

RIA

GRUPO 3

LIBRO : "Flex in Action"
Capítulos : 14, 15 y 19

¿Qué es Flex ?

- ⇒ Es un conjunto de librerías – o framework – para desarrollo de UI (user interface).
- ⇒ Entorno de trabajo para crear aplicaciones en Action Script
- ⇒ Está específicamente creado para RIA y es compatible con DB en XML y más

Estados

- Un estado se puede definir como una apariencia visual, con un comportamiento particular, y la representación para una vista (UI).
- Las vistas tienen al menos un estado, que es conocido como el estado por defecto o la base estado de la vista. Se puede definir cualquier número de estados adicionales en función de sus necesidades.

- Los estados pueden ser utilizados para muchos otros escenarios, tales como:
 - Una vista de búsqueda
 - Una vista de diseño-personalización
 - Vista que muestra datos de usuario en un estado y luego permite la edición en diferentes estados.

Estados de Flex (¿Como cambiar sus propiedades?)

Listing 14.1 Changing properties with states

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/halo">

  <s:layout>
    <s:VerticalLayout />
  </s:layout>
  <s:states>
    <s:State name="orange" />
    <s:State name="black" />
  </s:states>
  <s:HGroup>
    <s:Button label="Orange" click="currentState = 'orange'" />
    <s:Button label="Black" click="currentState = 'black'" />
  </s:HGroup>
  <s:Rect width="200" height="200">
    <s:fill>
      <s:SolidColor color.black="#000000" color.orange="#da7800" />
    </s:fill>
  </s:Rect>
</s:Application>
```

State declaration named orange

State declaration named black

Orange button changes state to orange

Black button changes state to black

200x200 rectangle

Rectangle's fill, color changed by state

Adición de un componente

Listing 14.6 Using includeIn

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/halo">

  <s:layout>
    <s:VerticalLayout />
  </s:layout>
  <s:states>
    <s:State name="orange" stateGroups="box" />
    <s:State name="black" stateGroups="box" />
    <s:State name="green" stateGroups="circle" />
    <s:State name="blue" stateGroups="circle" />
  </s:states>
  <s:Button label.orange="Black" label.black="Green"
            label.green="Blue" label.blue="Orange"
            click.orange="currentState='black'"
            click.black="currentState='green'"
            click.green="currentState='blue'"
            click.blue="currentState='orange'" />
  <s:Rect width="200" height="200" includeIn="box">
    <s:fill>
      <s:SolidColor color.black="black" color.orange="#de7800" />
    </s:fill>
  </s:Rect>
  <s:Ellipse width="200" height="200" includeIn="circle">
    <s:fill>
      <s:SolidColor color.green="green" color.blue="blue" />
    </s:fill>
  </s:Ellipse>
</s:Application>
```

Include Rect
in box group

Include Ellipse
in circle group

Listing 14.10 Real-world state example

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/halo"
               xmlns:views="views.*"
               applicationComplete="init()">

  <s:layout>
    <s:VerticalLayout paddingLeft="20" paddingTop="20" />
  </s:layout>

  <s:states>
    <s:State name="login" stateGroups="loggedOut" />
    <s:State name="computers" stateGroups="loggedIn" />
    <s:State name="info" stateGroups="loggedIn" />
    <s:State name="tv" stateGroups="loggedIn" />
  </s:states>

  <s:Panel includeIn="loggedOut" title.login="Get in there!">
    <s:layout>
      <s:VerticalLayout />
    </s:layout>
    <mx:Form>
      <mx:FormItem label="Username">
        <s:TextInput />
      </mx:FormItem>
      <mx:FormItem label="Password">
        <s:TextInput />
      </mx:FormItem>
    </mx:Form>
    <mx:ControlBar>
      <s:Button label="Login" click="currentState='computers'"/>
    </mx:ControlBar>
  </s:Panel>

  <s:HGroup includeIn="loggedIn">
    <s:ButtonBar dataProvider="{contentStack}" />
    <s:Button label="log out" color="black" click="currentState='login'"/>
  </s:HGroup>

  <mx:ViewStack id="contentStack" includeIn="loggedIn">
    <views:ComputersView label="Computers" />
    <views:InfoForm label="Info" />
    <views:TVView label="TVs" />
  </mx:ViewStack>
</s:Application>
```

State declaration for
loggedOut group

State declarations
for loggedIn
group

Login view

Application menu

ViewStack
showing content

Resumen: Estados

- Son de gran valor para su caja de herramientas
- Usados para manipular propiedades de los componenetes:
 - Configurar los manejadores de clic específicos.
 - Añadir y eleminiar objetos de la pantalla.
 - Controlar cuando los elementos se crean y/o eliminan de la lista de visualización.
 - Mover objetos entre los padres.
 - Controlar cada fase de los estados a través de eventos.

Trabajar con los servicios de datos

Integrar los servicio de datos en su aplicación:

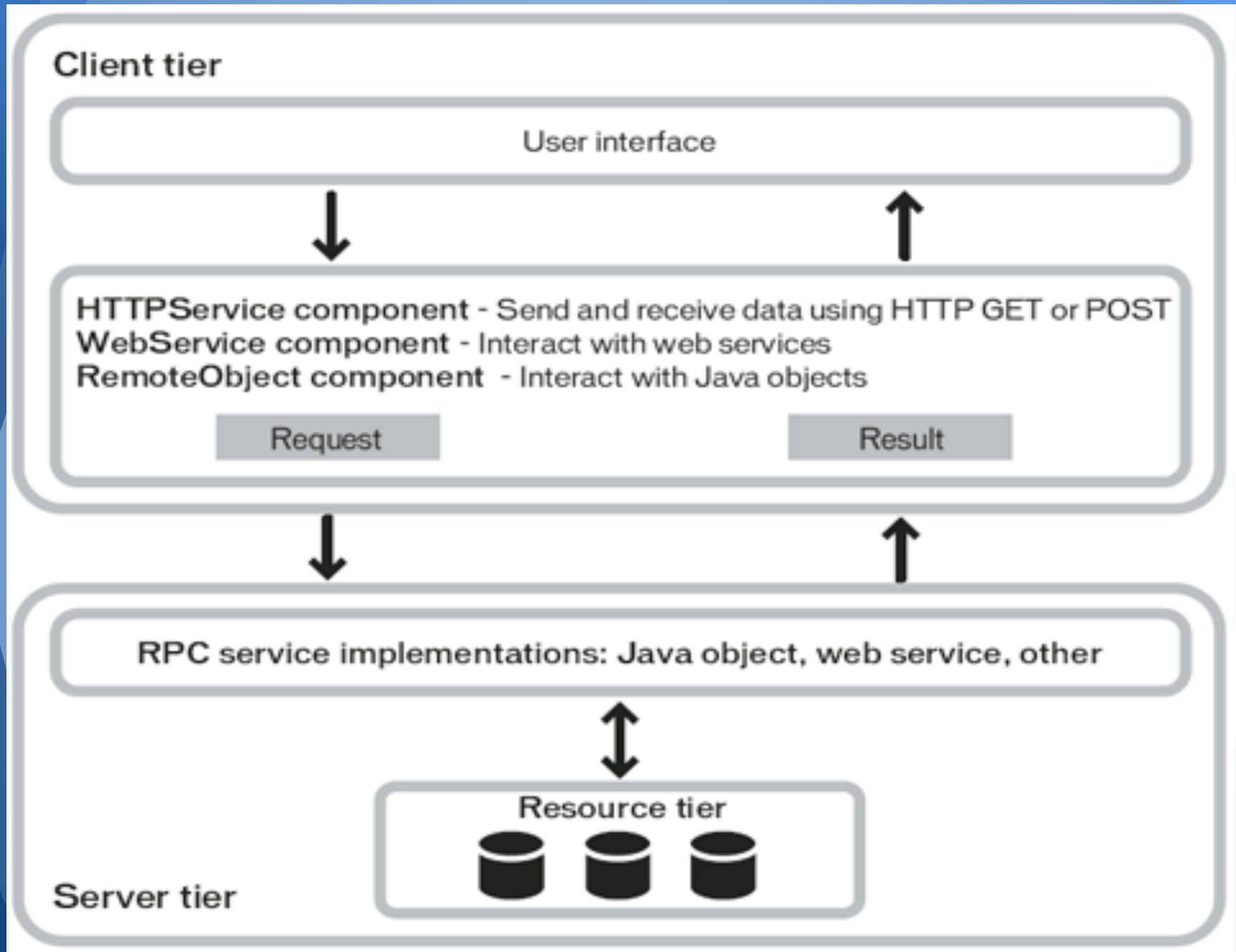
-HTTPService

-WebService

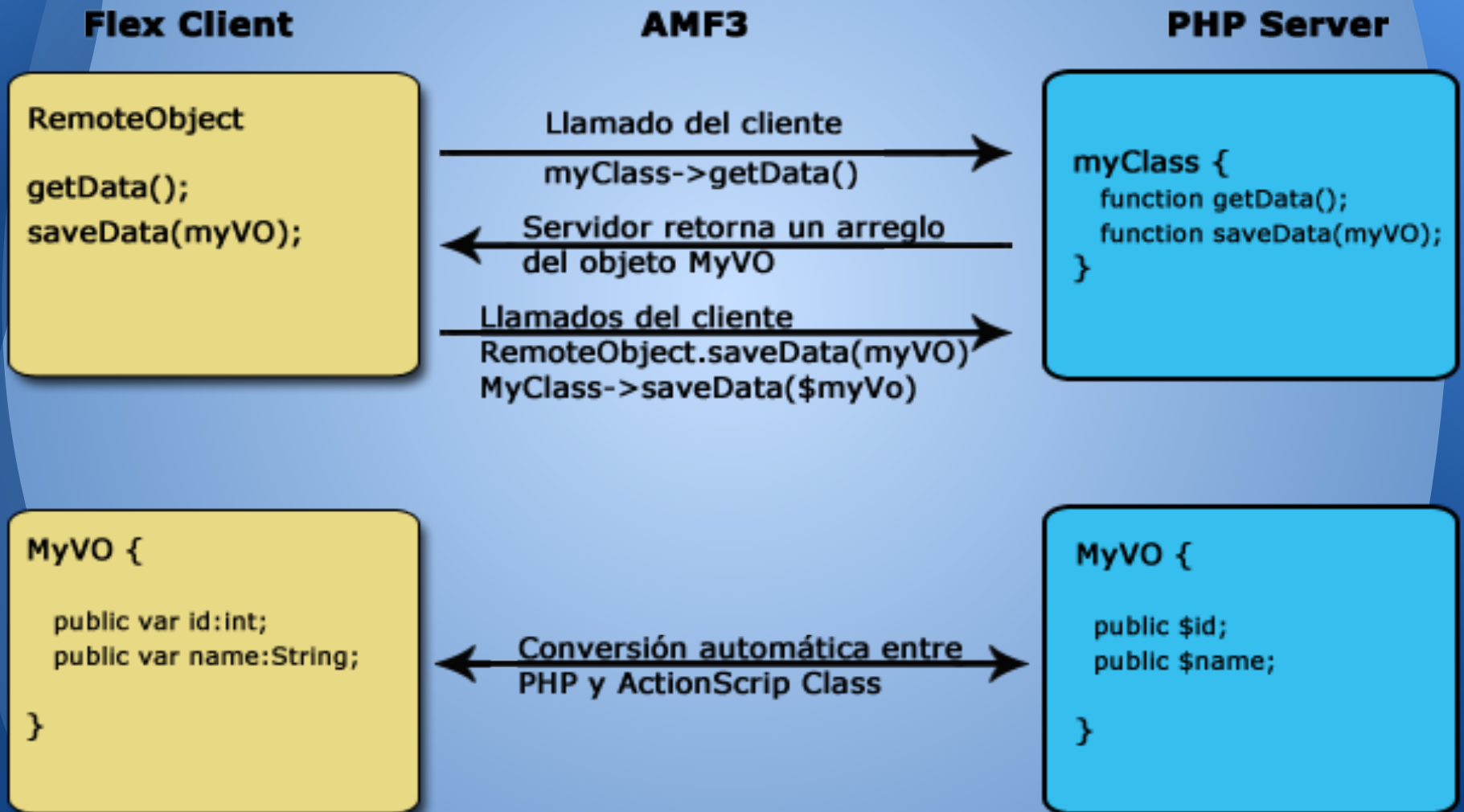
-AMF (Action Message Format):

- **Open Source**
- **PHP**
- **ColdFusion**
- **BlazeDS**
- **LiveCycle Data Services**

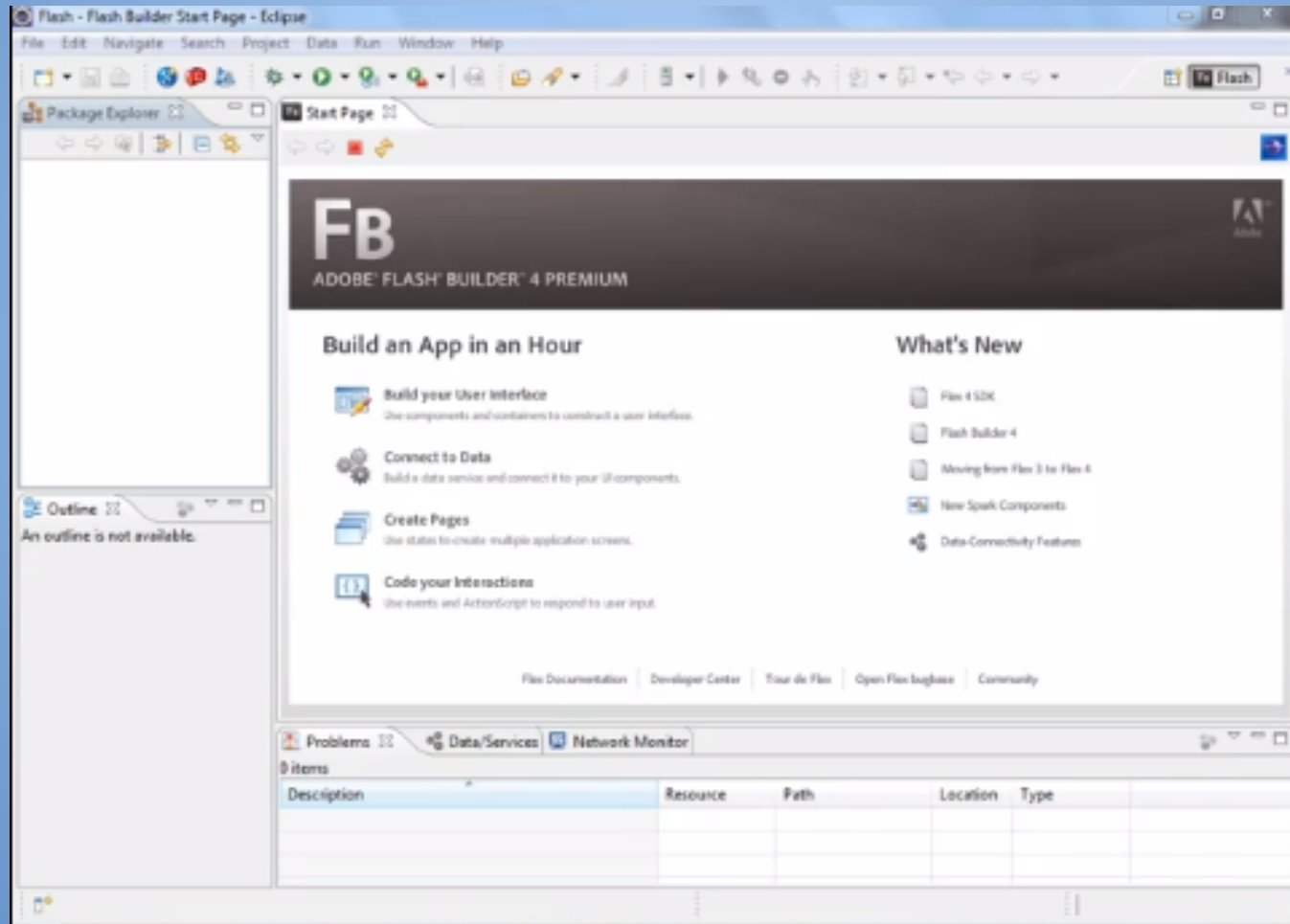
Comunicación Cliente/Servidor RIA



La figura muestra el flujo de datos entre Flex y PHP por medio de AMF



Creación de aplicaciones Flex centradas en datos con Flash Builder



Características Destacadas

- Ahorro en el tiempo de desarrollo
- Generación de código en segundo plano sin salir del IDE
- Flujo de trabajo en general independiente de la tecnología de servidor
- Coherencia con la metodología MVC

Creación de un entorno adecuado

- El IDE refleja la plataforma tanto del cliente como del servidor (Matriz IDE)

Servicios

- Generación de servicios básicos con operaciones CRUD
- Configuración de los tipos de datos de envío y retorno a partir de la base de datos
- Drag and Drop para el enlace de los datos

Patrones de diseño impulsadas en Flex

Arquitecturas eficaces para Flex:

- MVC.
- Patrón de diseño modelo-presentación.

Patrones adaptados a las necesidades y usabilidad de flex.

- Orientado a eventos.
- Vinculan objetos.

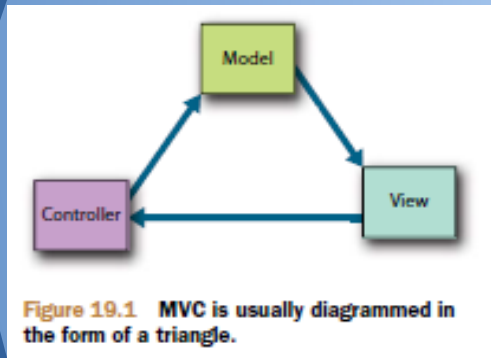
MVC

El código está organizado en tres capas:

- Los datos encapsulados(Modelo).
- De cara al usuario la lógica de visualización(Vista).
- La tercera capa actúa de mediadora entre las partes (Controlador).

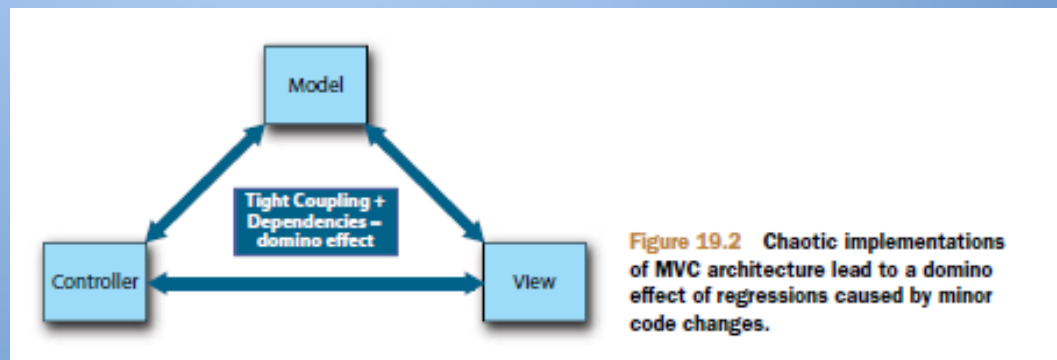
MVC

La vista a menudo tiene la capacidad de afectar directamente el modelo, pero debe pasar por el controlador para tener un efecto sobre él.



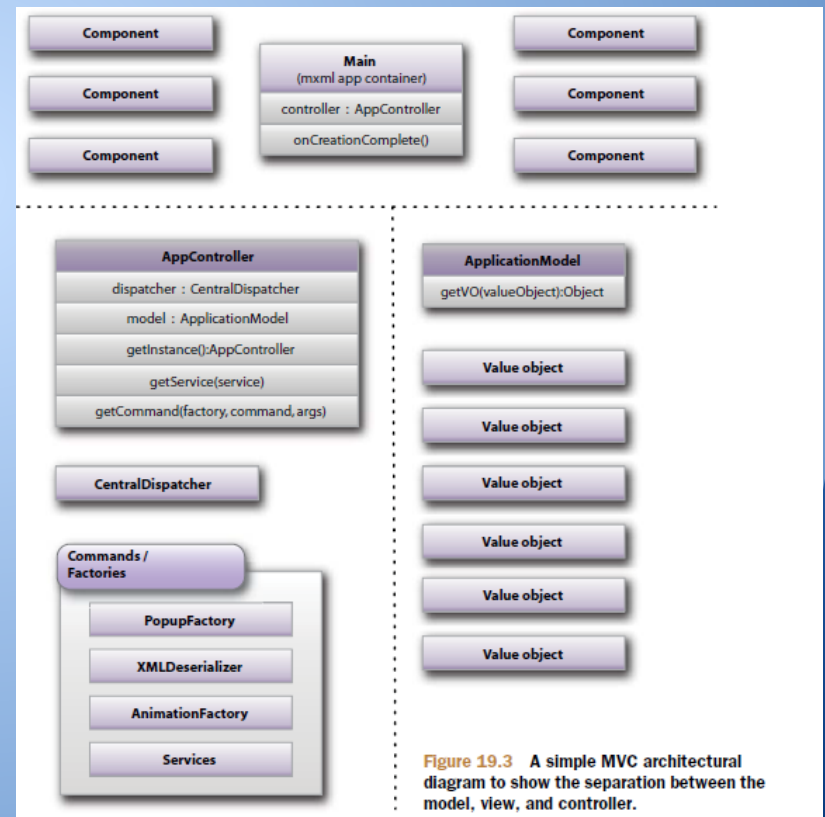
Flujo de control MVC.

MVC implementación caótica.



Como crear su propia arquitectura

A medida que la aplicación crece, también lo hace su nivel de complejidad. Ejemplo de una improvisada aplicación MVC, suele ser reprogramada después de la primera iteración empleando Microarquitectura como Robotlegs.



Microarquitectura

- Arquitectura dentro de una arquitectura mucho más grande.
- Múltiples arquitecturas conforman el marco global.
- Costos/Beneficios:
 - Capa de complejidad.
 - Efectos desastrosos.
 - Curva de aprendizaje
 - Estudio para mantener.
 - + Fuerza a utilizar normas consistentes.
 - + Normas para organización y nombres.
 - + Fácil mantenimiento del código.
 - + Mantenido por varios programadores.
 - + Provee ventajas, bajas dependencias, bajo acoplamiento, rendimientos y reusabilidad.

Microarquitectura

- **Segunda generación:**

- Construidas para capacitar a los desarrolladores en lugar de dominar.
 - Mate.
 - Swiz Framework.
 - Robotlegs.
-
- Utilizan Control of Inversion(COI) y Dependency Injection (DI).

Robotlegs

- Atiende específicamente a nivel empresarial Flex y desarrollo AS3 puro.
- Ofrece una arquitectura sólida:
- Optimización de dependencias.
- Inversión del flujo de control a través contextManaged DI.
- Mediación dinámica de los componentes para reducir código repetitivo.

- Inyección de dependencias a través de SwiftSuspenders.
- Agregando metadatos para definirlas.
 - [Inject] : Especificar puntos de inyección.
 - [PostConstruct] : Marcar métodos de ejecución

Robotlegs MVCS classes

- **Injector**
- Utiliza por defecto SwiftSuspenders.
- **Injector**
- Es un adaptador directamente a la solución DI que la aplicación está utilizando.
- La interfaz proporciona cuatro métodos:
 - **mapSingleton** : Asignar una única instancia de una clase.
 - **mapSingletonOf** Asignar una única instancia de una clase base o interfaz.
 - **mapValue** : Es utilizado para asignar una instancia específica de una clase.
 - **mapClass** Asignar muchas instancias únicas de la clase.

Robotlegs MVCS classes

- MediatorMap

- Utilizada para definir la relación entre los componentes de la vista y los mediadores que los conectan con el resto de la aplicación.

- CommandMap

- Es utilizado para definir la relación entre los eventos ActionScript.

- ViewMap

- Permite mapear componentes de vista para una inyección.

Creación de aplicaciones con MVCS Robotlegs



•En MVCS:

- El modelo almacena la información
- La vista proporciona un mecanismo para que el usuario pueda interactuar con el controlador y este responda las acciones
- El servicio se conecta a los recursos externos, tales como un servicio web o un archivo local de sistema.
- No hay clases de modelo o Servicio. Estas, son utilizadas para separar las responsabilidades y ambas utilizan la clase **Actor**.

Contexto	Mediador	Comando	Actor
Es un hub que facilita la comunicación entre niveles y proporciona la secuencia de arranque inicial de la estructura.	Sirve como conectores entre los componentes de vista. Sus usuarios interactúan con el framework a través de las vistas.	Sirve como un mecanismo para reducir los niveles de su aplicación y encapsular la lógica. Se representan al nivel del controlador.	Proporciona parte de la funcionalidad básica que es utilizada por las clases, tanto en el servicio y como en el modelo.

CONTEXTO



Es el corazón de la aplicación Robotlegs.

Al crear el Contexto de la aplicación, reemplazaremos el método startup().

Aquí mapeamos:

- Modelo (ACTOR)
- Servicio (ACTOR)
- Vistas, y sus mediadores, (MEDIADOR)
- Diversos comandos a sus respectivos eventos. (CONTROLADOR)

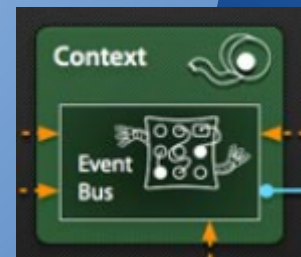
Se inicializa dentro de la vista.

En MXML, Hay que añadirlo dentro de las declaraciones. (Por ser no visual)

Se puede crear con Acción-Script con un evento con un ciclo de vida, pero es conveniente utilizar MXML

.

ContextView: Provee la referencia a una vista y hay que indicarle si se iniciara automáticamente o no.



VISTAS y MEDIADORES



VISTAS:

- Representan a los mediadores.
- Punto de entrada de los usuarios.

MEDIADOR:

- Entrega y recepción de mensajes entre las clases de la aplicación.
- Está a la escucha de estos eventos.
- Tiene una relación uno-a-uno con su respectiva vista. Por esta razón es importante asociar el componente de vista en el mediador.

MediatorMap:

- Crea la relación entre la vista y su mediador en el contexto.
- Además relaciona el Modelo, que controla el estado de la aplicación y se accede por el mediador.

onRegister:

- Es un metodo que se reemplazara en casi todos los casos.
- Es el gancho en el mediador.
- Utiliza el EventMap para añadir escuchas a la vista.
- Sirve como un puente y proporciona una separación limpia de sus responsabilidades.
- Distribuye un evento.

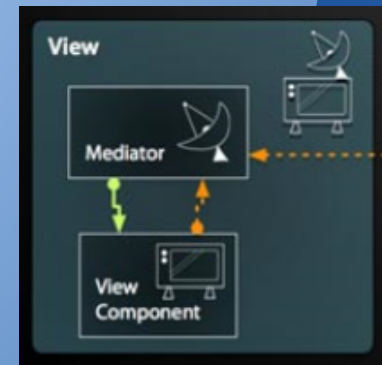
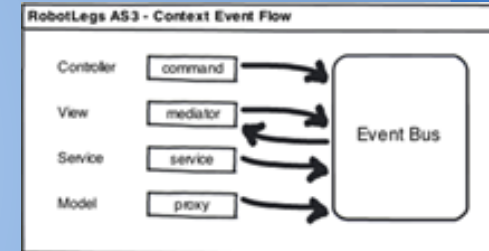
EventMap:

- Es un objeto local creado por MVCS Mediador y clases Actor.
- Proporciona el método unmapListeners para eliminar a la vez todos los detectores registrados. Un mediador invoca esta función automáticamente cuando es eliminado.
- Es muy cómodo a la hora de asignar detectores de eventos dentro de las clases Robotlegs MVCS.
- Escucha eventos de la propiedad de eventDispatcher del Mediador.

eventDispatcher:

- Se encuentra dentro de todas las clases MVCS.
- Es un bus de eventos centralizado que es utilizado para la comunicación entre las clases.
- Cada contexto proporciona un único IEventDispatcher para facilitar esta comunicación.

Estos eventos son asignados en el mediador a través del EventMap.
Actores, Mediadores y comandos están equipadas con acceso a este bus de eventos para permitir la comunicación desacoplada.



COMANDOS



Se utiliza para encapsular una solicitud y realizar operaciones.

Cada acción da lugar a algún tipo de trabajo que debe ser realizado.

Los comandos son realizados para un único fin.

Al ver el paquete del controlador, se es capaz de ver todo lo que hace la aplicación.

Intentar que sus responsabilidades sean las mínimas posibles.

Al distribuir un evento a través del eventDispatcher del Contexto, se asigna a un comando, es manejado por el CommandMap.

CommandMap:

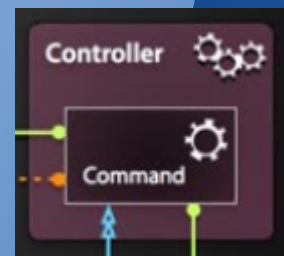
Ejecutara el método execute() del comando , luego de recibir un evento mapeado que lo desencadena.

El único requisito de un comando es que tenga un método execute().

También incluyen el mapeo de utilidades de Robotlegs, asignar servicios, un método dispatch(), y el acceso a la ContextView del contexto.

Están destinados a ser de corta duración.

Son creados para hacer su trabajo, y luego se destruyen rápidamente.



SERVICIOS



SERVICIO:

Tenemos que acceder o interactuar con servicios externos.

Ejemplos: Servicio web con acceso remoto a una base de datos.

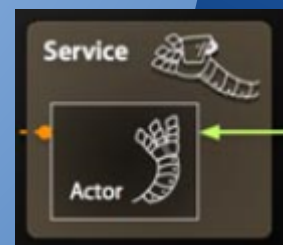
Aplicaciones remotas como LiveCycle Data Services, APIs web desde sitios como Google o Flickr.

ACTOR:

Es una clase de utilidad que proporciona eventos del contexto y el método dispatch().

Al hacer uso de servicios externos es importante convertir los datos que se recuperan tan pronto como sea posible.

Si se conecta a un servicio remoto, estaría llevando a cabo estas acciones de forma asíncrona con controladores de eventos o respuestas de peticiones a una HTTPService, RemoteObject o NetConnection.



MODELO



Representan los datos y estos el estado de su solicitud.

Al igual que los servicios, los modelos no tienen una clase que los representen específicamente.

La clase Actor Robotlegs MVCS los crea.

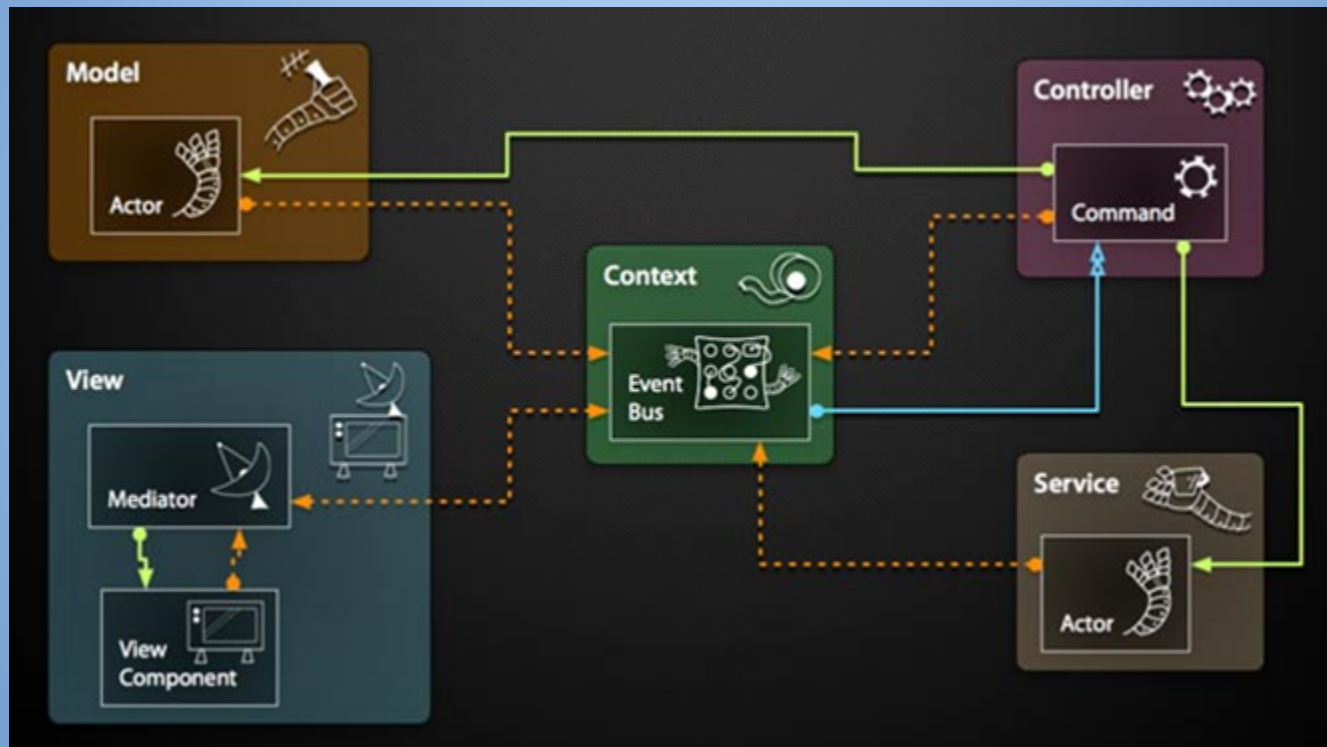
Ambos servicios y modelos son los niveles conceptuales en Robotlegs MVCS.

Además de la gestión de temas relacionados con el Estado, el modelo se encarga de manipulación de los datos.

Cuando por ejemplo se crea o se elimina un contacto de una lista de contactos, se hace en el modelo.



Modelo simple



FIN...

MUCHAS GRACIAS!!!