Artículo
Javier Lorenzo Mesa · 25 jun, 2020 Lectura de 15 min

Node.js: Cómo crear una aplicación web básica con React (Parte 2)

¡Hola desarrolladores!

El desarrollo de una aplicación web Full-Stack en JavaScript con Caché requiere que juntes los bloques correctos para construirla. En la primera parte de este artículo, creamos una aplicación básica para desarrollar el frontal de usuario (front-end) en React. En esta segunda parte, mostraré cómo elegir la tecnología más adecuada para desarrollar la parte servidora (back-end) de tu aplicación. Verás que Caché te permite utilizar diferentes enfoques para vincular tu front-end con tu servidor de Caché, dependiendo de las necesidades de tu aplicación. También configuraremos un back-end mediante Node.js/QEWD y CSP/REST. En la siguiente parte, vamos a mejorar nuestra aplicación básica web y la conectaremos a Caché usando estas tecnologías.

Caché es una base de datos multi-modelo muy potente y flexible, además de un servidor para aplicaciones que cuenta con un enfoque único. Permite conectar el front-end de tu aplicación utilizando muchas tecnologías diferentes. Esta es una gran ventaja, pero también puede hacer que sea más difícil decidir cuál de estas tecnologías se debe utilizar para desarrollar el back-end.

Como desarrollador, quieres elegir una tecnología que permanezca a lo largo del tiempo, te haga ser productivo durante el desarrollo, mantenga estable la base de tu código y permita que tu aplicación se ejecute en tantos dispositivos y plataformas como sea posible utilizando el mismo código fuente.

Estos requisitos te conducen hoy en día, inevitablemente, a escribir las aplicaciones con JavaScript porque es el lenguaje que se utiliza en los navegadores (lo que hace que esté disponible en todo tipo de plataformas y sea tan popular). La aparición de Node.js hizo que también fuera posible utilizar JavaScript del lado del servidor, lo que permite a los desarrolladores usar el mismo lenguaje tanto en el front-end como en el back-end.

Básicamente, puedes escribir un sitio web o una aplicación web de dos maneras diferentes: usando la tecnología del lado del servidor o del lado del cliente (rendering). El renderizado del lado del servidor es una excelente opción para los sitios web habituales, mientras que el renderizado del lado del cliente es la mejor elección cuando se escribe una aplicación web (dado que también necesita funcionar sin conexión). Las páginas CSP de Caché son un buen ejemplo de la tecnología del lado del servidor, mientras que React es una tecnología del lado del cliente (aunque actualmente también puede utilizarse para el renderizado del lado del servidor). Esta es una decisión que deberás tomar y depende totalmente de tu aplicación.

Por ahora, nos centraremos en la escritura de las aplicaciones web, no hablaré de las páginas CSP aquí.

En el back-end, también deberás elegir qué tecnologías utilizarás para tu aplicación web: ¿usarás únicamente el servidor para aplicaciones de Caché y las llamadas CSP/REST o usarás Node.js como tu servidor para aplicaciones de modo que funcione con Caché? Al utilizar Node.js como tu servidor para la aplicación tendrás lo mejor de ambos mundos: te permite escribir el código de tu back-end en JavaScript con Caché, como una base de datos y un servidor para aplicaciones detrás de él.

¿Qué hace que esta combinación sea tan potente?

- Utiliza el mismo lenguaje (JavaScript) tanto para el front-end como para el back-end
- Utiliza todos los módulos estándares disponibles de Node.js en tus aplicaciones (esto aumenta la funcionalidad de tu aplicación hacia todos los tipos de dispositivos y servicios externos que puedas imaginar)
- Te permite reutilizar tu código COS existente mediante envoltorios de función muy pequeños
- Integra tu aplicación con otros servicios y protocolos estándar

- ...

Escribir tus aplicaciones con estas tecnologías no te limita a una única tecnología: de hecho, ¡puedes combinarlas todas en la misma aplicación!

La siguiente pregunta es cómo puedes vincular tu front-end con tu back-end. Actualmente, existen diversas opciones, cada una con sus (des)ventajas:

- WebSockets: crea un canal bidireccional (con estados) para el back-end y tu servidor puede "impulsar" (push) un mensaje hacia el front-end, no es necesario que tu aplicación inicie primero una solicitud. Esta es una excelente opción para las aplicaciones internas que cuentan con una conexión estable a la red. La biblioteca socket.io de Node.js también proporciona una "degradación elegante", ya que cuenta con una función de retroceso para las llamadas de Ajax
- REST: tu front-end está ligeramente acoplado a su back-end (sin estados), no es posible la comunicación bidireccional y la aplicación tiene que iniciar cada solicitud. Esta es una buena opción para las aplicaciones que no requieren de sesiones, redes inestables,...
- Ajax: aún se sigue utilizando ampliamente, y también sirve como una función de retroceso para las conexiones de WebSockets

Como primer ejemplo, presentaré las características de un back-end en Node.js usando el módulo QEWD:

- crea un canal de comunicación seguro entre tu aplicación web y el back-end a través de WebSockets, de manera predeterminada, mediante la biblioteca que está incorporada en socket.io, y te permite cambiar al modo Ajax de forma transparente (¡sin que sea necesario modificar el código de tu aplicación!)
- conecta el código de tu aplicación desde el back-end hacia tu base de datos en Caché
- contiene un potente servidor Express REST estándar, que proporciona los servicios REST en el mismo servidor de back-end, y también te permite crear llamadas federadas hacia los servidores remotos subyacentes y combinarlo todo en una sola respuesta (¡incluso te permite interceptar y modificar la respuesta según tus propias necesidades!)
- es completamente modular y se integra con los módulos de Node.js que ya existen; también le permite utilizar todos los demás módulos de Node.js en tus aplicaciones. Incluso te permite reemplazar el servidor Express por algún otro, como Ember.js
- simplifica enormemente el desarrollo de código en el back-end, ya que resuelve el problema de la falta de sincronización durante la codificación en JavaScript: esto te permite escribir tu código de una manera completamente sincronizada
- crea un procesamiento múltiple del back-end para ti, con tantos procesos de Node.js como sean necesarios - la funcionalidad de Node.js en sí no lo proporciona de manera predeterminada
- te permite utilizar cualquier tipo de estructura del lado del cliente (front-end), como React, Angular, ExtJS, Vue.js,...

Antes de que empecemos, hay que tener en cuenta una consideración importante: en la parte 1 construimos una pequeña demo de una aplicación en React mediante el módulo create-react-app. Notarías que este módulo contiene un servidor de programación en Node.js que se ejecuta en el localhost:3000. Es importante tener en cuenta que este servidor únicamente sirve para el front-end - el servidor back-end QEWD que configuraremos aquí es un servidor back-end (para aplicaciones) completamente distinto y que se ejecuta por separado en un puerto diferente (normalmente en el localhost:8080 para el desarrollo). Durante el desarrollo, tanto el servidor de programación React en el puerto 3000 como el servidor para aplicaciones QEWD en el puerto 8080 se ejecutarán en la misma máquina, ¡no te confundas con los dos servidores! En producción, la situación será diferente: tu aplicación React se ejecutará desde el navegador del cliente y tu servidor QEWD recibirá en el servidor de Caché las conexiones que provengan del cliente.

A continuación, mostraré todos los pasos necesarios para comenzar nuestro ejemplo de una aplicación en React.

Primero, abre una línea de comandos en Windows para instalar QEWD en tu sistema:

```
Administrator: Command Prompt

C:\qewd>npm install qewd qewd-monitor cors
C:\qewd
+-- cors@2.8.3
| `-- object-assign@4.1.1
+-- qewd@2.6.2
| `-- socket.io@1.7.3
|   `-- object-assign@4.1.0
`-- qewd-monitor@2.1.0

npm WARN enoent ENOENT: no such file or directory, open 'C:\qewd\package.json'
npm WARN qewd No description
npm WARN qewd No repository field.
npm WARN qewd No README data
npm WARN qewd No license field.

C:\qewd>_
```

Cuando se termine la instalación, verás un par de advertencias que puedes ignorar. Observa cómo el módulo QEWD utiliza los módulos Node.js que ya existen como bloques de construcción (echa un vistazo dentro de la carpeta C:\qewd\node_modules). El módulo qewd-monitor se utilizará más adelante para monitorizar tu servidor QEWD y el módulo cors será necesario para las solicitudes REST.

El QEWD necesita conectarse con Caché, por lo que debemos **copiar el conector cache.node que usamos durante el primer script de prueba entre Node.js y Caché en C:\qewd\node_modules** . Esto permitirá que QEWD inicie nuestros nuevos procesos o "procesos hijos" en Node.js, que se conectarán con Caché.

Solo necesitamos crear un pequeño script de inicio en JavaScript, que nos permita iniciar el servidor para aplicaciones QEWD. Primero, añade la carpeta del proyecto C:\qewd en tu editor Atom. Después, dentro de la carpeta C:\qewd, crea un nuevo archivo **qewd-start.js** y ábrelo para editarlo.

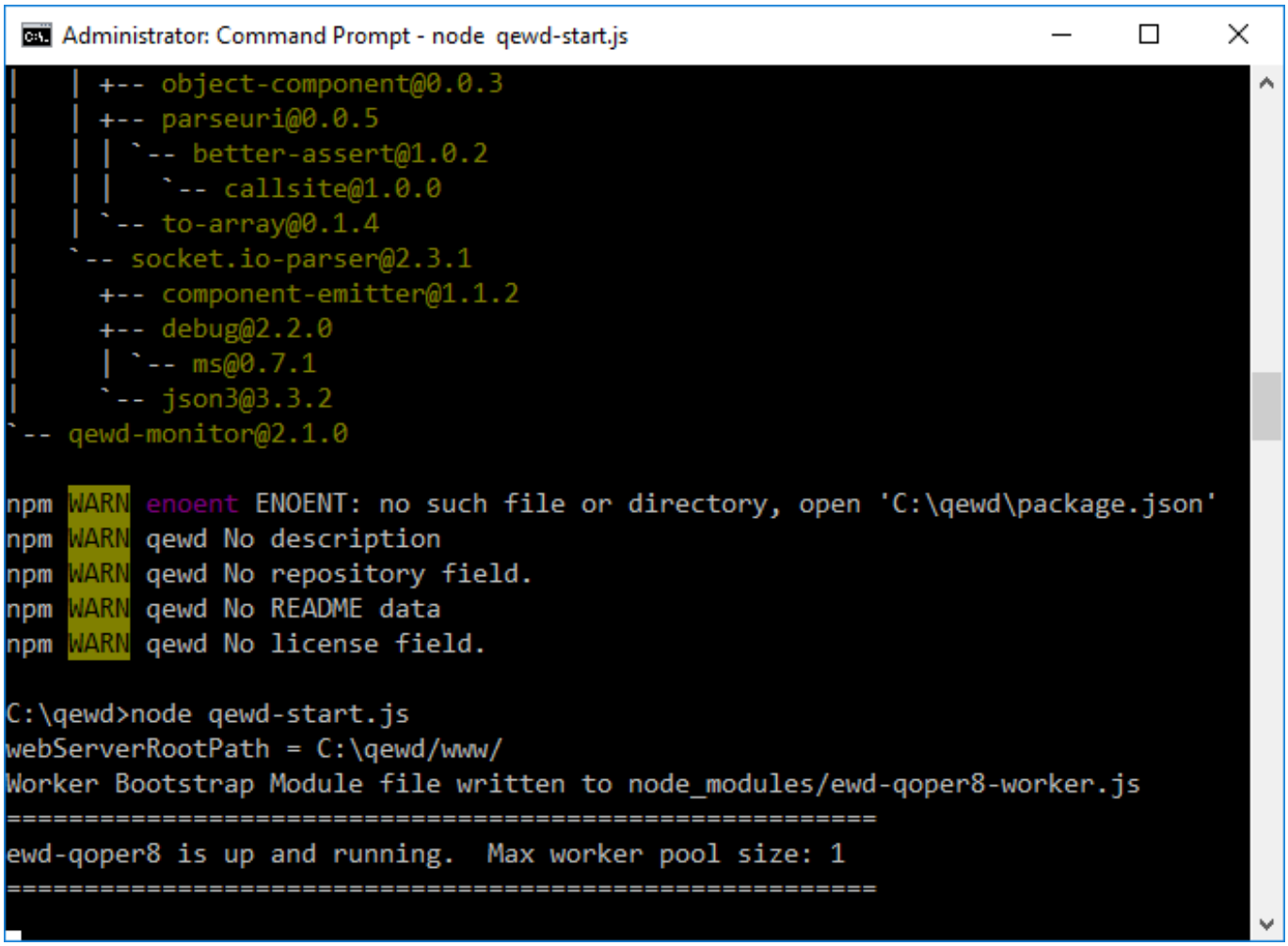
Copia/pega el siguiente código en este archivo y guárdalo:

```
// define the QEWD configuration (adjust directories if needed)
let config = {
  managementPassword: 'keepThisSecret!',
  serverName: 'My first QEWD Server',
  port: 8080,
  poolSize: 1,
  database: {
    type: 'cache',
    params: {
      path:"C:\\InterSystems\\Cache\\Mgr",
      username: "_SYSTEM",
      password: "SYS",
```

```
      namespace: "USER"
    }
  }
};
// include the cors module to automatically add CORS headers to REST responses
let cors = require('cors');
// define the QEWD Node.js master process variable
let qewd = require('qewd').master;
// define the internal QEWD Express module instance
let xp = qewd.intercept();
// define a basic test path to test if our QEWD server is up- and running
xp.app.get('/testme', function(req, res) {
  console.log('*** /testme query: ', req.query);
  res.send({
    hello: 'world',
    query: req.query
  });
});
// start the QEWD server now ...
qewd.start(config);
```

* Para tener una introducción completa al servidor para aplicaciones QEWD para Node.js/Caché, recomiendo consultar la documentación, disponible en el apartado "Training" en el menú superior.

Ahora podemos iniciar nuestro servidor para aplicaciones QEWD en la línea de comandos:

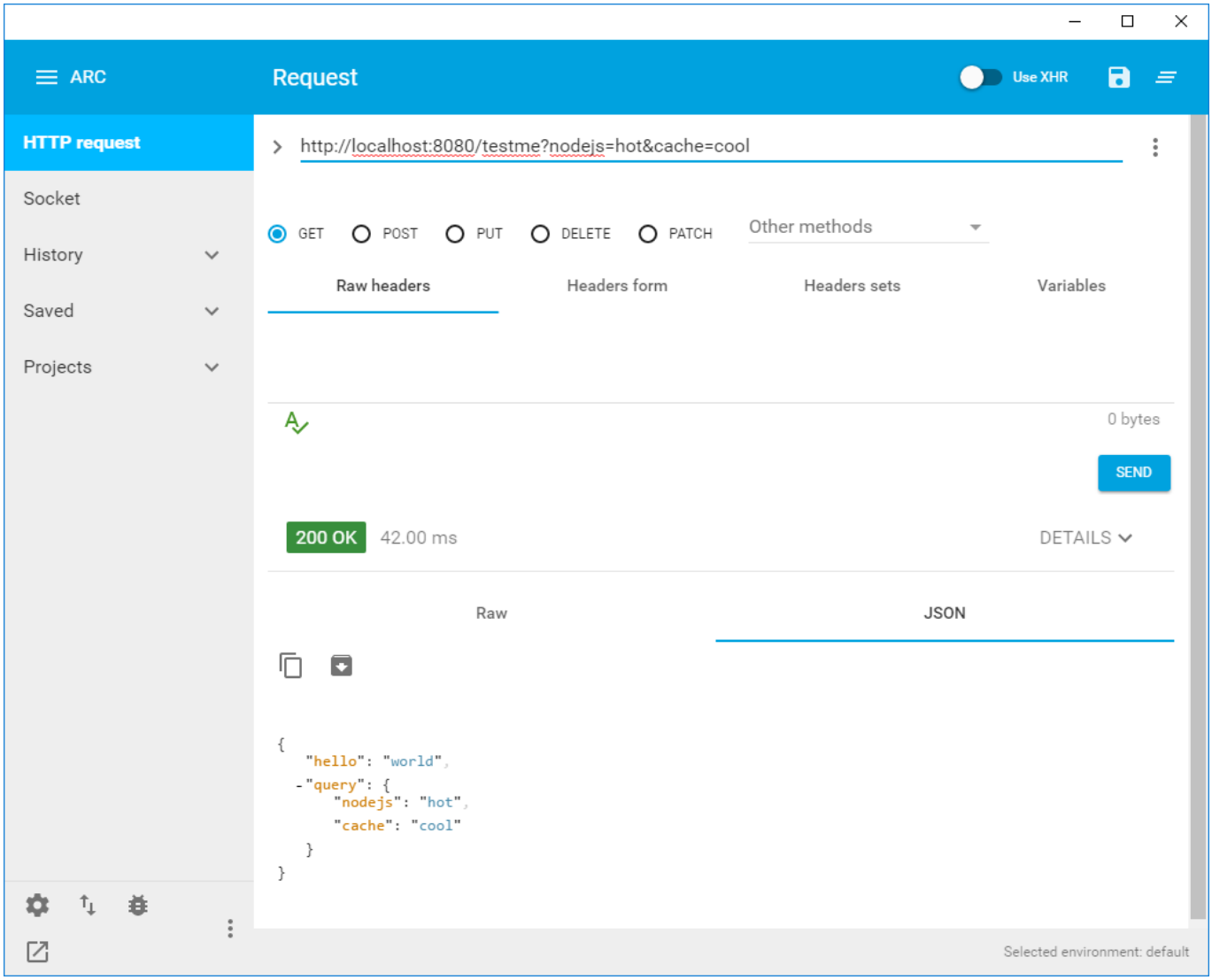


Si recibes alguna advertencia del firewall de Windows, concede permiso a los procesos de Node.js para que tengan acceso a la red.

Ahora abre Chrome, e instala la extensión (app) Advanced REST client (ARC) para depurar las solicitudes REST que enviaremos hacia nuestro servidor para aplicaciones.

Abre una nueva pestaña en la página y haz clic sobre la opción "Apps" en la barra de herramientas. Verás que la nueva extensión está disponible allí. Haz clic en el icono para iniciar la aplicación ARC.

Ahora escribe la url http://localhost:8080/testme?nodejs=hot&cache=cool en la solicitud de línea de texto que se encuentra en la parte superior y observa la respuesta del módulo QEWD/Express:



¡Enhorabuena! Si ves esta respuesta, tu servidor QEWD se inició satisfactoriamente ¡y el módulo Express en su interior también está funcionando!

Hasta ahora, aún no nos hemos conectado a Caché: solo probamos el proceso maestro QEWD, en el que se recibirán todas las solicitudes. Este proceso está recibiendo las conexiones de WebSockets y el módulo Express está listo para atender las solicitudes REST.

La **primera opción** que implementaremos ahora para acceder a nuestro back-end será mediante el servidor QEWD REST (consulta la documentación): crear un archivo para el módulo **testrest.js** dentro de C:\qewd\node_modules

```
module.exports = {
  restModule: true,
```

```
  handlers: {
    isctest: function(messageObj, finished) {
      // this isctest handler retrieves text from the ^nodeTest global and returns it
      let incomingText = messageObj.query.text;
      let nodeTest = new this.documentStore.DocumentNode('nodeTest');
      let d = new Date();
      let ts = d.getTime();
      nodeTest.$(ts).value = incomingText;
      finished({text: 'You sent: ' + incomingText + ' at ' + d.toUTCString()});
    }
  }
};
```

Guarda este archivo y edita el archivo **qewd-start.js** en C:\qewd. Aquí necesitamos añadir un endpoint para nuestro módulo REST:

```
// define the QEWD configuration (adjust directories if needed)
let config = {
  managementPassword: 'keepThisSecret!',
  serverName: 'My first QEWD Server',
  port: 8080,
  poolSize: 1,
  database: {
    type: 'cache',
    params: {
      path:"C:\\InterSystems\\Cache\\Mgr",
      username: "_SYSTEM",
      password: "SYS",
      namespace: "USER"
    }
  }
};
// include the cors module to automatically add CORS headers to REST responses
let cors = require('cors');
// define the QEWD Node.js master process variable
let qewd = require('qewd').master;
// define the internal QEWD Express module instance
let xp = qewd.intercept();
// define a basic test path to test if our QEWD server is up- and running
xp.app.get('/testme', function(req, res) {
  console.log('*** /testme query: ', req.query);
  res.send({
    hello: 'world',
    query: req.query
  });
});
// define REST endpoint for /testrest requests
xp.app.use('/testrest', cors(), xp.qx.router());
// start the QEWD server now ...
qewd.start(config);
```

La línea que añadimos enrutará todas las solicitudes que empiecen con la ruta /testrest hacia nuestro módulo testrest.js. Para activar la nueva ruta, es necesario que reiniciemos el servidor QEWD presionando Ctrl-C en la ventana de la línea de comandos y reiniciarlo:

* Ten en cuenta que incluso puedes reiniciar un servidor QEWD, cuando los

clientes estén conectados, en caso de que lo necesites (no se recomienda hacer este procedimiento en un sistema ocupado): todas las conexiones de los clientes se reiniciarán/reconectarán automáticamente. Por cierto, también existe una manera más sencilla para añadir/(re)definir dinámicamente las rutas en un servidor QEWD que esté en ejecución, aunque esto requiere de un pequeño módulo de enrutamiento, que le proporcionará más flexibilidad sin la necesidad de detener el servidor QEWD.

Ahora, vuelve al ARC REST client y envía una nueva solicitud para
http://localhost:8080/testrest/isctest?text=REST+call+to+cache

En la línea de comandos donde se está ejecutando el servidor QEWD, verás que aparecen mensajes de la llamada entrante:

Ahora puedes ver que QEWD comenzó un proceso de trabajo para esta solicitud. ¿Por qué? El proceso maestro (el servidor Express) recibe tu solicitud y se la entrega al middleware QEWD Express cors(), en primer lugar, para añadir los encabezados CORS y después para agregar la función de enrutamiento QEWD/REST xp.qx.router() en este momento, que añade la solicitud a la lista de solicitudes en espera e inicia un "proceso hijo" (el worker). Estos workers se encargan de gestionar tus solicitudes y procesan el código de tu aplicación (en testrest.js). Esto permite el procesamiento múltiple de las solicitudes, ya que QEWD puede iniciar tantos workers como sea necesario.

Probablemente te diste cuenta de que, en la salida del espacio de trabajo, el proceso maestro recibió la solicitud REST y entregó dicha solicitud al worker. Entonces, el worker carga el módulo de tu aplicación, llama al controlador de las solicitudes correcto y envia la respuesta.

Ahora que ya definimos un endpoint para QEWD REST, añadiremos una **segunda opción** para llamar a nuestro back-end utilizando WebSockets: crear un archivo para el módulo **test.js** en C:\qewd\node_modules, para procesar las solicitudes (mensajes) que entran mediante las conexiones de WebSockets:

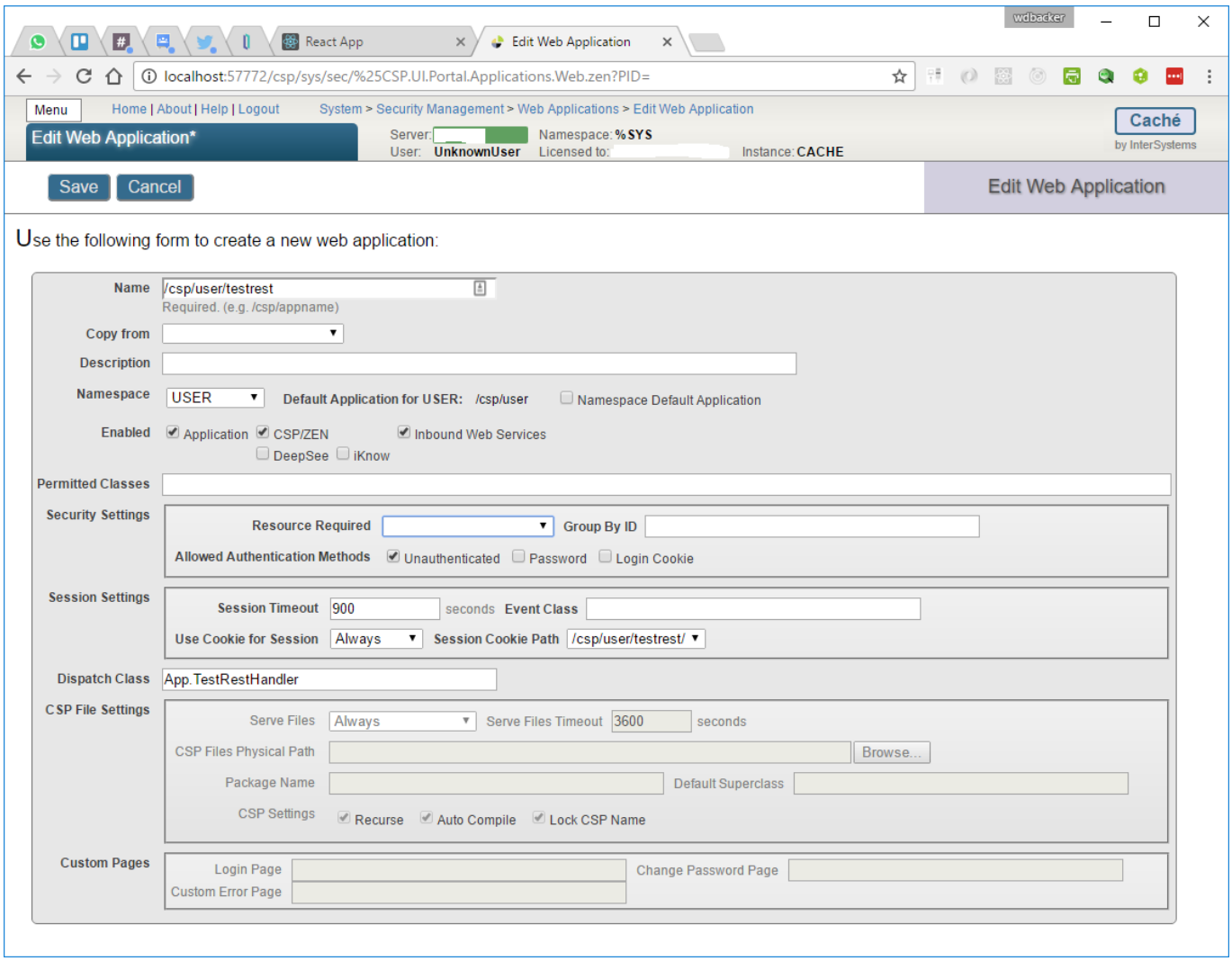
```
module.exports = {
  handlers: {
    isctest: function(messageObj, session, send, finished) {
      // get the text coming in from the message request
      var incomingText = messageObj.params.text;
      // instantiate the global node ^nodeTest as documentStore abstraction
      let nodeTest = new this.documentStore.DocumentNode('nodeTest');
      // get the current date & time
      let d = new Date();
      let ts = d.getTime();
      // save the text from the request in the ^nodeTest global (subscripted by the c
urrent timestamp)
      nodeTest.$(ts).value = incomingText;
      // return the response to the client using WebSockets (or Ajax mode)
      finished({text: 'You sent: ' + incomingText + ' at ' + d.toUTCString()});
    }
  }
};
```

Este controlador de solicitudes es casi igual al controlador de solicitudes REST, sin embargo, notarás que hay algunas diferencias sutiles: en los parámetros que definen las funciones del controlador, ahora están disponibles un objeto de sesión y un método send(). El método send() permite que el back-end envíe mensajes "push" hacia el front-end, y también tenemos un objeto de sesión ahora, porque las conexiones de WebSockets tienen estados.

No es posible probar este controlador de solicitudes de WebSockets mediante la herramienta ARC, porque necesitamos configurar una conexión de WebSockets para probarlo. En la siguiente parte, donde mejoraremos el front-end de nuestra aplicación en React, verás cómo funciona el controlador para iniciar una conexión de WebSockets con nuestro back-end QEWD.

Para tener información más detallada y formación sobre el funcionamiento del servidor QEWD/Express, te recomiendo que consultes la cdocumentación, disponible en el apartado "Training" en el menú superior.

La **tercera opción** para establecer una conexión con nuestro back-end es utilizar las llamadas CSP/REST: ve al portal del Sistema y en la sección Administración del sistema, ve a las Seguridad - Aplicaciones - Aplicaciones web y define una nueva aplicación web (consulta también la documentación sobre CSP/REST):



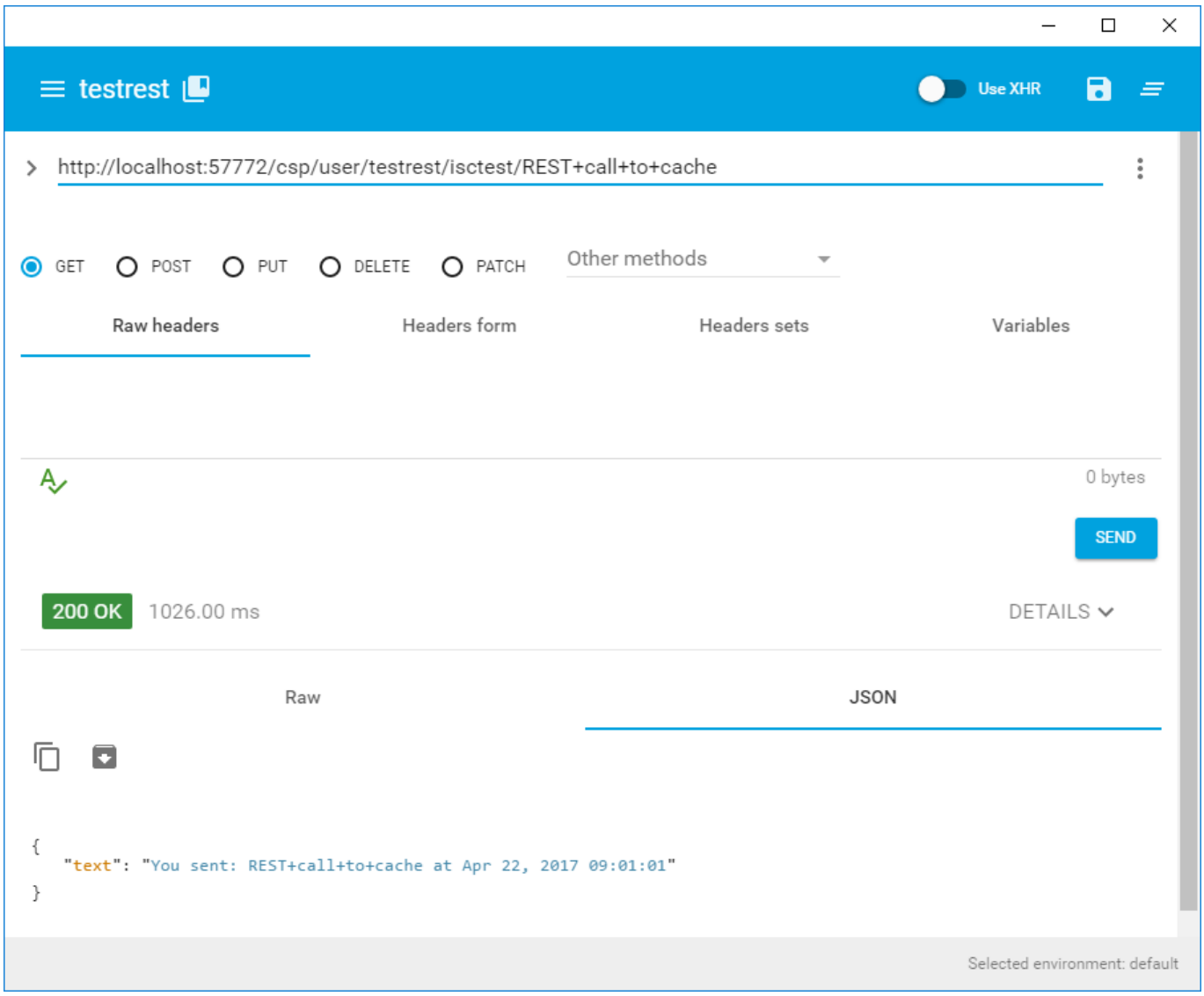
Abre Caché Studio, crea una clase **App.TestRestHandler** en el namespace **USER** y añade este código:

```
Class App.TestRestHandler Extends %CSP.REST
{
  Parameter HandleCorsRequest = 1;
  XData UrlMap
  {
    <Routes>
      <Route Url="/isctest/:text" Method="GET" Call="GetIscTest"/>
    </Routes>
  }
  ClassMethod GetIscTest(text As %String = "") As %Status
  {
    #Dim e as %Exception.AbstractException
    #Dim status as %Status
    Try {
      Set d = $now()
      Set ts = $zdt(d,-2)
      Set ^nodeTest(ts) = text
      If $Data(%response) Set %response.ContentType="application/json"
      Write "{"
      Write " ""text"" : ""You sent: ",text," at ",$zdt(d,5,1),"""
      Write "}"
    } Catch (e) {
      Set status=e.AsStatus()
```

```
      Do ..ErrorHandler(status)
    }
    Quit $$$OK
  }
  ClassMethod ErrorHandler(status)
  {
    #Dim errorcode, errormessage as %String;
    set errorcode=$piece(##class(%SYSTEM.Status).GetErrorCodes(status),",")
    set errormessage=##class(%SYSTEM.Status).GetOneStatusText(status)
    Quit ..ErrorHandlerCode(errorcode,errormessage)
  }
  ClassMethod ErrorHandlerCode(errorcode, errormessage) As %Status
  {
    Write "{"
    Write " ""ErrorNum"" : """,errorcode,""","
    Write " ""ErrorMessage"" : """,errormessage,"""
    write "}"
    If $Data(%response) {
      Set %response.ContentType="application/json"
    }
    quit $$$OK
  }
}
```

* Gracias a Danny Wijnschenk por este ejemplo sobre CSP/REST

Vamos a probar este back-end REST también, usando ARC:



¡Enhorabuena! Hemos creado tres posibles maneras para establecer una conexión entre nuestro back-end y Caché:

- utilizando el servidor para aplicaciones Node.js/QEWD/Express mediante llamadas REST
- utilizando el servidor para aplicaciones Node.js/QEWD/Express mediante WebSockets (con funciones de retroceso opcionales para las llamadas de Ajax)
- utilizando el puerto de enlace CSP/REST con Caché como el servidor para aplicaciones, mediante clases para el controlador CSP/REST

Como puedes ver, con Caché hay muchas maneras de lograr nuestro objetivo. Puedes decidir cuál de las opciones es la más apropiada para ti.

Si tú (o la mayoría de tu equipo de desarrollo) estás familiarizado con JavaScript y quieres desarrollar tu aplicación utilizando el mismo lenguaje, pero con los potentes frameworks de JavaScript para el front-end, y/o tu aplicación necesita las funciones o servicios externos que ya estén disponibles en los módulos de Node.js, entonces un servidor para aplicaciones Node.js que funcione con Caché es tu mejor opción, porque tendrás lo mejor de ambos mundos y podrás conservar y reutilizar tu código, las clases y el SQL en Caché. El servidor para aplicaciones QEWD se diseñó específicamente para escribir aplicaciones empresariales con Caché y es tu mejor opción para integrar el front-end y el back-end.

Si tu front-end está escrito en HTML simple, no estás utilizando los potentes frameworks de JavaScript y tu aplicación únicamente necesita llamadas REST hacia Caché, puedes conectarte directamente a Caché usando CSP/REST. Solamente recuerda que en este caso no podrás usar todos los módulos fácilmente disponibles de

Node.js.

Ahora, ya estamos listos para la parte 3: ¡conectaremos el front-end de nuestra aplicación en React con los tres posibles back-ends!

#API REST #JavaScript #JSON #Node.js #React #Caché

URL de fuente:https://es.community.intersystems.com/post/nodejs-c%C3%B3mo-crear-una-aplicaci%C3%B3n-web-b%C3%A1sica-con-react-parte-2