Aplicaciones Web de Servidor

ARQUITECTURA Y DISEÑO: PATRÓN MVC

El patrón Modelo-Vista-Controlador

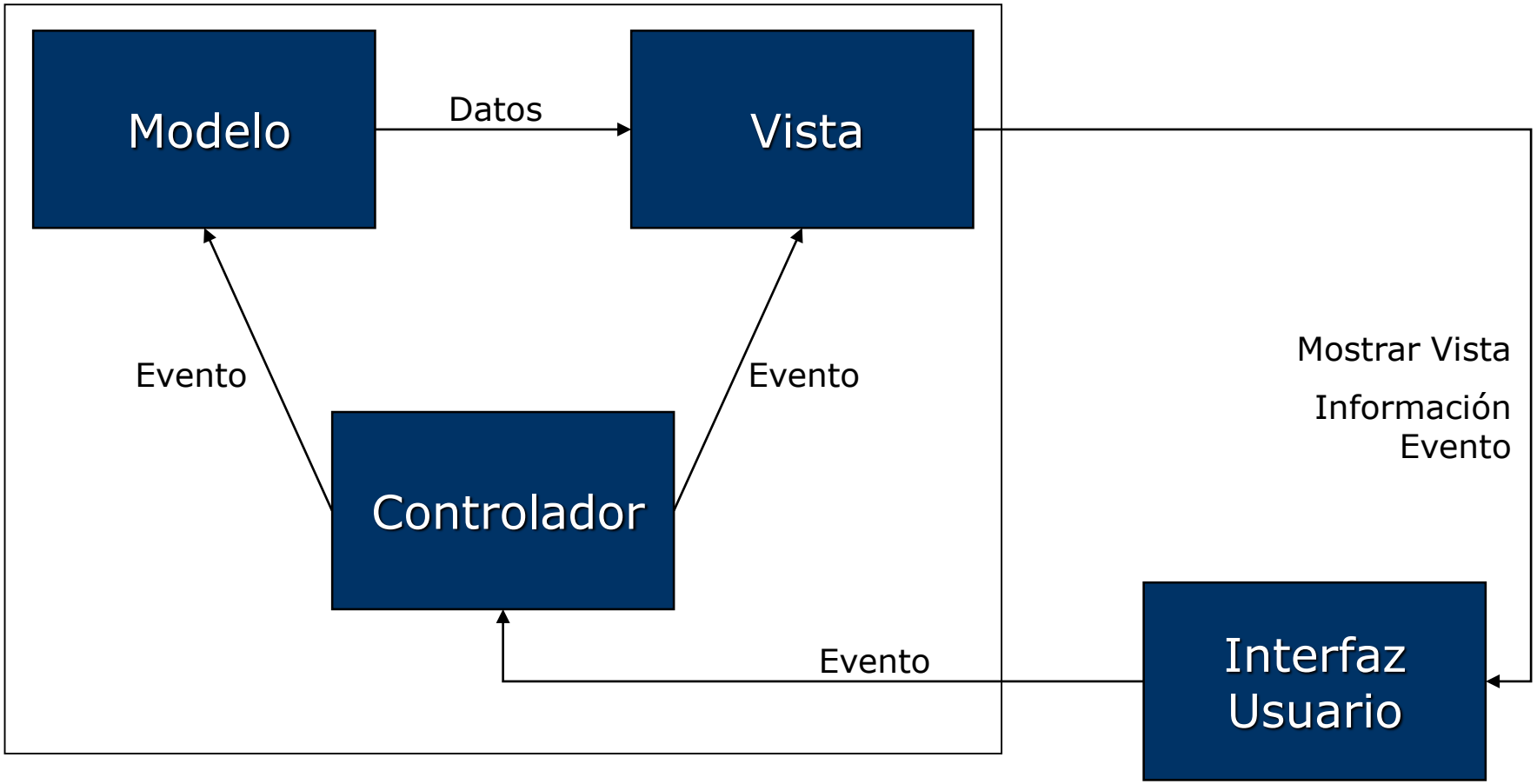
Arquitectura y diseño: Patrón MVC

El patrón Modelo-Vista-Controlador se originó en la comunidad Smalltalk para implementar interfaces de usuario en los que las responsabilidades están bien distribuidas entre distintas partes (componentes) del diseño.

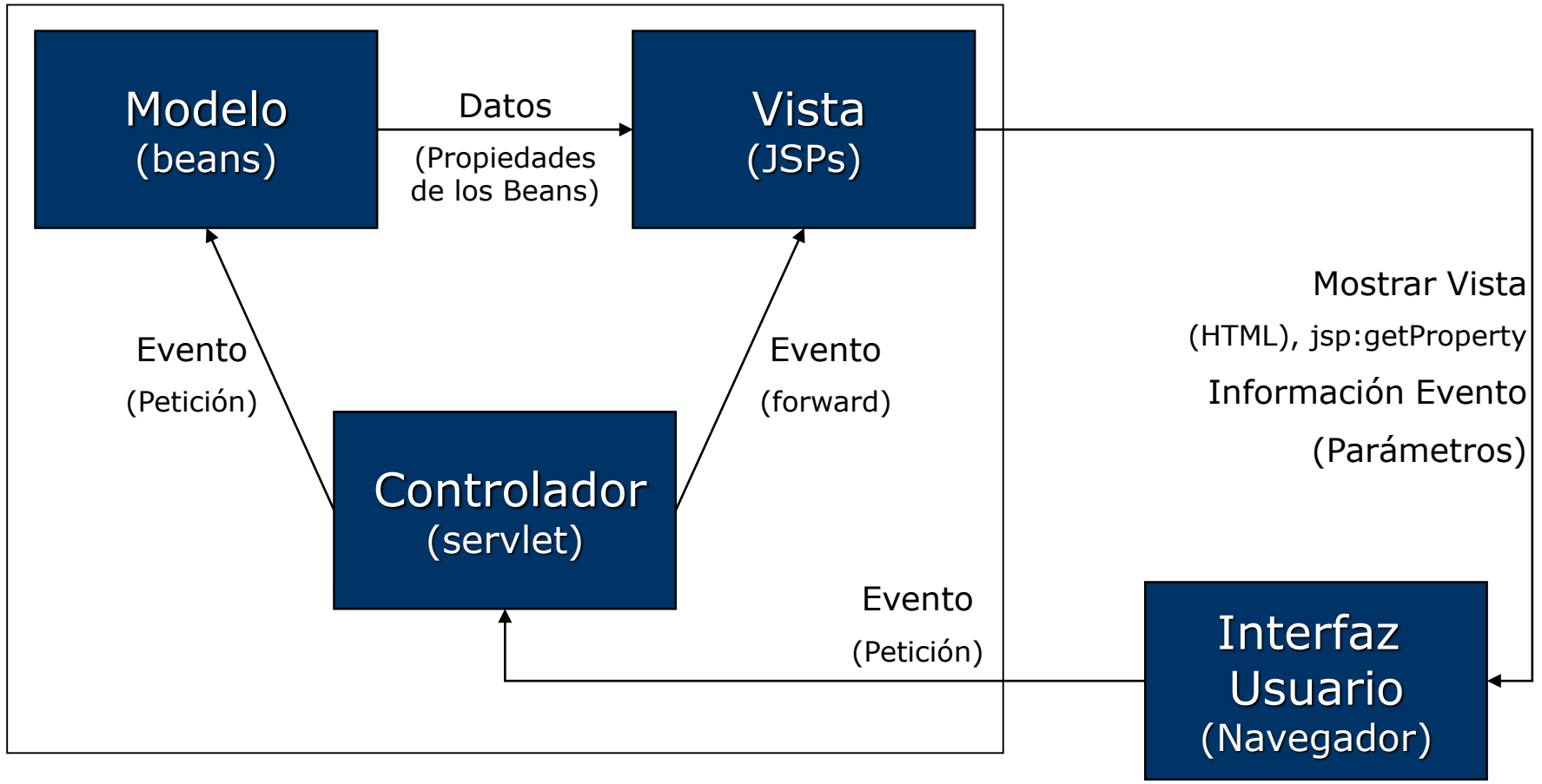
Así, se decidió, distinguir tres responsabilidades distintas:

- Lógica de negocio ➔ Modelo.
- Gestión de eventos de usuario ➔ Controlador.
- Presentación ➔ Vista.

Arquitectura y diseño: Patrón MVC



Arquitectura y diseño: Patrón MVC - Tecnologías Java



Arquitectura y diseño: Patrón MVC - El modelo

El modelo representa la lógica de negocio de la aplicación.

Encapsular el modelo de una aplicación en componentes facilita la depuración, mejora la calidad y favorece la reutilización de código.

Puede dividirse en dos tipos de componentes:

- De estado.
- De acción.

Arquitectura y diseño: Patrón MVC - El modelo

Los componentes de estado encapsulan el estado de la aplicación y exponen métodos para el acceso y cambio de éste.

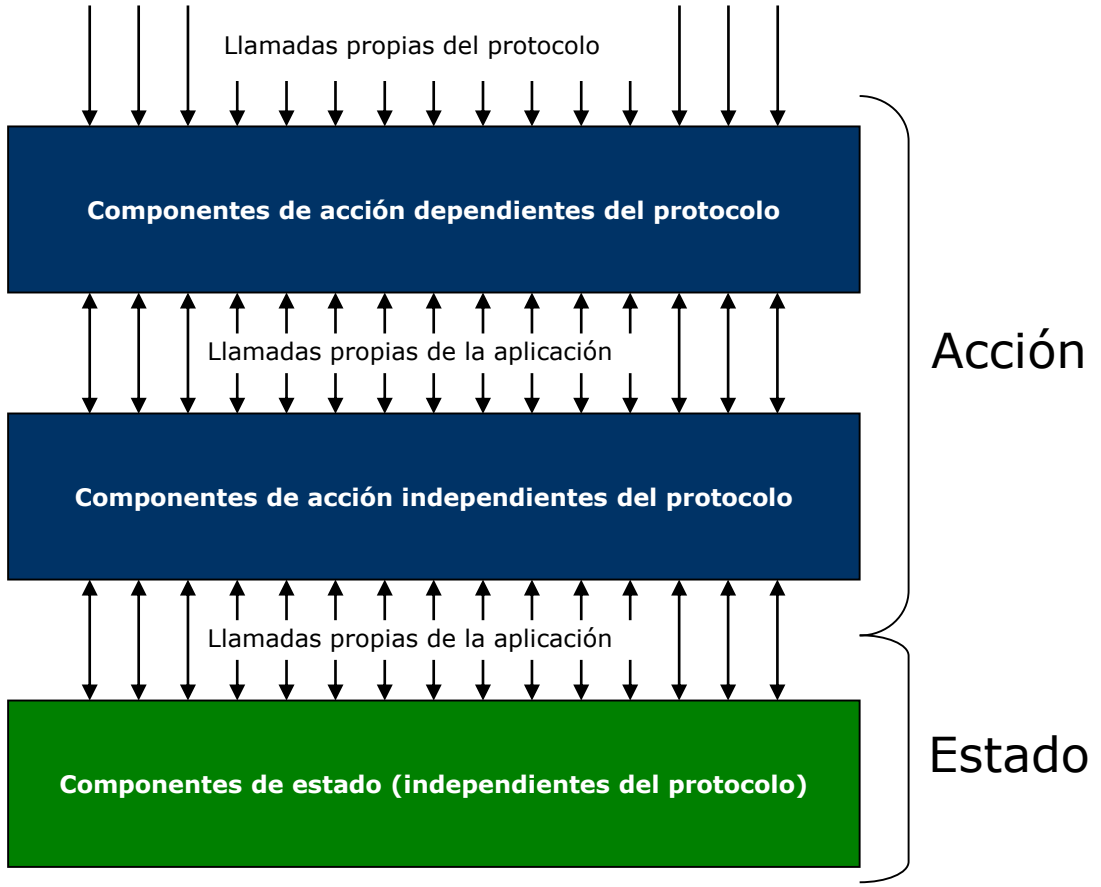
Al estar una capa por debajo de la capa de acción, los componentes de estado deben ser completamente independientes del protocolo. Así, podrán ser reutilizados en otro tipo de aplicaciones (RMI, etc…).

Arquitectura y diseño: Patrón MVC - El modelo

La capa de componentes de acción define los cambios permisibles del estado en respuesta a los eventos.

Los componentes de acción no pueden ser completamente independientes del protocolo, pero, aún así, se debe intentar reducir el acoplamiento al máximo o incluso construir dos subcapas, una dependiente del protocolo que transforme los eventos y delegue el procesamiento a otra capa de componentes de acción independientes del protocolo.

Arquitectura y diseño: Patrón MVC - El modelo



Arquitectura y diseño: Patrón MVC - El controlador

El controlador es responsable de recibir los eventos, determinar el procesador del evento, invocar al procesador y finalmente provocar la generación de la vista apropiada.

En una aplicación web java la tecnología más adecuada para implementar los controladores son los Servlets.

Estos servlets actúan como direccionadores (dispatchers) de las peticiones.

Arquitectura y diseño: Patrón MVC - El controlador

Los controladores deben realizar las siguientes tareas:

- Control de la seguridad.
- Identificación de eventos.
- Preparar el modelo.
- Procesar el evento.
- Manejar los errores.
- Provocar la generación de la respuesta.

Arquitectura y diseño: Patrón MVC - La vista

La vista representa la lógica de presentación de la aplicación.

Los componentes de la vista extraen el estado actual del sistema del modelo y proporcionan la interfaz de usuario para el protocolo que se está usando.

Como parte de la generación la vista debe presentar al usuario el conjunto de eventos que puede generar en ese momento concreto.

La tecnología Java indicada para la generación de vistas en aplicaciones web son las JSPs.

Separar el modelo y la vista permite la construcción de interfaces con diferentes apariencias.

Delegación de peticiones: RequestDispatcher.

Arquitectura y diseño: Delegación de peticiones - RequestDispatcher

Al construir un aplicación web suele ser necesario delegar el procesamiento de una petición a otros Servets (o JSPs), o incluir la salida de otros Servlets en la respuesta (para generación modulada de la respuesta).

Para este tipo de procesamiento el API Servlet proporciona la interfaz javax.servlet.RequestDispatcher.

Arquitectura y diseño: Delegación de peticiones - RequestDispatcher

Se puede recuperar un RequestDispatcher de tres manaras diferentes:

- ServletContext.getNamedDispatcher(String name) ➔ Devuelve un RequestDispatcher para redirigir la petición a un servlet declarado en el DD con el nombre *name*.
- ServletContext.getRequestDispatcher(String path) ➔ Devuelve un RequestDispatcher para redirigir la petición al recurso determinado por *path*.
- ServletRequest.getRequestDispatcher(String path) ➔ Devuelve un RequestDispatcher para redirigir la petición al recurso determinado por *path*.

Si cualquiera de estos métodos no pueden determinar el destino de la redirección devolverán *null*.

Arquitectura y diseño: Delegación de peticiones - RequestDispatcher

La interfaz RequestDispatcher define los siguientes métodos:

- public void forward(ServletRequest req, ServletResponse res) throws ServletException, IOException
- public void include(ServletRequest req, ServletResponse res) throws ServletException, IOException.

Aspectos de arquitectura y diseño en el contenedor Web – Delegación de peticiones: forward.

El método forward delega la petición en el servlet destino.

El servlet origen no debe haber escrito nada en la respuesta, es decir, se supone que toda la generación de la respuesta la va a llevar a cabo el servlet destino.

Si se ha escrito algo en la respuesta, cualquier llamada al método forward lanzará *IllegalStateException.*

Antes de que la llamada al método forward termine el contenedor habrá “cometido” la respuesta y cerrado el stream.

Aspectos de arquitectura y diseño en el contenedor Web – Delegación de peticiones: include.

Incluye toda la salida generada por el servlet destino en la respuesta.

El servlet destino tiene acceso a todos los métodos de la petición, pero tiene ciertas limitaciones a la hora de interactuar con la respuesta (el objeto ServletResponse), ya que cualquier intento de modificar o establecer cabeceras en la respuesta serán ignorados.

A no ser que el RequestDispatcher haya sido recuperado por medio del método getNamedDispatcher los siguientes atributos serán añadidos al objeto ServletRequest:

- javax.servlet.include.request_uri
- javax.servlet.include.context_path
- javax.servlet.include.servlet_path
- javax.servlet.include.path_info
- javax.servlet.include.query_string

Aspectos de arquitectura y diseño en el contenedor Web – Delegación de peticiones: errores.

Si durante una llamada a forward o a include se produce una excepción, la especificación indica que:

- Si la excepción es de tipo IOException o ServletException se propagará hacia el servlet origen.
- Si la excepción es de otro tipo se envolverá en una ServletException y el servlet origen podrá recuperar la excepción por medio del método Throwable getRootCause() de ServletException.

Aplicación de Ejemplo de MVC