Uso de plantillas HTML

En este nuevo documento, describimos cómo se utilizan las plantillas para generar contenido HTML, lo que permite que una aplicación adapte la visualización del contenido al usuario para reflejar la solicitud que se está procesando o los datos de estado de la aplicación.

Al igual que muchos de los temas descritos en este curso, las plantillas son mucho más fáciles de entender una vez que se ve cómo funcionan. Por lo tanto, comenzaremos creando un sistema de plantillas personalizado simple utilizando solo las características proporcionadas por las API de JavaScript y Node.js, solo para explicar cómo encajan las piezas.

Mostraremos las plantillas del lado del servidor, donde el servidor backend genera el contenido HTML, y las plantillas del lado del cliente, donde el navegador genera el contenido.

Las plantillas personalizadas de este documento son ‘educativas’ pero demasiado limitadas para su uso en un proyecto real, por lo que también presentamos un paquete de plantillas popular que tiene muchas más características y un rendimiento mucho mejor, y que es adecuado para su uso en un proyecto real.

La tabla 1 pone las plantillas HTML en contexto y la tabla 2 resume el documento completo.

Tabla 1: Poner las plantillas HTML en contexto.

Pregunta	Respuesta
¿Qué son?	Las plantillas HTML son documentos HTML que contienen marcadores de posición que se reemplazan con contenido dinámico para reflejar el estado de la aplicación.
¿Por qué son útiles?	Las plantillas permiten que el contenido presentado al usuario refleje cambios en el estado de la aplicación y son un componente clave en la mayoría de las aplicaciones web.
¿Cómo se utilizan?	Hay muchos paquetes de plantillas buenos disponibles y los frameworks populares generalmente incluyen un sistema de plantillas.
¿Existen limitaciones o inconvenientes?	Es importante encontrar un paquete con un formato que te resulte fácil de leer, pero, de lo contrario, los motores de plantillas son una adición positiva a un proyecto de una aplicación web.
¿Existen alternativas?	Puedes generar contenido completamente usando código JavaScript, pero esto tiende a ser difícil de mantener. Es posible que no puedas evitar el uso de plantillas si estás usando un framework, como React o Angular.

Tabla 2: Resumen del capítulo.

Problema	Solución	Listado
Representar (renderizar) dinámicamente elementos HTML	Utilizar un motor de plantillas que combine elementos HTML y expresiones que se evalúen para producir valores de datos.	1-4, 11-15, 21-27
Evaluar expresiones de plantilla	Utilizar la palabra clave eval para evaluar expresiones de cadena como declaraciones de JavaScript.	5, 6
Dividir las plantillas en contenido más manejable	Utilizar plantillas/vistas parciales.	7-10
Representar dinámicamente elementos HTML en el navegador	Compilar plantillas en código JavaScript que se incluye en el paquete cargado por el navegador.	16-20, 28-31

Preparación para este documento

Este segundo documento de la parte 2 utiliza el proyecto part2app creado previamente. No se requieren cambios para prepararse para este documento. Abre un command prompt y ejecuta el comando que se muestra en el listado 1 en la carpeta part2app para iniciar las herramientas de desarrollo.

Listado 1: Iniciando las herramientas de desarrollo.

```
npm start
```

Abre un navegador web, solicita http://localhost:5000 y haz clic en el botón Enviar solicitud. El navegador enviará una solicitud al servidor backend y mostrará detalles de los resultados, como se muestra en la figura 1.

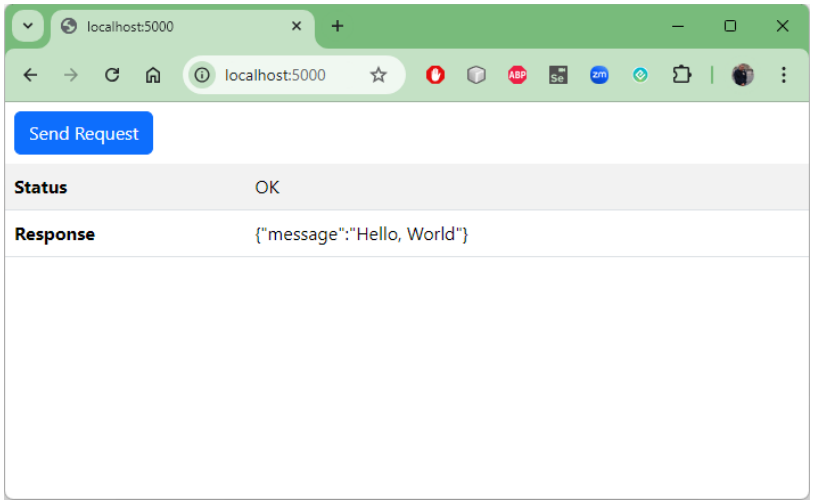

Figura 1: Ejecución de la aplicación de ejemplo.

Uso de plantillas HTML del lado del servidor

Las plantillas HTML del lado del servidor permiten que el servidor backend genere contenido de forma dinámica para enviar al navegador contenido adaptado a una solicitud individual. La adaptación puede adoptar cualquier forma, pero un ejemplo típico es incluir contenido específico para el usuario, como incluir el nombre del usuario.

Se requieren tres cosas para una plantilla HTML: un archivo de plantilla que tenga secciones de marcador de posición en las que se insertará contenido dinámico, un diccionario de datos o contexto que proporcione los valores que determinarán el contenido dinámico específico que se generará y un motor de plantilla que procese la vista y el diccionario para producir un documento HTML en el que se haya insertado contenido dinámico y que se pueda usar como respuesta a una solicitud HTTP, como se muestra en la figura 2.

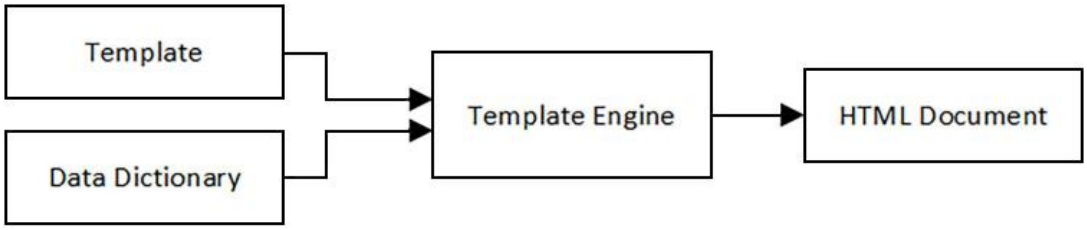

Figura 2: Los componentes de una plantilla HTML.

La tarea de procesar una plantilla se conoce como *renderizado* y ocurre completamente en el servidor backend.
El renderizado produce un documento HTML normal que, desde la perspectiva del navegador, no parece diferente del contenido estático normal. (Existe un tipo diferente de plantilla que se envía al navegador como JavaScript, donde se renderiza para crear contenido HTML por parte del cliente, como se describe en la sección uso de plantillas HTML del lado del cliente).

Creación de un motor de plantillas simple

Es fácil crear un motor de plantillas simple para ayudar a comprender cómo funcionan, aunque es mucho más difícil crear uno que esté listo para producción. En esta sección, crearemos algo simple y luego presentaremos un paquete de motor de plantillas de código abierto que es mejor, más rápido y tiene muchas más funciones.

Comenzaremos creando la plantilla, que ayudará a poner todo en contexto. Creamos la carpeta part2app/templates/server y agrega un archivo llamado basic.custom con el contenido que se muestra en el listado 2.

Sugerencia

La mayoría de los editores de código se pueden configurar para que comprendan que los archivos con una extensión no estándar, como .custom, contienen un formato conocido, como HTML. Si estás utilizando Visual Studio Code, por ejemplo, haz clic en Texto sin formato (*Plain Text*) en la esquina inferior derecha de la ventana y elije el formato para un solo archivo o configura una asociación para que todos los archivos .custom se traten como HTML, lo que facilitará la detección de errores al seguir los ejemplos.

Listado 2: El contenido del archivo basic.custom en la carpeta templates/server.

```
<!DOCTYPE html>
<html>
   <head><link href="/css/bootstrap.min.css" rel="stylesheet" /></head>
   <body>
      <h3 class="m-2">Message: {{ message }}</h3>
   </body>
</html>
```

Esta plantilla es un documento HTML completo con un marcador de posición, que se indica mediante llaves dobles (los caracteres {{ y }}). El contenido dentro de las llaves es una expresión de plantilla que se evaluará cuando se represente la plantilla y se use para reemplazar el marcador de posición.

No todos los motores de plantillas utilizan los caracteres {{ y }}, aunque es una opción popular, y lo que es importante es que es poco probable que la secuencia de caracteres que denota un marcador de posición se encuentre en las partes estáticas de la plantilla, por lo que normalmente verás secuencias de caracteres repetidos o caracteres inusuales.

Creación del motor de plantillas personalizado

El paquete Express tiene soporte integrado para motores de plantillas, lo que facilita la experimentación y el aprendizaje de cómo funcionan. Agrega un archivo llamado custom_engine.ts a la carpeta src/server con el contenido que se muestra en el listado 3.

Listado 3: El contenido del archivo custom_engine.ts en la carpeta src/server.

```
import { readFile } from "fs";
import { Express } from "express";

const renderTemplate = (path: string, context: any,
   callback: (err: any, response: string | undefined) => void) => {

   readFile(path, (err, data) => {
     if (err != undefined) {
```

```
            callback("Cannot generate content", undefined);
        } else {
            callback(undefined, parseTemplate(data.toString(), context));
        }
    });
};

const parseTemplate = (template: string, context: any) => {
    const expr = /{{(.*)}}/gm;
    return template.toString().replaceAll(expr, (match, group) => {
        return context[group.trim()] ?? "(no data)"
    });
}
export const registerCustomTemplateEngine = (expressApp: Express) =>
    expressApp.engine("custom", renderTemplate);
```

Express llamará a la función renderTemplate para renderizar una plantilla. Los parámetros son una cadena que contiene la ruta del archivo de plantilla, un objeto que proporciona datos de contexto para renderizar la plantilla y una función de devolución de llamada que se utiliza para proporcionar a Express el contenido renderizado o un error si algo sale mal.

La función renderTemplate utiliza la función readFile para leer el contenido del archivo de plantilla y luego invoca la función parseTemplate, que utiliza una expresión regular para buscar los caracteres {{ y }}. Para cada coincidencia, una función de devolución de llamada inserta un valor de datos del objeto de contexto en el resultado, de esta manera:

```
...
const expr = /{{(.*)}}/gm;
return template.toString().replaceAll(expr, (match, group) => {
    return context[group.trim()] ?? "(no data)"
});
...
```

Este es un enfoque rudimentario y los motores reales son más complejos y tienen más cuidado para encontrar expresiones de plantilla, pero esto es suficiente para demostrar la idea. La función registerCustomTemplateEngine registra el motor de plantilla con Express, lo que se hace llamando al método Express.engine, especificando la extensión del archivo y la función renderTemplate:

```
...
export const registerCustomTemplateEngine = (expressApp: Express) =>
    expressApp.engine("custom", renderTemplate);
...
```

Esta declaración le indica a Express que utilice la función renderTemplate para renderizar archivos de plantilla que tengan una extensión de archivo .custom.

Configuración del motor de plantillas personalizado

La parte final del proceso es configurar Express y crear una ruta que coincida con las solicitudes que se manejarán con una plantilla, como se muestra en el listado 4.

Listado 4: Configuración del motor de plantillas en el archivo server.ts en la carpeta src/server.

```
import { createServer } from "http";
import express, {Express } from "express";
import { testHandler } from "./testHandler";
import httpProxy from "http-proxy";
import helmet from "helmet";
import { registerCustomTemplateEngine } from "./custom_engine";

const port = 5000;

const expressApp: Express = express();

const proxy = httpProxy.createProxyServer({
  target: "http://localhost:5100", ws: true
});

registerCustomTemplateEngine(expressApp);
expressApp.set("views", "templates/server");

expressApp.use(helmet());
expressApp.use(express.json());

expressApp.get("/dynamic/:file", (req, resp) => {
   resp.render(`${req.params.file}.custom`, { message: "Hello template" });
});

expressApp.post("/test", testHandler);
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
expressApp.use((req, resp) => proxy.web(req, resp));

const server = createServer(expressApp);

server.on('upgrade', (req, socket, head) => proxy.ws(req, socket, head));
```

```
server.listen(port,
  () => console.log(`HTTP Server listening on port ${port}`));
```

Al llamar al registerCustomTemplateEngine definido en el listado 4 se configura el motor de plantillas personalizado. De manera predeterminada, Express busca archivos de plantilla en la carpeta views.

Views y view engine son nombres alternativos para plantillas y motores de plantillas, pero para mantener la terminología consistente, utilizamos el método ExpressApp.set para cambiar la ubicación del archivo de plantilla:

```
...
expressApp.set("views", "templates/server");
...
```

El conjunto completo de propiedades de configuración de Express se puede encontrar en https://expressjs.com/en/4x/api.html#app.set y la propiedad views se utiliza para especificar el directorio que contiene los archivos de plantilla.

El enrutador Express se utiliza para hacer coincidir las solicitudes que se manejarán mediante plantillas, como la siguiente:

```
...
expressApp.get("/dynamic/:file", (req, resp) => {
    resp.render(`${req.params.file}.custom`, { message: "Hello template" });
});
...
```

El método get crea una ruta que coincide con las rutas que comienzan con /dynamic y captura el siguiente segmento de ruta hasta un parámetro de ruta llamado file. El controlador de solicitudes invoca el método Response.render, que es responsable de representar una plantilla. El parámetro de ruta file se utiliza para crear el primer argumento para el método render, que es el nombre del archivo de plantilla. El segundo argumento es un objeto que proporciona al motor de plantillas datos de contexto para ayudarlo a generar contenido. En este ejemplo, el objeto de contexto define una propiedad de mensaje, cuyo valor se incluirá en la salida representada.

Para probar el motor de plantillas personalizado, utiliza un navegador para solicitar http://localhost:5000/dynamic/basic. La parte dinámica de la URL se corresponderá con la nueva ruta Express, y la parte básica corresponde al archivo basic.custom en la carpeta de

plantillas. El motor de vista personalizado procesará el archivo de plantilla y los resultados se escribirán en la respuesta, como se muestra en la figura 3.

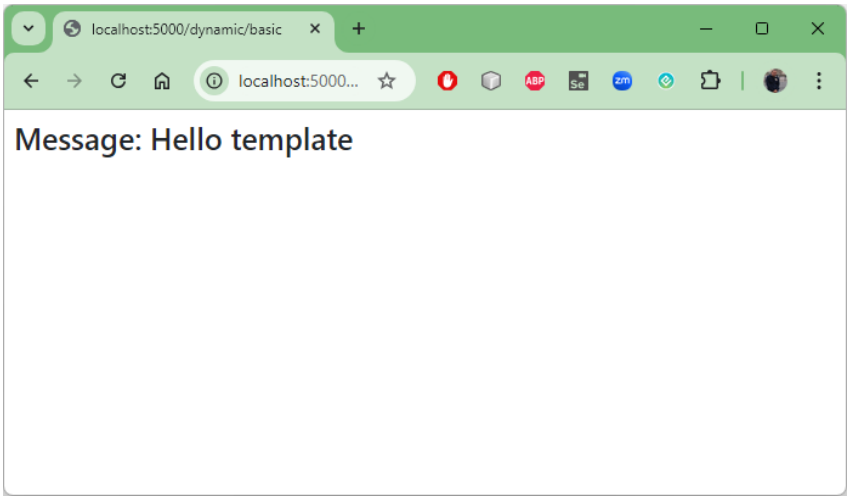

Figura 3: Uso de un motor de plantillas personalizado.

Evaluación de expresiones en plantillas

Insertar valores de datos en plantillas es un buen comienzo, pero la mayoría de los motores de plantillas admiten la evaluación de fragmentos de código JavaScript e inserción de los resultados en la salida. El listado 5 agrega algunas expresiones de plantilla a la plantilla.

Listado 5: Agregar expresiones al archivo basic.custom en la carpeta templates/server.

```
<!DOCTYPE html>
<html>
  <head><link href="/css/bootstrap.min.css" rel="stylesheet" /></head>
  <body>
    <h3 class="m-2">Message: {{ message }}</h3>
            <h3 class="m-2">Lower: {{ message.toLowerCase() }}</h3>
    <h3 class="m-2">Count: {{ 2 * 3 }}</h3>
  </body>
</html>
```

El listado 6 agrega compatibilidad para evaluar expresiones al motor de plantillas, mediante la función eval de JavaScript.

Listado 6: Evaluación de expresiones en el archivo custom_engine.ts en la carpeta src/server.

```
import { readFile } from "fs";
import { Express } from "express";

const renderTemplate = (path: string, context: any,
```

```
    callback: (err: any, response: string | undefined) => void) => {

    readFile(path, (err, data) => {
      if (err != undefined) {
          callback("Cannot generate content", undefined);
      } else {
          callback(undefined, parseTemplate(data.toString(), context));
      }
    });
};

const parseTemplate = (template: string, context: any) => {
    const ctx = Object.keys(context)
      .map((k) => `const ${k} = context.${k}`)
      .join(";");
    const expr = /{{(.*)}}/gm;
    return template.toString().replaceAll(expr, (match, group) => {
      return eval(`${ctx};${group}`);
    });
}

export const registerCustomTemplateEngine = (expressApp: Express) =>
    expressApp.engine("custom", renderTemplate);
```

Precaución

La función eval de JavaScript es peligrosa, especialmente si existe la posibilidad de que se la utilice con contenido o datos proporcionados por los usuarios, ya que se puede utilizar para ejecutar cualquier código JavaScript. Esto por sí solo es razón suficiente para utilizar un paquete de motor de plantillas bien probado, como el que se presenta en la sección uso de un paquete de plantillas.

La dificultad de utilizar eval es asegurarse de que los datos de contexto estén disponibles como variables locales al evaluar una expresión. Para asegurarnos de que los datos de contexto estén dentro del alcance, creamos una cadena para cada propiedad del objeto de contexto y combinamos esas cadenas con la expresión que se va a evaluar, de esta manera:

```
...
"const message = context.message; message.toLowerCase()"
...
```

Este enfoque garantiza que haya un valor de mensaje para que la expresión lo utilice, por ejemplo. Existen algunos peligros graves al utilizar eval, pero está bien para la aplicación de ejemplo, aunque vale la pena repetir que se debe utilizar un paquete de plantillas real en

proyectos reales, especialmente cuando se trabaja con datos proporcionados por el usuario. Utiliza una ventana del navegador para solicitar http://localhost:5000/dynamic/basic y verás los resultados que se muestran en la figura 4. (El navegador no se recargará automáticamente, por lo que deberás realizar una nueva solicitud o recargar el navegador).

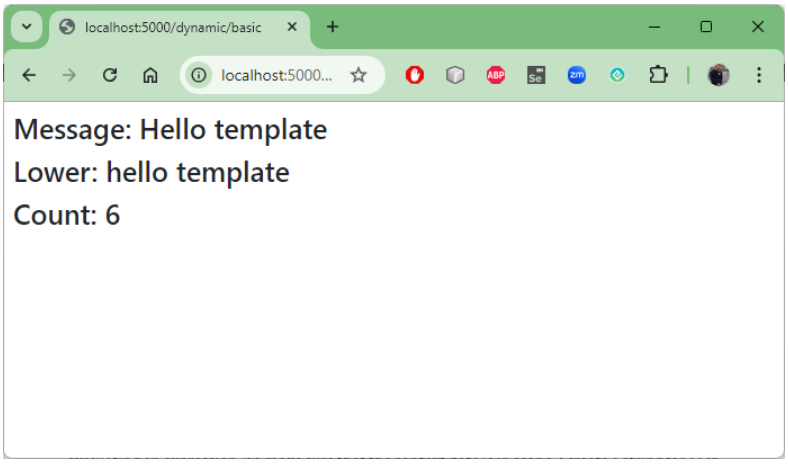

Figura 4: Evaluación de expresiones de JavaScript en una plantilla.

Adición de características de plantilla

La capacidad de evaluar expresiones proporciona una base para crear características adicionales, que se pueden escribir fácilmente como funciones de JavaScript y agregar al contexto utilizado para analizar la plantilla. Agrega un archivo llamado custom_features.ts a la carpeta src/server con el contenido que se muestra en el listado 7.

Compilación de plantillas

La mayoría de los motores de plantillas reales compilan sus plantillas, lo que significa que las plantillas se convierten en una serie de funciones de JavaScript que se pueden invocar para generar contenido. Esto no cambia el contenido que se genera, pero puede mejorar el rendimiento porque se puede crear una salida sin necesidad de leer y buscar el archivo de plantilla. Las plantillas del lado del cliente también se compilan para que las funciones de JavaScript se puedan presentar al navegador. Puedes ver un ejemplo de este proceso en la sección uso de un paquete de plantillas, más adelante en el documento.

Listado 7: El contenido del archivo custom_features.ts en la carpeta src/server.

```
import { readFileSync } from "fs";

export const style = (stylesheet: string) => {
  return `<link href="/css/${stylesheet}" rel="stylesheet" />`;
}
export const partial = (file: string, context: any) => {
```

```
    const path = `./${context.settings.views}/${file}.custom`;
    return readFileSync(path, "utf-8");
```

Este archivo define una función de estilo que acepta un nombre de hoja de estilo y devuelve un elemento de enlace. La función partial lee otro archivo de plantilla y devuelve su contenido para incluirlo en el contenido general. La función partial recibe un objeto de contexto, que utiliza para localizar el archivo solicitado:

```
...
const path = `./${context.settings.views}/${file}.custom`;
...
```

El objeto de contexto que Express proporciona al motor de plantillas tiene una propiedad de configuración, que devuelve un objeto que contiene la configuración de la aplicación. Una de las propiedades de configuración es views, que devuelve la ubicación de los archivos de plantilla (la carpeta templates/server). El listado 8 revisa la plantilla para utilizar estas nuevas funciones.

Nota

La función partial del listado 7 realiza una operación de bloqueo para leer el contenido del archivo. Como se explicó anteriormente, esto es algo que se debe evitar tanto como sea posible, y hemos utilizado la función readFileSync solo por simplicidad.

Listado 8: Uso de funciones de plantilla en el archivo basic.custom en la carpeta templates/server.

```
<!DOCTYPE html>
<html>
  <head>{{ @style("bootstrap.min.css") }}</head>
  <body>
    {{ @partial("message") }}
    <h3 class="m-2">Message: {{ message }}</h3>
    <h3 class="m-2">Lower: {{ message.toLowerCase() }}</h3>
    <h3 class="m-2">Count: {{ 2 * 3 }}</h3>
  </body>
</html>
```

Se accede a las nuevas funciones con un prefijo @, lo que facilita su búsqueda al analizar plantillas. En el listado 8, la expresión @style invocará la función de estilo para crear un elemento de enlace para el archivo CSS de Bootstrap, y la expresión @partial invocará la función parcial para cargar una plantilla llamada mensaje. Para crear la plantilla, conocida como plantilla parcial, que se cargará con la expresión @partial, crea un archivo llamado

mensaje.custom en la carpeta templates/server con el contenido que se muestra en el listado 9.

Listado 9: El contenido del mensaje.custom en la carpeta templates/server.

```
<div class="bg-primary text-white m-2 p-2">
   {{ message }}
</div>
```

Asignación de expresiones a funciones

Todo lo que queda es traducir las expresiones @ en la plantilla a instrucciones de JavaScript que invocan las funciones del listado 7. Toma la siguiente expresión:

```
...
{{ @partial("message") }}
...
```

La expresión anterior se traducirá a lo siguiente:

```
...
features.partial("message", context);
...
```

Una vez que se complete la traducción, el resultado se puede evaluar como cualquier otra expresión. El listado 10 cambia el motor de plantillas para admitir las nuevas funciones.

Listado 10: Compatibilidad con funciones de plantilla en el archivo custom_engine.ts en la carpeta src/server.

```
import { readFile } from "fs";
import { Express } from "express";
import * as features from "./custom_features";

const renderTemplate = (path: string, context: any,
   callback: (err: any, response: string | undefined) => void) => {

   readFile(path, (err, data) => {
     if (err != undefined) {
          callback("Cannot generate content", undefined);
     } else {
          callback(undefined, parseTemplate(data.toString(),
            { ...context, features }));
     }
   });
```

```
};

const parseTemplate = (template: string, context: any) => {
   const ctx = Object.keys(context)
     .map((k) => `const ${k} = context.${k}`)
     .join(";");
   const expr = /{{(.*)}}/gm;
   return template.toString().replaceAll(expr, (match, group) => {
     const evalFunc= (expr: string) => {
        return eval(`${ctx};${expr}`)
     }
     try {
        if (group.trim()[0] === "@") {
          group = `features.${group.trim().substring(1)}`;
          group = group.replace(/\)$/m, '', context, evalFunc)");
        }
        let result = evalFunc(group);
        if (expr.test(result)) {
          result = parseTemplate(result, context);
        }
        return result;
     } catch (err: any) {
        return err;
     }
  });
}

export const registerCustomTemplateEngine = (expressApp: Express) =>
   expressApp.engine("custom", renderTemplate);
```

Las funciones definidas en el listado 7 se importan y se les asigna el prefijo features. La manipulación de cadenas realiza la traducción de la expresión @ al nombre de la función, con la adición de la propiedad context y una función eval. Esto permite que las expresiones accedan al objeto de contexto, las configuraciones que incluye y la capacidad de evaluar expresiones con el contexto.

El resultado de una función @ puede contener otras expresiones de plantilla; por lo tanto, la expresión regular se utiliza para analizar recursivamente el resultado.

Utiliza un navegador para solicitar http://localhost:5000/dynamic/basic y verás el resultado que producen las nuevas características, como se muestra en la figura 5.

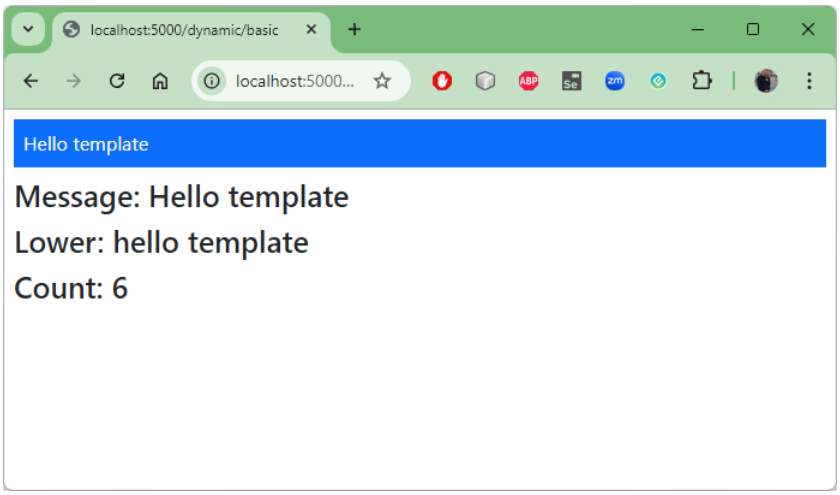

Figura 5: Agregar características de plantilla.

Uso de plantillas para crear una aplicación de ida y vuelta simple

El motor de plantillas es simple, pero tiene la funcionalidad suficiente para crear una aplicación básica que modifica el HTML que muestra en función de la interacción del usuario, que es la función clave de cualquier aplicación web. Para demostrarlo, vamos a presentar al usuario un botón que incrementa un contador, donde el valor del contador dará como resultado que se presente un contenido diferente al usuario.

Este es un ejemplo de una aplicación de ida y vuelta (*round-trip*), donde cada interacción requiere una solicitud HTTP al servidor para obtener un nuevo documento HTML para mostrar al usuario.

El primer paso es agregar el objeto que representa la solicitud HTTP a los datos de contexto proporcionados al motor de plantillas personalizado, como se muestra en el listado 11.

Listado 11: Adición de datos de contexto en el archivo server.ts en la carpeta src/server.

```
…
expressApp.get("/dynamic/:file", (req, resp) => {
    resp.render(`${req.params.file}.custom`, {
        message: "Hello template", req
    });
});
...
```

A continuación, agrega un archivo llamado counter.custom a la carpeta templates/server con el contenido que se muestra en el listado 12.

Listado 12: El contenido del archivo counter.custom en la carpeta templates/server.

```
<!DOCTYPE html>
<html>
  <head>{{ @style("bootstrap.min.css") }}</head>
  <body>
    <a class="btn btn-primary m-2"
      href="/dynamic/counter?c={{ Number(req.query.c ?? 0) + 1}}">
        Increment
    </a>
    <div>
      {{ @conditional("(req.query.c ?? 0) % 2", "odd", "even") }}
    </div>
  </body>
</html>
```

Esta plantilla contiene un elemento de anclaje (la etiqueta a) que, cuando se hace clic en él, solicita un nuevo documento HTML del servidor backend mediante una URL que contiene un parámetro de cadena de consulta llamado c. El valor de c incluido en la URL de solicitud siempre es uno más que el valor que se muestra al usuario, de modo que hacer clic en el botón tiene el efecto de incrementar el contador.

La plantilla contiene una expresión @conditional, que se utilizará para representar diferentes plantillas parciales para valores pares e impares de c. Los argumentos de @conditional son una expresión que se evaluará y dos nombres de plantillas parciales que se utilizarán para los resultados verdaderos y falsos cuando se evalúe la expresión.

Para crear la plantilla partial que se utilizará para los valores impares, agrega un archivo llamado odd.custom a la carpeta templates/server con el contenido que se muestra en el listado 13.

Listado 13: El contenido del archivo odd.custom en la carpeta templates/server.

```
<h4 class="bg-primary text-white m-2 p-2">
  Odd value: {{ req.query.c ?? 0}}
</h4>
```

Para crear la plantilla partial que se utilizará para los valores pares, agrega un archivo llamado even.custom a la carpeta templates/server con el contenido que se muestra en el listado 14.

Listado 14: El contenido del archivo even.custom en la carpeta templates/server.

```
<h4 class="bg-secondary text-white m-2 p-2">
  Even value: {{ req.query.c ?? 0}}
</h4>
```

El paso restante es implementar la expresión @conditional como una característica de plantilla, como se muestra en el listado 15.

Listado 15: Agregar una característica condicional en el archivo custom_features.ts en src/server.

```
import { readFileSync } from "fs";

export const style = (stylesheet: string) => {
  return `<link href="/css/${stylesheet}" rel="stylesheet" />`;
}
export const partial = (file: string, context: any) => {
  const path = `./${context.settings.views}/${file}.custom`;
  return readFileSync(path, "utf-8");
}
export const conditional = (expression: string,
  trueFile: string, falseFile: string, context: any,
  evalFunc: (expr: string) => any) => {
  return partial(evalFunc(expression) ? trueFile : falseFile, context);
}
```

La función condicional acepta una expresión, dos rutas de archivo, un objeto de contexto y una función utilizada para evaluar expresiones. La expresión se evalúa y el resultado se pasa a la función parcial, seleccionando efectivamente una vista parcial en función de si la expresión se evaluó como verdadera o falsa.

Usa un navegador para solicitar http://localhost:5000/dynamic/counter y haz clic en el botón Incrementar. Cada clic hace que el navegador solicite una URL como http://localhost:5000/dynamic/counter?c=1 y el valor de c se utiliza para seleccionar el contenido HTML en la respuesta, como se muestra en la figura 6.

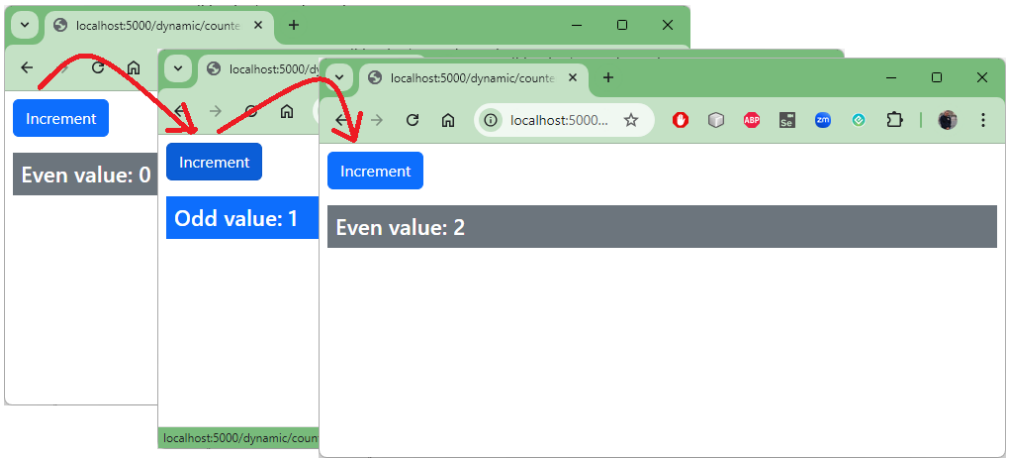

Figura 6: Uso de plantillas para crear una aplicación simple de ida y vuelta.

Uso de plantillas HTML del lado del cliente

Un inconveniente del ejemplo anterior es que se genera un documento HTML completamente nuevo y se envía al navegador cada vez que se hace clic en Increment, aunque solo cambia una sección del HTML.

Las plantillas HTML del lado del cliente realizan la misma tarea que sus contrapartes del lado del servidor, pero la plantilla es analizada por el código JavaScript que se ejecuta en el navegador. Esto permite un enfoque dirigido, donde se modifican los elementos seleccionados, lo que puede ser más receptivo que esperar un nuevo documento HTML. Esta es la base de las aplicaciones de una sola página (SPA, *single-page applications*), donde se entrega un solo documento HTML al cliente y luego se modifica mediante código JavaScript.

La principal dificultad de las plantillas del lado del cliente es que deben escribirse completamente en JavaScript, lo que puede dificultar la expresión de contenido HTML de una manera que sea fácil de leer y mantener.

Los frameworks del lado del cliente más populares, como React y Angular, utilizan formatos de plantilla del lado del cliente que son más fáciles de leer que JavaScript puro, pero utilizan un compilador para transformar la plantilla en una función de JavaScript para que pueda agregarse al paquete de JavaScript proporcionado al navegador.

Las plantillas utilizadas por los grandes frameworks tienen otros beneficios, como facilitar la combinación de plantillas para crear contenido complejo y garantizar que las actualizaciones de los elementos HTML se realicen de la manera más eficiente posible.

Pero, dejando de lado estas características, el proceso de generación de contenido en el cliente es similar a hacerlo en el servidor. Una buena forma de comprender los problemas involucrados en la creación de plantillas del lado del cliente es recrear el ejemplo de contraparte de la sección anterior utilizando JavaScript del lado del cliente. Para comenzar, agrega un archivo llamado counter_custom.js a la carpeta src/client con el contenido que se muestra en el listado 16.

Listado 16: El contenido del archivo counter_custom.js en la carpeta src/client.

```
import { Odd } from "./odd_custom";
import { Even } from "./even_custom";

export const Counter = (context) => `
  <button class="btn btn-primary m-2" action="incrementCounter">
     Increment
  </button>
```

```
<div>
  ${ context.counter % 2 ? Odd(context) : Even(context) }
</div>`
```

Plantillas del lado del cliente versus del lado del servidor

La mayoría de los proyectos de aplicaciones web tienden a combinar plantillas del lado del servidor y del lado del cliente porque cada tipo de plantilla resuelve un problema diferente.

Las plantillas del lado del servidor requieren una conexión HTTP para cada documento HTML, lo que puede afectar el rendimiento. Sin embargo, el déficit de rendimiento se puede compensar con la rapidez con la que el navegador puede mostrar el contenido del documento HTML una vez que lo ha recibido.

Las plantillas del lado del cliente responden a los cambios de manera más eficiente y sin la necesidad de realizar solicitudes HTTP adicionales, pero esta ventaja puede verse socavada por la necesidad de transferir el código JavaScript y los datos de estado en primer lugar. Cuando se utiliza un framework como React o Angular, también se debe transferir el JavaScript para el framework, y esto puede ser una barrera en regiones donde son comunes los dispositivos menos capaces y las redes poco confiables.

Para cerrar la brecha y brindar lo mejor de ambos mundos, algunos frameworks ofrecen renderizado del lado del servidor (SSR, *server-side rendering*), donde las plantillas se renderizan en el servidor para crear una versión de ida y vuelta de la aplicación, que el navegador puede mostrar rápidamente. Una vez que se muestra el contenido generado por el servidor, el navegador solicita el código JavaScript y pasa a una aplicación de una sola página. SSR ha mejorado en los últimos años, pero aún es complicado y no se adapta a todos los proyectos.

Las plantillas en este ejemplo son funciones de JavaScript que devuelven cadenas HTML, que es la forma más sencilla de crear una plantilla del lado del cliente y no requiere un compilador. Las funciones de plantilla de JavaScript recibirán un parámetro de contexto que contiene el estado actual de la aplicación.

Las características de cadena de JavaScript facilitan la inserción de valores de datos en cadenas HTML. En este caso, el valor de la propiedad counter en el objeto de contexto recibido por la función se utiliza para elegir entre las funciones Odd y Even, que es un enfoque más simple que la funcionalidad equivalente en el ejemplo de plantilla del lado del servidor.

Un problema con este enfoque es que el manejo de eventos de elementos puede ser difícil. No solo la política de seguridad de contenido de la aplicación de ejemplo impide los controladores de eventos en línea, sino que también puede ser difícil definir funciones de controlador que utilicen datos de contexto en cadenas HTML.

Para solucionar esta limitación, hemos agregado un atributo de acción al elemento de botón en el listado 16, al que se le asigna el valor incrementCounter. Los eventos del botón podrán propagarse por el documento HTML y utilizaremos el valor del atributo action para decidir cómo responder.

Para crear la vista parcial que mostrará los valores pares, agrega un archivo llamado even_custom.js a la carpeta src/client con el contenido que se muestra en el listado 17.

Listado 17: El contenido del archivo even_custom.js en la carpeta src/client.

```
export const Even = (context) => `
  <h4 class="bg-secondary text-white m-2 p-2">
    Even value: ${ context.counter }
  </h4>`
```

La cadena HTML devuelta por esta función incluye el valor de la propiedad counter.couter. Para crear la plantilla para los valores impares, crea un archivo llamado odd_custom.js en la carpeta src/client con el contenido que se muestra en el listado 18.

Listado 18: El contenido del archivo odd_custom.js en la carpeta src/client.

```
export const Odd = (context) => `
  <h4 class="bg-primary text-white m-2 p-2">
    Odd value: ${ context.counter }
  </h4>`
```

El listado 19 reemplaza el código en el archivo client.js para utilizar las nuevas funciones de plantilla y definir las características que requieren.

Listado 19: Reemplazo del contenido del archivo client.js en la carpeta src/client.

```
import { Counter } from "./counter_custom";

const context = {
  counter: 0
}

const actions = {
  incrementCounter: () => {
    context.counter++; render();
  }
}

const render = () => {
  document.getElementById("target").innerHTML = Counter(context);
}
```

```
document.addEventListener('DOMContentLoaded', () => {
  document.onclick = (ev) => {
    const action = ev.target.getAttribute("action")
    if (action && actions[action]) {
      actions[action]()
    }
  }
  render();
});
```

Cuando se emite el evento DOMContentLoaded, que indica que el navegador ha terminado de analizar el documento HTML, se crea un detector de eventos para los eventos de clic y se invoca la función render.

La función render invoca la función de plantilla Counter y utiliza la cadena HTML que recibe para establecer el contenido de un elemento HTML cuyo id es target. Cuando se recibe un evento de clic, se verifica el target del evento en busca de un atributo action y su valor se utiliza para seleccionar una función para ejecutar desde el objeto de acciones. Hay una acción en el ejemplo, que incrementa la propiedad counter del objeto de contexto y llama a la función render para actualizar el contenido presentado al usuario.

El paso final es eliminar el contenido existente del documento HTML estático y crear el elemento que se completará con el contenido de la plantilla del lado del cliente, como se muestra en el listado 20.

Listado 20: Preparación del documento HTML en el archivo index.html en la carpeta static.

```
<!DOCTYPE html>
<html>
 <head>
    <script src="/bundle.js"></script>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
 </head>
  <body>
    <div id="target"></div>
    <!-- <button id="btn" class="btn btn-primary m-2">Send Request</button>
    <table class="table table-striped">
      <tbody>
        <tr><th>Status</th><td id="msg"></td></tr>
        <tr><th>Response</th><td id="body"></td></tr>
      </tbody>
    </table> -->
  </body>
```

```
</html>
```

Utiliza un navegador para solicitar http://localhost:5000 y se te presentará el mismo contenido producido por las plantillas del lado del servidor. La diferencia es que cuando se hace clic en el botón Incrementar, el cambio de estado se maneja mediante la representación de las plantillas del lado del cliente, como se muestra en la figura 7, sin la necesidad de solicitar un nuevo documento HTML del servidor backend.

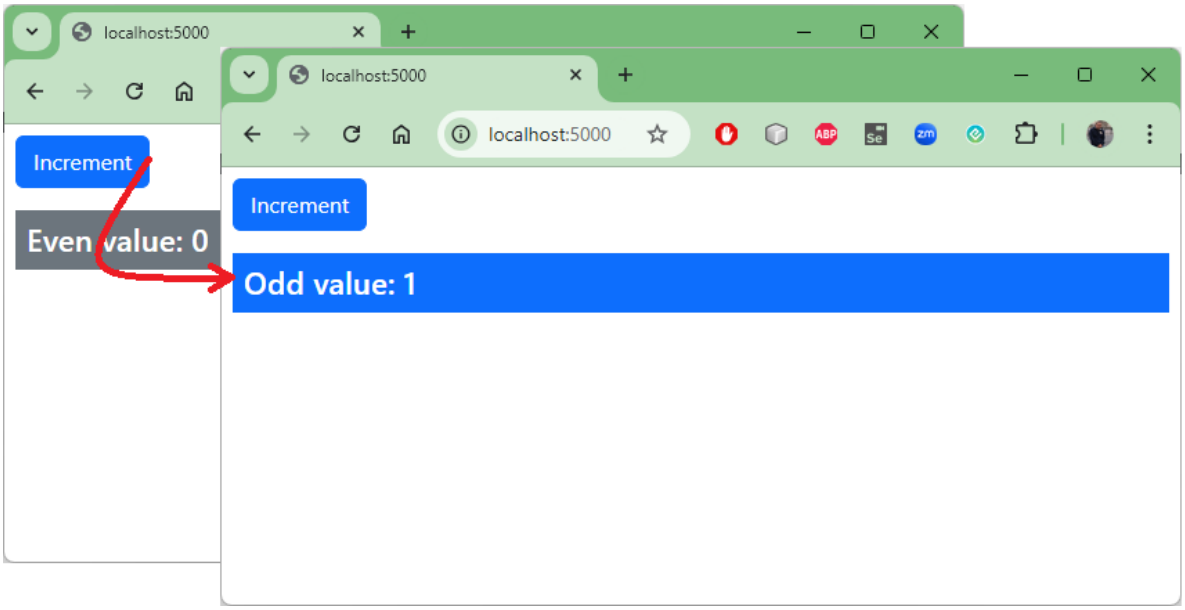

Figura 7: Uso de plantillas simples del lado del cliente

Uso de un paquete de plantillas

Los ejemplos hasta ahora en este documento han demostrado cómo se pueden utilizar las plantillas para representar contenido y muestran cómo se pueden crear fácilmente algunas características básicas. Para proyectos reales, tiene más sentido adoptar uno de los excelentes paquetes de plantillas disponibles para JavaScript. Ejecuta los comandos que se muestran en el listado 21 en la carpeta part2app para instalar uno de los paquetes de plantillas más utilizados, que se llama Handlebars, y un paquete que lo integra en Express.

Listado 21: Instalación de un paquete de plantillas.

```
npm install handlebars@4.7.8
npm install express-handlebars@7.1.2
```

Hay muchos paquetes de plantillas disponibles y todos ofrecen características similares. La principal diferencia entre los paquetes es la forma en que se escriben las plantillas y cómo se

denotan las expresiones. Los caracteres {{ y }} son una forma común de denotar expresiones y se conocen como plantillas de bigote porque las llaves recuerdan a un bigote. El paquete Handlebars (https://handlebarsjs.com) usa este estilo de expresión, como sugiere su nombre. Este es el estilo de plantilla de JavaScript al que debemos acostumbrarnos (según sugieren los expertos) y la familiaridad es muy importante a la hora de elegir un paquete de plantillas.

Nota

Hay otras opciones si no te gustan las plantillas de estilo bigote. El paquete Pug (https://pugjs.org) se basa en la sangría para estructurar las plantillas, que es una opción popular, y el paquete Embedded JavaScript (EJS) (https://ejs.co) usa secuencias <% y %>. Dejando de lado las preferencias estilísticas, todos estos paquetes están bien escritos y tienen buenos niveles de soporte.

Uso de un paquete para plantillas del lado del servidor

Las plantillas Handlebars no tienen lógica, lo que significa que no pueden contener fragmentos de JavaScript que se evalúen para producir contenido. En su lugar, se definen funciones auxiliares para implementar la lógica necesaria para generar contenido. Agrega un archivo llamado template_helpers.ts a la carpeta src/server con el contenido que se muestra en el listado 22.

Listado 22: El contenido del archivo template_helpers.ts en la carpeta src/server.

```
export const style = (stylesheet: any) => {
  return `<link href="/css/${stylesheet}" rel="stylesheet" />`;
}

export const valueOrZero = (value: any) => {
  return value !== undefined ? value : 0;
}

export const increment = (value: any) => {
  return Number(valueOrZero(value)) + 1;
}

export const isOdd = (value: any) => {
  return Number(valueOrZero(value)) % 2;
}
```

La función style acepta el nombre de una hoja de estilo y genera un elemento de enlace para ella. La función valueOrZero verifica si un valor está definido y, si no lo está, devuelve cero. La función increment incrementa un valor. La función isOdd devuelve verdadero si un valor es impar.

Definición de las plantillas

El paquete que integra Handlebars en Express admite diseños, que son plantillas que contienen los elementos comunes que, de lo contrario, se repetirían en todas las plantillas. Crea la carpeta templates/server/layouts y agrégale un archivo llamado main.handlebars con el contenido que se muestra en el listado 23.

Listado 23: El contenido del archivo main.handlebars en la carpeta templates/server/layouts.

```
<!DOCTYPE html>
<html>
  <head>
    {{{ style "bootstrap.min.css" }}}
  </head>
  <body>
    {{{ body }}}
  </body>
</html>
```

Hay dos expresiones en este diseño. La primera invoca la función auxiliar de estilo definida en el listado 22, utilizando la cadena bootstrap.min.css como argumento (los argumentos para los auxiliares están separados por espacios y no por paréntesis). La otra expresión es body, en el que se inserta la plantilla de contenidos que se ha solicitado.

Las expresiones en el diseño se indican con llaves triples ({{{ y }}}), que indican a Handlebars que los resultados deben insertarse en la plantilla sin que se escapen para la seguridad del HTML. Se debe tener cuidado al tratar con datos que se han recibido de los usuarios, y la mayoría de los motores de plantillas dan formato automáticamente al contenido para que el navegador no lo interprete como HTML. Una secuencia de tres llaves indica a Handlebars que el resultado debe pasarse sin formato, lo que es necesario cuando una expresión produce HTML.

Para crear la plantilla principal del lado del servidor para el proyecto de ejemplo, agrega un archivo llamado counter.handlebars a la carpeta templates/server con el contenido que se muestra en el listado 24.

Listado 24: El contenido de counter.handlebars en la carpeta templates/server.

```
<a class="btn btn-primary m-2"
   href="/dynamic/counter?c={{ increment req.query.c }}">
     Increment
</a>

{{#if (isOdd req.query.c) }}
```

```
    {{> odd }}
{{else}}
    {{> even }}
{{/if}}
```

Esta es la plantilla más compleja requerida por el ejemplo. El asistente de incremento se utiliza para crear la URL que el navegador solicitará cuando se haga clic en el elemento de anclaje:

```
...
href="/dynamic/counter?c={{ increment req.query.c }}">
...
```

Las llaves dobles indican una expresión de plantilla que se puede formatear para la seguridad HTML. Esta expresión invoca el asistente de incremento y utiliza el valor del parámetro de consulta como argumento. El asistente incrementará el valor que recibe y el resultado se incluirá en el valor del atributo href del elemento de anclaje.

Las otras expresiones son más complejas. Primero, hay una expresión if/else, como esta:

```
...
{{#if (isOdd req.query.c) }}
    {{> odd }}
{{else}}
    {{> even }}
{{/if}}
...
```

Se evalúa la expresión #if y el resultado se utiliza para determinar si el contenido del primer o segundo bloque se incluye en el resultado. En este ejemplo, los resultados aplican otras expresiones de plantilla:

```
...
{{#if (isOdd req.query.c) }}
    {{> odd }}
{{else}}
    {{> even }}
{{/if}}
...
```

El carácter > indica al motor de plantillas que cargue una plantilla parcial. Si la expresión #if es verdadera, se utilizará la parte impar; de lo contrario, se utilizará la parte par. La tabla 3 describe las características de plantilla más útiles.

Nota

No debes insertar un espacio (ni ningún otro carácter) entre la secuencia {{ y el resto de la expresión; de lo contrario, el motor de plantillas informará un error. Por lo tanto, {{/if}} está bien, pero {{ /if }} no funcionará.

Tabla 3: Características útiles de la plantilla.

Nombre	Descripción
{{#if val}	El contenido se incluirá en la salida si el valor de la expresión es verdadero. También hay una cláusula {{else}} que se puede utilizar para crear un efecto if/then/else.
{{#unless val}}	El contenido se incluirá en la salida si el valor de la expresión es falso.
{{> partial}}	Esta expresión inserta la plantilla parcial especificada en el resultado.
{{each arr}}	Esta expresión repite un conjunto de elementos para cada elemento de un arreglo, como se demuestra más adelante.

Para crear la plantilla partial para valores pares, crea la carpeta templates/server/partials y agrégale un archivo llamado even.handlebars con el contenido que se muestra en el listado 25.

Listado 25: El contenido del archivo even.handlebars en la carpeta templates/server/partials.

```
<h4 class="bg-secondary text-white m-2 p-2">
   Handlebars Even value: {{ valueOrZero req.query.c }}
</h4>
```

La plantilla partial contiene una expresión que utiliza el asistente valueOrZero para mostrar el valor c de la cadena de consulta o cero si no hay ningún valor. Agrega un archivo llamado odd.handlebars a la carpeta templates/server/partials con el contenido que se muestra en el listado 26.

Listado 26: El contenido del archivo odd.handlebars en la carpeta templates/server/partials.

```
<h4 class="bg-primary text-white m-2 p-2">
   Handlebars Odd value: {{ valueOrZero req.query.c}}
</h4>
```

Hay otras formas de recrear el ejemplo utilizando Handlebars, que tiene algunas características excelentes, pero este enfoque es el que más se asemeja al motor personalizado

que se creó anteriormente. El paso final es configurar la aplicación para utilizar Handlebars, como se muestra en el listado 27.

Listado 27: Configuración del motor de plantillas en el archivo server.ts en la carpeta src/server.

```
import express, {Express } from "express";
import { testHandler } from "./testHandler";
import httpProxy from "http-proxy";
import helmet from "helmet";
//import { registerCustomTemplateEngine } from "./custom_engine";
import { engine } from "express-handlebars";
import * as helpers from "./template_helpers";

const port = 5000;

const expressApp: Express = express();

const proxy = httpProxy.createProxyServer({
  target: "http://localhost:5100", ws: true
});

//registerCustomTemplateEngine(expressApp);
expressApp.set("views", "templates/server");

expressApp.engine("handlebars", engine());
expressApp.set("view engine", "handlebars");

expressApp.use(helmet());
expressApp.use(express.json());

expressApp.get("/dynamic/:file", (req, resp) => {
  resp.render(`${req.params.file}.handlebars`,
    { message: "Hello template", req,
      helpers: { ...helpers }
    });
});

expressApp.post("/test", testHandler);
expressApp.use(express.static("static"));
expressApp.use(express.static("node_modules/bootstrap/dist"));
expressApp.use((req, resp) => proxy.web(req, resp));

const server = createServer(expressApp);
```

```
server.on('upgrade', (req, socket, head) => proxy.ws(req, socket, head));

server.listen(port,
  () => console.log(`HTTP Server listening on port ${port}`));
```

El paquete express-handlebars se utiliza para integrar el motor de plantillas Handlebars en Express. Una diferencia es que las funciones de ayuda se agregan al objeto de contexto que se utiliza para representar la plantilla, pero por lo demás, la configuración es similar al motor personalizado.

Nota

La integración de Handlebars con Express proporciona soporte para proporcionar valores de datos adicionales, conocidos como variables locales, fuera de la llamada al método de representación. Más adelante, mostraremos el uso de esta función para incluir detalles de autenticación en la plantilla.

Utiliza un navegador para solicitar http://localhost:5000/dynamic/counter y verás la aplicación de ida y vuelta, pero representada por un paquete de plantillas real, con la adición de la palabra "Handlebars" en las plantillas parciales para enfatizar el cambio, como se muestra en la figura 8.

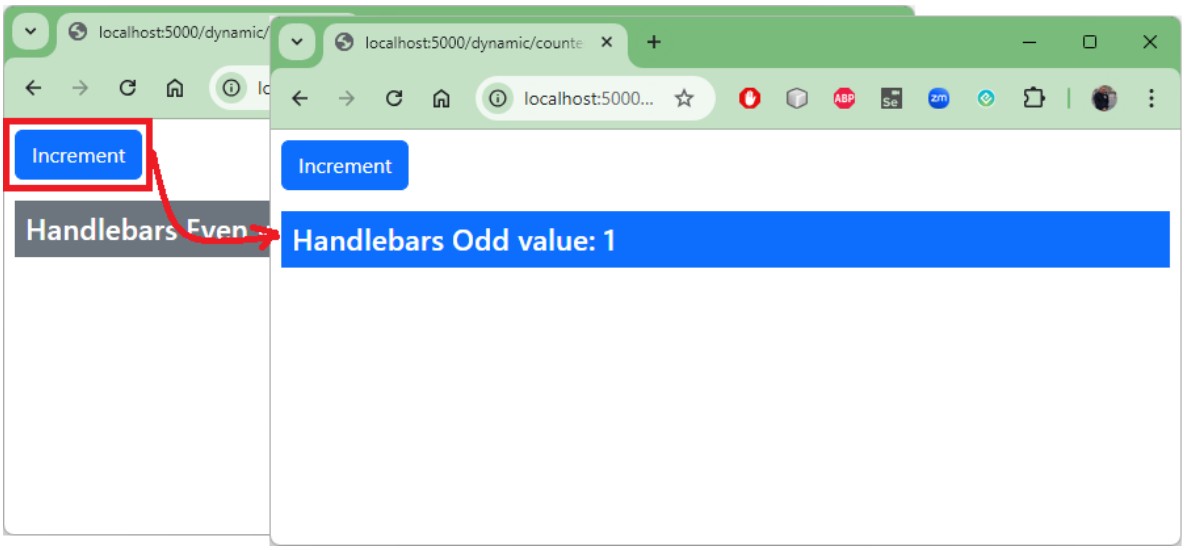

Figura 8: Uso de un paquete para plantillas del lado del servidor.

Uso de un paquete para plantillas del lado del cliente

Muchos paquetes de plantillas también se pueden usar en el navegador para crear plantillas del lado del cliente, pero esto requiere envolver las plantillas en elementos de script en los documentos HTML enviados al navegador, lo cual es complicado de hacer. Por este motivo,

la mayoría de los paquetes de plantillas ofrecen integraciones con herramientas de compilación y empaquetadores populares, como webpack, que compilan plantillas en código JavaScript. Ejecuta el comando que se muestra en el listado 28 en la carpeta part2app para agregar un paquete que integre Handlebars en webpack.

Listado 28: Instalación de un paquete de integración.

```
npm install --save-dev handlebars-loader@1.7.3
```

Se requiere un cambio en el archivo de configuración de webpack para agregar compatibilidad con la compilación de plantillas Handlebars, como se muestra en el listado 29.

Listado 29: Cambio de la configuración en el archivo webpack.config.mjs en la carpeta part2app.

```
import path from "path";
import { fileURLToPath } from 'url';

const __dirname = path.dirname(fileURLToPath(import.meta.url));

export default {
  mode: "development",
  entry: "./src/client/client.js",
  devtool: "source-map",
  output: {
    path: path.resolve(__dirname, "dist/client"),
    filename: "bundle.js"
  },
  devServer: {
    static: ["./static"],
    port: 5100,
    client: { webSocketURL: "http://localhost:5000/ws" }
  },
  module: {
    rules: [
      { test: /\.handlebars$/, loader: "handlebars-loader" }
    ]
  },
  resolve: {
    alias: {
      "@templates": path.resolve(__dirname, "templates/client")
    }
  }
};
```

La sección de configuración de module agrega soporte para procesar plantillas Handlebars. La sección resolve crea un alias para que los archivos JavaScript creados a partir de plantillas se puedan importar con @templates, en lugar de usar una ruta relativa en una declaración de importación.

Webpack no detecta cambios en su archivo de configuración, por lo que detén las herramientas de compilación y ejecuta el comando npm start nuevamente para que la nueva configuración surta efecto.

Para definir la plantilla del lado del cliente, agrega un archivo llamado counter_client.handlebars en la carpeta templates/client con el contenido que se muestra en el listado 30.

Listado 30: El contenido del archivo counter_client.handlebars en la carpeta templates/client.

```
<button class="btn btn-primary m-2" action="incrementCounter">
   Increment
</button>
<div>
   {{#if (isOdd counter) }}
      <h4 class="bg-primary text-white m-2 p-2">
         Client Odd Value: {{ counter }}
      </h4>
   {{else}}
      <h4 class="bg-secondary text-white m-2 p-2">
         Client Even Value: {{ counter }}
      </h4>
   {{/if}}
</div>
```

Todas las características de Handlebars están disponibles en las plantillas del lado del cliente, incluidas las plantillas partial, pero hemos combinado todo para simplificar. Las restricciones de la política de seguridad de contenido en los controladores de eventos en línea aún se aplican, por lo que hemos utilizado el atributo de acción en el elemento del botón para identificar qué acción se debe realizar cuando se hace clic en el botón.

Solo se requiere un asistente en el lado del cliente. Agrega un archivo llamado isOdd.js a la carpeta templates/client con el contenido que se muestra en el listado 31.

Listado 31: El contenido del archivo isOdd.js en la carpeta templates/client.

```
export default (value) => value % 2;
```

La ubicación de los archivos está especificada por el paquete del cargador handlebars y la configuración predeterminada tiene funciones auxiliares de plantilla definidas en archivos individuales con el nombre del asistente utilizado como nombre de archivo, junto con las plantillas que las utilizan. El listado 32 actualiza el archivo client.js para utilizar la plantilla Handlebars.

Listado 32: Uso de una plantilla en el archivo client.js en la carpeta src/client.

```
//import { Counter } from "./counter_custom";
import * as Counter from "@templates/counter_client.handlebars";

const context = {
  counter: 0
}

const actions = {
  incrementCounter: () => {
    context.counter++; render();
  }
}

const render = () => {
  document.getElementById("target").innerHTML = Counter(context);
}

document.addEventListener('DOMContentLoaded', () => {
  document.onclick = (ev) => {
    const action = ev.target.getAttribute("action")
    if (action && actions[action]) {
      actions[action]()
    }
  }
  render();
});
```

La plantilla compilada es un reemplazo directo de la función personalizada que definimos anteriormente en el documento. Cuando webpack crea el paquete del lado del cliente, los archivos de plantilla Handlebars se compilan en JavaScript. (Puedes recibir un error de compilación cuando guardes los cambios en el listado 32. Si eso sucede, detén las herramientas de desarrollo y vuelve a iniciarlas utilizando el comando npm start).

Utiliza un navegador para solicitar http://localhost:5000 y verás la aplicación del lado del cliente, como se muestra en la figura 9.

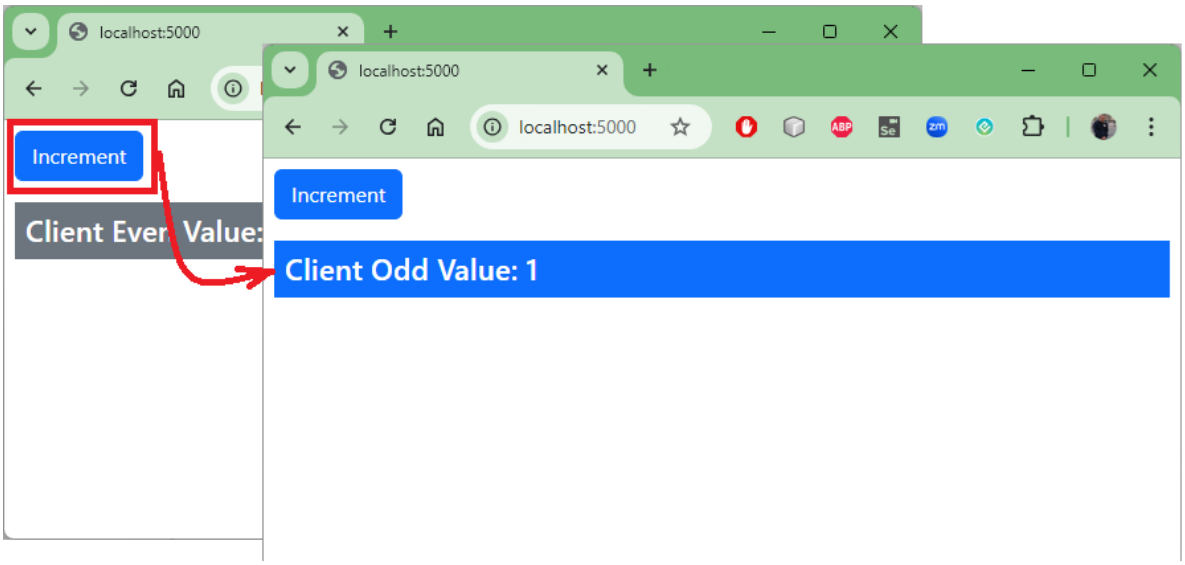

Figura 9: Uso de un paquete para plantillas del lado del cliente.

Resumen

En este documento, mostramos cómo funcionan las plantillas del lado del servidor y del lado del cliente, y cómo se pueden utilizar para generar contenido HTML. También se cubrió la siguiente información:

- Las plantillas son una mezcla de contenido estático con marcadores de posición para valores de datos.
- Cuando se representa una plantilla, el resultado es un documento HTML o un fragmento que refleja el estado actual de la aplicación.
- Las plantillas se pueden representar mediante Node.js, como plantillas del lado del servidor, o mediante JavaScript ejecutándose en el navegador, como plantillas del lado del cliente.
- Las plantillas del lado del cliente suelen compilarse en funciones de JavaScript para que el navegador pueda reproducirlas fácilmente.
- Hay muchos paquetes de plantillas de código abierto buenos disponibles, todos los cuales ofrecen características similares, pero utilizan diferentes formatos de archivo de plantilla.

En el próximo documento, explicaremos cómo se pueden utilizar formularios HTML para recibir datos del usuario y cómo validar los datos cuando se reciben.