



Curso React con datos inmutables

Carlos de la Orden
redradix

Contenido

- 1. Visión general y arquitectura
- 2. Átomo: gestión de estado global
- 3. Vistas con React
- 4. Lógica de negocio
- 5. Testing
- 6. Infraestructura

Contenido

- 1. Visión general y arquitectura
 - Origen: de página a aplicación
 - Arquitectura
 - Motivos y ventajas
 - Empaquetar la aplicación
 - Ejercicio / ejemplo

Origen: de página a aplicación

El pasado: páginas web

- Server-side: TODA la lógica en el servidor
- Cliente sólo pide y visualiza páginas completas
- Para interactuar:
 - Cambiar URL (HTTP GET)
 - Formulario (HTTP POST)

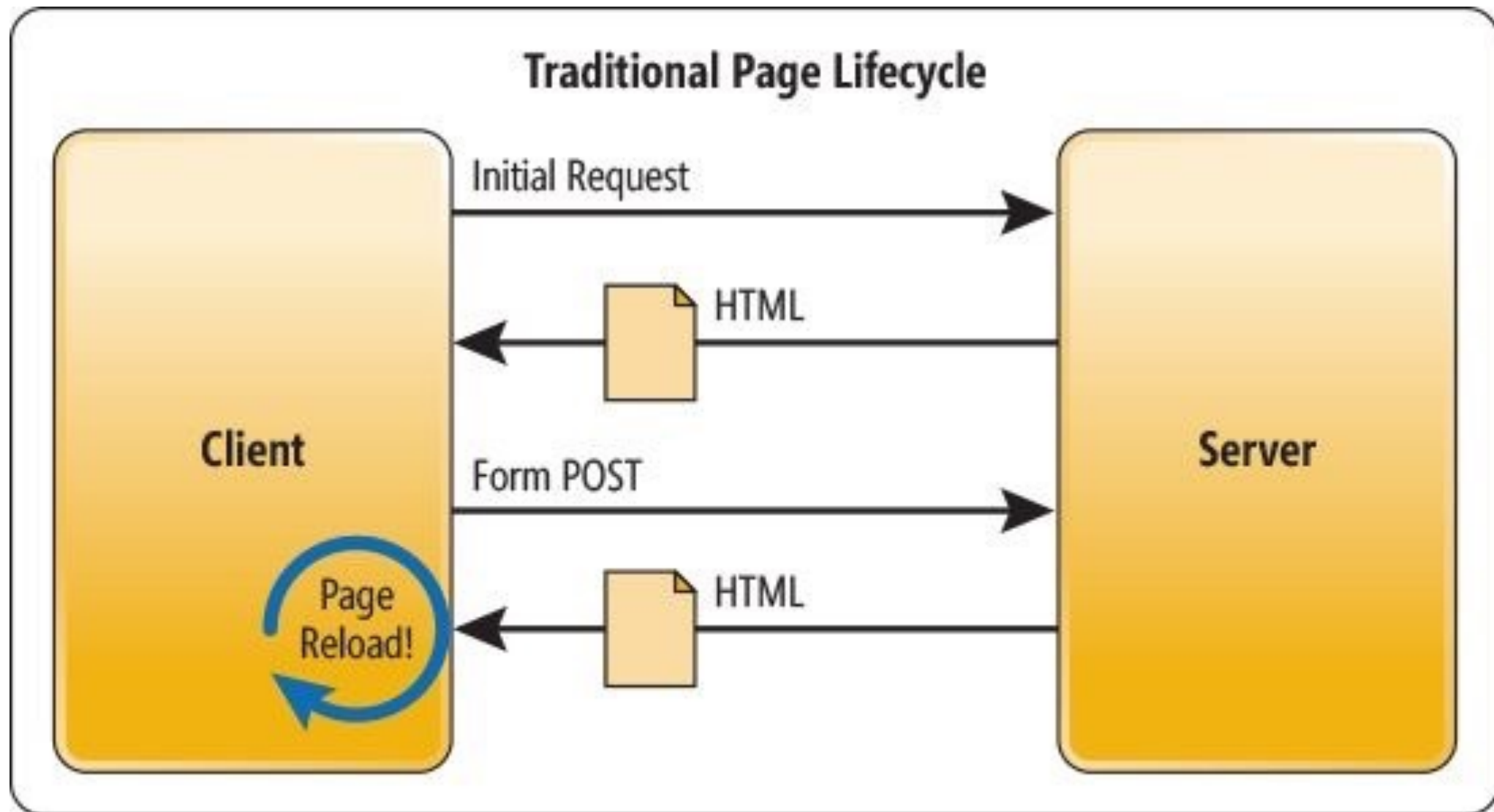
Origen: de página a aplicación

El presente y futuro: Single Page Applications

- Actualmente: ~~páginas~~ aplicaciones Web
- Cliente **construye** la aplicación completa con **componentes**
- Para interactuar: peticiones asíncronas de datos
 - Ajax: HTTP Request + JSON / XML (RESTful APIs)
 - Websockets (Realtime APIs)
- Ejemplos: Gmail, Facebook, Google Maps...

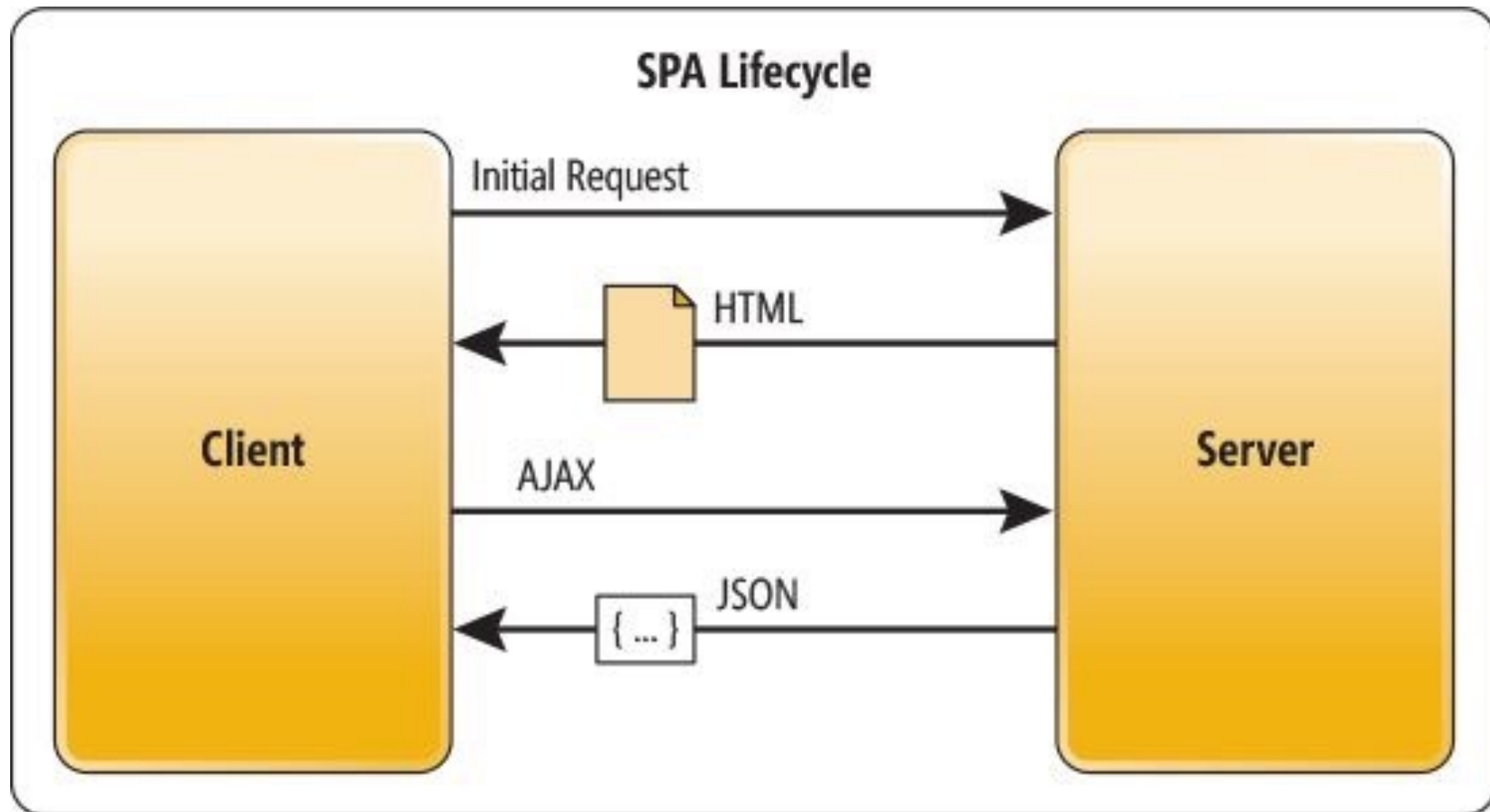
Origen: de página a aplicación

Interacción cliente <-> servidor



Origen: de página a aplicación

Interacción cliente <-> servidor



Origen: de página a aplicación

Frameworks

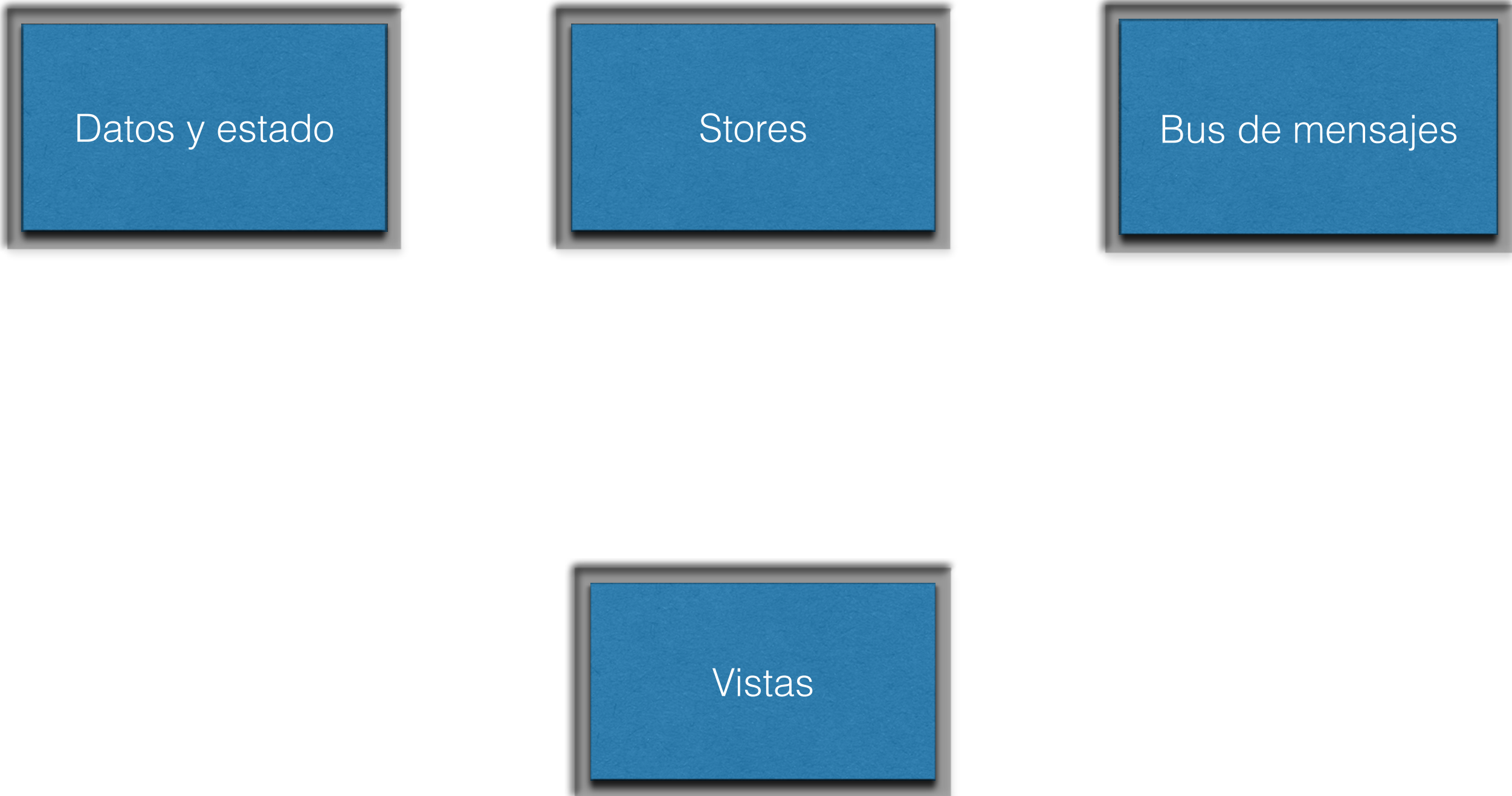
- Aplicaciones de cliente complejas
- Mismos problemas que aplicaciones de escritorio
- Mismas soluciones
 - Model View Controller
 - Decenas de implementaciones (jQuery, Backbone, Ember, Angular...)
 - Mismas ventajas / inconvenientes

Arquitectura

- Una alternativa **diferente**
- Sencilla de entender
- Aprovecha tecnologías recientes

Arquitectura

Las piezas del puzzle



Datos y estado

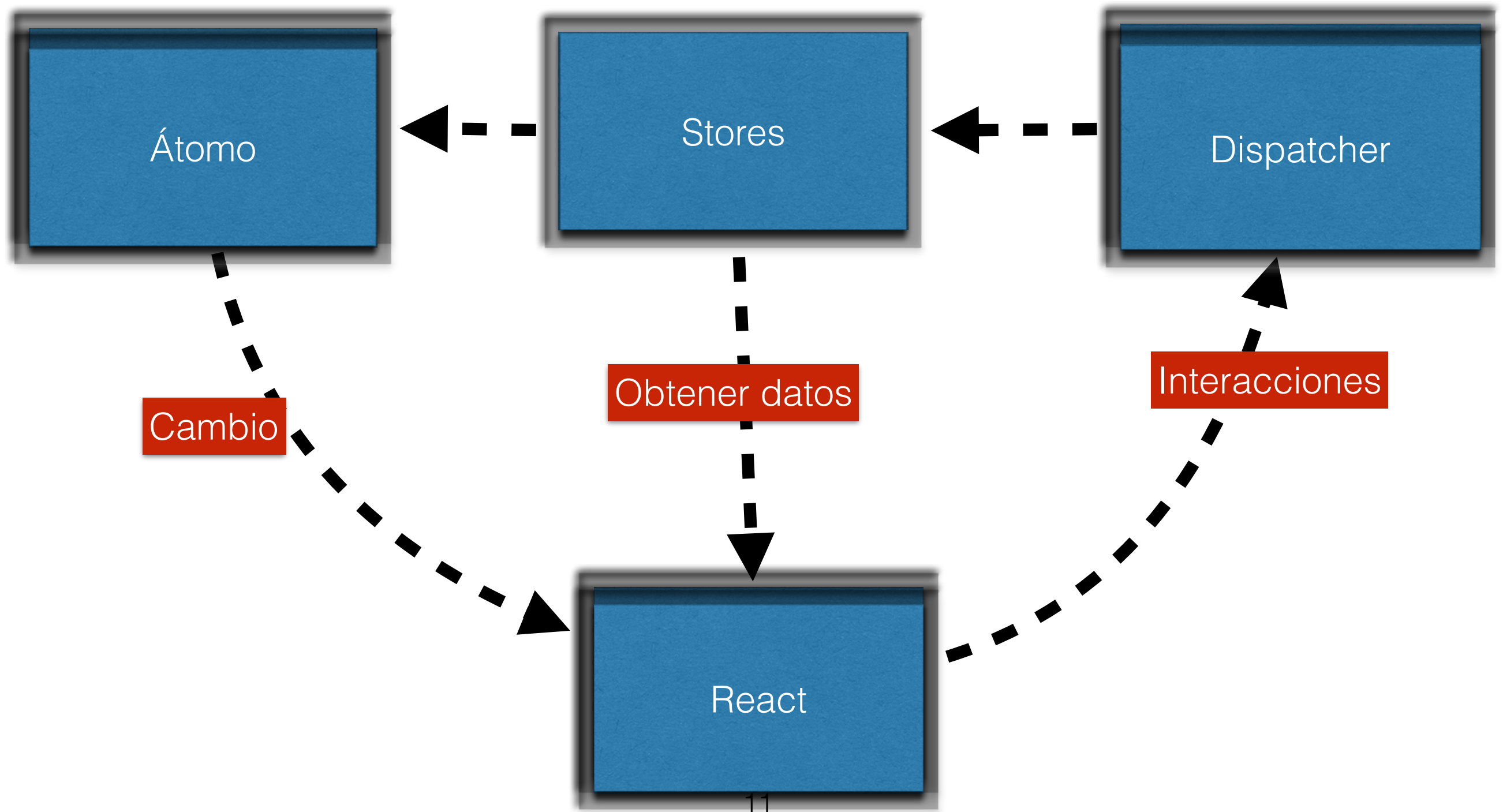
Stores

Bus de mensajes

Vistas

Arquitectura

Comunicación



Arquitectura

Átomo: datos y estado

- Átomo (súper estructura)
- Guarda **todo el estado de la aplicación**
- Utilizando estructuras de datos **inmutables**

Arquitectura

Stores: acceso y modificación del átomo

- Responsables de una parte del átomo global
- Ofrecen **consultas** a las vistas para recuperar información concreta
- Implementan **comandos** recibidos por el Dispatcher para:
 - Modificar el átomo
 - Acceder a servicios externos
 - Emitir otros comandos (orquestración)

Arquitectura

React: UI declarativa

- Las vistas son una **función del estado** (átomo): cada componente utilizará un subconjunto de los datos existentes en el átomo
- Se construye una UI compleja a partir de una jerarquía de **componentes sencillos**
- Los componentes padre pueden configurar los componentes hijo
- Cualquier interacción dispara uno o varios mensajes a través del Dispatcher

Arquitectura

Dispatcher: bus de mensajes

- Es la **única forma** en que las Vistas notifican cambios a los demás actores
- Los mensajes pueden llevar datos asociados
- Un mismo mensaje puede ser escuchado desde **diferentes Stores**
- Cada mensaje es atendido de forma **síncrona**

Motivos y ventajas

Flujo unidireccional

- La UI sólo se actualiza **por un motivo**: el átomo ha cambiado
- Los componentes consultan los datos que necesitan a uno o varios Stores: estos son “sus” datos
- Si estos datos no han cambiado, la vista no cambiará
- Si estos datos han cambiado, la vista cambia / aparece / desaparece (y con ella sus componentes hijos)

Motivos y ventajas

Estado contenido en el átomo

- El estado completo se puede ver en un momento determinado
- Estado explícito en lugar de estado **implícito**
- El estado se puede **serializar**, manipular de forma externa (testing)
- Permite que un cambio en **cualquier parte del átomo** reconstruya la UI completa

Motivos y ventajas

Interfaz de usuario con React

- Modificar el DOM es una operación **costosa**
- **Queremos** modificaciones selectivas
- **No queremos** escribir nosotros esa lógica
- Con React nos despreocupamos: que decida React
- ¿Cómo lo hace? **Virtual DOM**

Motivos y ventajas

React - Virtual DOM

- Los componentes de React no generan HTML directamente
- Generan **código**
- React compara la **salida** de los componentes cuando hay cambios, para decidir **qué debe cambiar en el DOM**
- React garantiza que se realizarán las mínimas operaciones necesarias en el DOM, de la forma más eficiente

Empaquetar la aplicación

- Empaquetar nos permite programar nuestra aplicación cliente de forma modular
- Archivos separados, una estructura de carpetas clara, etc
- Pero a la hora de incluir el JS en el navegador, queremos **un único archivo** con toda la aplicación

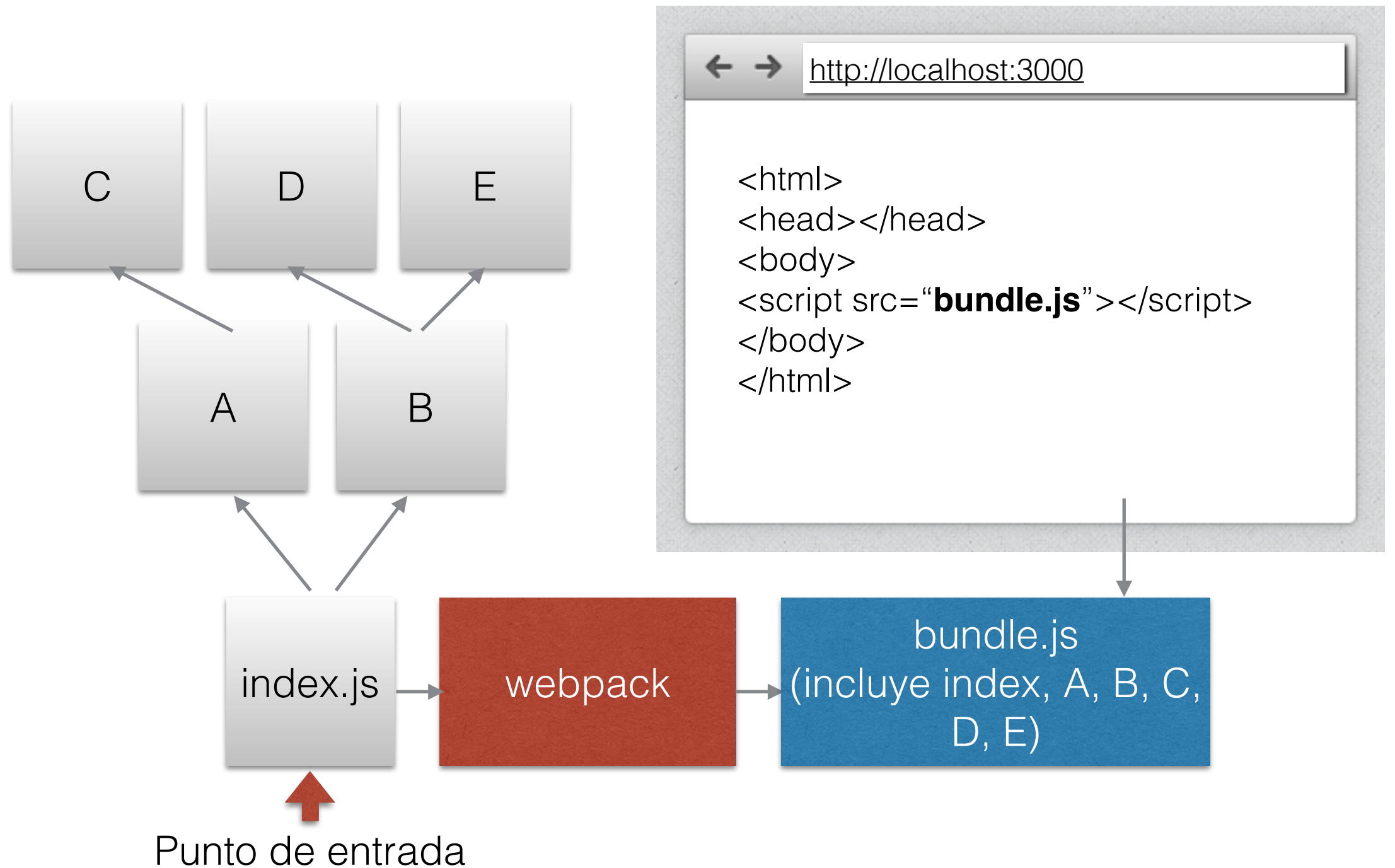
Empaquetar la aplicación

- Para esto necesitamos una herramienta que “compile” nuestra aplicación con sus dependencias y nos genere el *bundle* completo
- Herramientas: **webpack**
- <https://webpack.github.io/>

Empaquetar la aplicación

- Utilizamos **npm** para gestionar dependencias (node.js)
- Utilizamos sintaxis **require(xxx)** de node.js
- Y por último **webpack** se encargará de recorrer el árbol de dependencias desde nuestro “main” para generar el *bundle* completo

Empaquetar la aplicación



Empaquetar la aplicación

- Instalar webpack globalmente:
➔ `npm install -g webpack`
- Lanzar webpack:
➔ `webpack [entrada.js] [/ruta/bundle.js]`
- O bien configuración en **webpack.config.js** y sólo invocar **webpack**

webpack.config.js

```
'use strict';

//configuracion de webpack para compilar JSX y crear el bundle
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: './dist',
    sourceMapFilename: './dist/[file].map',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      { test: /\.js$/, loader: 'babel-loader', include: path.join(__dirname, './src') }
    ]
  }
};
```

webpack.config.js

```
'use strict';

//configuracion de webpack para compilar JSX y crear el bundle
var path = require('path');

module.exports = {
  ➔ entry: './src/index.js',
  ➔ output: {
    path: './dist',
    sourceMapFilename: './dist/[file].map',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
  ➔     { test: /\.js$/, loader: 'babel-loader', include: path.join(__dirname, './src') }
    ]
  }
};
```

Punto de entrada

Archivo y ruta de salida

Procesadores de código

Servir la aplicación

- Cuando desarrollamos una SPA, necesitaremos un micro servidor Web de desarrollo que nos cargue el archivo HTML con nuestro bundle
- Probaremos la aplicación abriendo **localhost:xxx**
- Además queremos que webpack se ejecute con cada cambio en un archivo (*watching*)

➡ `webpack -w`

Servir la aplicación

- webpack permite su uso desde código
- Para los ejercicios (temas 3 y 4), ejecutaremos el archivo `index.js` dentro de cada carpeta
- Incluye la compilación con webpack con *watching* y un servidor web sencillo

➔ `node index.js`

- ¡¡Aplicación lista en <http://localhost:3000>!!

Ejemplo

➔ `cd ./soluciones/tema4 && node index.js`

- <http://localhost:3000>

