Implementar la Arquitectura Base de un Proyecto Genérico de Software como Servicio (SaaS)

Procesos de Ingeniería del Software

Objetivo del sistema

El sistema solicitado por el cliente consiste en una aplicación de propósito general orientada a servicios y que incluya gestión de usuarios.

Podemos concretar la gestión de usuarios en la siguiente funcionalidad (en posteriores sprints la completaremos):

- Agregar usuario por nick (es una versión ligera del clásico registro e inicio de sesión de usuarios)
- Obtener los usuarios del sistema: obtiene una lista de usuarios
- Usuario activo: devuelve cierto si el usuario está activo
- Eliminar usuario por nick
- Otros: diferentes consultas sobre los usuarios

Este primer sprint se estructura en 7 bloques:

1. Puesta en marcha del proyecto
2. Implementar la capa lógica y las pruebas
3. Implementar el backend
4. Implementar la capa Rest
5. Implementar el componente de comunicación Rest
6. Implementar el componente de visualización
7. Desplegar el prototipo

Los siguientes apartados describen cada uno de esos bloques.

Bloque 1: Puesta en marcha del proyecto

1.1 Diseño de la arquitectura de la solución

En primer lugar creamos el diagrama de clases inicial (véase Figura 1) tomando como referencia el apartado anterior.

La idea que aplicamos en este modelo inicial es el principio “el diseño más simple que pueda funcionar”. Este modo de proceder nos permite crear una solución muy abierta ya que son muy pocas las decisiones que adoptamos al principio.

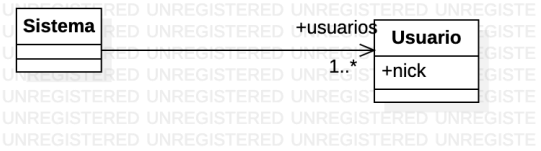


Figura 1. Diagrama de clases inicial

En cuanto a la arquitectura de la solución, tenemos una indicación clara del cliente, que indica que la solución debe ser “orientada a servicios”. Por ese motivo, diseñamos un diagrama de despliegue inicial (véase Figura 2) que nos ayude a tener una visión de conjunto de la solución.

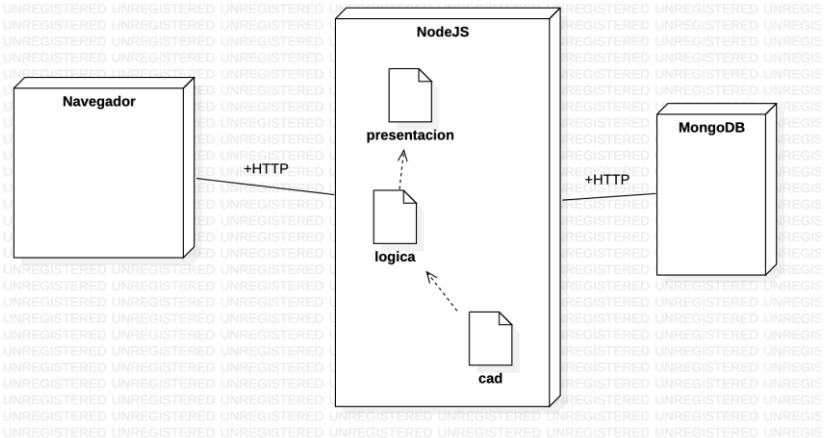


Figura 2. Diagrama de despliegue inicial

1.2 Crear el repositorio en GitHub

El siguiente paso es crear un repositorio público en GitHub. Este repositorio es público para poder realizar un seguimiento del trabajo de cada equipo y también porque la arquitectura base del proyecto es de propiedad colectiva.

Este repositorio estará vacío, lo único que contiene es el archivo Readme.

- Necesitamos credenciales de GitHub. Iniciamos sesión
- Pulsamos la opción Repositories (en las opciones de GitHub)
- Pulsamos el botón New
- Aparece las opciones de Create a new repository
    - Escribimos el nombre, por ejemplo, “arquitecturabase”
    - Escribimos el campo Description
    - Seleccionamos Public
    - Seleccionamos la opción Add a Readme file
    - También es conveniente elegir una licencia (MIT License, por ejemplo). Para más información repasar la asignatura API

1.3 Clonar el repositorio localmente

Una vez tenemos un repositorio vacío en GitHub, para poder trabajar con él tenemos que clonarlo localmente.

- Selecciona el repositorio “arquitecturabase” (o el nombre que le hayas puesto) en GitHub
- Despliega el desplegable que aparece al pulsar el botón verde “<>Code” y elige la primera opción que aparece (HTTPS) y copia la url que aparece (le llamamos ruta-copiada)
- Abre un terminal o cmd en la carpeta donde alojas los repositorios de GitHub (por ejemplo podría ser /Developer/git (MacOS o Linux) o c:\proyectos\git (Windows) o una ruta similar
- En el terminal escribes el comando Git: git clone ruta-copiada (se supone que has instalado las herramientas en línea de comando de Git)

A partir de este momento ya podemos trabajar con el repositorio utilizando nuestro editor favorito (Visual Studio Code, Sublime Text 3, etc).

1.4 Estructura de la solución en el sistema de archivos local

A continuación, vamos a crear la estructura de la solución en nuestro sistema de archivos local.

- Abrimos la carpeta del repositorio local (arquitecturabase) en nuestro editor
- Los siguientes elementos están todos al mismo nivel:
    - Creamos la carpeta servidor (representada en el diagrama de despliegue por el nodo servidor)
    - Creamos la carpeta cliente (representada en el diagrama de despliegue por el nodo cliente
    - Creamos el archivo index.js (punto de entrada del Backend)

La carpeta “servidor” contendrá todos los componentes y artefactos del Backend, a excepción del archivo index.js que estará ubicada en el raíz de la solución.

La carpeta “cliente” contendrá los componentes del Frontend de la solución. Este proyecto no tiene por qué ser parte de la solución general, es decir, no tiene por qué desplegarse junto al Backend ni estar en el mismo nodo servidor.

Bloque 2: Implementar la lógica y las pruebas

2.1. Implementación del modelo: agregar usuario

Un modo rápido de implementar y probar la lógica de la aplicación es hacerlo como si fuera a ser parte de los artefactos del Frontend (carpeta “cliente”). Sabemos que la lógica de la aplicación es un artefacto del Backend (carpeta “servidor”), pero resulta engorroso y complicado implementar y probar la lógica si la encapsulamos desde el principio en la capa lógica del Backend. Por ese motivo, vamos a implementar y probar la lógica con artefactos de cliente.

Este modo de proceder tiene la gran ventaja (y tiene algunos inconvenientes) de que podemos interactuar con los objetos de dominio desde la consola del navegador. Esto nos proporciona una gran flexibilidad para fijar el funcionamiento de la lógica.

Para implementar y probar la lógica en el cliente, seguimos estos pasos:

- Crea un archivo “index.html” dentro de la carpeta “cliente”. Incluye en el archivo el siguiente código (véase Codigo 1).

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title></title>
  <!--<script src="./modelo.js"></script>-->
</head>
<body>

</body>
</html>
```

Código 1. Contenido inicial del archivo index.html.

- Escribe un título para la página (etiqueta <title>)
- Almacena el archivo
- Comprueba el resultado en el navegador utilizando las opciones de tu editor. (en Visual Studio Code selecciona Ejecutar y luego Iniciar sin depuración. Otra opción también de VS Code es pulsar el botón derecho sobre el archivo index.html y elegir la opción “Open with Live Server”. En Sublime Text pulsa el botón derecho y elige “Open in Browser”)
- En el navegador aparece una página en blanco. Si inspeccionas (botón derecho, inspeccionar) el resultado de la estructura del HTML se puede ver en Elements. Otras pestañas con las que debemos familiarizarnos son estas: Console y Network

El siguiente paso es implementar el modelado inicial. Para ello nos fijamos en el diagrama de clases inicial. Conviene observar que vamos a utilizar Javascript sin clases, al estilo más sencillo. Javascript es un lenguaje interpretado y orientado a prototipos. Esto significa que cada objeto es único o prototípico y que la solución está orientada a funciones. De este modo, cuando escribamos una función estamos codificando un prototipo, un objeto único.

Esta decisión puede resultar controvertida, ya que muchos de los frameworks de cliente utilizan Typescript que es un Javascript con clases. Sin embargo, creo conveniente trabajar con un lenguaje sin tipos y dinámico porque obliga al desarrollador a tener un mayor control de la solución.

Vamos a mostrar la implementación en Javascript del diagrama de clases inicial y posteriormente lo explicaremos.

- Crea un nuevo archivo llamado “modelo.js“, dentro de la carpeta “cliente”
- Incluye el código que aparece en Codigo 2, en el archivo modelo.js

```
function Sistema(){
   this.usuarios={};
   this.agregarUsuario=function(nick){
      this.usuarios[nick]=new Usuario(nick);
   }
}

function Usuario(nick){
   this.nick=nick;
}
```

Código 2. Implementación del diagrama de clases inicial.

- La función Sistema en realidad es un objeto y se corresponde, en el diagrama de clases, a la clase Sistema. Es el punto de entrada de la capa lógica de nuestra solución. Utilizar esta clase tiene la ventaja de que nos permite acceder de forma sencilla al resto de objetos de la solución.
- El objeto Sistema “tiene-muchos” objetos de tipo Usuario, es decir, tiene una colección de objetos Usuario. Esta colección se puede implementar de muchas maneras, en nuestro caso hemos elegido una colección tipo array asociativo (esto se ve por la inicialización utilizada this.usuarios={};) de modo que los nick serán un campo único en nuestro sistema. Más adelante podemos hacer que el campo único sea el email en vez del nick.
  Un array asociativo es una colección formada por objetos de tipo clave-valor. También se puede considerar como Diccionario.
- El objeto Usuario por ahora no tiene más que un atributo denominado “nick”. Puedes agregar los atributos que consideres oportuno.

A continuación, vamos a interactuar con los objetos del modelo.

- En el archivo index.html hay una línea comentada (la línea 7). Hay que quitar el comentario.
- Ejecuta el archivo index.html
- En el navegador, abre las herramientas de desarrollo (botón derecho, inspeccionar), selecciona Console
- En la consola ejecuta (pulsar Enter):
    - sistema=new Sistema()
    - sistema.agregarUsuario("Pepe")
- Comprueba la colección "usuarios" de la instancia "sistema":
    - sistema.usuarios
- Agrega otros usuarios y comprueba el resultado
- Intenta agregar un usuario con Nick en uso
    - ¿Tenemos información en la consola del resultado?
    - Deberíamos anotar esta incidencia en el Backlog como tarea

2.2. Implementación del modelo (2): obtener todos los usuarios

Para obtener la lista de usuarios, tenemos que modificar el modelo. Incluye el nuevo método (véase Código 3) en el objeto Sistema.

```
this.obtenerUsuarios=function(){
    return this.usuarios;
}
```

Código 3. Método para obtener la colección de usuarios.

Comprueba la nueva funcionalidad:

- Comprueba que, al crear el objeto Sistema, al invocar el método obtenerUsuarios(), devuelve una colección vacía
- Agrega usuarios y comprueba el resultado invocando el nuevo método

2.3. Implementación del modelo (3): usuario activo

La nueva funcionalidad a implementar, es el método "usuarioActivo(nick)" del objeto Sistema. Este método debería devolver true si el usuario existe en el sistema y false si no existe ningún usuario con ese Nick.

Implementa el método de modo que compruebe si existe un usuario determinado (le pasamos el nick).

Comprueba el funcionamiento en la consola del navegador.

2.4. Implementación del modelo (4): eliminar usuario

La siguiente funcionalidad consiste en eliminar el usuario cuyo nick se pasa como argumento. El método sería eliminarUsuario(nick).

Implementa el método utilizando el operador delete.

Comprueba el funcionamiento en la consola del navegador.

2.5. Proyecto de pruebas del modelo

Vamos a preparar el proyecto de pruebas de la funcionalidad desarrollada hasta el momento.

El proyecto de pruebas se puede implementar utilizando diferentes frameworks. En el proyecto de clase vamos a utilizar Jasmine (https://jasmine.github.io/).

Cuando tengamos el proyecto de pruebas bien definido, lo integraremos en el Backend de nuestra solución. Mientras tanto, es preferible tenerlo en una carpeta diferente a la de la solución.

- Se sugiere utilizar una localización diferente a la solución, por ejemplo: /developer/Jasmine (versión Unix), o c:\developer\jasmine (versión Windows).
- Utiliza el enlace del Campus Virtual (Jasmine standalone) para descargar la última versión (5.1.1). Hay que descargar el archivo Jasmine-standalone-5.1.1.zip y lo ubicas en la carpeta elegida en el paso anterior.
- Renombra la carpeta para que se llame pruebas-arquitecturabase
- Abre la carpeta del proyecto de pruebas en el editor
- En la carpeta src:
    - Elimina los archivos Player.js y Song.js
    - Copia el archivo modelo.js del proyecto arquitecturabase y lo pegas en esta carpeta
- Edita en VS Code el archivo SpecRunner.html
    - En la línea 17 cambia Player.js por modelo.js
    - Elimina la línea en la que aparece “src/Song.js”
    - Elimina la línea en la que aparece “spec/SpecHelper.js”
    - Cambia “spec/PlayerSpec.js” por “spec/modeloSpec.js”
- En la carpeta spec:
    - Elimina el archivo SpecHelper.js
    - Renombra PlayerSpec.js por modeloSpec.js
- Cambia el contenido del archivo modeloSpec.js por el código mostrado en Código 4.

```
describe('El sistema', function() {
  let sistema;

  beforeEach(function() {
    sistema=new Sistema()
  });
```

```
  it('inicialmente no hay usuarios', function() {
    expect(sistema.numeroUsuarios()).toEqual(0);
  });
})
```

Código 4. Contenido inicial del archivo modeloSpec.js

- El bloque beforeEach() se ejecuta antes de cada bloque it()
- Los bloques it() contienen cláusulas de tipo expect para comprobar propiedades del código.
- Como se puede ver en el código, tenemos un nuevo método que hay que implementar. El nuevo método es “numeroUsuarios()” y nos devuelve el número de usuarios que hay en la colección. Como utilizamos un array asociativo o diccionario, para contar los elementos de la colección tenemos que contar las claves (un array asociativo se compone de objetos clave-valor). Para ello utilizaremos este código:

  Object.keys(sistema.usuarios).length

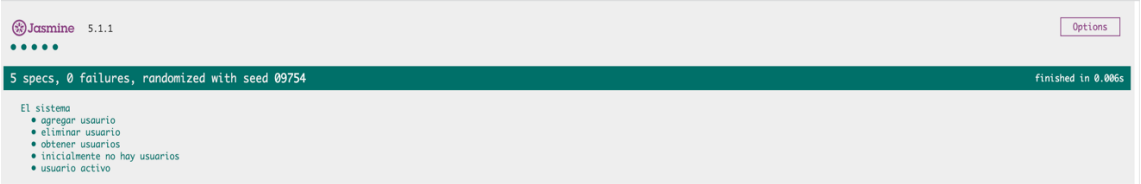
- Crea el nuevo método y comprueba su funcionamiento en la consola del navegador
- Ejecuta las pruebas (selecciona el archivo SpecRunner.html y lo ejecutas siguiendo el método que más te guste)

El resultado debería ser algo como esto:

Ejercicios:

Escribe un bloque it() por cada método (agregarUsuario, obtenerUsuarios, usuarioActivo y eliminarUsuario).

El resultado esperado debería ser algo como esto:



Conviene realizar ciertas tareas al finalizar cada sesión:

- Asegúrate de que el código de modelo.js de la solución (en la carpeta arquitecturabase) tiene los cambios introducidos en el proyecto de pruebas

- Consolida los cambios y actualiza el repositorio de GitHub. Esto se hacía con los siguientes comandos:

  git add .
  git commit -m”mensaje corto que representa lo codificado”
  git push

Bloque 3. Implementar el Backend

En esta segunda parte vamos a preparar los artefactos que forman parte del backend. En primer lugar, veremos las tecnologías que necesitamos, después mostraremos un ejemplo sencillo de aplicación de servidor y finalmente integraremos nuestro modelo y las pruebas en el servidor.

El backend se puede implementar de muchas maneras. En nuestro caso vamos a seguir el esquema de una arquitectura en 3 capas:

- Capa de presentación: API Rest (archivo index.js)
- Capa lógica: archivo modelo.js
- Capa de acceso a datos: archivo cad.js

La capa de presentación se implementa como un API Rest. Para implementar el API Rest tenemos diferentes opciones y diferentes tecnologías. En nuestro caso vamos a utilizar NodeJS y un estilo lo más sencillo posible (es decir, sin rutas ni controladores) para permitir al alumno un acceso gradual a estas tecnologías.

3.1. Prerrequisitos

- Descargar e instalar NodeJS (https://nodejs.org/en/download)
- Comprobar instalación:
    - Iniciar un terminal o cmd
    - Ejecutar npm -v
    - Ejecutar node –versión

Npm (Node Package Manager) es una herramienta para gestionar (localizar, instalar, actualizar) los módulos que necesita una aplicación NodeJS para su ejecución.

3.2. Crear y configurar el archivo package.json

El archivo package.json es un archivo que utilizan las aplicaciones NodeJS para almacenar información sobre la aplicación (descripción, dependencias etc).

- Inicia un terminal o cmd
- Sitúate en el raíz de la solución (ejemplo: c:\developer\git\arquitecturabase)
- Ejecuta npm init
    - Puedes completar los campos o pulsar Enter para dejar la información por defecto

3.3. Crear el archivo .gitignore

El archivo .gitignore se utiliza para aligerar la sincronización con los repositorios remotos ya que evita tener que incluir en el sistema de gestión de versiones aquellos archivos o carpetas que se no es necesario versionar.

- Abre la solución con el editor de referencia
- Crea un nuevo archivo y le pones de nombre .gitignore
- En el contenido, podemos incluir:
    - node_modules
    - otros: se puede incluir la carpeta .vscode

3.4. Crear el servidor "Hola Mundo" con NodeJS y Express

Abre el archivo index.js de la solución e incluye el código que se muestra en Código 5.

```
const fs=require("fs");
const express = require('express');
const app = express();
//const modelo = require("./servidor/modelo.js");

const PORT = process.env.PORT || 3000;

app.use(express.static(__dirname + "/"));

app.get("/", function(request,response){
    response.statusCode = 200;
    response.setHeader('Content-Type', 'text/plain');
    response.end('Hola Mundo!');
});

app.listen(PORT, () => {
  console.log(`App está escuchando en el puerto ${PORT}`);
  console.log('Ctrl+C para salir');
});
```

Código 5. Ejemplo de aplicación Express

En las primeras líneas aparece la función "require". Esta función sirve para declarar dependencias (librerías) que se utilizan en el código. Normalmente, las dependencias se deben instalar en línea de comandos utilizando npm (npm install express). Cuando se instala una dependencia, se almacena una referencia en el archivo package.json.

La constante PORT contiene el puerto que utilizará nuestro servidor para escuchar. Nuestro código elige la variable de entorno PORT en caso de que esté definida, o el puerto 3000 en caso de que la variable de entorno PORT no está definida.

Las sentencias de tipo app.use sirven para declarar directivas o cargar algún middleware que necesitemos. En nuestro caso indicamos como ruta raíz de nuestro servidor, la ruta local del sistema de archivos donde tenemos instalada la aplicación.

A partir de esta línea nos encontraremos las diferentes rutas de nuestro API Rest, con estas posibles funciones:

app.get
app.post
app.put
app.delete

Cada una de estas cuatro funciones se corresponden con cada uno de los métodos del protocolo HTTP (GET, POST, PUT, DELETE).

Abre un terminal y ejecuta: node index.js

- En el terminal debe aparece el siguiente mensaje:
  App está escuchando en el puerto 3000
  Ctrl+C para salir
  - Si aparece algún error puede deberse a que no has instalado el módulo de Express (npm install express) o algún error de sintaxis.
- Abre un navegador y en la barra de navegación escribes:
  http://localhost:3000
- Inspecciona la página. En las DevTools, selecciona la pestaña Network y refresca la página

3.5. Integrar el modelo en nuestro servidor

- Copia el archivo modelo.js (que está en la carpeta “cliente”) y lo pegas en la carpeta “servidor”
- En el archivo modelo.js (el de la carpeta servidor) debemos exportar el nombre que usaremos en la capa Rest (que está en el archivo index.js), para ello hay que incluir al final esta expresión:
  module.exports.Sistema=Sistema
- Descomenta la línea 4 del archivo index.js
- Incluye la siguiente expresión después de la línea 8:
  let sistema = new modelo.Sistema();
- Comprueba que funciona correctamente, que no salta ningún error

3.6. Conectar cliente y el servidor

A continuación, vamos a cambiar la ruta determinada (el raíz de nuestro servidor) para que, en vez de devolver la cadena “Hola Mundo”, devuelva el index.html de la carpeta “cliente”.

Para conseguirlo, cambia el app.get del raíz por el contenido mostrado en Código 6.

```
var contenido=fs.readFileSync(__dirname+"/cliente/index.html");
  response.setHeader("Content-type","text/html");
  response.send(contenido);
```

Código 6. Contenido del app.get("/"…{}).

Como se puede ver, ahora utilizamos el sistema de archivos para localizar el archivo index.html y enviarlo al navegador que realiza la petición.

Elimina la referencia a modelo.js del archivo index.html. Esto lo hacemos porque el modelo ahora forma parte de la capa lógica del backend.

3.7. Integrar las pruebas en el servidor

Ya que hemos integrado el modelo en el servidor, necesitamos que el proyecto de pruebas también esté en el servidor.

Utilizando el Explorador de Windows, copia el archivo modeloSpec.js (está ubicado en nuestro proyecto de pruebas) en la carpeta "servidor" de nuestra solución.

Incluye una línea al principio de modeloSpec.js que tenga una referencia al archivo modelo.js (similar a la que tenemos en index.js). También hay que crear de forma diferente la instancia de Sistema (véase Código 7).

```
const modelo = require("./modelo.js");

describe('El sistema...', function() {
   let sistema;

   beforeEach(function() {
     sistema=new modelo.Sistema();
   });
```

Código 7. Contenido inicial del archivo "modeloSpec.js".

Observa que al crear la instancia de Sistema(), ahora tenemos que usar la variable con la que importamos la capa lógica (modelo.js).

Para ejecutar las pruebas, hay que modificar el package.json (véase Código 8).

```
 "scripts": {
    "testW": "node_modules\\.bin\\jasmine-node servidor",
    "test": "./node_modules/.bin/jasmine-node servidor"
  },
```

Código 8. Posible contenido de la sección scripts del package.json

El contenido de la sección scripts tiene dos alternativas. Lo ideal es ejecutar los tests con el comando npm test (en un terminal). En ocasiones, según el sistema operativo y otros factores difíciles de concretar, los terminales de Windows (cmd, PowerShell, bash) no reconocen la ruta estilo Unix, con lo cual en esos casos hay que ejecutar el comando npm run testW.

Bloque 4. Exponer la lógica en la capa Rest

En esta tercera parte vamos a seleccionar y exponer (publicar en nuestro API Rest) la funcionalidad de la capa lógica que hemos implementado en “modelo.js”.

Antes de nada, vamos a cambiar el modo de iniciar la aplicación para que se asemeje al que se utiliza en el despliegue.

El objetivo es lanzar la aplicación (el backend) con el comando “npm start”. Para ello tenemos que modificar la sección de scripts del package.json. En el Código 9 se muestra la nueva línea en esa sección.

```
"scripts": {
    "start": "node index.js",
    "testW": "node_modules\\.bin\\jasmine-node servidor",
    "test": "./node_modules/.bin/jasmine-node servidor"
  },
```

Código 9. Nuevo modo de iniciar la aplicación (npm start).

4.1. Exponer el método “agregarUsuario”

El modo de exponer la funcionalidad del modelo consiste en incluir una nueva ruta en la que aceptamos peticiones del cliente. Necesitaremos un bloque de código que gestione las peticiones que lleguen a través de esa nueva ruta.

El primer método que vamos a exponer es agregarUsuario(nick). Este método tiene un parámetro de tipo String, con lo cual tenemos que permitir que los clientes puedan pasar el Nick en la propia petición. Esta problemática es muy común en el desarrollo de aplicaciones y servicios en Internet. Hay dos métodos principales para que los clientes pasen información al servidor: Query String y Request Body.

Para este método vamos a utilizar QS. Revisa el Código 10 e intégralo en tu solución.

Una decisión inseparable de lo anterior es el tipo de método HTTP que vamos a utilizar para implementar la petición. Si nos decantamos por QS, entonces necesariamente utilizaremos una petición GET, que en nuestro código se refleja porque usamos un bloque de código app.get(ruta,function(req,res){}).

Para el resto de peticiones en las que necesitemos pasar información en el Body de la petición, utilizaremos los otros métodos: POST, PUT, etc.

```
app.get("/agregarUsuario/:nick",function(request,response){
  let nick=request.params.nick;
  let res=sistema.agregarUsuario(nick);
  response.send(res);
});
```

Código 10. Bloque de código app.get que se encarga de las peticiones que llegan a la ruta "/agregarUsuario/nick".

Como se puede ver en Código 10, hay un formato específico para escribir la ruta e indicar los parámetros. Observa el modo en que tenemos que escribir el parámetro "Nick" en la url.

Ese parámetro lo recuperamos con la expresión "request.params.nick", que genéricamente sería "request.params.parametro" para una ruta del tipo "/peticion/:parametro".

Obsérvese que la petición en el navegador no utiliza ":". En el navegador escribiríamos "http://localhost:3000/agregarUsuario/pepe".

También se puede observar que el método "agregarUsuario(nick)" ahora devuelve un valor. Esto es así porque necesitamos recibir información que nos indique si la operación se ha podido realizar o no.

Esto implica modificar el código de la lógica. Observa e integra el nuevo método "agregarUsuario" de Sistema que se muestra en Código 11.

```
this.agregarUsuario=function(nick){
        let res={"nick":-1};
        if (!this.usuarios[nick]){
            this.usuarios[nick]=new Usuario(nick);
            res.nick=nick;
        }
        else{
            console.log("el nick "+nick+" está en uso");
        }
        return res;
    }
```

Código 11. Nueva versión del método agregarUsuario.

El método también controla que el "Nick" que se pasa como parámetro no está en uso.

Lanza el servidor y comprueba el resultado en el navegador ejecutando estas dos peticiones:

- http://localhost:3000/agregarUsuario/pepe
- http://localhost:3000/agregarUsuario/pepe

Comprueba el valor devuelto. Comprueba las peticiones en la pestaña Network de las DevTools.

4.2. Exponer el método “obtenerUsuarios”

Teniendo en cuenta la experiencia adquirida en el apartado anterior, implementa esta petición en la capa Rest (index.js).

Comprobar el resultado:

- Agrega varios usuarios utilizando el navegador
- Realiza la petición para la lista de usuarios: http://localhost:3000/obtenerUsuarios

4.3. Exponer el método “usuarioActivo”

Ahora toca exponer el método “usuarioActivo(nick)”. Respecto al valor que devuelve el método, se puede mantener el estilo de devolver siempre un objeto JSON.

Realiza los cambios necesarios en el modelo.

Comprueba el funcionamiento en el navegador. El resultado esperado se muestra en la Figura

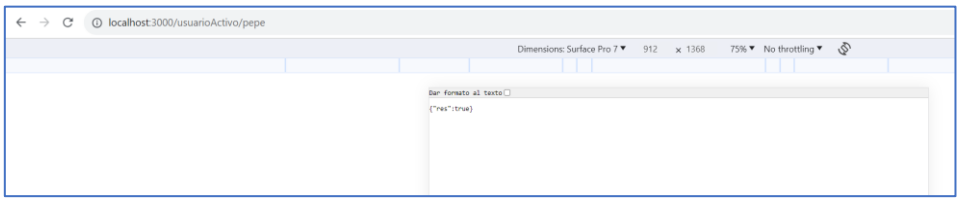


Figura 3. Resultado esperado de la petición /usuarioActivo/nick

4.4. Exponer el método “numeroUsuarios”

El siguiente método a exponer es “numeroUsuarios()”.

Realiza los cambios necesarios en el modelo para que la petición Rest devuelva un objeto JSON con esta estructura:

{“num”:numero_usuarios}

Comprueba el resultado.

4.5. Exponer el método “eliminarUsuario”

El siguiente método a publicar es “eliminarUsuario(nick)”.

4.6. Revisión de las pruebas

Debido al refactoring (hemos incluido objetos JSON de respuesta a las peticiones) realizado en los métodos del modelo, las pruebas no funcionan correctamente.

Comprueba los errores que aparecen ejecutando alguno de estos dos comandos (en un terminal):

```
npm test
npm run testW
```

Revisa las pruebas (modeloSpec.js) de cada método para contemplar los cambios que hemos realizado.

Bloque 5. Implementar el cliente de comunicación Rest

A partir de este momento dejamos el servidor y nos centramos en el cliente. En concreto, vamos a implementar el componente de comunicación con la capa Rest.

5.1 Implementar la petición “agregarUsuario(nick)”

En primer lugar, creamos el archivo “clienteRest.js” y lo situamos en la carpeta “cliente” de nuestra solución. En el archivo recién creado debes escribir el código mostrado en Código 12.

```
function ClienteRest(){
    this.agregarUsuario=function(nick){
        var cli=this;
        $.getJSON("/agregarUsuario/"+nick,function(data){
            if (data.nick!=-1){
                console.log("Usuario "+nick+" ha sido registrado")
            }
            else{
                console.log("El nick ya está ocupado");
            }
        })
    }
}
```

Código 12. Objeto ClienteRest y el primer método “agregarUsuario(nick)”

Como se puede ver, ahora tendremos un componente de comunicación que llamamos “ClienteRest”. Ese componente por ahora tiene un solo método, se trata del método que invoca al primero de los métodos expuestos en la capa Rest.

La petición al servidor la realizamos con AJAX mediante la librería Jquery.

El valor devuelto por el servidor lo identificamos con la variable “data”. Revisa el valor que devuelve el método “agregarUsuario” del servidor. Para ello tienes que localizar el app.get que se corresponde con esa petición y ahí descubrirás cómo la capa Rest delega en el modelo. En el modelo es donde realmente se realiza la respuesta.

Para poder utilizar el nuevo componente, tenemos que referenciarlo en el index.html.

```
<script src="./cliente/clienteRest.js"></script>
```

Y para que funcionen las llamadas de Jquery, tenemos que cargar la librería. Para ello, utiliza Google para localizar una url que nos permita cargar esa librería. En el navegador puedes escribir:

cdnjs jquery

Uno de los primeros resultados será la web cdnjs con la referencia de la url con el código ya preparado (aparece estos símbolos </>) para incluir en el index.html.

Una vez incluidas esas referencias, ejecuta la aplicación (el backend), abre el navegador en http://localhost:3000 y después abre las DevTools (botón derecho, inspeccionar) y ejecuta estas expresiones en la consola del navegador:

rest=new ClienteRest()
rest.agregarUsuario("Pepe")
rest.agregarUsuario("Luis")
rest.agregarUsuario("Pepe")

Comprueba los resultados obtenidos tras cada llamada.

También puedes inspeccionar la pestaña Network para comprobar con más detalle la comunicación cliente-servidor.

Las llamadas AJAX se pueden realizar de otra manera. Examina el Código 13 y crea una nueva función ("agregarUsuario2") en ClienteRest que utilice el nuevo método.

```
    $.ajax({
    type:'GET',
    url:'/agregarUsuario/'+nick,
    success:function(data){
     if (data.nick!=-1){
       console.log("Usuario "+nick+" ha sido registrado")
     }
     else{
       console.log("El nick ya está ocupado");
     }
    },
    error:function(xhr, textStatus, errorThrown){
       console.log("Status: " + textStatus);
       console.log("Error: " + errorThrown);
    },
    contentType:'application/json'
});
```

Código 13. Método alternativo para realizar peticiones Rest.

Comprueba el resultado en la consola del navegador tras ejecutar estas expresiones:

rest=new ClienteRest()

rest.agregarUsuario(“Pepe”)
rest.agregarUsuario2(“Luis”)
rest.agregarUsuario(“Pepe”)
rest.agregarUsuario2(“Luis”)

Comprueba lo que ocurre si invocamos alguno de los métodos de ClienteRest sin ningún parámetro. Por ejemplo, rest.agregarUsuario() o también rest.agregarUsuario2().

5.2 Implementar el resto de peticiones

Ejercicio: Implementa el resto de peticiones en ClienteRest. Para cada caso, muestra el objeto JSON que se obtiene como respuesta del servidor en la consola del navegador:

- obtenerUsuarios()
- numeroUsuarios()
- usuarioActivo(nick)
- eliminarUsuario(nick)

Comprueba el resultado utilizando la consola del navegador. Es conveniente observar que para probar los componentes de cliente, en principio no es necesario reiniciar el servidor.

Bloque 6. Implementar el componente de visualización

El siguiente paso es implementar el componente de visualización, el encargado de mostrar los formularios que necesitemos en nuestra aplicación.

6.1. Introducción

Para implementar el componente de visualización vamos a seguir el modelo **Single Page Application**, un tipo de aplicación web que utiliza una sola página HTML para mostrar todo el contenido.

Veamos ventajas y desventajas de utilizar SPAs (tomado de www.codeacademy.com).

Las SPAs tienen la ventaja de proporcionar una experiencia interactiva más ágil que las aplicaciones basadas en múltiples páginas, ya que hacen peticiones de cantidades de datos más pequeñas. También facilitan la reutilización de código entre diferentes proyectos debido a que es posible construir componentes de uso general. Finalmente, facilitan migrar el código de web a móvil.

Entre las desventajas de las SPAs se pueden enumerar las siguientes: la primera carga puede resultar algo más costosa; también tiene desventajas respecto al posicionamiento (Search Engine Optimization, SEO), ya que los motores de búsqueda indexan las páginas según el contenido y una SPA tiene mucho contenido en "segundo plano"; la última desventaja tiene que ver con el modo de operar en el navegador, el botón de volver atrás puede no funcionar correctamente.

Puestos a implementar SPAs, tenemos diferentes alternativas:

- Utilizar JavaScript plano (se suele denominar vanilla JavaScript)
- React: una librería para construir componentes que se renderizan
- Vue.js: es un framework que utiliza plantillas dentro de un HTML
- Otros: AngularJS, MeteorJS, EmberJS etc

En nuestra solución vamos a utilizar JavaScript plano. Esta opción tiene la ventaja de que tenemos el control de los componentes pero tiene el inconveniente de que resulta complejo de manejar cuando el proyecto crece y tenemos muchos componentes de visualización.

También utilizaremos Bootstrap, un framework responsive que utiliza HTML, CSS y JavaScript. Un framework responsive facilita la creación de aplicaciones web cuyo aspecto se ajusta automáticamente para mostrarse correctamente en todos los

dispositivos, desde pequeños teléfonos a grandes pantallas (véase www.w3schools.com/bootstrap4).

Bootstrap se ofrece en tres versiones. En esta solución vamos a utilizar la versión 4. No hay un criterio especial para esta decisión, la 3 puede resultar algo antigua y la 5 demasiado moderna.

Las aplicaciones que utilizan Bootstrap (o un framework similar) siguen el principio "primero la solución móvil" (Mobile-first approach), que significa que el diseño de la aplicación prioriza la versión móvil y luego la adapta a grandes pantallas.

6.2. Implementar el componente visual "mostrarAgregarUsuario"

El componente visual web que permite al usuario entrar en el sistema (agregarUsuario), en su versión más sencilla, necesita sólo dos elementos: un elemento HTML que le permita escribir el Nick (un input text) y un botón que sirva para solicitar el ingreso.

En primer lugar, debemos crear un nuevo archivo en la carpeta "cliente" que se llame "controlWeb.js".

Con el nuevo archivo abierto, escribimos la estructura de nuestro componente ControlWeb siguiendo el estilo que hemos utilizado hasta el momento:

```
function ControlWeb(){
}
```

Antes de renderizar nuestros componentes visuales, debemos cargar las dependencias necesarias de Bootstrap. Estas dependencias se pueden cargar directamente de la web (CDN, content delivery network) o localmente. Las dependencias necesarias son:

- Las CSS de Bootstrap compiladas y comprimidas (minified)
- Jquery (que ya la tenemos en nuestra solución)
- PopperJS: utilizado para mostrar tooltips
- El JavaScript de Bootstrap

En la web de w3schools, en la sección Bootstrap versión 4, en el apartado BS4 Get Started puedes encontrar esas referencias (la de Jquery ya la tenemos en el index.html).

Incluye ese código en el index.html

Vamos a crear nuestro primer método de ControlWeb que tendrá la misión de mostrar un input de tipo texto para que el usuario escriba el nick y un botón. La idea es que cuando el usuario pulse el botón, realizaremos la llamada al método "agregarUsuario(nick)" del componente ClienteRest.

El HTML necesario lo tomamos de la sección BS3 Inputs de w3schools (el primer ejemplo). Observa el código mostrado en Código 14.

```
<div class="form-group">
      <label for="usr">Name:</label>
      <input type="text" class="form-control" id="usr">
</div>
```

Código 14. HTML tomado de w3schools para mostrar un input de tipo texto.

En ese código HTML tenemos que incluir un botón. El Código 15 muestra el código del botón.

```
<button type="submit" class="btn btn-primary">Submit</button>
```

Código 15. HTML para mostrar un botón.

El botón lo debemos incluir después del input text y antes de cerrar el div de "form-group".

En las SPAs el modo de funcionar consiste en insertar y quitar componentes visuales de forma dinámica. Para insertar esos componentes, debemos "inyectarlos" en alguna zona de nuestro HTML. Un modo de inyectar el HTML es construir una cadena e insertarla en una etiqueta previamente identificada.

La manera de construir una cadena es sencilla:

```
let cadena='<div class="form-group">';
cadena = cadena + '<label for="usr">Name:</label>';
…
```

Observa que para construir la cadena utilizamos comillas simples ya que el código que hemos copiado de w3schools utiliza comillas dobles para las propiedades de las etiquetas HTML.

De esa manera, ya podemos construir nuestro primer método mostrarAgregarUsuario en el objeto ControlWeb.

Si utilizas copiar y pegar desde este documento al editor, debes asegurarte de que el editor interpreta correctamente las comillas simples. Con las comillas dobles no suele haber problemas.

Faltaría mostrar el nuevo formulario en nuestra página HTML. Para ello debemos incluir la siguiente expresión:

```
$("#au").append(cadena);
```

Esa función “append” inserta la cadena (el formulario para agregar usuario) en un punto concreto de nuestra página. Ese punto concreto es una etiqueta HTML que tiene una propiedad id que es “au”. Debemos incluir ese elemento en el index.html (dentro del body):

El siguiente paso consiste en controlar el evento click en el botón. Para conseguirlo lo adecuado es incluir una propiedad en el botón que nos permita identificarlo. Una manera de identificar el botón es ponerle un id que resulte representativo de su función, por ejemplo btnAU (botón AgregarUsuario).

Incluye esa propiedad en el botón. Quedaría algo así

<button id=”btnAU” type…>

A continuación, tenemos que controlar el evento de hacer clic en ese botón. Para eso nos resulta útil utilizar Jquery, que nos permite localizar el botón y asociarle una función que se ejecuta cuando el usuario haga click en ese botón.

La función comienza de este modo:

$(“#btnAU”).on(“click”,function(){ … }

El carácter $ representa a la función de Jquery. La cadena “#btnAU” tiene dos partes, en segundo lugar muestra el identificador del botón (btnAU) y primero aparece el carácter # que sirve para indicar a la función de Jquery que localice el elemento HTML cuyo id es btnAU. A ese elemento, si existe, le asocia la función que maneja el evento “click”.

Dentro de la función de manejo del evento, debemos hacer dos cosas:

- Recoger el valor del input text (el nick)
- Invocar al método del cliente rest para que solicite agregar a ese usuario

El código correspondiente sería el siguiente:

1. let nick=$(“#nick”).val();
2. rest.agregarUsuario(nick)

Para poder recoger el valor del input text, necesitamos identificar ese elemento HTML de la misma manera que hemos hecho con el botón. Para que funcione el código anterior debes cambiar el id del input text para que sea “nick” en vez de “usr”.

Además de lo anterior, necesitamos acceder de forma global a los objetos de cliente (tanto ClienteRest como ControlWeb). Para ello, crearemos dos instancias de esos objetos en el index.html.

Incluye el siguiente código justo antes de </body> en el index.html

<script>

```
        rest=new ClienteRest();
        cw=new ControlWeb();
</script>
```

También se debe incluir una referencia al nuevo artefacto controlWeb.js en el index.html (de la misma manera que hacemos con clienteRest.js).

Comprueba el resultado de esta manera:

- Lanza el servidor
- En la consola del navegador ejecuta esta expresión:
  cw.mostrarAgregarUsuario()

  El resultado esperado es que se muestra el formulario en la página.

Agrega un usuario y comprueba el resultado.

Los formularios hay que mostrarlos y quitarlos. Hemos visto cómo mostrarlos, vamos a ver cómo quitarlos.

Para poder borrar un formulario tenemos que identificarlo, esto es, debemos agregarle un atributo id en la etiqueta que englobe a todo el formulario. En nuestro caso tenemos la etiqueta `<div class="form-group">` a la que añadimos el atributo id=”mAU” (que serían las siglas de mostrarAgregarUsuario).

Lo siguiente que hacemos es agregar la expresión para que elimine el formulario, y lo hacemos dentro de la función on(“click”…):

```
$(“#mAU”).remove();
```

Comprueba el resultado recargando la página.

Observa que ahora desaparece el formulario.

Ejercicio: Modifica la solución para que el formulario aparezca la primera vez que el usuario navega a nuestro sitio.

6.3. Proporcionar estructura a la página de inicio

En el apartado anterior hemos visto como inyectar un formulario de forma dinámica en el index.html.

En este apartado vamos a proporcionar algo de estructura a nuestra página de inicio.

Para poder proporcionar estructura es imprescindible tener nociones del funcionamiento de Bootstrap, que es el framework que hemos seleccionado para visualizar nuestro contenido. Es conveniente repasar la documentación sobre este framework responsive con el objetivo de ser capaz de hacer pequeñas adaptaciones y modificaciones. Se recomienda realizar el tutorial de Bootstrap de w3schools.

Tenemos diferentes opciones para organizar nuestra página de inicio. En este tutorial voy a escoger una de las mostradas en el apartado BS3 Navbar de w3schools, en concreto, el ejemplo que aparece con el título Brand / Logo (véase Código 16).

```
<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
  <!-- Brand/logo -->
  <a class="navbar-brand" href="#">Logo</a>

  <!-- Links -->
  <ul class="navbar-nav">
    <li class="nav-item">
      <a class="nav-link" href="#">Link 1</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Link 2</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Link 3</a>
    </li>
  </ul>
</nav>

<div class="container-fluid">
  <h3>Brand / Logo</h3>
  <p>The .navbar-brand class is used to highlight the brand/logo/project name of your page.</p>
</div>
```

Código 16. Posible estructura del HTML de la página de inicio.

Inserta el contenido del Código 16 en el <body> de nuestra página de inicio y comprueba el resultado.

Incluye el dentro del contenedor (class=”container-fluid”) y comprueba el resultado.

La nueva página de inicio de nuestra aplicación debería tener un aspecto similar al mostrado en la Figura

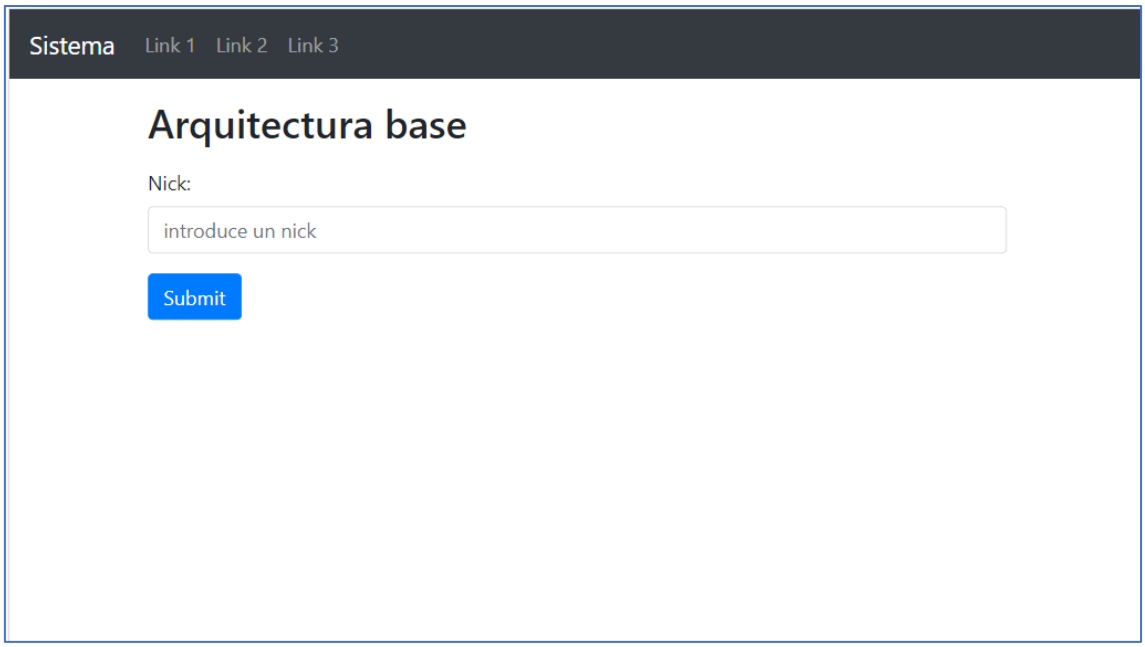


Figura 4. Resultado final del formulario “agregarUsuario” junto con el grupo NavBar.

Como ejercicios optativos finales quedaría elegir un componente de visualización para el resto de funciones:

- obtenerUsuarios
- numeroUsuarios
- usuarioActivo(nick)
- eliminarUsuario(nick)

Bloque 7. Despliegue de la aplicación en Cloud Run

El último paso del Sprint 1 es desplegar el prototipo en Google Cloud Run. Para conseguirlo vamos a seguir las pautas que proporciona el propio proveedor (https://cloud.google.com/run/docs/quickstarts/build-and-deploy/deploy-nodejs-service).

7.1 Requisitos previos

Para poder desplegar en Cloud Run hay que tener una cuenta de Google y tener habilitados los créditos de educación de GCP (en el campus hay un mensaje que explica cómo obtener créditos de uso de GCP).

- Accede a la consola de GCP (https://console.cloud.google.com) con tu cuenta del dominio de la UCLM.
- En la parte superior, junto al logo Google Cloud, aparece el proyecto seleccionado. Puedes crear un nuevo proyecto, accediendo al panel de proyectos desde ese mismo control.
- En el menú de la izquierda (representado por tres líneas horizontales), selecciona la opción Facturación. Aparece la información de facturación. Hay un panel en la zona de la derecha, abajo, que se llama Créditos. Ahí deben aparecerte los créditos educativos de uso de GCP.
- También debes tener instalado Google Cloud CLI (para comprobarlo, abre un terminal y escribes gcloud -v
- La primera vez debes ejecutar gcloud init (en un terminal o cmd)

7.2 Desplegar en Cloud Run en línea de comandos

- Abre un terminal o cmd
- Sitúate en la carpeta raíz de tu solución (por ejemplo /developer/git/arquitecturabase)
- Ejecuta el comando: gcloud run deploy
    - Ubicación del código fuente: puedes pulsar Enter
    - Nombre del servicio: puedes pulsar Enter
    - La primera vez solicita habilitar API de Artifact Registry: pulsamos y
    - Seleccionar región: escribimos 15 (europe-west1)
    - Permitir peticiones anónimas: pulsamos y
    - En este momento comienza la implementación (véase Figura 5). Este proceso tarda unos minutos (es más largo la primera vez)
- Al terminar el proceso se genera una url con la que podemos probar la aplicación.

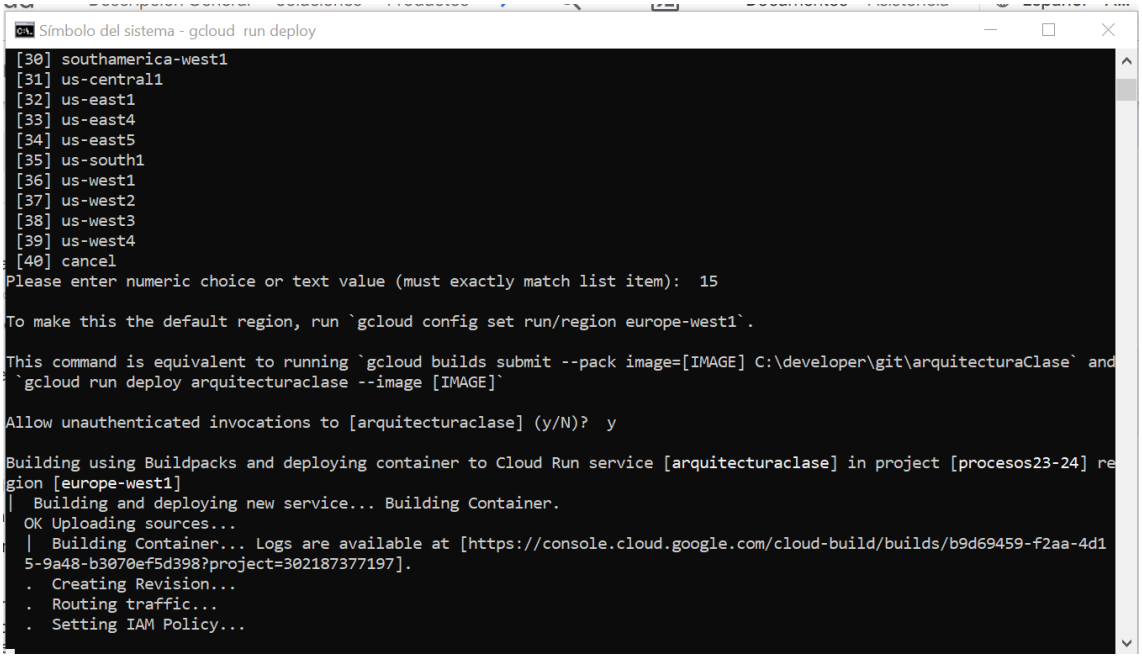


Figura 5. Proceso de implementación de la aplicación como servicio Cloud Run.