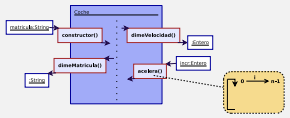
Grado Tecnologías Interactivas

Programación 2



Práctica 6

UNIVERSIDAD POLITECNICA DE VALENCIA

Escola Politècnica Superior de Gandia

DSIC

Departament de Sistemes Informàtics i Computació

Práctica 6

¡ Atención !

▷ Se recuerda que las prácticas deben prepararse antes de acudir al aula informática, anotando en el enunciado las dudas que se tengan.

▷ Los diseños y algortimos que se piden en esta práctica deben escribirse en la libreta de apuntes para poder ser revisados.

▷ Utiliza *git* en cada ejercicio haciendo *commit* cada vez que consigas un "hito".

▷ La realización de las prácticas es un trabajo individual y original. En caso de plagio se excluirá al alumno de la asignatura. Por tanto, es preferible presentar el trabajo realizado por uno mismo aunque éste tenga errores.

1

Bases de datos relacionales y SQL

Una *base de datos* es un sistema capaz de almacenar información de forma *organizada* y *persistente*. Además, debe ofrecer mecanismos para gestionar la información guardada. Estos mecanismos incluyen: crear la estructura en la que se guardan los datos; insertar, modificar y borrar datos; y realizar consultas.

Las bases de datos más utilizadas son las *relacionales*. Aunque hace unos años no tenían alternativa, han aparecido otras opciones que, aún no siendo iguales entre ellas, se engloban bajo el poco original nombre de bases de datos *no relacionales* o *no SQL*.

En esta práctiva vamos a aprender a usar, de forma elemental, bases de datos relacionales.

Una base de datos relacional organiza la información en tablas con un número *fijo* de columnas, pero tendrá tantas filas como información queramos guardar.

Por ejemplo, esta podría ser una tabla Persona para guardar datos de distintas personas.

Persona		
dni	nombre	apellidos
20123456A	Juan	Perez Perez
20123457B	Maria	Garcia
20123458C	Jose	Perez
20123458C	Jose	Ramirez

En esta tabla, todas las filas deben tener valor para cada columna. Además, una (al menos) de las columnas debe ser una *clave*. El valor de la columna clave *no* se puede repetir. En nuestro ejemplo, la clave es *dni* y por tanto no puede haber dos pesonas (dos filas) con el mismo dni.

Siempre es asi o solo en este caso ?

Igualmente, podríamos tener una tabla Asignatura, teniendo como clave la columna *código*.

Asignatura	
codigo	nombre
13928	Programacion 1
13929	Programacion 2
13898	Algebra

El calificativo *relacional* se debe al hecho que para establecer relaciones entre la información guardada, algunas tablas deben incorporar claves de otras: *claves ajenas*. Por ejemplo, para guadar en qué asignatura está matriculado cada alumno tendríamos la siguiente tabla Matrícula.

Matricula	
dni	codigo
20123456A	13928
20123456A	13929
20123457B	13898
20123457B	13929

En esta tabla la columna *dni* es una clave *ajena*: viene de la tabla *Persona*; y la columna *código* también es una clave ajena, proveniente de la tabla *Asignatura*. ¿Y cuál es la clave de la tabla *Matricula*? La combinación de ambas columnas *dni+codigo*.

1.1 SQL

Las bases de datos relacionales disponen de un lenguaje llamado *Structured Query Language* (SQL) para crear tablas; insertar, cambiar o borrar información; y para consultarla mediante interrogaciones.

¡ Atención !

Tras realizar los ejercicios de esta sección, busca en internet un tutorial sobre SQL para completar tu formación y como material de consulta futuro.

Ejercicio.

Hay muchos sistemas de gestión de bases de datos relacionales, incluso on-line (como "sql on-line"). Aquí vamos a usar uno muy sencillo y fácil de instalar: sqlite3.

- Entra en www.sqlite.org/download.html y descarga el fichero ".zip" del "Precompiled Binaries for ..." adecuado. Dentro del .zip, busca el ejecutable sqlite3 y copialo al directorio de trabajo.
- El código para crear la tabla Persona es el siguiente.

```
-- crearPersona.sql

create table Persona (
 dni char(9) not null,
 nombre varchar(20) not null,
 apellidos varchar(80) not null,
 primary key (dni)
);
```

Copialo en un fichero crearPersona.sql y ejecuta

.

```
sqlite3 datos.bd < crearPersona.sql
```
Get-Content .\crearPersona.sql | .\sqlite3.exe datos.bd

Este comando crea un fichero datos.bd para la base de datos y, dentro de ella, crea la tabla Persona.

- El código para insertar datos en la tabla Persona es el siguiente.

```
-- insertarPersonas.sql

insert into Persona values ('20123456A', 'Juan', 'Perez Perez');
insert into Persona values ('20123457B', 'Maria', 'Garcia');
insert into Persona values ('20123458C', 'Jose', 'Perez');
insert into Persona values ('20123459D', 'Jose', 'Ramirez');
```

Copialo en un fichero insertarPersonas.sql y ejecuta

```
sqlite3 datos.bd < insertarPersonas.sql
```
Get-Content .\insertarPersonas.sql | .\sqlite3.exe datos.bd

- Ahora vamos a entrar en el "shell" de comandos de sqlite3. Ejecuta:

```
sqlite3 datos.bd
```
.\sqlite3 datos.bd

Más tarde, saldremos del entorno mediante

```
.quit
```

La orden

```
.tables
```

nos muestra la lista de tablas que tiene la base de datos. Debe salir Persona.
Desde aquí, se puede también ejecutar ficheros con código SQL haciendo

```
.read fichero.sql
```

Si queremos consultar la estructura que tiene la tabla Persona, haremos

```
.schema Persona
```

- Para realizar consultas se usa la sentencia de SQL select. Por ejemplo, estando en el shell de sqlite3,
  - obtener toda la información de la tabla Persona

```
select * from Persona;
```

Debe salir

```
20123456A|Juan|Perez Perez
20123457B|Maria|Garcia
20123458C|Jose|Perez
20123459D|Jose|Ramirez
```

- Obtener los apellidos de todas las personas

```
select apellidos from Persona;
```

- Obtener los datos de la persona apellidada "Garcia".

```
select * from Persona where apellidos='Garcia';
```

Comillas simples ''

Ejercicio.

1. Inserta tus datos en la tabla Persona y luego búscalos.
2. Añade a la base de datos la tabla Asignatura e inserta algunas asignaturas como en el ejemplo que vimos más arriba.
3. Fíjate cómo se crea una tabla, como Matricula, que tiene claves ajenas:

```
-- crearMatricula.sql
create table Matricula (
 dni char(9) not null,
 codigo char(8) not null,
 foreign key (dni) references Persona(dni),
 foreign key (codigo) references Asignatura(codigo),
 primary key (dni,codigo)
);
```

4. Crea dicha tabla y "matrículate en programación 2".

Consultas (select)

Antes hemos visto un ejemplo de cómo realizar una búsqueda en una única tabla.

```
select * from Persona where apellidos='Garcia';
```

Veamos ahora cómo realizar una consulta que involucre a dos tablas. Por ejemplo, *apellidos de las personas matriculadas en la asignatura cuyo código es 13928*.

```
select Persona.apellidos
from Persona, Matricula
where Matricula.codigo='13928' and Matricula.dni=Persona.dni;
```

Esta es la explicación de la anterior consulta:

- sabemos que hay dos tablas involucradas porque el código de asignatura podemos encontrarlo en la tabla Matricula; y, por otro lado, el dato que hemos de obtener (apellidos) está en la tabla Persona. Los nombre de las tablas afectadas se indican en la parte from.
- La primera condición de la parte where exige que el código de matrícula sea 13928. De las filas de Matricula que cumplan lo anterior pedimos que su dni sea el mismo tanto en la tabla Matrícula como en la tabla Persona. Es importante darse cuenta que justamente dni es la columna que tiene ambas tablas en común.

Ejercicio.

1. Prueba las anteriores consultas.
2. Escribe una consulta que involucre a las tres tablas: *obtener el nombre de todas las personas matriculadas de Programación 2*.

select Persona.nombre from Persona, Matricula, Asignatura where Asignatura.nombre = 'Programacion2' and Persona.dni = Matricula.dni and Matricula.codigo = Asignatura.codigo;

Ejercicio.

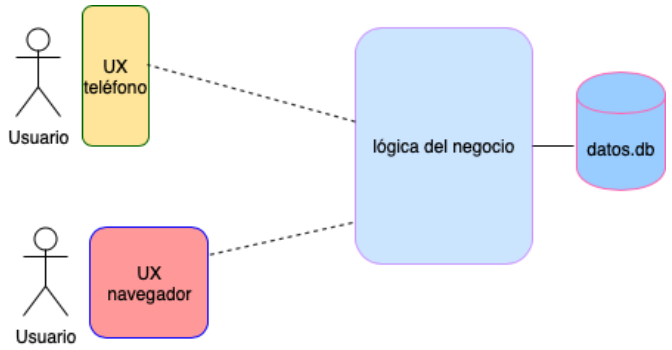
Aprende a realizar otras operaciones en SQL como: borrar filas, actualizar una fila (sin borrar y volver a insertar), borrar un tabla, etc.

delete from Persona where nombre = 'Juan';
drop table Matricula;
update Asignatura set nombre = 'Programacion3' where codigo = 'PROG1';

2

Bases de datos desde node.js

Al desarrollar una aplicación sobre una base de datos, su esquema lo crearemos (`create table`) con algún shell o utilidad de la propia base de datos; más o menos como lo hemos hecho antes, porque las tablas sólo se crean una vez. Pero las consultas (`select`), altas (`insert`), actualizaciones (`update`), y borrados (`delete`) serán enviadas a la base de datos desde un programa que desarrollemos y que, indirectamente, atenderá al usuario. Este programa se conoce como "lógica del negocio".



Vamos a ver una versión elemental de una lógica del negocio, siguiendo con el ejemplo de las asignaturas.

Ejercicio.

1. En un nuevo directorio de trabajo, crea esta estructura de directorios.

```
|-- docs
|-- ux
|-- servidorREST
|     |-- test
|-- bd
|-- logica
      |-- test
```

2. Copia a la carpeta bd el fichero con la base de datos datos.bd, así como los demás scripts *.sql y el programa sqlite3.
3. Lee y copia a la carpeta logica el siguiente fichero package.json

```
{
 "//": "npm run test",
 "name": "logica",
 "description": "2019",
 "license": "LGPL-3.0",
 "version": "0.0.1",
 "repository": "no hay",
 "scripts": {
  "test": "./node_modules/mocha/bin/mocha"
 },
 "dependencies": {
  "sqlite3": "*"
 },
 "devDependencies": {
  "mocha": "*"
 }
}
```

4. En el directorio logica ejecuta

```
npm install
```

para instalar las dependencias (sqlite3 y mocha).

5. Copia el siguiente fichero, Logica.js, a la carpeta logica

```
// ..................................................................
// Logica.js
// ..................................................................
const sqlite3 = require( "sqlite3" )
```

```
// ..............................................................
// ..............................................................
module.exports = class Logica {

	// ..........................................................
	// nombreBD: Texto
	// -->
	//    constructor ()  -->
	// ..........................................................
	constructor( nombreBD, cb  ) {
		this.laConexion = new sqlite3.Database(
			nombreBD,
			( err ) => {
				if( ! err )  {
					this.laConexion.run( "PRAGMA foreign_keys = ON" )
				}
				cb( err)
			})
	} // ()

	// ..........................................................
	// nombreTabla:Texto
	//     -->
	//         borrarFilasDe() -->
	// ..........................................................
	borrarFilasDe( tabla ) {
		return new Promise( (resolver, rechazar) => {
			this.laConexion.run(
				"delete from " + tabla + ";",
				(err)=> ( err ? rechazar(err) : resolver() )
```

```
                )
        })
} // ()

// ..............................................................
//          borrarFilasDeTodasLasTablas() -->
// ..............................................................
async borrarFilasDeTodasLasTablas() {
        await this.borrarFilasDe( "Matricula" )
        await this.borrarFilasDe( "Asignatura" )
        await this.borrarFilasDe( "Persona" )
} // ()

// ..............................................................
// datos:{dni:Texto, nombre:Texto: apellidos:Texto}
//     -->
//         insertarPersona() -->
// ..............................................................
insertarPersona( datos ) {
        var textoSQL =
                'insert into Persona values( $dni, $nombre, $apellidos );'
        var valoresParaSQL = { $dni: datos.dni, $nombre: datos.nombre,
                                                $apellidos: datos.apellidos }

        return new Promise( (resolver, rechazar) => {
                this.laConexion.run( textoSQL, valoresParaSQL, function( err ) {
                        ( err ? rechazar(err) : resolver() )
                })
        })
} // ()
```

```
// ..............................................................
// dni:Texto
//         -->
//            buscarPersonaPorDNI() <--
//         <--
// {dni:Texto, nombre:Texto: apellidos:Texto}
// ..............................................................
buscarPersonaConDNI( dni ) {
        var textoSQL = "select * from Persona where dni=$dni";
        var valoresParaSQL = { $dni: dni }

        return new Promise( (resolver, rechazar) => {
                this.laConexion.all( textoSQL, valoresParaSQL,
                                                        ( err, res ) => {
                                                                ( err ? rechazar(err) : resolver(res) )
                                                        })
        })
} // ()

// ..............................................................
//          cerrar() -->
// ..............................................................
cerrar() {
        return new Promise( (resolver, rechazar) => {
                this.laConexion.close( (err)=>{
                        ( err ? rechazar(err) : resolver() )
                })
        })
} // ()
```

```
} // class
// ..................................................................
// ..................................................................
```

6. Copia el siguiente fichero, mainTest1.js, a la carpeta logica/test

```
// ....................................................
// mainTest1.js
// ....................................................
const Logica = require( "../Logica.js" )

var assert = require ('assert')

// ....................................................
// main ()
// ....................................................
describe( "Test 1: insertar una persona", function() {

        // ................................................
        // ................................................
        var laLogica = null

        // ................................................
        // ................................................
        it( "conectar a la base de datos", function( hecho ) {
                laLogica = new Logica(
                        "../bd/datos.bd",
                        function( err ) {
                                if ( err ) {
                                        throw new Error ("No he podido conectar con datos.db")
                                }
```

```
                        hecho()
                })
}) // it

// ...................................................
// ...................................................
it( "borrar todas las filas", async function() {

        await laLogica.borrarFilasDeTodasLasTablas()

}) // it

// ...................................................
// ...................................................
it( "puedo insertar una persona",
        async function() {

                await laLogica.insertarPersona(
                        {dni: "1234A", nombre: "Pepe",
                         apellidos: "García Pérez" } )

                var res = await laLogica.buscarPersonaConDNI( "1234A" )

                assert.equal( res.length, 1, "¿no hay un resulado?" )
                assert.equal( res[0].dni, "1234A", "¿no es 1234A?" )
                assert.equal( res[0].nombre, "Pepe", "¿no es Pepe?" )
        }) // it

// ...................................................
// ...................................................
```

```
        it( "no puedo insertar una persona con dni que ya está",
                async function() {
                        var error = null

                        try {
                                await laLogica.insertarPersona(
                                        {dni: "1234A", nombre: "Pepa",
                                         apellidos: "Pérez Pérez" } )

                        } catch( err ) {
                                error = err
                        }

                        assert( error, "¿Ha insertado el dni que ya estaba 1234A? (¿No ha pasado por el catch()?" )
                }) // it

        // ................................................
        // ................................................
        it( "cerrar conexión a la base de datos",
                async function() {
                        try {
                                await laLogica.cerrar()
                        } catch( err ) {
                                // assert.equal( 0, 1, "cerrar conexión a BD fallada: " + err)
                                throw new Error( "cerrar conexión a BD fallada: " + err)
                        }
                }) // it

}) // describe
```

7. Finalmente, en el directorio logica ejecuta

```
npm test
```

Si todo ha ido bien, deberías ver lo siguiente:

```
?? npm test

> logica@0.0.1 test /home/epsg/tmp/01-logica/logica
> ./node_modules/mocha/bin/mocha



  Test 1: insertar una persona
    ✓ conectar a la base de datos
    ✓ borrar todas las filas
    ✓ puedo insertar una persona
    ✓ no puedo insertar una persona con dni que ya está
    ✓ cerrar conexión a la base de datos


  5 passing (38ms)
```

Si no es así, revisa con atención todos los pasos.

2.1 Análisis de Logica.js

Este fichero contiene una clase con métodos "significativos", relacionados con la aplicación que se pretende desarrollar. En nuestro caso, gestión de matrículas en asignaturas.

- El constructor recibe el nombre del fichero que contiene la base de datos y abre una conexión con ella usando `new sqlite3.Database`. Este constructor recibe un callback para indicarnos cuándo termina y si hay algún error, cosa que transladamos también mediante un callback a quien ha llamado al constructor.

```
    constructor( nombreBD, cb  ) {
     this.laConexion = new sqlite3.Database(
     nombreBD,
     ( err ) => {
      if( ! err )  {
       this.laConexion.run("PRAGMA foreign_keys=ON")
      }
      cb( err)
     })
    } // ()
```

-

```
    cerrar() {
     return new Promise( (resolver, rechazar) => {
      this.laConexion.close( (err)=>{
       ( err ? rechazar(err) : resolver() )
      })
     })
    } // ()
```

Este método cierra la conexión con la base de datos usando `this.laConexion.close()`. Como este método es asíncrono, le enviamos una función anónima `(err)=>...` como callback, pero nosotros devolvemos una promesa para posteriormente simplificar nuestro código usando `await`.

- Disponemos de un método de utilidad para borrar todas las filas de una tabla dado su nombre. Para enviar el comando SQL `delete` a a la base de datos utilizamos el método `this.laConexion.run()` que, de nuevo, es asíncrono. Como antes, le damos una función anónima `(err) => ...` pero devolvemos una promesa para luego poder usar `await`.

```
borrarFilasDe( tabla ) {
 return new Promise( (resolver, rechazar) => {
  this.laConexion.run( "delete from " + tabla + ";",
   (err)=> ( err ? rechazar(err) : resolver() )
  )
 })
} // ()
```

-
```
async borrarFilasDeTodasLasTablas() {
  await this.borrarFilasDe( "Matricula" )
  await this.borrarFilasDe( "Asignatura" )
  await this.borrarFilasDe( "Persona" )
 } // ()
```

Gracias a que `borrarFilasDe()` devuelve una promesa, ahora podemos aprovecharnos y escribir un nuevo método `borrarFilasDeTodasLasTablas()` que llame a varias veces a aquél método asíncrono de forma clara usando `await`.

- El método para realizar una alta en la tabla Persona recibe un objeto JSON con los valores de dni, nombre y apellidos. El texto SQL insert ... se envía a la base de datos con this.laConexion.run(). En este texto utilizamos comodines (empiezan por $) como $dni. Y para indicar los valores para sustituir a cada comodín hemos de preparar un objeto JSON. Como es habitual, devolvemos una promesa.

```
insertarPersona( datos ) {
 var textoSQL =
  'insert into Persona values( $dni, $nombre, $apellidos );'
 var valoresParaSQL = { $dni : datos.dni, $nombre : datos.nombre,
        $apellidos : datos.apellidos }
 return new Promise( (resolver, rechazar) => {
  this.laConexion.run( textoSQL, valoresParaSQL,
      function( err ) {
   ( err ? rechazar(err) : resolver() )
    })
 })
} // ()
```

- Finalmente, tenemos un método para recuperar todos los datos de una persona a partir de su dni. Como antes, preparamos un texto SQL con un comodín y un objeto con el valor de éste. Usamos this.laConexion.all() para enviar el texto SQL. Hay que notar que all() devuelve un array con los resultados (está vacío si no hay).

```
buscarPersonaConDNI( dni ) {
 var textoSQL =
      "select * from Persona where dni=$dni";
 var valoresParaSQL = { $dni : dni }
 return new Promise( (resolver, rechazar) => {
  this.laConexion.all( textoSQL, valoresParaSQL,
        ( err, res ) => {
         ( err ? rechazar(err) : resolver(res) )
        })
 })
} // ()
```

Ejercicio.

Haciendo ingenería inversa a partir de su código, deduce el diseño de la clase Logica para poder ampliarlo en el futuro. Recuerda que en un diseño lógico *nunca* aparecen ni callbacks, ni promesas, ni punteros.

2.2 Test para Logica.js

Vamos a analizar los test de ejemplo que hemos escrito para la anterior versión de la lógica en mainTest1.js.

- El primer paso del test consiste en crear un objeto Logica, cuyo constructor se conecta a la base de datos que le indicamos. Le pasamos un callback para saber cuándo la conexión está hecha y si ha habido un error, en cuyo caso disparamos una excepción (throw new Error(...). Como el constructor ha de devolver un objeto, no pudimos devolver una promesa y aquí no podemos usar await.

```
it("conectar a la base de datos", function(hecho){
      laLogica = new Logica( "../bd/datos.bd",
        function( err ) {
          if ( err ) {
            throw new Error ("No he podido conectar
                                con datos.db")
          }
          hecho() // avisamos que el it esta hecho
        })
}) // it
```

-
```
it( "borrar todas las filas", async function() {
 await laLogica.borrarFilasDeTodasLasTablas()
}) // it
```

El siguiente paso consiste en borrar todas las filas de las tres tablas. Hacemos esto para estar seguros que no hay datos guardados previamente. Como borrarFilasDeTodasLasTablas() devuelve al promesa podemos usar await (dentro de la función marcada como async. Aquí no es necesario llamar a hecho()).

- Ahora insertamos una persona y la buscamos dando su dni. Si la encuentra, obtendremos en res un array de una única casilla con un JSON con sus datos. Por eso planteamos los tres assert() que vemos.

```
it("puedo insertar una persona", async function(){
  await laLogica.insertarPersona( {dni : "1234A",
      nombre : "Pepe", apellidos : "García Pérez" })

  var res = await laLogica.buscarPersonaConDNI( "1234A" )

  assert.equal( res.length, 1, "¿no hay un resulado?" )
  assert.equal( res[0].dni, "1234A", "¿no es 1234A?" )
  assert.equal( res[0].nombre, "Pepe", "¿no es Pepe?" )
}) // it
```

- Por último, tratamos de insertar una persona utilizando un dni ya utilizado. Esta operación debe fallar, disparando una excepción que nos lleve al catch(). Para comprobar que el catch ocurre, allí copiamos err en error, para luego verificar en el assert() que error ha dejado de valer null.

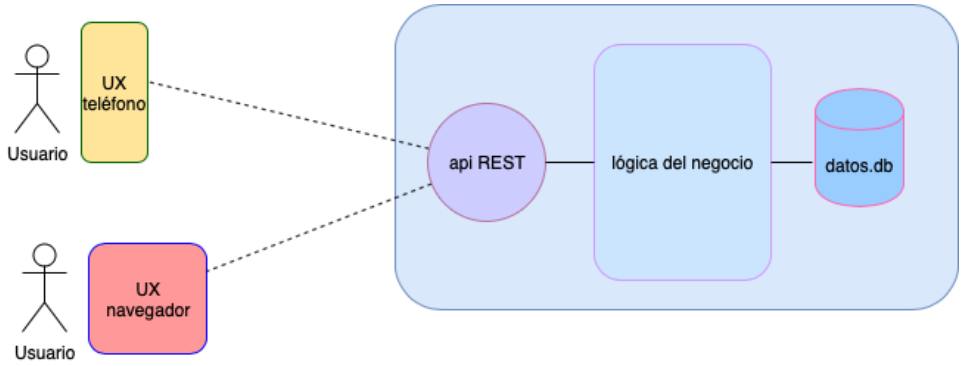
```
it( "no puedo insertar una persona con dni que ya está",
 async function() {
 var error = null

 try {
   await laLogica.insertarPersona(
        {dni : "1234A", nombre : "Pepa",
         apellidos : "Pérez Pérez" } )

 } catch( err ) {
   error = err
 }

assert( error, "¿Ha insertado el dni que ya estaba 1234A?
        (¿No ha pasado por el catch()?" )
}) // it
```

Ejercicio.

1. Amplia la lógica del negocio para poder dar de alta nuevas asignaturas: diseña los métodos necesarios, impleméntalos y escribe test para probarlos en mainTest2.js.
2. Amplia la lógica del negocio para poder realizar matrícula: diseña los métodos necesarios, impleméntalos y escribe test para probarlos en mainTest3.js.
3. Amplia la lógica del negocio para poder realizar una consulta que devuelva los códigos de asignatura en que una persona está matriculada, dando los apellidos de dicha persona. Diseña los métodos necesarios, impleméntalos y escribe test para probarlos en mainTest4.js.

3 Acceder a la lógica mediante un servidor REST

Es evidente que la lógica del negocio y su base de datos están instaladas en un ordenador distinto del que los usuarios puedan utilizar. Sin embargo, las aplicaciones de usuario debe poder comunicarse con la lógica del negocio. Por ello, debemos exponer la lógica para que pueda ser utilizada *remotamente*. Para este propósito hay multitud de sistemas y bibliotecas de comunación que podríamos utilizar. Aquí vamos a explicar cómo hacerlo mediante una api REST.



Un servidor REST es en realidad un servidor HTTP que en lugar de responder con páginas html, invoca métodos de la lógica y devueve su respuesta. Es decir, cuando un servidor REST recibe la petición

```
GET /persona/20123456A
```

llama al método de la lógica `buscarPersonaConDNI( 20123456A )` y devuelve el JSON que obtiene:

```
{dni: "20123456A", nombre: "Juan", apellidos: "Perez Perez"}
```

(Un servidor web normal hubiera intentado devolver la página `20123456A.html` que, en realidad, no existe).

Hoy en día, los servicios de internet, con muy pocas excepciones se ofrecen a través de una api REST. Ello se debe a tres importantes ventajas:

- son fáciles de escribir porque, entre otras cosas, se basan en el muy conocido protocolo HTTP.
- tienen una buena estructura porque separan la lógica de la interfaz de usuario.
- permite que la lógica sea usada por distintas interfaces de usuario, escritas en lenguajes diferentes para plataformas diferentes.

En contraposición a la segunda ventaja, hay entornos que propician el caos mezclando lógica e interfaz de usuario. Un ejemplo de esto son las páginas PHP donde es habitual mezclar código HTML con scripts PHP. Lo cierto es que no hay nada que impida la separación de intereses (interfaz vs. lógica) en PHP, e incluso es perfectamente posible y fácil escribir un servidor REST en PHP. Pero generalmente, la preparación autodidacta de muchos programadores provoca que las cosas se hagan desordenadamente ya que desconocen principios fundamentales del desarrollo de software como la arquitectura Modelo-Vista-Controlador y el fundamental principio de diseño separación de intereses.

3.1 Servidor REST sencillo

Como hemos dicho, un servidor REST es en realidad un servidor HTTP que, en vez de devolver páginas html, realiza acciones y devuelve otro tipo de respuestas. Los clientes de un servidor HTTP son los navegadores, pero en el caso de un servidor REST son programas excritos ex profeso (que se pueden ejecutar en un teléfono o también dentro de una página html). Veamos primero cómo son las peticiones y respuestas según el protocolo HTTP.

Una petición de HTTP es un trozo de texto que un cliente envía al servidor, como el siguiente:

```
POST /alta HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 66

{ "dni": "1234A", "nombre": "Pepe", "apellidos", "García Pérez"}
```

- La petición tiene dos partes separadas por una línea en blanco. La primera parte se llama *cabecera* ("header"), y la segunda *cuerpo* o *carga* ("body" o "payload").
- La primera línea de la cabecera indica el verbo y el nombre del recurso a que afecta dicho verbo. Los verbos pueden ser `GET`, `POST`, `PUT` o `DELETE` (aunque hay algunas posibilidades más). El nombre del recurso tiene el aspecto de una ruta de directorio (nombres separados por `/`) y puede terminar en una interrogación como `?clave1=valo1r&clave2=valor2`
- La carga es información que queremos enviar al servidor. En principio, `GET` y `DELETE` no necesitan carga. La carga puede estar codificada de muchas formas, por ejemplo texto plano `text/plain` o JSON `application/json`. El formato de la carga *debe* indicarse mediante el campo `Content-Type` en la cabecera, para que el servidor sepa cómo debe interpretar la carga a su recepción.

Cuando un servidor HTTP termina el procesamiento de una petición, debe responder un trozo de texto como el siguiente:

```
HTTP/1.1 200 OK
Date: Sun, 10 Mar 2019 18:40:53 GMT
Content-Length: 88
Content-Type: text/html

<html>
<body>
<h1>Hola, World!</h1>
</body>
</html>
```

- La respuesta también tiene cabecera y, opcionalmente, carga; separadas por una línea en blanco.
- La primera línea de la cabecera es el *estado*. Si la petición ha tenido éxito veremos `200 OK`. En caso contrario, hay una larga lista de códigos de error como `400 BAD REQUEST` o el famoso `404 NOT FOUND`.
- Si hay carga en la respuesta, es necesario indicar su formato mediante `Content-Type` en la cabecera.

Express

Para escribir un servidor REST en *node.js* utilizaremos la biblioteca `express`. Sucintamente, los pasos son

```
// 1. crear el servidor
var servidorExpres = express()

// 2. configurar sus reglas
servidorExpres.get( ...
...
servidorExpres.post( ...

// 3. arrancarlo
servidorExpress.listen( 8080, ...
```

La parte más importante es configurar las *reglas* para que el servicio sepa qué hacer cuando llegan peticiones HTTP como GET /persona/1234A. Por ejemplo, esta es una regla para responder "¡Funciona!" cuando llegue GET /prueba.

```
servidorExpress.get('/prueba', function( peticion, respuesta ){
 console.log( " * GET /prueba " )
 respuesta.send( "¡Funciona!" )
}) // get /prueba
```

En los ejemplos que veremos en los siguientes ejercicios, debes aprender a consultar la información que llega en una petición (nombre del recurso, interrogación, carga y transformación de la carga) y cómo crear una respuesta y enviarla.

Request

Para enviar peticiones a un servidor REST y poder probarlo de forma automática utilizaremos la biblioteca request. Por ejemplo, el siguiente código envía una petición POST (con carga) y comprueba la respuesta:

```
var datosPersona = { dni : "1234A", nombre : "Pepe", apellidos : "García Pérez" }
request.post(
 { url : "http://localhost:8080/alta",
   headers : { 'User-Agent' : 'jordi', 'Content-Type' : 'application/json' },
   body : JSON.stringify( datosPersona ) },
 function( err, respuesta, carga ) {
  assert.equal( err, null, "¿ha habido un error?" )
  assert.equal( respuesta.statusCode, 200, "¿El ócdigo no es 200 (OK)" )
)}
```

Para probar un servidor, recuerda que éste debe estar arrancado antes de ejecutar los test.

Ejercicio.

1. Crea un directorio de trabajo como éste:

```
|-- servidorREST
      |-- test
```

2. Lee y copia a la carpeta servidorREST el siguiente fichero package.json

```
{
 "//": "npm run test",
 "name": "servidorREST",
 "description": "2019",
 "license": "LGPL-3.0",
 "version": "0.0.1",
 "repository": "no hay",
 "scripts": {
  "servidor": "node mainServidorREST.js",
  "test": "./node_modules/mocha/bin/mocha"
 },
 "dependencies": {
  "express": "*",
  "request": "*",
  "sqlite3": "*"
 },
 "devDependencies": {
  "mocha": "*"
 }
}
```

3. En el directorio servidorREST ejecuta

```
npm install
```

para instalar las dependencias (express, request y mocha).

4. Lee detenidamente y copia el siguiente fichero, mainServidorREST.js, a la carpeta servidorREST

```
// .....................................................................
// mainServidorREST.js
// .....................................................................
const express = require( 'express' )
const bodyParser = require( 'body-parser' )

// .....................................................................
// main()
// .....................................................................
function main() {
	// creo el servidor
	var servidorExpress = express()

	// para poder acceder a la carga de la petición http, asumiendo que es JSON
	servidorExpress.use ( bodyParser.text({type : 'application/json'}) )

	// cargo las reglas REST
	var reglas = require( "./ReglasREST.js")
	reglas.cargar( servidorExpress )

	// arrancao el servidor
	var servicio = servidorExpress.listen( 8080, function() {
		console.log( "servidor REST escuchando en el puerto 8080 ")
	})
```

```
        // capturo control-c para cerrar el servicio ordenadamente
        process.on('SIGINT', function() {
         console.log (" terminando ")
         servicio.close ()
        })
} // ()

// ..................................................................
// ..................................................................
main()
// ..................................................................
// ..................................................................
```

5. Estuda con atención y copia el siguiente fichero, ReglasREST.js, a la carpeta servidorREST

```
// ..................................................................
// ReglasREST.js
// ..................................................................
module.exports.cargar = function( servidorExpress ) {
    // ..................................................
    // GET /prueba
    // ..................................................
    servidorExpress.get('/prueba/', function( peticion, respuesta ){
              console.log( " * GET /prueba " )
              respuesta.send( "¡Funciona!" )
       }) // get /prueba

    // ..................................................
    // GET /longitud/<palabra>
    // ..................................................
    servidorExpress.get(
```

```
                '/longitud/:palabra',
                function( peticion, respuesta ){
                        console.log( " * GET /longitud " )
                        var palabra = peticion.params.palabra

                        var solucion = { palabra : palabra, longitud : palabra.length }

                        respuesta.send( JSON.stringify( solucion ) )
                }) // get /longitud

// ....................................................
// GET /dividir?a=<num>&b=<num>
// ....................................................
servidorExpress.get(
                '/dividir',
                function( peticion, respuesta ){
                        console.log( " * GET /dividir " )

                        var a = peticion.query.a
                        var b = peticion.query.b

                        if( isNaN(a) || isNaN(b) || b == 0 ) {
                                // si a o b no son números, o b es 0
                                // no se puede dividir
                                // (400 = bad request)
                                respuesta.status(400).send(" no puedo dividir ");
                                return
                        }

                        var solucion = { a : a, b : b, division : a/b }
```

```
                        respuesta.send( JSON.stringify( solucion ) )
                }) // get /dividir

        // ....................................................
        // POST /alta
        // ....................................................
        servidorExpress.post(
                '/alta',
                function( peticion, respuesta ){
                        console.log( " * POST /alta " )

                        var datos = JSON.parse( peticion.body )

                        console.log( datos.dni )
                        console.log( datos.nombre )
                        console.log( datos.apellidos )

                        // supuesto procesamiento
                        if( datos.dni == "1234A" ) {
                                respuesta.send( "OK" )
                        } else {
                                // 404 = not found
                                respuesta.status( 404 ).send( "no acertaste con el dni" )
                        }

                }) // get /dividir
} // ()

// ....................................................................
// ....................................................................
```

```
// .....................................................................
// .....................................................................
```

6. Analiza y copia el siguiente fichero, mainTest1.js, a la carpeta mainServidorREST/test

```
// ....................................................
// mainTest1.js
// ....................................................
var request = require ('request')
var assert = require ('assert')

// ....................................................
// ....................................................
const IP_PUERTO="http://localhost:8080"

// ....................................................
// main ()
// ....................................................
describe( "Test 1 : Recuerda arrancar el servidor", function() {

      // ................................................
      // ................................................
      it( "probar que GET /prueba responde ¡Funciona!", function( hecho ) {
            request.get(
                  { url : IP_PUERTO+"/prueba", headers : { 'User-Agent' : 'jordi' }},
                  function( err, respuesta, carga ) {
                        assert.equal( err, null, "¿ha habido un error?" )
                        assert.equal( respuesta.statusCode, 200, "¿El código no es 200 (OK)" )
                        assert.equal( carga, "¡Funciona!", "¿La carga no es ¡Funciona!?" )
                        hecho()
                  } // callback()
```

```
        ) // .get
}) // it

// ..............................................
// ..............................................
it( "probar GET /longitud", function( hecho ) {
        request.get(
                { url : IP_PUERTO+"/longitud/hola",
                  headers : { 'User-Agent' : 'jordi' }},
                function( err, respuesta, carga ) {
                        assert.equal( err, null, "¿ha habido un error?" )
                        assert.equal( respuesta.statusCode, 200, "¿El código no es 200 (OK)" )
                        var solucion = JSON.parse( carga )
                        assert.equal( solucion.longitud, 4, "¿La longitud no es 4?" )

                        hecho()
                } // callback
        ) // .get
}) // it

// ..............................................
// ..............................................
it( "probar GET /dividir", function( hecho ) {
        request.get(
                { url : IP_PUERTO+"/dividir?a=10&b=2.5",
                  headers : { 'User-Agent' : 'jordi' }},
                function( err, respuesta, carga ) {
                        assert.equal( err, null, "¿ha habido un error?" )
                        assert.equal( respuesta.statusCode, 200, "¿El código no es 200 (OK)" )
                        var solucion = JSON.parse( carga )
```

```
                                assert.equal( solucion.division, 4, "¿El cociente es no es 4?" )
                                hecho()
                        } // callback
                ) // .get
        }) // it

        // ................................................
        // ................................................
        it( "probar POST /alta", function( hecho ) {
                var datosPersona = { dni : "1234A", nombre : "Pepe", apellidos : "García Pérez"
                }

                request.post(
                        { url : IP_PUERTO+"/alta",
                          headers : { 'User-Agent' : 'jordi', 'Content-Type' : 'application/json' },
                          body : JSON.stringify( datosPersona )
                        },
                        function( err, respuesta, carga ) {
                                assert.equal( err, null, "¿ha habido un error?" )
                                assert.equal( respuesta.statusCode, 200, "¿El código no es 200 (OK)" )
                                assert.equal( carga, "OK", "¿La carga no es OK" )
                                hecho()
                        } // callback
                ) // .post
        }) // it
}) // describe
```

7. Finalmente, en el directorio servidorREST ejecuta

```
npm run servidor
```

y en *otra* ventana

```
npm test
```

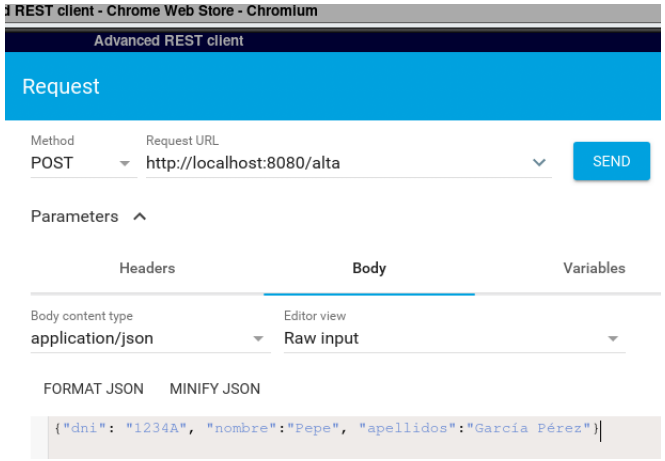
Si todo ha ido bien, deberías ver lo siguiente:

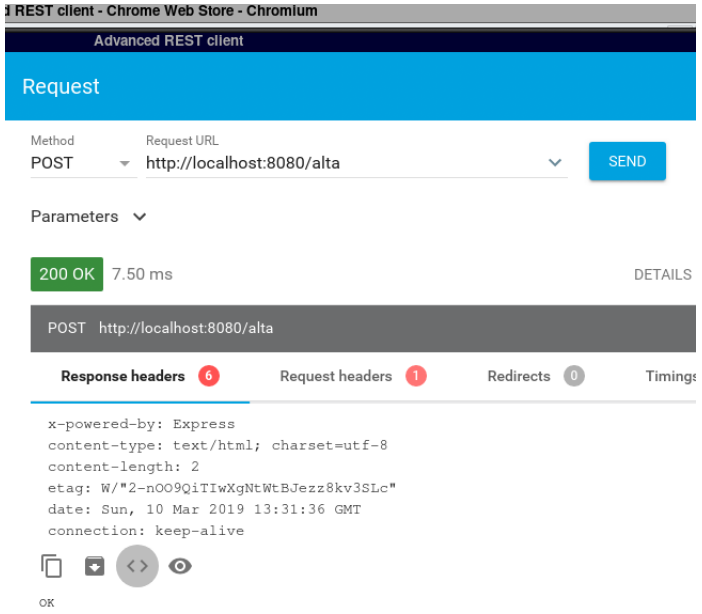
```
Test 1: Recuerda arrancar el servidor
  ✓ probar que GET /prueba responde ¡Funciona!
  ✓ probar GET /longitud
  ✓ probar GET /dividir
  ✓ probar POST /alta


4 passing (68ms)
```

Si no es así, revisa con atención todos los pasos.

Ejercicio.

Instala en tu navegador un *cliente REST* y aprende a enviar peticiones a nuestro servidor:





Ejercicio.

Utilizando el código proporcionado, amplia nuestro servidor de juguete añadiendo una regla de tipo POST que realice una división de dos números recibiendo y devolviendo los datos en formato JSON.

Prueba dicha regla utilizando un cliente REST y escribiendo un test automático.

3.2 Servidor REST para la lógica del negocio

Sabiendo ya cómo escribir y probar tanto un servidor REST como la lógica del negocio de una aplicación, vamos a ver cómo conectarlos. Para ello, vamos a retomar la aplicación de matrículas (Persona-Matricula-Asignatura).

Los puntos a los que hay que prestar atentción son:

- En el fichero mainServidorREST.js
  1. Importamos la lógica:

```
var Logica = require( "../logica/Logica.js" )
```

  2. Escribimos una función que cargue la lógica devolviendo una promesa

```
function cargarLogica( fichero ) {
   return new Promise( (resolver, rechazar) => {
      var laLogica = new Logica( fichero,
      ...
```

  3. Ahora la función main la etiquetamos con async y hacemos

```
async function main() {
  var laLogica = await cargarLogica( "../bd/datos.bd" )
  var servidorExpress = express()
  var reglas = require( "./ReglasREST.js")
  reglas.cargar( servidorExpress, laLogica )
```

- En el fichero Reglas.js, la función cargar recibe, además del objeto servidorExpress para configurar sus rutas, el objeto laLogica. Así, preparamos una regla GET /persona/<dni> que llame a laLogica.buscarPersonaConDNI().

```
module.exports.cargar = function( servidorExpress, laLogica ) {
 // regla GET /persona/:dni
 servidorExpress.get( "/persona/:dni", async function( peticion, respuesta ){
     var dni = peticion.params.dni
```

```
        var res = await laLogica.buscarPersonaConDNI( dni )
        ...
```

Ejercicio.

Recordemos el árbol de directorios de la aplicación de matrículas que ya tiene el contenido correcto en las carpetas bd y logica:

```
|-- docs
|-- ux
|-- servidorREST
|     |-- test
|-- bd
|-- logica
      |-- test
```

1. Lee y copia a la carpeta servidorREST el siguiente fichero package.json

```
{
        "//": "npm run test",
        "name": "servidorREST",
        "description": "2019",
        "license": "LGPL-3.0",
        "version": "0.0.1",
        "repository": "no hay",
        "scripts": {
                "servidor": "node mainServidorREST.js",
                "test": "./node_modules/mocha/bin/mocha"
        },
        "dependencies": {
                "express": "*",
```

```
                "request": "*"
        },
        "devDependencies": {
                "mocha": "*"
        }
}
```

2. En el directorio servidorREST ejecuta

```
npm install
```

para instalar las dependencias.

3. Lee y copia el siguiente fichero, mainServidorREST.js, a la carpeta servidorREST

```
// ..................................................................
// mainServidorREST.js
// ..................................................................

// ..................................................................
// ..................................................................
const express = require( 'express' )
const bodyParser = require( 'body-parser' )

const Logica = require( "../logica/Logica.js" )

// ..................................................................
// ..................................................................
function cargarLogica( fichero ) {
        return new Promise( (resolver, rechazar) => {
                var laLogica = new Logica( fichero,
                                                        function( err ) {
                                                                if ( err ) {
                                                                        rechazar( err )
```

```
                                                                        } else {
                                                                                resolver( laLogica )
                                                                        }
                                                                }) // new
        }) // Promise
} // ()

// ..............................................................
// main()
// ..............................................................
async function main() {
        var laLogica = await cargarLogica( "../bd/datos.bd" )

        // creo el servidor
        var servidorExpress = express()

        // para poder acceder a la carga de la petición http
        // asumiendo que es JSON
        servidorExpress.use ( bodyParser.text({type: 'application/json'}) )

        // cargo las reglas REST
        var reglas = require( "./ReglasREST.js")
        reglas.cargar( servidorExpress, laLogica )

        // arranco el servidor
        var servicio = servidorExpress.listen( 8080, function() {
                console.log( "servidor REST escuchando en el puerto 8080 ")
        })
```

```
        // capturo control-c para cerrar el servicio ordenadamente
        process.on('SIGINT', function() {
         console.log (" terminando ")
         servicio.close ()
        })
} // ()
// ....................................................................
// ....................................................................
main()
// ....................................................................
// ....................................................................
```

4. Lee y copia el siguiente fichero, ReglasREST.js, a la carpeta servidorREST

```
// ....................................................................
// ReglasREST.js
// ....................................................................
module.exports.cargar = function( servidorExpress, laLogica ) {

    // ..................................................
    // GET /prueba
    // ..................................................
    servidorExpress.get('/prueba', function( peticion, respuesta ){
              console.log( " * GET /prueba " )
              respuesta.send( "¡Funciona!" )
       }) // get /prueba

    // ..................................................
    // GET /persona/<dni>
    // ..................................................
    servidorExpress.get(
```

```
                '/persona/:dni',
                async function( peticion, respuesta ){
                        console.log( " * GET /persona " )

                        // averiguo el dni
                        var dni = peticion.params.dni

                        // llamo a la función adecuada de la lógica
                        var res = await laLogica.buscarPersonaConDNI( dni )

                        // si el array de resultados no tiene una casilla ...
                        if( res.length != 1 ) {
                                // 404: not found
                                respuesta.status(404).send( "no encontré dni: " + dni )
                                return
                        }

                        // todo ok
                        respuesta.send( JSON.stringify( res[0] ) )
                }) // get /persona
        } // cargar()
        // .....................................................................
        // .....................................................................
```

5. Utiliza un cliente REST para comprobar manualmente que la regla GET /persona/:dni funciona correctamente.
6. Escribe un test automático en servidorREST/test/mainTest1.jss para probar que la regla GET /persona/:dni funciona correctamente.

Ejercicio.

Ya dispones de los métodos en la lógica para dar de alta nuevas asiganturas, realizar matrículas, efectuar consultas, etc. Fueron implementados y probados en la anterior sección sobre la lógica. Ahora, de forma progresiva (de uno en uno), añade una regla REST para poder cada utilizar cada método remotamente. Implementa un test automático para cada nueva regla REST que añadas.

Ejercicio Final.

Utilizando lo aprendido en el proyecto del cuatrimestre 1B y en la asignatura sobre interfaces de usuario, desarrolla una aplicación de usuario para navegador, preferiblemente de tipo Single Page Application, para nuestra aplicación sobre matrículas. Hay que realizar obligatoriamente los siguientes pasos:

1. Diseño de la arquitectura.
   Deben confeccionarse los siguientes diseños usando herramientas apropiadas como por ejemplo draw.io:
   1. Arquitectura general (recordar la Tarea 1 "Arquitectura cliente-servidor").
   2. Esquema de la base de datos (tablas, columnas, tipos y claves).
   3. Lógica del negocio (Logica.js)
   4. Reglas REST (lista de VERBO+recurso, indicando cuál es y cómo se pasa la información entre cliente y servidor).
   5. Programa mainServidor.js y ReglasREST.js (métodos y funciones que contienen).
   6. Diseño (interno) del programa web de usuario. Recuerda que este programa debe tener un "proxy" de la lógica del negocio *exactamente* con los mismos métodos (excepto el constructor) que el original. La implementación de este proxy enviará peticiones REST a nuestro servidor REST.
   7. Diseño de la UX del programa web de usuario.
2. Planifica "hitos" ("sprints") muy concretos y lo más sencillos posible ("divide y vencerás"). Para cada hito establece previamente cuál es su criterio de aceptación.
3. Elige el hito que creas más importante e implementa el código necesario para lograrlo. Conseguir un hito suele implicar escribir código en cada nivel de la aplicación: base de datos, lógica del negocio, api

   REST, y programa de usuario. Recuerda que la lógica del negocio y el api REST deben tener sus test automáticos.
4. Repite el paso anterior hasta completar la aplicación.
5. En todo momento el código implementado y su diseño en papel deben coincidir. Si descubres fallos de diseño, corrige primero el diseño y luego haz la implementación. Nunca puede haber discrepancias entre ambos.

¡ Suerte !

A

Apuntes Adicionales sobre SQL

A.1

Diseños de Tablas: Formas Normales

Como tantos otras cuestiones en programación, las tablas de las bases de datos relacionales deben ser diseñadas de manera que tengan propiedades convenientes por lo que respecta a usabilidad y a la cantidad de almacenamiento necesario. Aunque el segundo criterio, dependiendo del caso, hoy en día puede no ser crucial; tener tablas bien diseñadas que faciliten su entendimiento y uso evitarán errores y permitirán su ampliación futura de forma sencilla.

En general los requerimientos básicos que debe cumplir el esquema de una base de datos (diseño de sus tablas) es que estén en *tercera forma normal*. Hay exigencias adicionales, tal vez menos relevantes en la práctica que cumplir los primeros tres requerimientos.

En formas normales podemos encontrar una explicación exhaustiva. De forma simplificada y práctica podemos decir que la normalización persigue evitar repeticiones innecesarias que, en una tabla, todos los campos dependan única y exclusivamente de la clave primaria en su totalidad. En cierto sentido, también podríamos decir que al diseñar una base de datos, una tabla que represente un "objeto real" no puden tener atributos (columnas) que no forman parte del objeto verdadero.

Ejemplo de tabla que no está en *primera forma normal* (por tener columnas "repetidas").

	Persona				
* dni	nombre	apellidos	teléfono1	teléfono2	teléfono3
20123456A	Juan	Perez Perez	612.45.23.43	-	-
20123457B	Maria	Garcia	695.35.28.24	912.23.82.47	-
20123458C	Jose	Perez	-	939.32.33.83	-

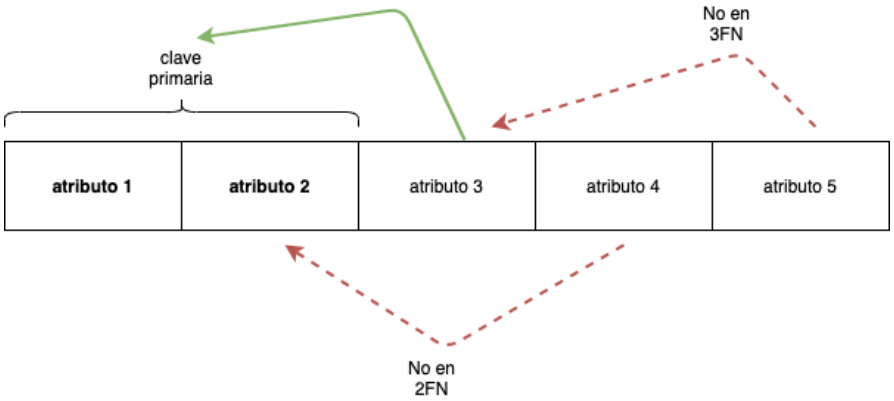
La tabla Persona debe tener únicamente los campos: dni, nombre, apellidos. Debe haber otra tabla llamata Telefono, con dos campos: dni y teléfono; siendo ambos conjuntamente la clave primaria.

Ejemplo de tabla que no está en *segunda forma normal* (porque un atributo depende solo de un campo de la clave primaria y no de toda ella).

	Docencia	
* dni profesor	* código asignatura	centro
20123456A	113-prg1	EPSG
20123456A	113-prg2	EPSG
20123777B	32-prg1	Fac. de Informática
20123777B	32-ia	Fac. de Informática

En tabla Docencia (cuya clave es dni+asignatura), el campo Centro solamente depende del código de asignatura (113=EPSG, 32=Fac. de Informática) por lo que hay información redundante. El centro donde se imparte un asignatura debería estar consignado en la tabla Asignatura (una única vez por asignatura).

En general, una forma sencilla de comprobar que el diseño de una tabla está normalizado es dibujar un pequeño diagrama de dependencias entre los campos de una tabla para poder verificar que dependen *exclusivamente* de toda la clave primaria.



A.2

Consultas Avanzadas

En los siguientes ejemplos usamos la base de datos online de W3Schools.

Cuando una consulta (select) involucra a dos o más tablas, se denomina *JOIN*. La notación original ("antigua") es esta:

```
-- Clientes y Proveedores del mismo país.
select CustomerName, Customers.City, SupplierName, Suppliers.City
from Customers, Suppliers
where Customers.Country=Suppliers.Country;
```

Hay un estándar de 1992 que cambia la notación para dar cabida a más tipos de *joins*. El original es el *inner join*.

```
-- Clientes y Proveedores del mismo país.
select CustomerName, Customers.City, SupplierName, Suppliers.City
from Customers inner join Suppliers on Customers.Country=Suppliers.Country;
```

Otro ojemplo, con cuatro tablas.

```
-- Pedidos que contienen productos de tipo "SeaFood" (67) (con inner join)
select Orders.OrderId, OrderDetails.ProductId, Products.ProductName
from
    ( (
        Orders inner join OrderDetails on Orders.OrderId = OrderDetails.OrderId
       ) inner join Products on Products.ProductId = OrderDetails.ProductId
    ) inner join Categories on Categories.CategoryId = Products.CategoryId
where Categories.CategoryName = "SeaFood"
order by Orders.OrderId;
```

```
-- Pedidos  que contienen  productos de  tipo "SeaFood"  (67) (forma tradicional)
select Orders.OrderId, OrderDetails.ProductId, Products.ProductName
from Orders, OrderDetails, Products, Categories
where Orders.OrderId = OrderDetails.OrderId
      and Products.ProductId = OrderDetails.ProductId
      and Categories.CategoryId = Products.CategoryId
      and Categories.CategoryName = "SeaFood"
order by Orders.OrderId;
```

```
-- Pedidos que contienen productos de tipo "SeaFood" (60 diferentes)
select distinct Orders.OrderId
from
    ( (
        Orders inner join OrderDetails on Orders.OrderId = OrderDetails.OrderId
       ) inner join Products on Products.ProductId = OrderDetails.ProductId
    ) inner join Categories on Categories.CategoryId = Products.CategoryId
where Categories.CategoryName = "SeaFood"
order by Orders.OrderId;
```

A parte de los habituales *inner join*, son posibles los *LEFT JOIN*, *RIGHT JOIN*, y *FULL OUTER JOIN*.

Por ejemplo, un *left join* incluye resultados de la primera tabla aunque éstos no estén relacionados con ninguna fila de la segunda.

La siguiente consulta devuelve, para cada cliente, los números de pedido que ha realizado. Aunque un cliente no haya realizado ningún pedido, aparece en el resultado (sin ningún valor para el número de pedido).

```
-- Números de Pedido que cada Cliente ha realizado.
select Customers.CustomerName, Orders.OrderID
from Customers
     left join Orders on Customers.CustomerID=Orders.CustomerID
order by Customers.CustomerName;
```

Si en la anterior consulta hubiéramos usado *inner join* solamente aparecerían los clientes que han realizado al menos un pedido.

group by

En ocasiones es necesario discriminar los resultados de una consulta en grupos que tengan el mismo valor para una columna, de forma que se pueda contar la cantidad de elementos que forma el grupo.

```
-- Cantidad de clientes en cada país.
select Country, count(CustomerID) as CantidadClientes
from Customers
group by Country;
```

Subconsultas

Una consulta puede contener otras en el sección *FROM* o en la sección *WHERE* (por ejemplo, usando *IN* o *NOT IN*).

```
-- Identificadores de Pedidos que contienen productos de tipo "SeaFood" (60 diferentes)
select distinct Orders.OrderId
from Orders, OrderDetails
where Orders.OrderId = OrderDetails.OrderID
      and OrderDetails.ProductId in
         (select ProductId from Products -- id de productos de tipo SeaFood
          where CategoryId in
                (select CategoryId from Categories where CategoryName = "SeaFood") -- id de SeaFood
   );
```

```
-- países que solo ofrecen un tipo de producto
select *
from ( select Country, count(CategoryId) as cuantasCategorias
       from ( select distinct Suppliers.Country , Categories.CategoryId
              from Products, Suppliers, Categories
              where Products.CategoryId=Categories.CategoryId and
                    Products.SupplierId=Suppliers.SupplierId
            )
       group by Country
)
where cuantasCategorias=1
```

```
-- países que solo ofrecen un tipo de producto y qué producto es
select Suppliers.Country, Products.ProductName, Categories.CategoryName
from Products, Suppliers, Categories
where Products.SupplierId=Suppliers.SupplierId and Products.CategoryId=Categories.CategoryId
      and Suppliers.Country in
      (select Country
       from (select Country, count(CategoryId) as cuantasCategorias
             from (select  distinct  Suppliers.Country , Categories.CategoryId
```

```
                    from Products, Suppliers, Categories
                    where Products.CategoryId=Categories.CategoryId
                          and Products.SupplierId=Suppliers.SupplierId
                ) group by Country
            ) where cuantasCategorias=1
)
```

```
-- Productos que ofrece cada país (para comprobar las anteriores consultas)
select Suppliers.Country, Products.ProductName, Categories.CategoryName
from Suppliers, Products, Categories
where Suppliers.SupplierId=Products.SupplierId and Products.CategoryId=Categories.CategoryId
order by Suppliers.Country, Categories.CategoryName
```

```
-- pedidos que contienen algún producto de España
select distinct OrderID
from OrderDetails
where OrderDetails.ProductId in
      (select ProductID
       from Products
       where Products.SupplierID in
             (select SupplierID from Suppliers where Suppliers.Country='Spain')
      )
```

```
-- pedidos que contienen algún producto de España (para comprobar)
select distinct OrderId, ProductName
from OrderDetails, Products, Suppliers
where OrderDetails.ProductId=Products.ProductId
      and Products.SupplierId=Suppliers.SupplierId
      and Suppliers.Country='Spain'
```

```
-- pedidos con todos sus productos del mismo país
select *
from (select OrderDetails.OrderID, count(Suppliers.Country) as contador
      from OrderDetails, Products, Suppliers
      where OrderDetails.ProductID = Products.ProductID
            and Products.SupplierID=Suppliers.SupplierID
      group by OrderID
     )
where contador=1;
```

```
-- pedidos con todos sus productos del mismo país y qué país es éste
select oi.OrderId, Suppliers.Country
from (select OrderDetails.OrderId, count(Suppliers.Country) as contador
      from OrderDetails, Products, Suppliers
      where OrderDetails.ProductID = Products.ProductID
            and Products.SupplierID=Suppliers.SupplierID
      group by OrderID) as oi,
      OrderDetails, Products, Suppliers
where contador=1
      and oi.OrderId=OrderDetails.OrderId
      and OrderDetails.ProductId=Products.ProductId
      and Products.SupplierId=Suppliers.SupplierId
order by Country, oi.OrderId
```

24 mayo 2021