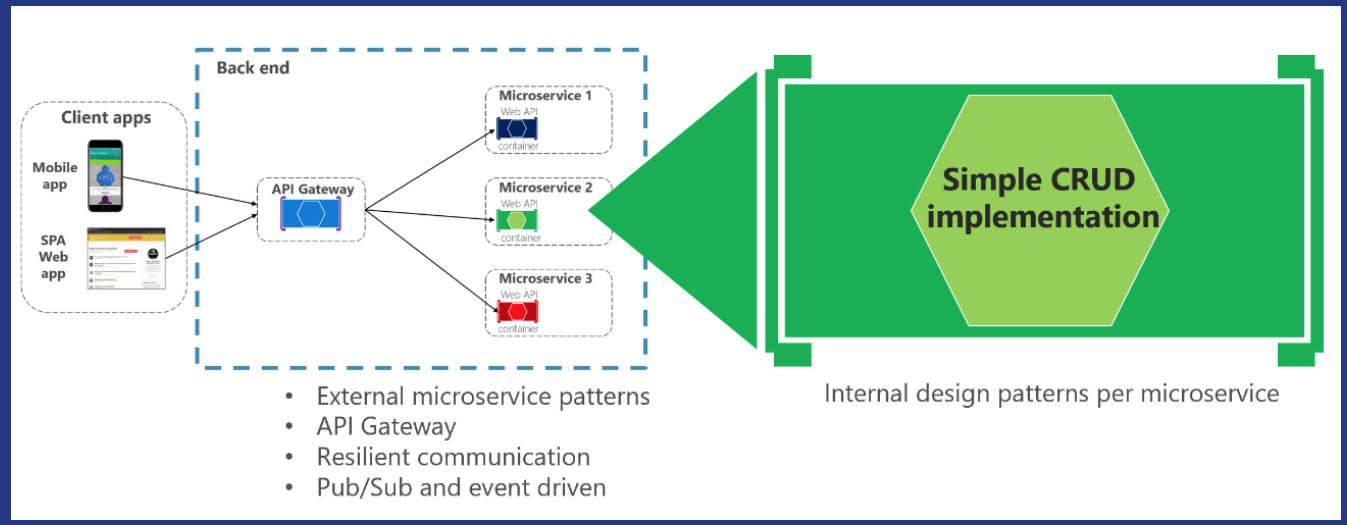
Aplicaciones monolíticas y contenedores

La implementación en los distintos hosts se puede administrar con técnicas de implementación tradicionales. Los hosts de Docker se pueden administrar con comandos como docker run ejecutados manualmente o a través de la automatización como canalizaciones de entrega continua (CD).



Creación de un microservicio CRUD sencillo controlado por datos

Un ejemplo de este tipo de servicio sencillo controlado por datos es el microservicio de catálogo de la aplicación de ejemplo eShopOnContainers. Este tipo de servicio implementa toda su funcionalidad en un solo proyecto de API Web de ASP.NET Core que incluye las clases para su modelo de datos, su lógica de negocios y su código de acceso a datos



Implementación de servicios API Web de CRUD con Entity Framework Core

- Entity Framework (EF) Core es una versión ligera, extensible y multiplataforma de la popular tecnología de acceso a datos Entity Framework. EF Core es un asignador relacional de objetos (ORM) que permite a los desarrolladores de .NET trabajar con una base de datos mediante objetos .NET.

- El microservicio de catálogo usa EF y el proveedor de SQL Server porque su base de datos se está ejecutando en un contenedor con la imagen de SQL Server para Linux Docker. Pero la base de datos podría implementarse en cualquier SQL Server, como en una base de datos SQL de Azure o Windows local. Lo único que debe cambiar es la cadena de conexión en el microservicio ASP.NET Web API.

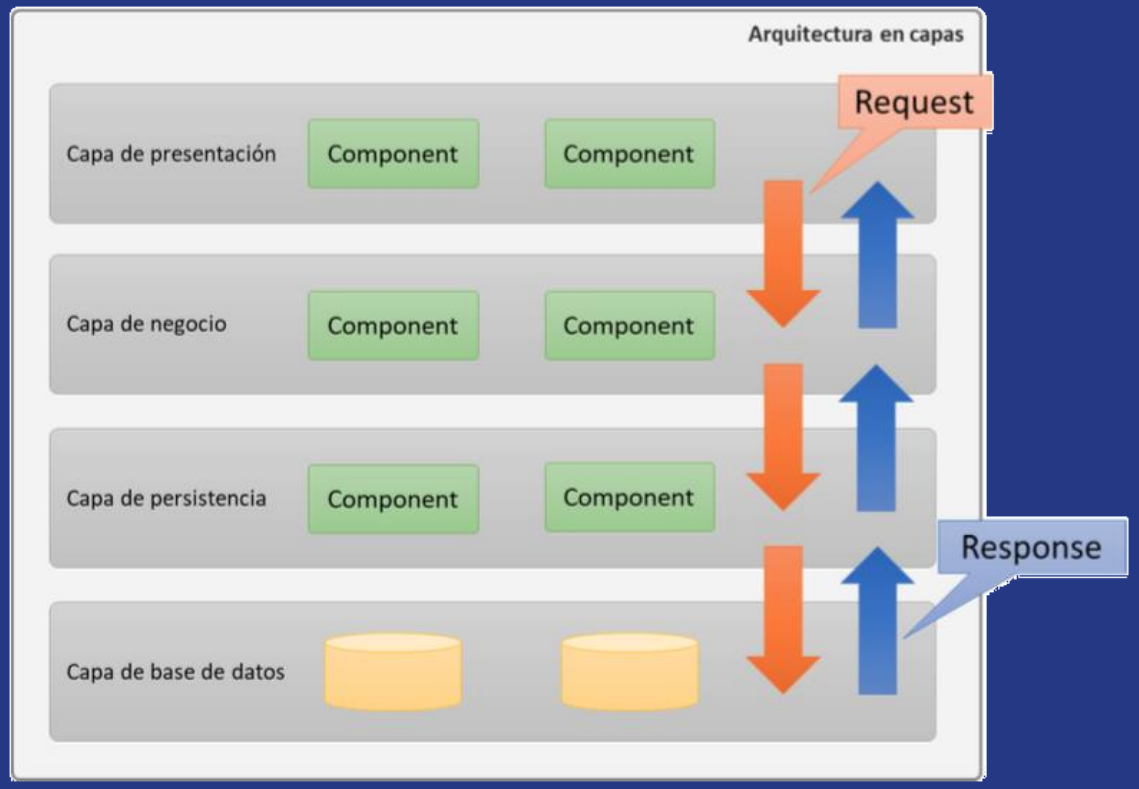
Implementación de servicios API Web de CRUD con Entity Framework Core

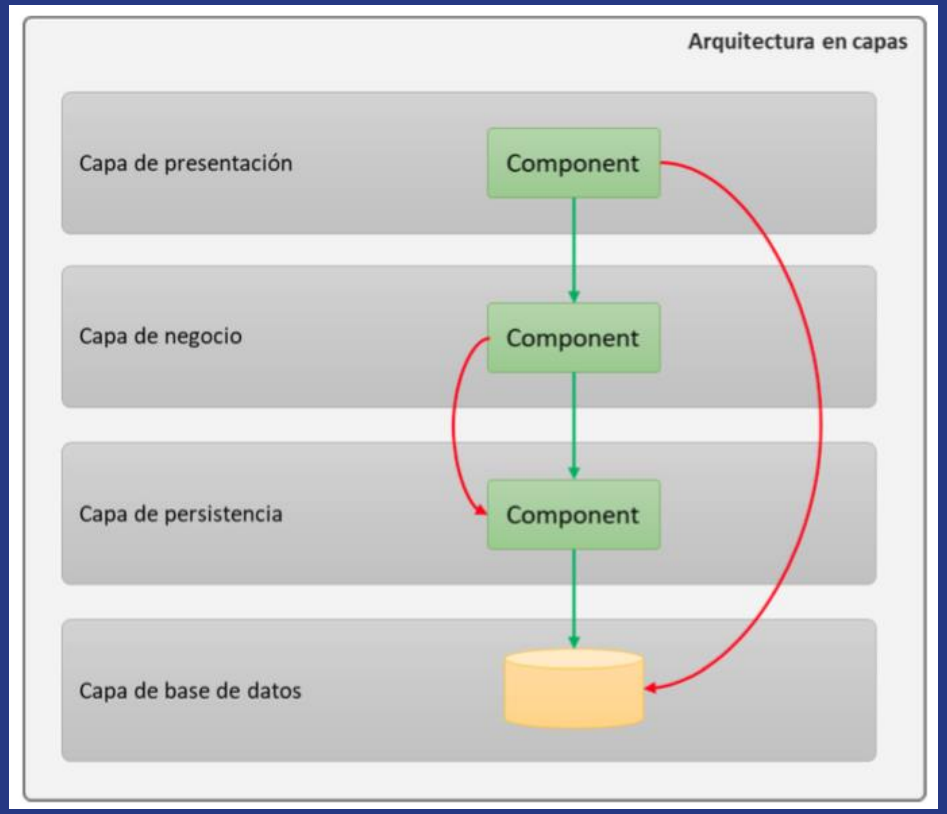
El modelo de datos

- Con EF Core, el acceso a datos se realiza utilizando un modelo. Un modelo se compone de clases de entidad (modelo de dominio) y un contexto derivado (DbContext) que representa una sesión con la base de datos, lo que permite consultar y guardar los datos.

- Puede generar un modelo a partir de una base de datos existente, codificar manualmente un modelo para que coincida con la base de datos, o bien usar técnicas de migración de EF para crear una base de datos a partir del modelo, mediante el enfoque Code First, que facilita que la base de datos evolucione a medida que el modelo cambia en el tiempo.

Arquitectura basada en capas





Arquitectura basada en capas

La capa de datos es la responsable de:

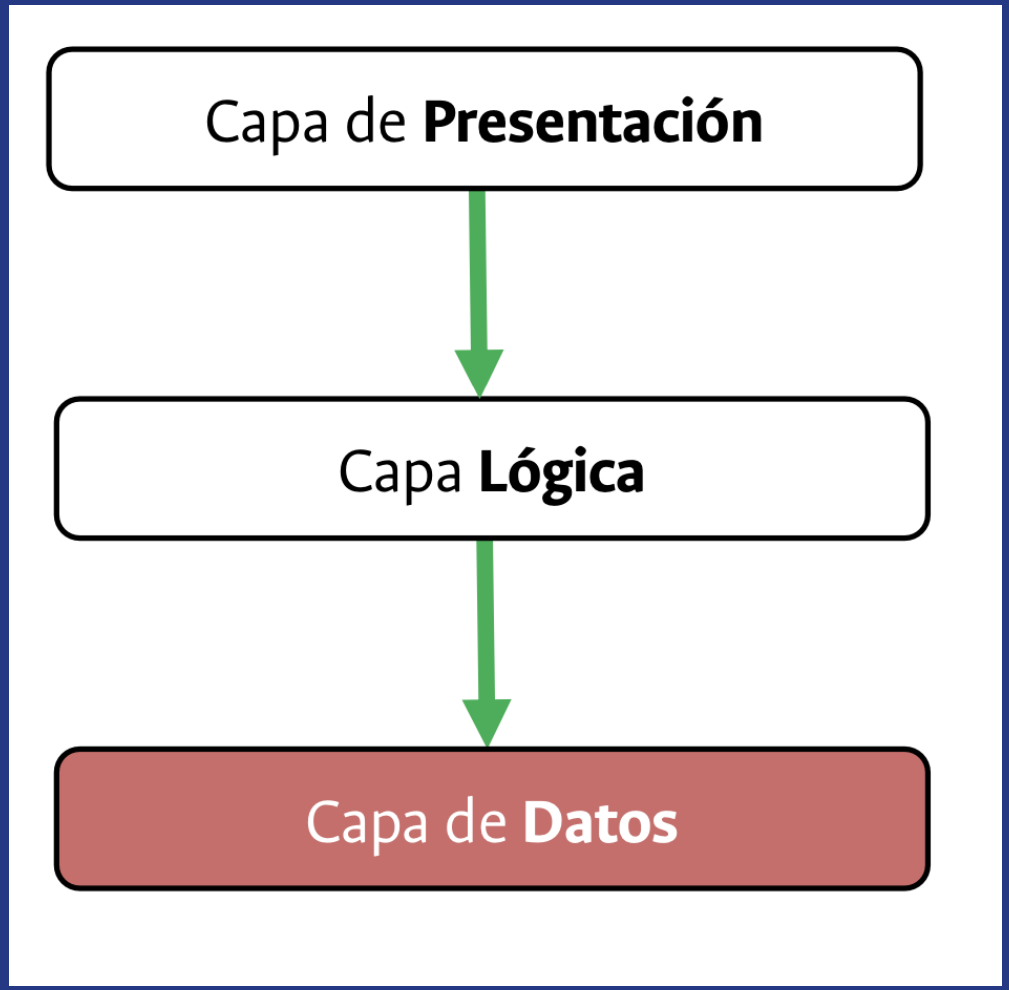
- Almacenar y gestionar los datos
- pertenecientes al sistema de software.
- Proporcionar los mecanismos de acceso necesarios para que la capa lógica pueda usar los datos.

La capa lógica es la responsable de:

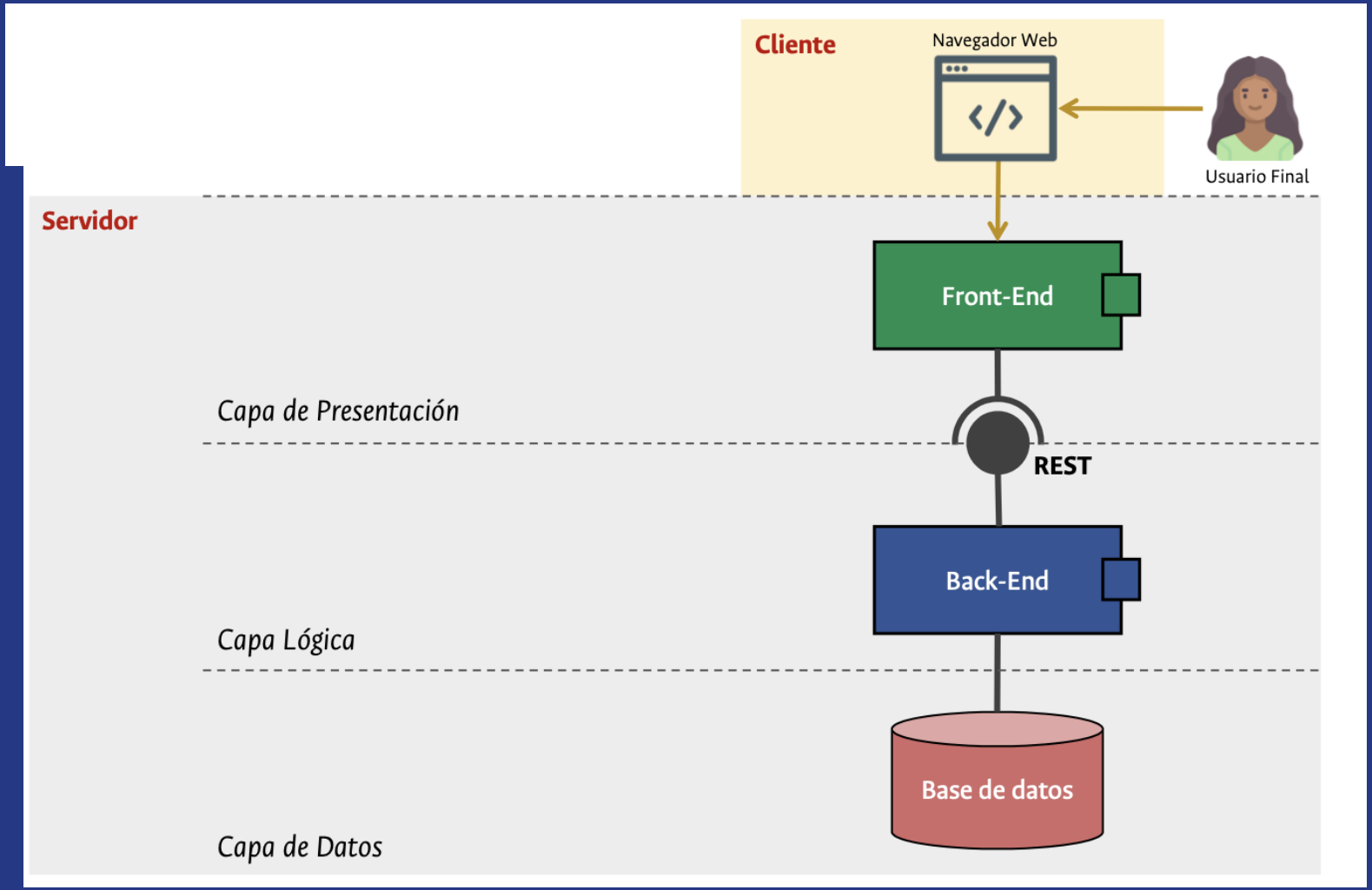
- Definir los algoritmos de lógica de negocio que comprenderán el sistema. (funcionalidades)
- Proporcionar los mecanismos de acceso necesarios para que la capa de presentación pueda usar la lógica de negocio.

La capa de presentación es la responsable de:

- Proveer los elementos necesarios para permitir la interacción efectiva entre el uusario final y el sistema.

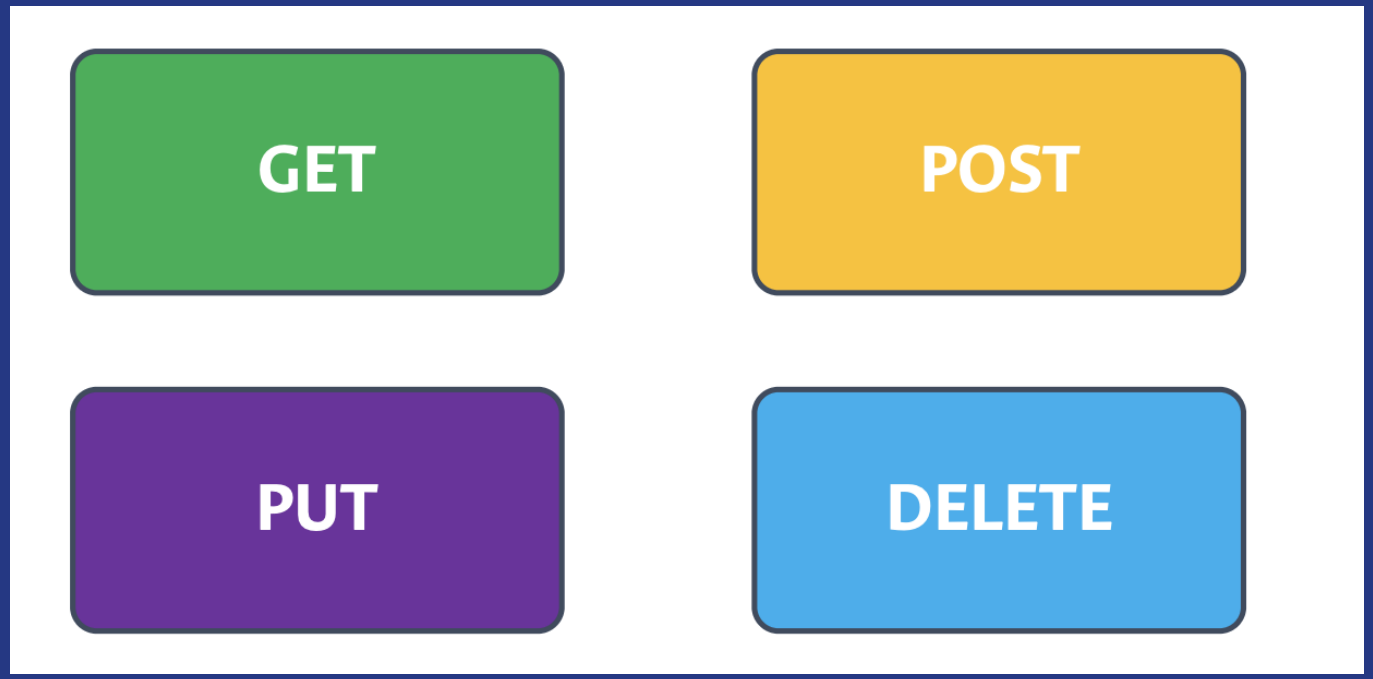


Arquitectura basada en capas



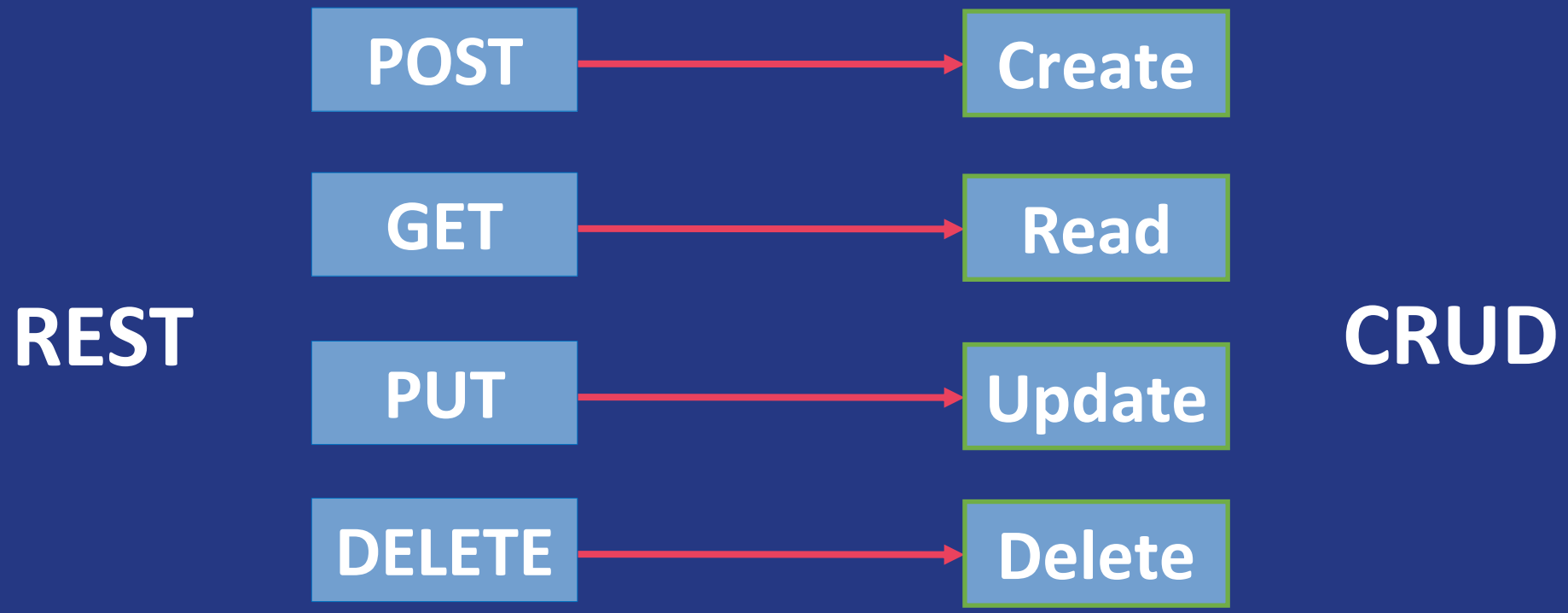
API - REST

Es una interfaz de aplicación que expone un servicio web, por medio de un conjunto de operaciones HTTP y siguiendo el estilo arquitec- tónico REST.
Las operaciones más comunes son:



API - REST

Podemos realizar un símil con las operaciones CRUD



Estructura de persistencia

- Nuestro CrudRepository para una entidad llamada Reservation se vería así inicialmente:

```
public interface ReservationCrudRepository extends CrudRepository<Reservation,Integer> {


}
```

Estructura de persistencia

- El CRUDRepository será utilizado por una clase Repository. Esta clase es la encargada de entenderse con la base de datos para ejecutar las operaciones de consulta, escritura y borrado.
- El Repository debe ser el único lugar en la aplicación que se comunica con la base de datos.
- Esta clase la decoramos con @Repository, de esa manera Spring la puede gestionar como un componente.
- En esta clase estableceremos todos los métodos necesarios para satisfacer las acciones de la base de datos.

Estructura de persistencia

- La Clase repository se vería así:
  - La anotación Autowired delega en Spring la inicialización del objeto ReservationCrudRepository

```
@Repository
public class ReservationRepository {
    @Autowired
    private ReservationCrudRepository reservationCrudRepository;

    public List<Reservation> getAll() { return (List<Reservation>) reservationCrudRepository.findAll(); }
    public Optional<Reservation> getReservation(int id) { return reservationCrudRepository.findById(id); }
    public Reservation save(Reservation reservation) { return reservationCrudRepository.save(reservation); }
    public void delete(Reservation reservation) { reservationCrudRepository.delete(reservation); }
```

Conexión con Servicio

Las clases @Service que se trataron previamente, son las encargadas de utilizar este repositorio como parte de su lógica.

Para el siguiente ejemplo, la lógica restringirá la creación de Reservas cuando en la petición se incluya el id (debido a que este es autogenerado), así que cuando lo incluya, se asegurará de evitar la colisión.

Si no se envía el id, se guardará directamente.
Si se envía el id, se validará que no se haya insertado antes un registro con ese id.
Si no hay otro registro con ese id, se guardará.

Conexión con Servicio

```java
@Service
public class ReservationService {
    @Autowired
    private ReservationRepository reservationRepository;

    public List<Reservation> getAll() { return reservationRepository.getAll(); }

    public Optional<Reservation> getReservation(int reservationId) {
        return reservationRepository.getReservation(reservationId);
    }

    public Reservation save(Reservation reservation){
        if(reservation.getIdReservation()==null){
            return reservationRepository.save(reservation);
        }else{
            Optional<Reservation> e= reservationRepository.getReservation(reservation.getIdReservation());
            if(e.isEmpty()){
                return reservationRepository.save(reservation);
            }else{
                return reservation;
            }
        }
    }
```

Controlador

- El controlador, es el encargado de gestionar las peticiones.
- Será la puerta de entrada a las peticiones hechas.
- Define la estructura de URLs, ya que a cada Controlador se le asigna una URL.
- Los métodos de la clase han de representar las diferentes peticiones, el tipo de petición (método http) y la url.

- Las peticiones a los métodos se alcanzan luego del llamado al controlador.

Controlador

```java
@RestController
@RequestMapping("/Reservation")
public class ReservationController {
    @Autowired
    private ReservationService reservationService;
    @GetMapping("/all")
    public List<Reservation> getReservations() { return reservationService.getAll(); }

    @GetMapping("/{id}")
    public Optional<Reservation> getReservation(@PathVariable("id") int reservationId) {
        return reservationService.getReservation(reservationId);
    }
    @PostMapping("/save")
    public Reservation save(@RequestBody Reservation reservation) { return reservationService.save(reservation); }
    @PutMapping("/update")
    public Reservation update(@RequestBody Reservation reservation) { return reservationService.update(reservation); }
    @DeleteMapping("/{id}")
    public boolean delete(@PathVariable("id") int reservationId) {
        return reservationService.deleteReservation(reservationId);
    }
```

Controlador

- En las URL es posible ver {id} lo que significa que es una variable, la cual ingresa en la url.
- En este controlador si quiero ver toda la lista, tendré que apuntar a:
    - URL_BASE/Reservation/all
- En este controlador si quiero ver el detalle del elemento con id=1 se consultará a través de la url:
    - URL_BASE/Reservation/1

Controlador

- Para guardar una entidad, se debe invocar la URL /Reservation/save.
  - ¡Tener en cuenta que la petición debe ser de tipo POST!
  - La información en este caso estará en el cuerpo de la petición.

- La anotación indica el método http que se debe utilizar.
- La actualización será de tipo PUT y apunta a /Reservation/update
- La eliminación será de tipo DELETE y apunta a /Reservation/delete/XX
- El borrado debe incluir el id al finalizar la url. (reemplazar XX por id)

Ejemplo: Solicitud de reservas entre fechas

- Entregar el conjunto de Reservas hechas en el intervalo de dos fechas.

Ejemplo: Solicitud de reservas entre fechas

- Para lograrlo, crearemos un método GET en el controlador de reservas que reciba las dos fechas.
- Luego crearemos un Servicio que convierta las fechas, valide que la inicial es previa a la final (restricción de lógica de negocio)
- El servicio invocará la consulta al repositorio.
- El repositorio hará uso de un JPA Query Method para traer los resultados.

Ejemplo: Solicitud de reservas entre fechas

- El controlador tendrá ahora el siguiente método

```
@GetMapping("/report-dates/{dateOne}/{dateTwo}")
public List<Reservation> getReservationsReportDates(
            @PathVariable("dateOne") String dateOne,@PathVariable("dateTwo") String dateTwo){
   return reservationService.getReservationsPeriod(dateOne,dateTwo);
}
```

- Los valores entre llaves { } son relacionados en los argumentos del método con @PathVariable

Ejemplo: Solicitud de reservas entre fechas

- El servicio tendrá ahora el siguiente método

```
public List<Reservation> getReservationsPeriod(String dateA, String dateB){
    SimpleDateFormat parser=new SimpleDateFormat( pattern: "yyyy-MM-dd");
    Date a= new Date();
    Date b=new Date();
    try {
        a = parser.parse(dateA);
        b = parser.parse(dateB);
    } catch (ParseException e) {
        e.printStackTrace();
    }
    if(a.before(b)){
        return reservationRepository.getReservationPeriod(a,b);
    }else{
        return new ArrayList<>();
    }

}
```

Ejemplo: Solicitud de reservas entre fechas

- El servicio toma los String con las fechas, los covierte en objetos tipo Date.
- Valida que las fechas sean una anterior a otra y luego invoca el repositorio.
- El repositorio devuelve una lista de Reservas.

Ejemplo: Solicitud de reservas entre fechas

- El Repositorio tendrá ahora el siguiente método

```
public List<Reservation> getReservationPeriod(Date a, Date b){
    return reservationCrudRepository.findAllByStartDateAfterAndStartDateBefore(a,b);
}
```

- El repositorio finalmente llama a CRUD Repository en el que hemos incluido el Query Method.

Ejemplo: Solicitud de reservas entre fechas

- El CRUDRepository tiene el siguiente Query Method

```
public List<Reservation> findAllByStartDateAfterAndStartDateBefore(Date dateOne,Date dateTwo );
```

- El Query Method, de manera camelizada me permite decirle a la base de datos: encontrar todos (¡las reservas!) por (criterio de búsqueda) StartDate (hay un atributo de Reservation que se llama startDate).
- El criterio es una operación lógica: startDate after AND startDate before. Lo que hace es comparar que startDate sea posterior al primer argumento y anterior al segundo argumento.