Spring.io, Websockets, ReactJs-18, P5.js, AWS: buenas prácticas de Diseño

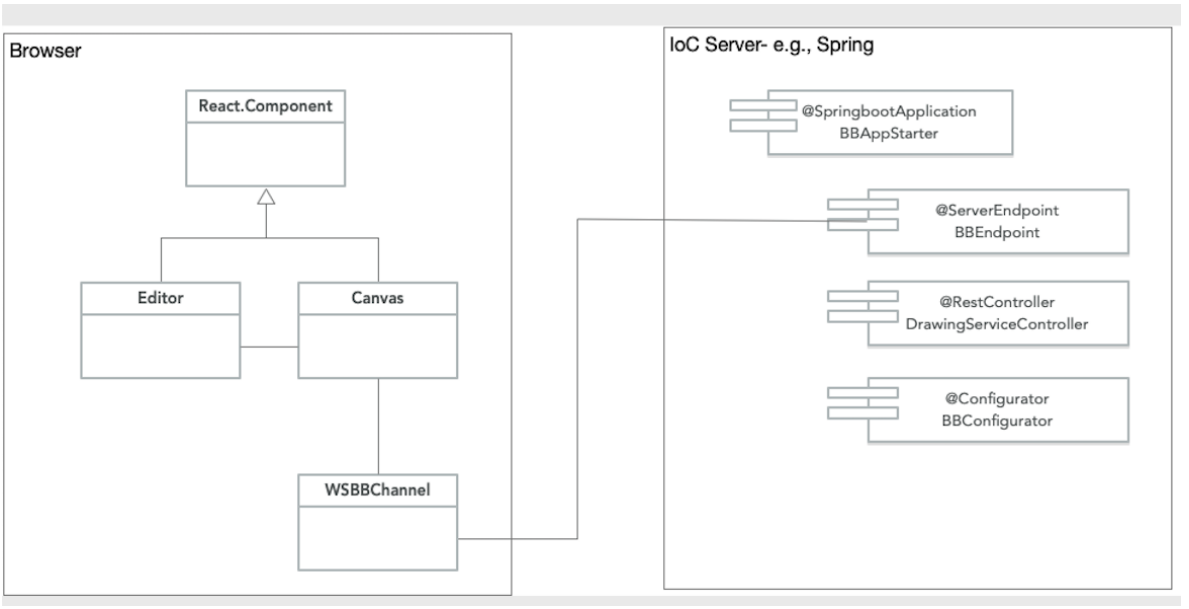
Vamos ahora a construir una aplicación interactiva en tiempo real usando una buena estrategia de diseño. Para esto vamos a construir una aplicación que permite dibujar de manera colaborativa en tiempo real.

La aplicación soporta múltiples clientes. La comunicación es en tiempo real.

Arquitectura

Queremos construir una aplicación web con comunicación bidirectional entre el cliente y el servidor. Los clientes inician su dibujo y se puede diferenciar su trazo del trazo de los clientes remotos.

La arquitectura usará ReactJs del lado del cliente y Spring.io del lado del servidor. En le taller le mostraremos cómo construir una arquitectura escalable y entendible usando estos elementos.



Vamos ahora a construir nuestro ejemplo.

Cree la estructura básica del proyecto.

Como siempre debemos partir de una aplicación java básica construida con Maven a la que le agregamos la dependencia web de Spring Boot.

Para crear su ambiente de trabajo vamos a utilizar un controlador simple de Spring que nos garantice que suba el servidor web y que empiece a servir código estático. Para esto debe:

1. Crear una aplicación java básica usando maven.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -
DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4
```

1. Actualizar el pom para utilizar la configuración web-MVC de spring boot. Incluya lo siguiente en su pom.

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<version>3.1.1</version>
</dependency>
</dependencies>
```

1. Cree la siguiente clase que iniciará el servidor de aplicaciones de Spring .

```
package co.edu.escuelaing.interactiveblackboard;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication
public class BBAppStarter {
public static void main(String[] args){
SpringApplication.run(BBAppStarter.class,args);
}
}
```

1. Cree un controlador Web que le permitirá cargar la configuración mínima Web-MVC

```
package co.edu.escuelaing.interactiveblackboard.controllers;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DrawingServiceController {

@RequestMapping(
value = "/status",
method = RequestMethod.GET,
produces = "application/json"
)
public String status() {
return "{\"status\":\"Greetings from Spring Boot. "
+ java.time.LocalDate.now() + ", "
+ java.time.LocalTime.now()
+ ". " + "The server is Runnig!\"}";
}
}
```

1. Cree un index html en la siguiente localización: /src/main/resources/static
2. Corra la clase que acabamos de crear y su servidor debe iniciar la ejecución
3. Verifique que se esté ejecutando accediendo a:

```
localhost:8080/status
```

1. Verifique que el servidor esté entregando elementos estáticos web entrando a:

```
localhost:8080/index.html
```

Nota: Spring una vez arranca los servicios Web empieza a servir recursos estáticos web que se encuentran en:

- /META-INF/resources/

- /resources/
- /static/
- /public/

Nota 2: Usted puede cambiar estos componentes estáticos de manera dinámica y el servidor los actualizará sin necesidad de reiniciarlos.

Ahora construimos el cliente Web

El index.html sería. Solo contiene un elemento “div” con identificador root. A Partir de este elemento construiremos la aplicación. Observe que esta página se encarga de cargar las librerías necesarias y el único script dónde estarán nuestros componentes. Observe que solo usaremos un elemento JSX, es decir no usaremos archivos Js y JSX, esto facilita la depuración y el mantenimiento.

```
<!DOCTYPE html>
<html>
<head>
<title>Interactive BB</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<!-- Load P5.js. -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.7.1/p5.min.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.7.1/addons/p5.dom.min.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.7.1/addons/p5.sound.min.js">
</script>
</head>
<body>
<div id="root"></div>

<!-- Load React. -->
<!-- Note: when deploying, replace "development.js" with "production.min.js". -->
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>

<!-- Load babel to translate JSX to js. -->
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>

<!-- Load our React component. -->
```

```
</body>
</html>
```

Construyamos el componente ReactJS paso a paso.

Primero construimos una versión simple

En le archivo js/bbComponents.jsx escriba:

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
<h1>Bienvenido</h1>
);
```

Este componente ya crea una primera versión de la aplicación

Ahora extendamos una poco y iremos los elementos principales de la interfaz gráfica

```
function Editor({name}) {
return (
<div>
<h1>Hello, {name}</h1>
<hr/>
<div id="toolstatus"></div>
<hr/>
<div id="container"></div>
<hr/>
<div id="info"></div>
</div>
);
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
<Editor name="Daniel"/>
);
```

Este elemento ya instancia un componente y muestra las partes principales de la aplicación. Observe que desde aquí es que estructuramos la pagina es decir si deseamos cambiar la interfaz cambiaremos los componentes y no las páginas.

Ahora creemos un componente para representar el canvas del tablero

Copie este código en le mismo archivo de sus componentes

```
function BBCanvas() {
const [svrStatus, setSvrStatus] = React.useState({loadingState: 'Loading Canvas
...'});
const myp5 = React.useRef(null);
const sketch = function (p) {
let x = 100;
let y = 100;
p.setup = function () {
p.createCanvas(700, 410);
}

p.draw = function () {
if (p.mouseIsPressed === true) {
p.fill(0, 0, 0);
p.ellipse(p.mouseX, p.mouseY, 20, 20);
}
if (p.mouseIsPressed === false) {
p.fill(255, 255, 255);
}
}

};

React.useEffect(() => {
myp5.current = new p5(sketch, 'container');
setSvrStatus({loadingState: 'Canvas Loaded'});
}, []);

return(
<div>
<h4>Drawing status: {svrStatus.loadingState}</h4>
</div>);
}
```

Este componente renderiza un estado del canvas y el canvas. Note que el componte necesita dos renderizaciones para estar totalmente operativo. En la primera simplemente se crean los componentes y en la segunda se carga el canvas. El canvas solo se monta cuando ya se realizo una renderización, es decir cuándo el método “componentDidMount” del ciclo de vida es llamado.

No olvide cargar este componente en el editor:

```
function Editor( {name}
) {
return (
<div>
<h1>Hello, {name}</h1>
<hr/>
<div id="toolstatus"></div>
<hr/>
<div id="container">
<BBCanvas />
</div>
<hr/>
<div id="info"></div>
</div>
);
}
```

En este momento su cliente ya debe estar funcionando.

Ahora vamos a modificarlo para poder interactuar con el servidor usando web sockets.

Construyamos una función y clase que nos servirán para manejar la conexión. Note que estas clases son clases y funciones estándar de Js.

```
// Retorna la url del servicio. Es una función de configuración.
function BBServiceURL() {
return 'ws://localhost:8080/bbService';
}


class WSBBChannel {
constructor(URL, callback) {
this.URL = URL;
this.wsocket = new WebSocket(URL);
this.wsocket.onopen = (evt) => this.onOpen(evt);
this.wsocket.onmessage = (evt) => this.onMessage(evt);
this.wsocket.onerror = (evt) => this.onError(evt);
this.receivef = callback;
}


onOpen(evt) {
console.log("In onOpen", evt);
}
```

```
onMessage(evt) {
console.log("In onMessage", evt);
// Este if permite que el primer mensaje del servidor no se tenga en cuenta.
// El primer mensaje solo confirma que se estableció la conexión.
// De ahí en adelante intercambiaremos solo puntos(x,y) con el servidor
if (evt.data != "Connection established.") {
this.receivef(evt.data);
}
}
onError(evt) {
console.error("In onError", evt);
}

send(x, y) {
let msg = '{ "x": ' + (x) + ', "y": ' + (y) + "}";
console.log("sending: ", msg);
this.wsocket.send(msg);
}


}
```

Modifiquemos el componente BBCanvas para utilizar este web socket

```
function BBCanvas() {
const [svrStatus, setSvrStatus] = React.useState({loadingState: 'Loading Canvas
...'});
const comunicationWS = React.useRef(null);
const myp5 = React.useRef(null);
const sketch = function (p) {
let x = 100;
let y = 100;
p.setup = function () {
p.createCanvas(700, 410);
}

p.draw = function () {
if (p.mouseIsPressed === true) {
p.fill(0, 0, 0);
p.ellipse(p.mouseX, p.mouseY, 20, 20);
comunicationWS.current.send(p.mouseX,p.mouseY);
}
if (p.mouseIsPressed === false) {
p.fill(255, 255, 255);
}
}

};
```

```
React.useEffect(() => {
myp5.current = new p5(sketch, 'container');
setSvrStatus({loadingState: 'Canvas Loaded'});
comunicationWS.current = new WSBBChannel(BBServiceURL(),
(msg) => {
var obj = JSON.parse(msg);
console.log("On func call back ", msg);
drawPoint(obj.x, obj.y);
});
return () => {
console.log('Clossing connection ...')
comunicationWS.current.close();
};
}, []);
function drawPoint(x, y) {
myp5.current.ellipse(x, y, 20, 20);
}

return(
<div>
<h4>Drawing status: {svrStatus.loadingState}</h4>
</div>);
}
```

Antes de ejecutar vamos a crear los componentes del servidor

Primero el Endpoint

```
package co.edu.escuelaing.interactiveblackboard.endpoints;


import java.io.IOException;
import java.util.logging.Level;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.logging.Logger;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import org.springframework.stereotype.Component;


@Component
@ServerEndpoint("/bbService")
```

```
public class BBEndpoint {


private static final Logger logger =
Logger.getLogger(BBEndpoint.class.getName());
/* Queue for all open WebSocket sessions */
static Queue<Session> queue = new ConcurrentLinkedQueue<>();


Session ownSession = null;

/* Call this method to send a message to all clients */
public void send(String msg) {
try {
/* Send updates to all open WebSocket sessions */
for (Session session : queue) {
if (!session.equals(this.ownSession)) {
session.getBasicRemote().sendText(msg);
}
logger.log(Level.INFO, "Sent: {0}", msg);
}
} catch (IOException e) {
logger.log(Level.INFO, e.toString());
}
}

@OnMessage
public void processPoint(String message, Session session) {
System.out.println("Point received:" + message + ". From session: " + session);
this.send(message);
}

@OnOpen
public void openConnection(Session session) {
/* Register this connection in the queue */
queue.add(session);
ownSession = session;
logger.log(Level.INFO, "Connection opened.");
try {
session.getBasicRemote().sendText("Connection established.");
} catch (IOException ex) {
logger.log(Level.SEVERE, null, ex);
}
}


@OnClose
public void closedConnection(Session session) {
/* Remove this connection from the queue */
queue.remove(session);
logger.log(Level.INFO, "Connection closed.");
```

```
}


@OnError
public void error(Session session, Throwable t) {
/* Remove this connection from the queue */
queue.remove(session);
logger.log(Level.INFO, t.toString());
logger.log(Level.INFO, "Connection error.");
}
}
```

Luego el configurador

```
package co.edu.escuelaing.interactiveblackboard.configurator;


import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.web.socket.server.standard.ServerEndpointExporter;


@Configuration
@EnableScheduling
public class BBConfigurator {


@Bean
public ServerEndpointExporter serverEndpointExporter() {
return new ServerEndpointExporter();
}
}
```

No olvide descargar las dependencias de web socket de springboot

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-websocket</artifactId>
<version>6.0.10</version>
<type>jar</type>
</dependency>
```

¿Podemos subirlo a AWS?

Para subirlo a AWS siga los siguientes pasos

Modifique el POM para copiar las dependencias:

```
<!-- build configuration -->
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-dependency-plugin</artifactId>
<version>3.6.0</version>
<executions>
<execution>
<id>copy-dependencies</id>
<phase>package</phase>
<goals><goal>copy-dependencies</goal></goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

Inicie el git y agregue un gitignore

```
> git init
> echo “#TODO copy gitignore from:
https://gist.github.com/dedunumax/54e82214715e35439227” > .gitignore
```

Actualice el git ignore como se indica en el enlace.

Corra su aplicación desde la línea de comandos

Este es el comando que debe usar para correr su proyecto en linux/MACOS

```
java -cp target/classes:target/dependency/*
co.edu.escuelaing.interactiveblackboard.BBAppStarter
```

Este es el comando que debe usar para correr su proyecto en Windows

```
java -cp target/classes;target/dependency/*
co.edu.escuelaing.interactiveblackboard.BBAppStarter
```

Prepare su aplicación para correr en un servidor desconocido

Calcule la dirección del servicio y utilice wss (Protocolo seguro) en cambio de ws (Protocolo no seguro). Utilice esto solo si el protocolo del servidor será seguro.

```
// Retorna la url del servicio. Es una función de configuración.
function BBServiceURL() {
var host = window.location.host;
console.log("Host: " + host);
// En heroku necesita conexiones seguras de web socket
var url = 'wss://' + (host) + '/bbService';
if(host.toString().startsWith("localhost")){
url = 'ws://' + (host) + '/bbService';
}
console.log("URL Calculada: " + url);
return url;
}
```

Inicie Spring en el puerto indicado por el entorno

Revise el concepto de app de 12 factores. Esto prepara su aplicación para ser ejecutada en entornos desconocidos. El administrador podrá por ejemplo pasar el puerto de publicación en una variable de entorno.

```
@SpringBootApplication
public class BBAppStarter {
public static void main(String[] args){
SpringApplication app = new SpringApplication(BBAppStarter.class);
app.setDefaultProperties(Collections
.singletonMap("server.port", getPort()));
app.run(args);
}
static int getPort() {
if (System.getenv("PORT") != null) {
return Integer.parseInt(System.getenv("PORT"));
}
return 8080; //returns default port if PORT isn't set (i.e. on localhost)
}
}
```

Realice un commit de su proyecto

Cree un servidor en aws, envíe los archivos necesarios y ejecute su proyecto. No olvide abrir los puertos.

Recuerde revisar las urls

- /status
- /bb.html

¿Preguntas?