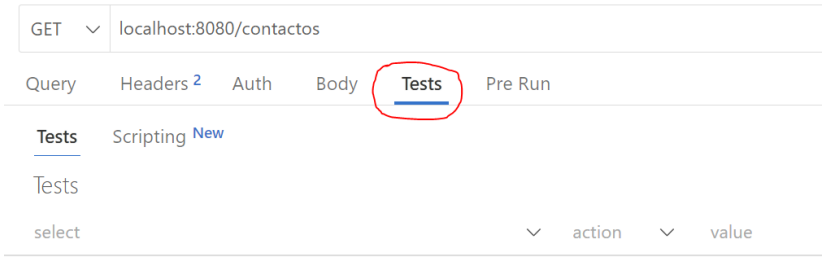
Testing de servicios REST

En este documento vamos a dar unas nociones básicas de cómo hacer *testing* de servicios REST, y pasar un conjunto de pruebas sobre una API REST y recoger los resultados.

1. Testing con *ThunderClient*

En documentos anteriores ya hemos explicado cómo utilizar la extensión *ThunderClient* de Visual Studio Code para realizar pruebas de servicios REST. Sin embargo, estas pruebas que hicimos entonces eran pruebas aisladas y "manuales": nosotros creábamos la petición, construíamos la URL y mirábamos la respuesta. Lo que veremos aquí es cómo podemos configurar colecciones para que se auto-ejecuten y evalúen sus resultados.

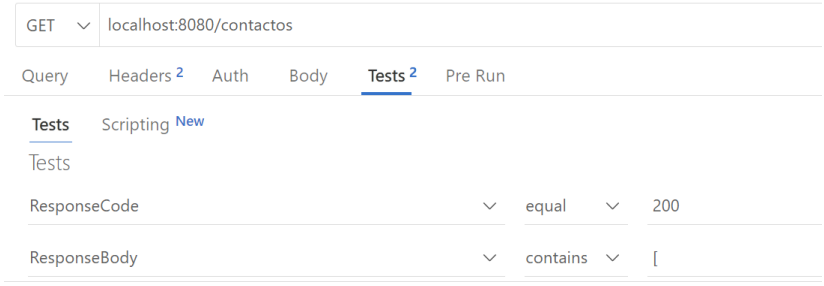
Los tests se van a gestionar desde la pestaña *Testing* de cada petición o *Request*. Ahí definiremos el parámetro (o parámetros) que queremos chequear de cada respuesta.



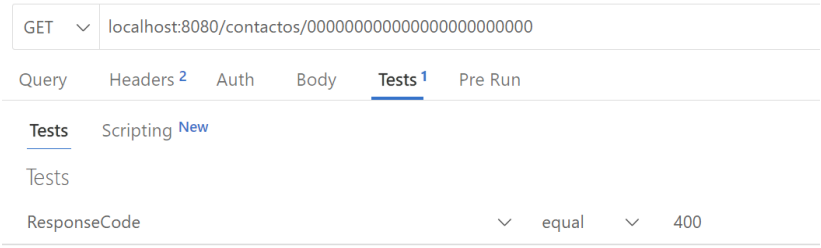
1.1. Definiendo tests básicos

Vamos a definir un test básico sobre el listado de contactos, de nuestro ejemplo *ContactosREST_v2*. En la colección de pruebas, vamos a la prueba del listado de contactos (*GET /contactos*) y en su pestaña *Testing* indicamos dos parámetros de comprobación:

- El código de estado de la prueba debe ser 200
- El contenido de la prueba debe contener un array (corchetes), aunque esté vacío.

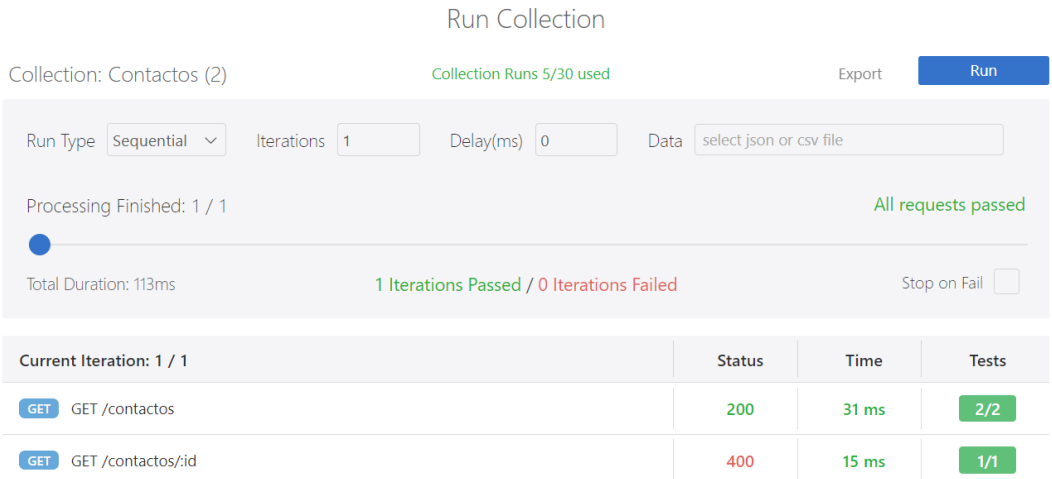


También podemos hacer otros tests sencillos en otras peticiones (*requests*). Por ejemplo, en la ficha de un contacto que sepamos que es incorrecto podemos ver que el código de estado sea un 400, si lo hemos especificado así en la respuesta:

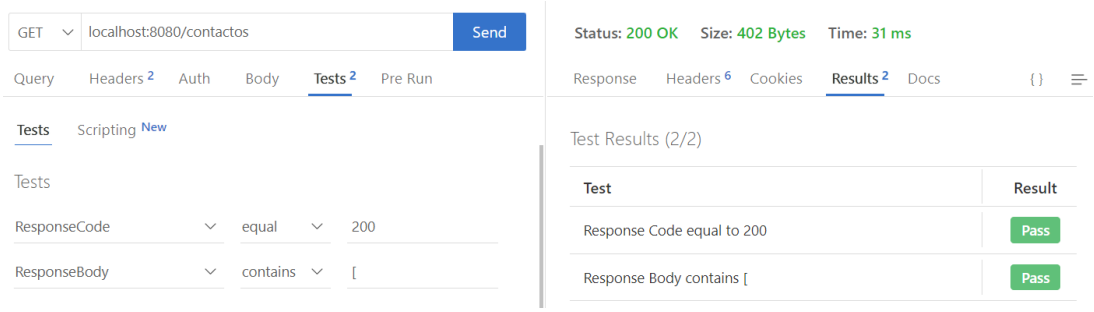


1.2. Ejecución y comprobación de tests

Para lanzar de golpe todos los tests que hemos definido hacemos clic derecho sobre la colección (en el panel izquierdo) y elegimos *Run All*, y después pulsamos el botón de *Run* que aparecerá. Esto pondrá en marcha todos los tests y, en el panel derecho, podremos ver los resultados generales y de cada test:



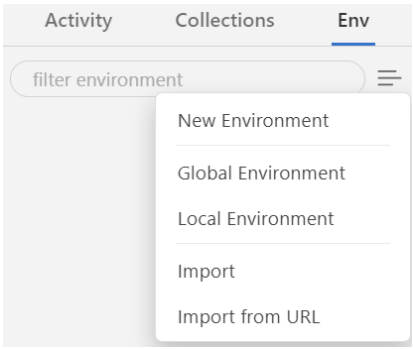
Haciendo clic en cada test podremos ver el desglose de las pruebas o parámetros que hemos pedido verificar:



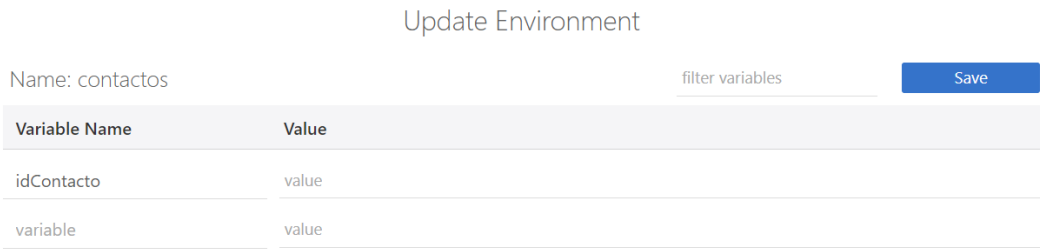
1.3. Utilizando variables en los tests

En algunas ocasiones lanzar un test no es tan sencillo, y necesitamos una parte variable que no podemos controlar de antemano. Por ejemplo, si insertamos un documento que luego queremos modificar, ¿cómo podemos guardarnos el *id* de ese documento para la prueba de modificación posterior?

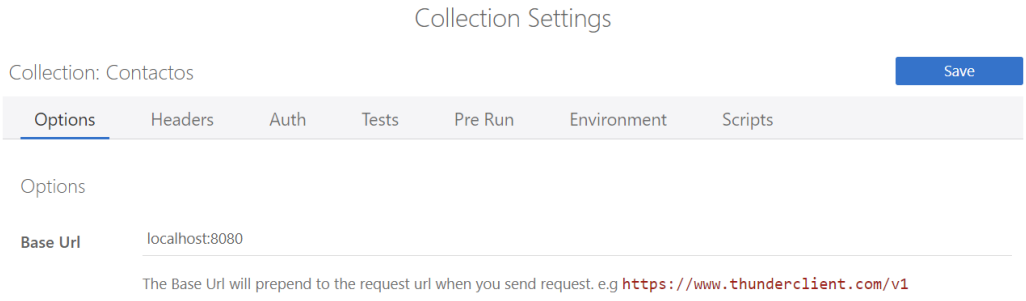
Para esto haremos uso de la pestaña ***Env*** del panel izquierdo, donde podemos crear variables de entorno, o bien globales, o bien asociadas a una colección en particular.



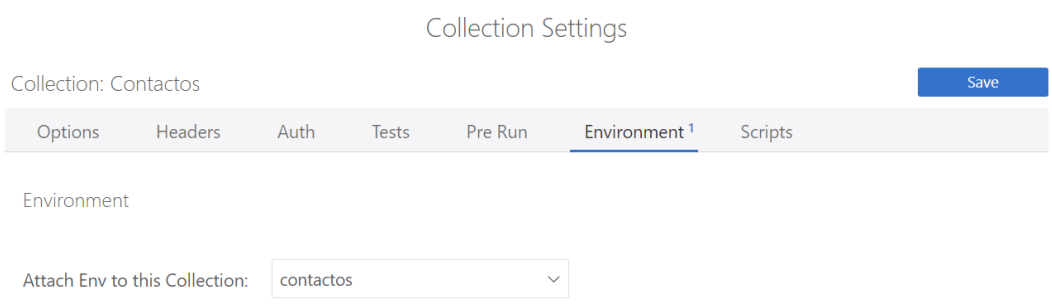
Crearemos un nuevo entorno (*New Environment*) y le pondremos un nombre. Por ejemplo, *contactos*. Dentro del entorno podemos crear variables y dejarlas preparadas para almacenar datos que necesitemos. Por ejemplo, el *id* del contacto con el que queramos trabajar:



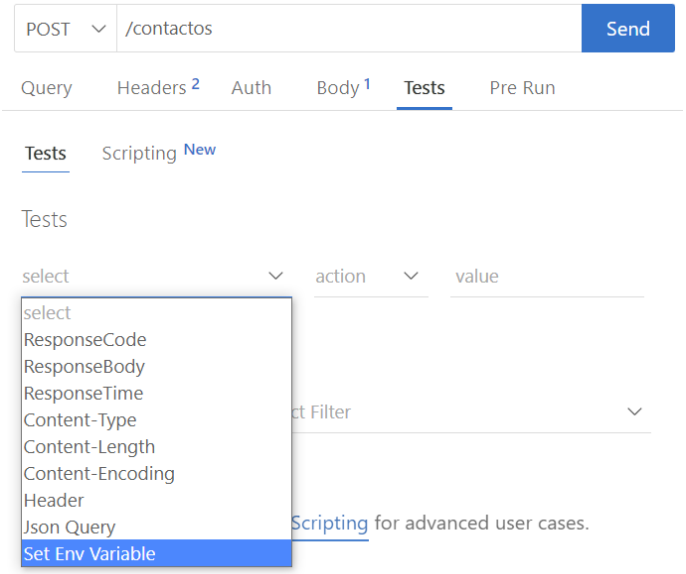
Ahora debemos ir a la colección de pruebas de contactos, hacer clic derecho en ella e ir a ***Settings***. En la sección *Options* podemos establecer (si queremos) una URL base para todas las peticiones de la colección, de modo que todas empezarán por este prefijo (y podremos cambiar fácilmente, por ejemplo, el número de puerto o la URL base en todas ellas):



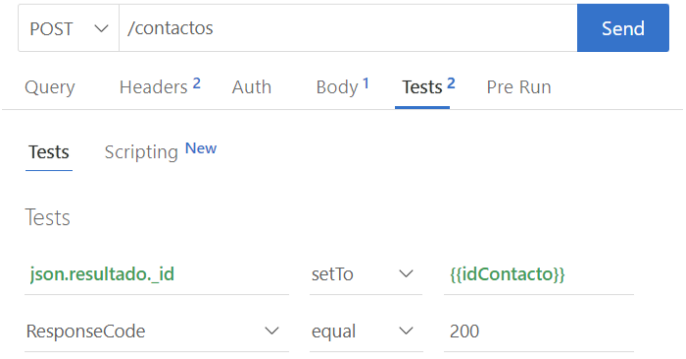
En la sección *Environment* podemos asociar un entorno a esta colección (en nuestro caso, el entorno *contactos*) que hemos creado antes. De este modo cualquier variable que hayamos definido en ese entorno se podrá utilizar directamente en esta colección.



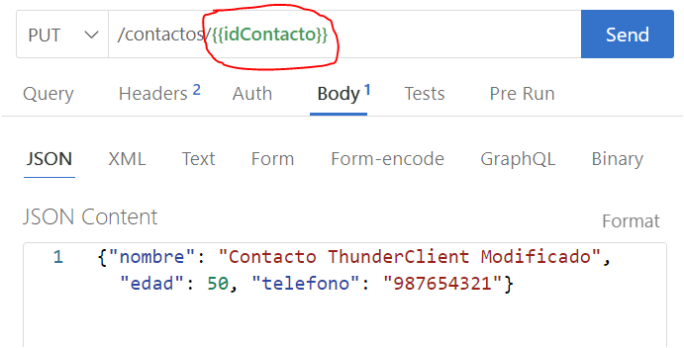
Veamos cómo funciona esto. En la prueba de POST para insertar un nuevo contacto, vamos a la pestaña *Tests* y elegimos la opción *Set Env Variable*.



Después elegimos qué parámetro JSON queremos recoger (en nuestro caso, el *_id* del contacto insertado) y en qué variable queremos guardarlo (en nuestra variable de entorno *idContacto*, expresada entre dobles llaves). Además, podemos indicar otros parámetros de testeo, como que el código de respuesta sea 200:



A partir de la prueba de este servicio, los que vengan después ya tendrán un valor almacenado en la variable *idContacto*, y podemos utilizarlo. Por ejemplo, así podríamos modificar algún dato del contacto recién insertado, sea cual sea su *id*:



1.4. Exportando tests

Si queremos exportar un test para poderlo utilizar en otros sistemas, deberemos tener en cuenta si tiene variables de entorno asociadas. En este caso deberemos exportar tanto el entorno (pestaña *Env*) como la colección, e importarlas en el sistema de destino.

2. Testing desde JavaScript: *Axios*

Uno de los principales inconvenientes que podemos encontrar a la hora de probar nuestras aplicaciones REST con herramientas como Thunder Client o Postman es que su uso gratuito tiene ciertas limitaciones, en cuanto al número de veces que podemos ejecutar una colección.

Para paliar esta deficiencia y complementar mejor nuestras pruebas podemos hacer uso de alguna librería específica en JavaScript que permita lanzarlas de forma gratuita e ilimitada desde el código. Es el caso de Axios.

2.1. Instalación y configuración

Las pruebas que hagamos en Axios las podemos definir en un proyecto aparte del que queramos probar, o bien podemos definir alguna carpeta *tests* dentro del propio proyecto. Aunque, eso sí, necesitaremos ejecutar el programa de tests desde un terminal, y puede que el terminal del proyecto principal ya esté ocupado ejecutando Express.

En el proyecto donde vayamos a hacer las pruebas necesitamos instalar el módulo *axios* con el correspondiente comando:

```
npm install axios
```

Una vez instalado lo incorporamos al fichero fuente (o ficheros) donde vayamos a hacer los tests (por ejemplo, `tests.js`), y realizamos una configuración previa donde indicamos la URL base a la que queremos conectar (aquella donde estará escuchando el proyecto a probar), y el tipo de contenido que vamos a utilizar (en nuestro caso, contenido JSON):

```
const axios = require('axios');

const axiosInstance = axios.create({
    baseURL: 'http://localhost:8080',
    headers: {
        'Content-Type': 'application/json',
    }
});
```

2.2. Definición de pruebas simples

Imaginemos que queremos probar el listado de contactos, que está escuchando por GET en la URI */contactos*. Podríamos definir un método en nuestro fichero de `tests.js` que lance la petición y recoja la respuesta. En el propio código JavaScript podemos evaluar qué parámetros consideramos correctos y cuáles un error. Por ejemplo, esperamos un código 200 de respuesta de estado, y un array (de tamaño 0 o superior).

```
const obtenerContactos = async () => {
    try {
        const respuesta = await axiosInstance.get('/contactos');
        if(respuesta.status == 200 && respuesta.data.resultado.length >= 0)
            console.log("OK - Listado contactos");
        else
            throw new Error();
    } catch (error) {
        console.log("ERROR - Listado contactos");
    }
};
```

En la variable `respuesta` donde obtenemos la respuesta disponemos tanto del código de estado devuelto (propiedad `status`) como del objeto devuelto en la petición (propiedad `data`). En nuestro caso tendríamos `respuesta.data.resultado` con el listado de habitaciones, o `respuesta.data.ok` con el booleano con el resultado de la operación. También podríamos consultar las cabeceras de respuesta a través de la propiedad `headers` (`respuesta.headers`).

Vamos con otra prueba simple: imaginemos que probamos a buscar un contacto que no existe. En este caso esperamos un código 400 como respuesta:

```
const contactoIncorrecto = async () => {
    try {
        const respuesta = await axiosInstance.get('/contactos/000000000000000000000(
        console.log("ERROR - Contacto incorrecto");
    } catch (error) {
        if(error.response.status == 400)
            console.log("OK - Contacto incorrecto");
        else
            console.log("ERROR - Contacto incorrecto");
    }
};
```

Notar que, en este caso, lo esperable es que se produzca una excepción, porque se envía un código de estado de error (400). Si no es así, la prueba falla. Y, en caso de recibir un error, también fallará si el código de error que se recibe (`error.response.status`) no es el esperado 400.

2.3. Ejecución de pruebas

Para ejecutar las pruebas que vayamos definiendo podemos definir una función adicional que las aglutine todas e invocarla desde el programa principal de `tests.js` :

```
// Función principal de ejecución de pruebas
const ejecutarPruebas = async() => {
    await obtenerContactos();
    await contactoIncorrecto();
    ...
}

// Main
ejecutarPruebas();
```

Los mensajes de ERROR y OK que pongamos en cada prueba son arbitrarios, podemos mostrar la información que consideremos. Pero es recomendable que se pueda ver cómodamente en la consola de salida qué pruebas han salido bien y cuáles no:

```
OK - Listado contactos
OK - Contacto incorrecto
...
```

2.4. Guardar datos entre pruebas

Igual que hemos visto en el caso de *Thunder Client*, en algunos casos nos puede interesar guardarnos una parte del resultado de una prueba para utilizarlo en pruebas sucesivas. Por ejemplo, el *id* de un contacto insertado para poderlo después buscar, modificar o borrar. O el *token* de validación de un usuario para poder lanzar pruebas que requieran autenticación.

La siguiente prueba se guarda el *token* del usuario autenticado en un servicio de *login*. Observad cómo enviamos el login y password del usuario en la petición, y cómo recogemos y guardamos el token en una variable que devolvemos.

```
const autenticarUsuario = async () => {
    try {
        const respuesta = await axiosInstance.post('/auth/login', {
            login: 'nacho',      // Reemplazar con el nombre de usuario real
            password: '1234567'  // Reemplazar con la contraseña real
        });
        if (respuesta.status == 200)
        {
            console.log("OK - Login");
            return respuesta.data.resultado;  // Devolvemos el token del resultado
        }
        else
            throw new Error();
    } catch (error) {
        console.log("Error - Login");
        return null;
    }
};
```

Podemos definir una función auxiliar que nos ayude a guardarnos el token en la cabecera correspondiente si es correcto, o a borrarlo de dicha cabecera si no lo es:

```
// Configuración del token de autenticación para las siguientes solicitudes
const setToken = (token) => {
    if (token) {
        axiosInstance.defaults.headers.common['authorization'] = `Bearer ${token}`;
    } else {
        delete axiosInstance.defaults.headers.common['authorization'];
    }
};
```

Ahora vamos a utilizar el token devuelto por esta prueba para insertar un contacto, enviando el token de autorización en la cabecera correspondiente.

```
const nuevoContacto = async (token) => {
    // Usamos la función anterior para guardar el token
    setToken(token);
    const contacto1 = {
        nombre: "Axios",
        edad: 20,
        telefono: "688888888"
    };

    try {
        const respuesta = await axiosInstance.post('/contactos', contacto1);
        if(respuesta.status == 200)
        {
            console.log("OK - Nuevo contacto");
            return respuesta.data.resultado._id;
        }
        else
            throw new Error();
    } catch(error) {
        console.log("ERROR - Nuevo contacto");
        return -1;
    }
}
```

En este caso devolvemos *-1* como identificador del contacto. Podríamos pasar este *id* como parámetro a las pruebas que lo necesiten, como por ejemplo la ficha del contacto que acabamos de insertar:

```
const fichaContacto = async (id) => {
    try {
        const respuesta = await axiosInstance.get(`/contactos/${id}`);
        if (respuesta.status == 200 && respuesta.data.resultado)
            console.log("OK - Ficha contacto");
        else
            throw new Error();
    } catch (error) {
        console.log("ERROR - Ficha contacto");
    }
};
```

Añadimos ahora estas pruebas a la función principal:

```
const ejecutarPruebas = async() => {
    await obtenerContactos();
    await contactoIncorrecto();
    let token = await autenticarUsuario();
    let id = await nuevoContacto(token);
    await fichaContacto(id);
}
```