Facultad de Ciencias de la Administración

TECNICATURA UNIVERSITARIA EN DESARROLLO WEB

Server side y Node.js

Semana N.º 2 – Lado del Servidor Parte II

Tecnicatura Universitaria en Desarrollo Web

Facultad de Ciencias de la Administración - Universidad Nacional de Entre Ríos

Objetivos de la clase

- **Objetivos**
  - Conocer las responsabilidades y características de la programación del lado del servidor.
  - Utilizar los métodos del protocolo HTTP para comunicar clientes y servidores web.
  - Crear vistas generadas a partir plantillas HTML renderizadas en el servidor.
  - Comprendan la noción de solicitud/respuesta y cómo se gestionan las rutas.
- **Temas a desarrollar:**
  - Revisión de conceptos de Programación Orientada a Objetos, programación sincrónica / asincrónica y gestión de dependencias.
  - Características de la programación del lado del servidor. Diferencias con programación del lado del cliente. Tecnologías comunes de programación del lado del servidor.
  - Programación del lado del servidor usando NodeJS. Event Loop. Programación basada en eventos.
  - Creación de un servidor web. Procesamiento de solicitudes HTTP.
  - Introducción al framework Express.js. Disposición de recursos estáticos. Motores de plantillas.

npm

- ***npm*** es un gestor de paquetes para **JavaScript** y por defecto, el gestor de paquetes de **Node.js**.
- En términos generales, las principales responsabilidades de un gestor de paquetes son la **instalación de paquetes** y la **administración de dependencias**.
- La sigla **npm** no se trata de un acrónimo sino que proviene de la abreviación recursiva ***"npm no es un acrónimo"***. Sin embargo, el primer commit del proyecto dice que **npm** son las siglas de *Node Package Manager*.
- **npm** es un administrador de paquetes rápido y confiable que es en gran parte responsable del rápido crecimiento y la diversidad del ecosistema de **Node.js**.

npm (2)

- **npm** consiste en un cliente de línea de comandos (CLI), que también se llama **`npm`**, y una base de datos en línea de paquetes públicos y privados, llamada ***registro npm***.
- Se accede al **registro** a través del cliente, y los paquetes disponibles se pueden explorar y buscar a través del sitio Web de **npm**: https://www.npmjs.com/
- Otro gestor de paquetes popular para **JavaScript** es ***Yarn***.
- El principal comando para utilizar **npm** es **`npm install`**.
- Por ejemplo, si queremos instalar **eslint** (herramienta que analiza el código buscando errores, malas prácticas y estilos inconsistentes. Opcionalmente realiza correcciones). En una terminal deberíamos ejecutar: **`npm install -g eslint`**
- El indicador **`-g`** le dice a **npm** que instale el paquete globalmente, lo que significa que está disponible en todo el sistema.
- La regla general es que las utilidades de **JavaScript** (como **eslint**) se instalarán globalmente, mientras que los paquetes específicos para cada aplicación web no se instalarán globalmente y por el contrario serán locales al proyecto según lo indique **package.json**.

Definición de paquetes en package.json

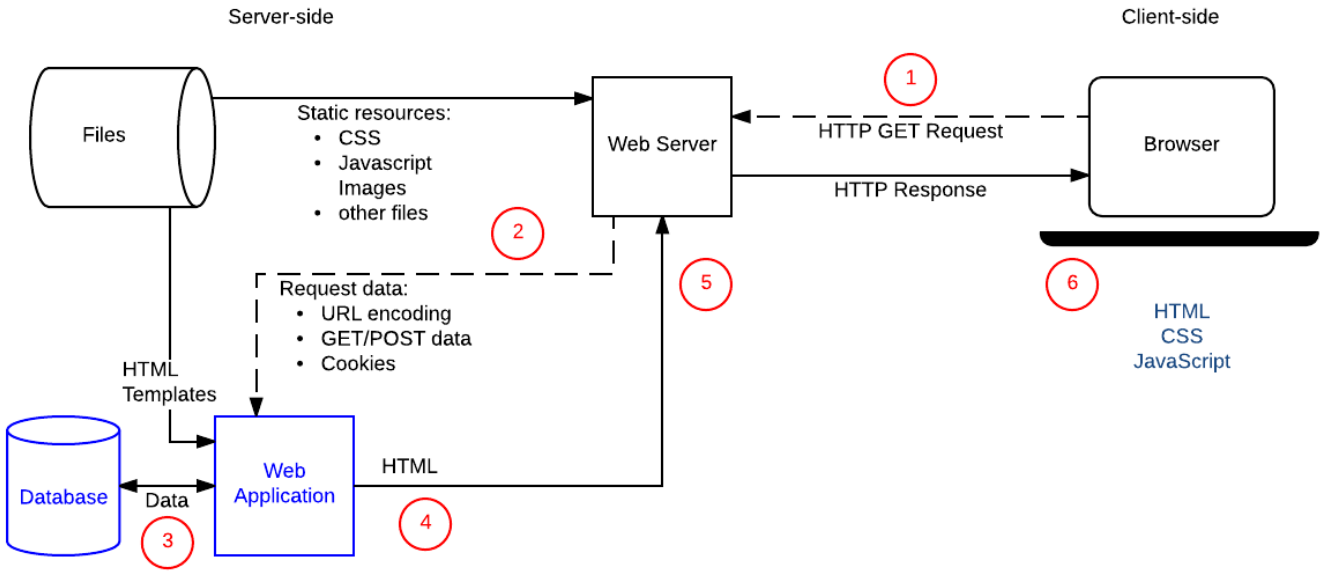
- En toda aplicación **Node** del mundo real el directorio raíz de la aplicación que contendrá un archivo llamado ***package.json***.
- Este archivo contiene metadatos sobre el proyecto incluyendo los paquetes de los que depende para ser ejecutado.
- Para crear una aplicación **Node** que use ***package.json*** ejecutamos: **`npm init`** ó **`npm init -y`**
- Si queremos agregar nuevos paquetes a nuestra aplicación usamos **`npm install [paquete]`**
  - En versiones anteriores a **`npm 5`** era necesario escribir **`npm install --save`**.
- Si el proyecto ya está iniciado (programado) y queremos descargar sus dependencias para ejecutarlo utilizamos el comando: **`npm install.`**
- Para usar módulos en vez de ***CommonJS*** debemos especificar la configuración **`{"type": "module"}.`**

```
// Importar usando Módulos
import { add } from './math.js';
```

```
// Importar usando CommonJS
const add = require('./math');
```

Renderizado del Lado del Servidor

- El esquema donde el servidor web procesa los datos de entrada y genera dinámicamente contenido en formato **HTML** se suele llamar ***Renderizado del Lado del Servidor (SSR)***.
- En este esquema, el servidor genera el **HTML** en función de las solicitudes del cliente, procesando datos y plantillas para enviar una página completa al navegador.



Primer servidor Web en Node (2)

- Para que nuestras páginas web estén disponibles para ser visitadas **Node** nos establece un esquema de trabajo muy diferente al de otros lenguajes y servidores web como **Java, PHP** ó **ASP.net**.
- En **Node**, es nuestra responsabilidad programar el servidor web. Sin embargo, a través del módulo integrado ***http***, **Node** se encarga que esto se solucione en un par de líneas.
- Vamos a construir el clásico ***Hola Mundo*** en el archivo **`01-holaMundo.js`**:

```
import http from 'http';

const port = 3000;

const server = http.createServer((req, res) => {
     res.writeHead(200, { 'Content-Type': 'text/plain' });
     res.end('Hola Mundo!');
});

server.listen(port, () => console.log(`Servidor iniciado en el puerto: ${port}`));
```

- Para ponerlo a correr, en la consola de comandos ejecutamos: **`node 01-holaMundo.js`**, **`npm run start`** o simplemente **`npm start`**.
- Para cortar la ejecución del programa usamos hacemos **“Control + C”** (como cuando copiamos un texto).

Rutas y recursos estáticos

- Para comprender mejor que estamos programando el servidor web, vamos a agregar algunas funcionalidades a nuestro **Hola Mundo**.
- En primer lugar a fin dar la opción al usuario de navegar a través de distintas páginas vamos a determinar cuál es la ruta deseada utilizando el objeto **`req`** (**`Request`** ó **Solicitud**).

```
…
const server = http.createServer((req, res) => {
    //Quito los query params. Si era /institucional?nombre=Juan => queda /institucional
    let processedPath  = req.url.indexOf('?') > 0? req.url.substring(0, req.url.indexOf('?')) : req.url;
    switch (processedPath) {
      case '/':  …
      case '/institucional': …
      case '/contacto': …
```

- Para que resulte más fácil trabajar con **HTML** cambiamos el programa. De esta forma resultará más fácil de modificar así como también acceder a recursos adicionales como archivos **CSS**, imágenes, etc. estos son llamados ***recursos estáticos***.
- Para cumplir con este cometido creamos un directorio con nombre ***public*** y en él programaremos todas las páginas web con las que queremos responder.
  - También creamos una carpeta ***img*** para las imágenes y ***css*** para los archivos **CSS**. Ver **`03-holamundo`**.

Express

- **Express.js** (o **Express**), es un framework web del lado del servidor que permite crear aplicaciones web y APIs con **Node**. Es software gratuito y de código abierto bajo la licencia MIT.
- Su autor original es TJ Holowaychuk lo describe como un proyecto inspirado **Sinatra** (framework web Ruby) ya que se trata de un framework relativamente pequeño con muchas funciones disponibles como complementos.
- Algunas características destacadas de **Express** son:
  - **Manejo de solicitudes y respuestas**: nos permite acceder de manera simple a parámetros de ruta y de consulta así como también al cuerpo de la solicitud.
  - **Enrutamiento**: permite definir cómo responderá la aplicación a diferentes rutas y métodos HTTP.
  - **Middleware**: permite ejecutar funciones antes que se maneje la solicitud enrutada.
  - **Templates**: si bien no incluye un sistema de plantillas nos permite seleccionar entre EJS, Pug ó Handlebars
  - **Gestión de sesiones**: nos permite almacenar información de sesión en el servidor así como también gestionar cookies.
  - **Manejo de errores**: Nos permite capturar y gestionar errores en nuestras aplicaciones.
  - **MVC y arquitectura modular**: facilita el patrón Modelo-Vista-Controlador.
  - **Facilidad de integración con otros módulos.**
  - **APIs REST**: es ideal para construir APIs debido a su enfoque de enrutamiento y facilidad para manejar métodos HTTP.

Primer servidor con Express

- Para construir un servidor web con **Express** seguimos los siguientes pasos:
  - Creamos una nueva carpeta o directorio. Entramos en él y abrimos VS Code.
  - Ejecutamos: **`npm init -y`** (versión interactiva **`npm init`**).
  - Instalamos **Express**: **`npm install express`**
  - Verificamos cuál es el **`main`** de nuestra aplicación en **`package.json`**. Por defecto es **`index.js`**.
  - Editamos nuestro archivo **`index.js`** para usar **Express**:


```
import express from 'express';
const app = express();
const port = 3000;
app.get('/', (req, res) => {
    res.type('text/plain');
    res.status(200);
    res.send('Hola soy una app Express!');
});
app.listen(port, () => console.log(`Express started on http://localhost:${port}`));
```

  - Para ejecutar usamos **`node index.js`** y para detener la ejecución **`Control + C`**.

Primer servidor con Express - Rutas

- **`app.get`** es un método por el cual podemos agregar nuevas rutas a nuestra aplicación.
- Si bien los más comunes son **`get`** y **`post`** con esta sintaxis podemos usar el resto de los métodos **HTTP**. Por ejemplo: **`app.post`**, **`app.put`**, **`app.delete.`**
- Los métodos reciben dos parámetros: ***ruta*** y ***función***.
- **Ruta**:
  - Si hay diferencias de mayúsculas/minúsculas, barras al final o query params la coincidencia tendrá lugar de todas maneras. Es decir, se determinan como rutas iguales: **`/jugadores, /Jugadores, /jugadores/, /jugadores?orden=dorsal, /jugadores/?orden=dorsal`**
- **Función:**
  - Recibe como parámetro dos objetos: **`request`** (solicitud) y **`response`** (respuesta).
  - Nótese que en vez de usar los métodos de **Node** estamos utilizando los de **Express**:
    - Express: **`res.send`** en vez de Node: **`res.end`**
    - Express: **`res.set`** en vez de Node: **`res.writeHead`**
    - Express: **`res.type`** para establecer el tipo de contenido de la respuesta**.**
  - No es necesario ni recomendable utilizar **`res.writeHead`** ni **`res.end`**

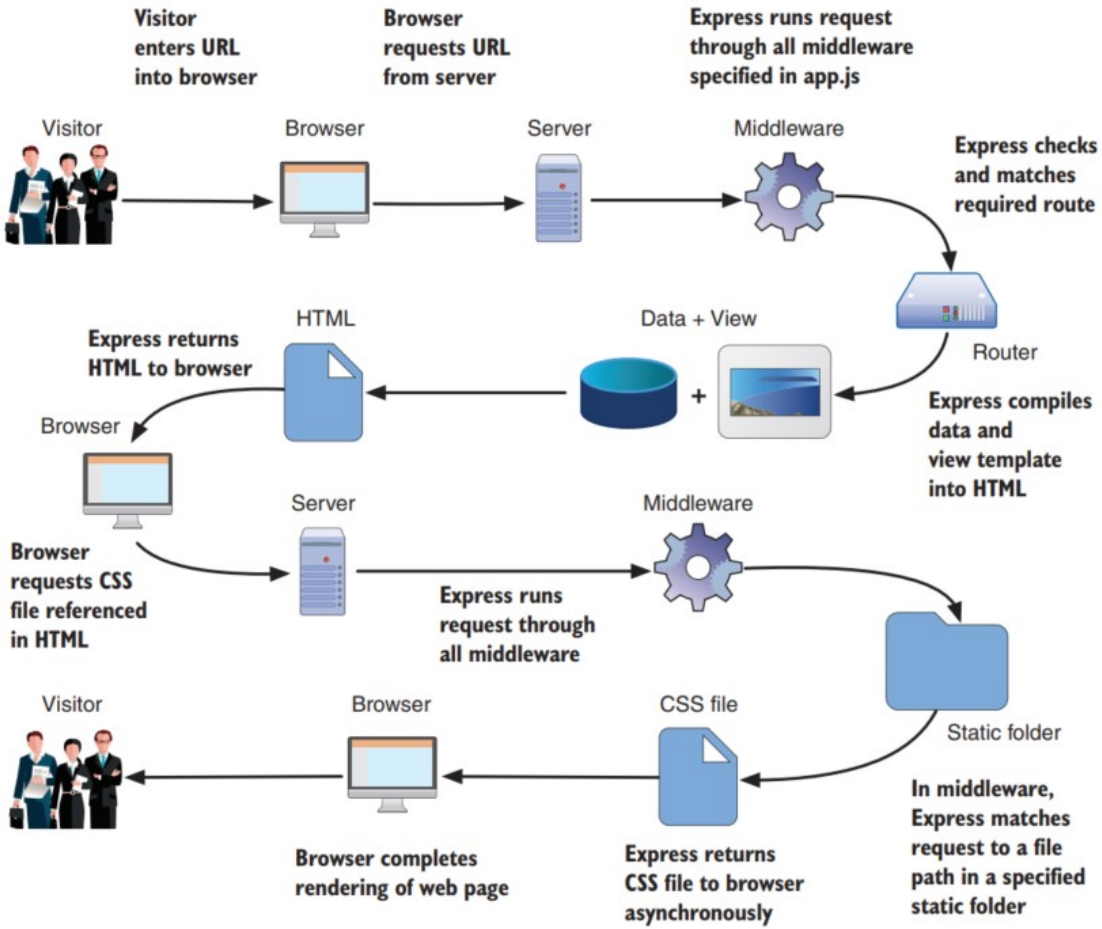
Primer servidor con Express – Rutas (2)

- Haciendo uso de **`app.use`** configuramos un ***middleware*** que nos permite entregar páginas de error.
  - Ya sea **404** (Recurso solicitado no encontrado) o **500** (Error interno del servidor).
- Desarrollaremos ***middlewares*** con mayor profundidad más adelante. Por el momento, coloquialmente podemos decir que se trata de un controlador general que gestionará una respuesta para cualquier ruta que no tuvo coincidencias con las rutas especificadas anteriormente.
- Debemos tener presente que el orden en que configuremos las rutas incide en la respuesta que va a entregar el servidor. Por tanto las rutas de error deberían ir últimas.
- Podemos utilizar el símbolo **`*`** como comodín siempre teniendo presente el orden en que se activan las rutas.

Plantillas con Handlebars

- Con lo visto hasta el momento de **Express** es muy difícil programar una aplicación web ya que no tenemos la posibilidad de estructurar documentos en **HTML** ni contamos con la ayuda del intellisense de VS Code para escribir código **HTML** dentro de los strings retornados.
- Existe una mejor alternativa de solución. Utilizar un ***motor de vistas (view engine)***. ¿Pero primero… qué es una ***vista***?
- Podemos decir que una ***vista*** es lo que se entrega a un cliente web. En el caso de un sitio web generalmente una vista es **HTML** pero es cualquier cosa que pueda ser renderizada por un navegador (JPEG, PNG, PDF, etc).
  - Para mantenerlo simple consideraremos que las vistas son en formato **HTML**.
- Una vista difiere de un recurso estático en que la vista no necesariamente tiene que ser estática. Puede ser construida a demanda de acuerdo al tipo de solicitud, entrada, etc.
- **Express** soporta múltiples tipos de motores de vistas: ***EJS***, ***Pug*** y ***Handlebars***. De esta lista ***Handlebars*** es el más fácil de aprender ya que nos permite estructurar los documentos utilizando HTML y no otro lenguaje.
- **Handlebars** está basado en **Moustache**. No nos abstrae de **HTML** sino que nos alienta a trabajar en **HTML** y usar etiquetas especiales para inyectar contenido.

Plantillas con Handlebars (2)



Plantillas con Handlebars (2)

- Para trabajar con **Handlebars** debemos:
  - Instalar el paquete **express-handlebars**: **`npm install express-handlebars`**
  - Crear en el directorio raíz de nuestro proyecto una carpeta **`views`** y dentro de ella otra que se llame **`layouts`.**
  - En **`layouts`** definiremos una plantilla general que será completada con las vistas solicitadas por el cliente.
    - Tendrá por nombre: **`main.handlebars`**.
  - En el directorio **`views`** creamos nuestras vistas en formato **HTML** y cada vez que necesitamos acceder a una variable utilizamos **`{{nombreVariable}}`**.
  - También podemos hacer uso de expresiones como lógicas que abren con **`#`** y cierran con **`/`**. Por ejemplo: **`#if - /if`**, **`#each - /each`**.
- Ejemplo de uso de **Handlebars**:

```
import express from 'express';

import expressHandlebars from 'express-handlebars';

const app = express();

const port = 3000;

app.engine('handlebars', expressHandlebars.engine({ defaultLayout: 'main' }));

app.set('view engine', 'handlebars');

app.get('/', (req, res) => res.render('inicio', { title: 'Inicio' })); //Rutas
```

Procesamiento de solicitudes HTTP

- Hasta aquí solo hemos procesado solicitudes **HTTP** de tipo **GET**. Pero no hemos procesado ningún parámetro de consulta. Procesar **query params** es útil para distintas tareas como especificar condiciones de filtrado, orden de elementos y otras cuestiones sobre el contenido que queremos visualizar.
- Accedemos a los **query params** usando **`req.query.[nombreDelParametro]`**
- **Express** también nos permite responder a solicitudes de tipo **POST** y ejecutar una acción asociada.
- La semántica de este método está asociada a agregar una nueva entidad del tipo al que hace referencia la **URL**. Por ejemplo, si estamos hablando de jugadores, hacer un **POST** a jugadores implica que queremos agregar un nuevo elemento.
- Para cumplir con este cometido usando **Express** y **Handlebars** hay varios pasos a seguir:
  - Creamos una vista que incluye un formulario. Debe poder accederse por **GET** para mostrar el formulario.
  - En la vista que incluye el formulario el elemento **`<form>`** debe tener el atributo **`method`** con valor **`POST`** y el atributo **`action`** coincidente con una URL en el servidor que espera solicitudes por este método. En caso que configuremos que la misma ruta acepta solicitudes **`GET`** y **`POST`** podemos no especificar el atributo **`action.`**
  - Posteriormente en el servidor configuramos la ruta.
  - Sin utilizar ningún **middleware** acceder a los datos es un trabajo tedioso que requiere leer los datos en chunks o partes. Por lo cual es recomendable utilizar un ***middleware*** como ***express.json()*** o similar.

Pug: Un enfoque diferente

- La mayoría de los motores de vistas o plantillas adoptan un enfoque centrado en **HTML**, **Pug** se destaca por abstraer los detalles de **HTML**.
- **Pug** es una creación de TJ Holowaychuk, la misma persona que creó **Express**. Por tanto, la integración de **Pug** con **Express** es muy buena.
- El enfoque que adopta **Pug** está en la afirmación que **HTML** es un lenguaje complicado y tedioso para escribir a mano.
- Aquí un ejemplo de cómo se ve una plantilla de **Pug**, junto con el **HTML** que generará:

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script.
      if (foo) {
         bar(1 + 5)
      }
  body

    h1 Pug
    #container
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
      p.
        Pug is a terse and
        simple templating
        language with a
        strong focus on
        performance and
        powerful features.
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Pug Demo</title>
<script>
    if (foo) {
        bar(1 + 5)
    }
</script>
<body>
<h1>Pug</h1>
<div id="container">

<p>You are amazing</p>


<p>
 Pug is a terse and
  simple templating
  language with a
  strong focus on
  performance and
  powerful features.
</p>
</body>
```

API Rest

- En **IDW** utilizamos **`fetch`** para recuperar datos desde una API externa. Llega el momento de saber cómo esas APIs son creadas.
- **REST** es el acrónimo de **RE**presentational **S**tate **T**ransfer. Define un estilo arquitectónico para diseñar sistemas distribuidos.
- El término fue introducido en la tesis doctoral de Roy Fielding en el año 2000.
- No es exactamente una arquitectura de software sino un ***conjunto de restricciones***:
  - Uniformidad.
  - Cliente – Servidor.
  - Sin estado.
  - Puede ser almacenado en una cache (cacheable).
  - Arquitectura en capas (opcional).
  - Código a demanda (opcional).

- Los términos **API REST, RESTful Web Service** y **REST Web Service** si bien no se refieren exactamente a lo mismo, son utilizados como sinónimos.

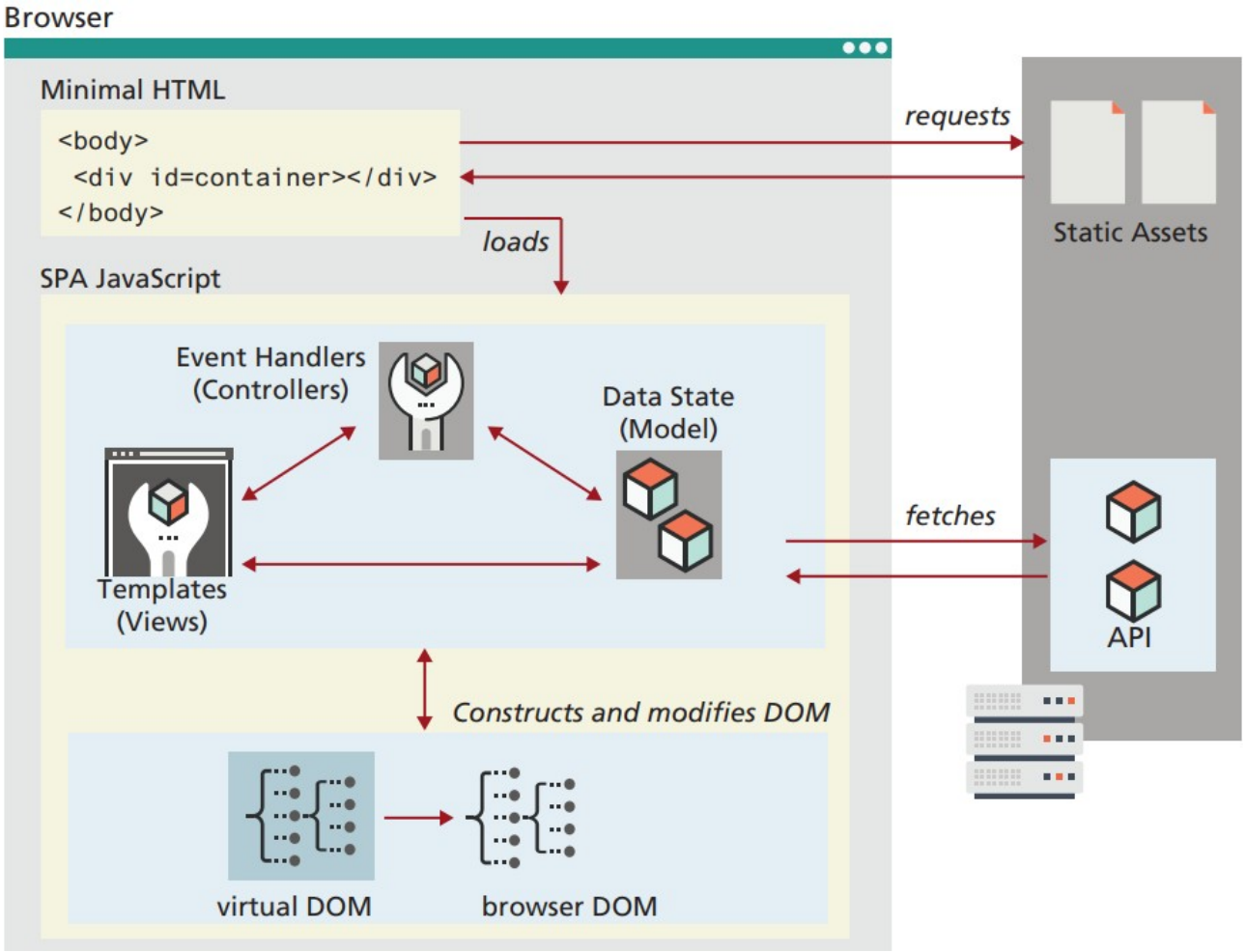
Recurso y representación

- Las abstracciones de información en **REST** se conocen con el nombre de ***Resource*** o ***Recurso***.
- Cada **recurso** debe tener un nombre que permita identificarlo.
- El ***estado de un recurso*** en un momento particular es conocido como ***Representación***.
- El formato de una **representación** se llama ***Media Type*** y determina como la representación debe ser procesada. El formato más utilizado es **JSON**.
- En el contexto de **REST**, generalmente escuchamos el término ***endpoint***.
- Un ***endpoint*** hace referencia a una **URL** específica que representa un **recurso** o una **colección de recursos**.
  - Los ***endpoints*** son ubicaciones específicas a los que se puede enviar una solicitud **HTTP** para interactuar con recursos.

API REST – Resource Methods

- Fielding no estableció una recomendación respecto de los métodos pero enfatizó que la API debe tener nombres uniformes.
- Los métodos de recurso son por lo general asociados con los métodos **HTTP**: **GET, POST, PUT, DELETE:**
  - **GET**: Recupera la información. Puede ser una colección o una única entidad.
  - **POST**: Solicita que el recurso cree una nueva entidad.
  - **PUT**: actualiza una entidad.
  - **DELETE**: remueve o elimina el elemento.

Esquema Renderizado del Lado del Cliente



Bibliografía

- **Libro:** Randy Connolly, Ricardo Hoar. ***“Fundamentals of Web Development, Global Edition”***. 3era Edition. Ed. Pearson. 2022.
- **Libro:** Ethan Brown. ***"Web Development with Node and Express"***. O’Reilly Media, Inc. 2020.
- **Libro:** Simon Holmes, Clive Harber. ***"Getting MEAN"***. 2da Edición. Manning. 2019.