Deacuerdo al libro ***Laravel Design Patterns and Best Practices***, los patrones de diseño en Laravel son:

1. Builder
2. Factory
3. Repository
4. Strategy
5. Provider
6. Facade

BUILDER

Este patrón de diseño pretende obtener objetos más simples y reutilizables. Su objetivo es separar las capas de construcción de objetos más grandes y complicadas del resto para que las capas separadas puedan ser usadas en diferentes capas de la aplicación.

En Laravel, la clase AuthManager necesita crear algunos elementos seguros para reutilizarlos con determinados controladores de almacenamiento como cookies, sesión o elementos personalizados. Para lograr esto, la clase AuthManager necesita usar funciones de almacenamiento como callCustomCreator () y getDrivers () de la claseManager.

FACTORY

El patrón de fábrica se basa en la creación de objetos de método de plantilla, que se basa en la definición del algoritmo de una clase en una subclase para implementar un algoritmo. Hay una subclase, que se deriva de una gran superclase, en este patrón. La clase principal, que podemos llamar superclase, sólo tiene lógica mayor y genérica; las subclases se derivan de esta superclase. Como resultado, puede haber más de una subclase heredada de esta superclase, que se dirigen a diferentes propósitos. Es beneficioso si la clase o sus componentes suelen cambiar, o si los métodos necesitan ser sustituidos, muy similar a la inicialización.

Laravel envía varios tipos de reglas de validación con la clase Validación. Cuando se desarrolla aplicaciones, normalmente se necesita validar los datos a medida que avanzamos. Para hacer esto, un enfoque común es establecer las reglas de validación en el Modelo y llamarlas desde el Controlador. Por "reglas" aquí nos referimos tanto al tipo de validación como a su alcance.

A veces, se necesita establecer reglas personalizadas y mensajes de error personalizados para validar los datos.

REPOSITORY

Se utiliza generalmente para crear una interfaz entre dos capas distintas de una aplicación. Los desarrolladores de Laravel usan este patrón para crear una capa abstracta entre NamespaceItemResolver (la clase que resuelve los namespaces y entiende qué archivo es en qué espacio de nombres) y Loader (una clase que requiere y carga otra clase en la aplicación). La clase Loader simplemente carga el grupo de configuración del espacio de nombres dado. Como es sabido, casi todo el código de Laravel Framework se desarrolla usando namespaces.

Supongamos que se intenta obtener un producto de una base de datos usando ORM Eloquent. El método será algo parecido a Producto::find(1) en el controlador. Para propósitos de abstracción, este enfoque no es ideal. Si ahora se pone un código como este, el controlador sabe que se está usando Eloquent, lo que idealmente no debería ocurrir en una estructura buena y abstracta. Si desea contener los cambios realizados en el esquema de la base de datos para que las llamadas fuera de la clase no se refieran a los campos directamente sino a través de un repositorio, tiene que buscar todos los códigos uno a uno.

STRATEGY

En este patrón de diseño, la lógica se extrae de clases complejas en componentes más fáciles para que puedan ser reemplazados fácilmente con métodos más simples. Por ejemplo, usted desea mostrar las entradas de blog populares en su sitio web. En un enfoque clásico, usted calculará la popularidad, hará la paginación, y listará los ítems relativos al offset de paginación actual y popularidad, y hará todos los cálculos en una clase simple. Este patrón tiene como objetivo separar cada algoritmo en componentes separados para que puedan ser reutilizados o combinados en otras partes de la aplicación fácilmente. Este enfoque también aporta flexibilidad y facilita el cambio de un sistema de algoritmos.

PROVIDER

Es un nivel medio entre una clase API y la capa de lógica empresarial/abstracción de datos. El proveedor es la implementación de la API separada de la propia API.

En Laravel, ambas clases AuthServiceProvider y HashServiceProvider extienden de ServiceProvider. La clase AuthServiceProvider proporciona todos los servicios a AuthManager cuando se envía una solicitud de autenticación, como comprobar si se ha creado una cookie y una sesión o si el contenido es inválido. Una vez solicitado el servicio de autenticación, el desarrollador puede definir si una sesión o cookie se configura a través de la respuesta de AuthDriver.

Sin embargo, HashServiceProvider proporciona los métodos relacionados cuando se hace una petición de hash segura para poder usar, buscar, revisar o hacer otras cosas con estos hash. Ambos proveedores devuelven los valores como array.

FACADE

Permite al desarrollador unir varias interfaces complicadas en una única interfaz de clase. Este patrón también permite envolver varios métodos de varias clases en una sola estructura.

En Laravel, casi todos los métodos se parecen a un método estático, por ejemplo, Input:: has(), Config:: get(), URL:: route(), View:: make(), y HTML:: style(). Sin embargo, no son métodos estáticos. Si fueran métodos estáticos, sería bastante difícil hacer pruebas para todos ellos. En realidad, son las imitaciones de este comportamiento. En segundo plano, con la ayuda del Contenedor de IoC (una forma de inyectar dependencias en una clase), Laravel en realidad llama a otra clase(s) a través de una clase de Facade. La clase base Facade se beneficia del propio método __callStatic() de PHP para llamar a los métodos requeridos, tales como métodos estáticos.