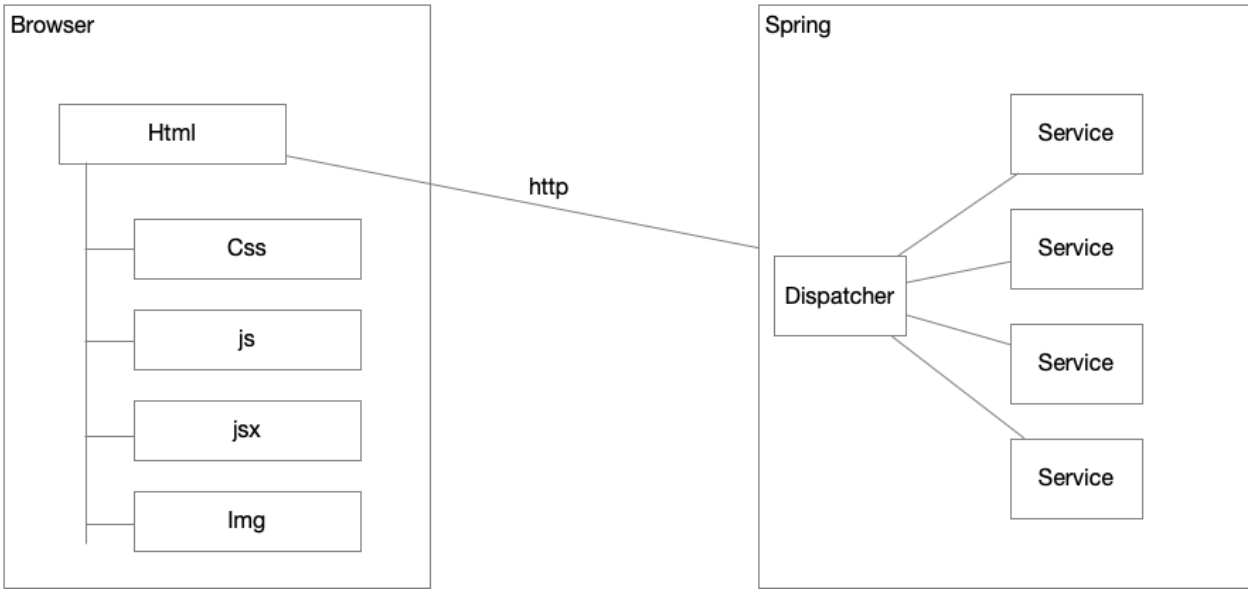
# Spring.io, Websockets y ReactJs

Vamos ahora a realizar un tutorial para implementar una aplicación pequeña que implementa un a aplicación Web que utiliza Web Sockets.

## Arquitectura

Queremos construir una aplicación web con comunicación bidirectional entre el cliente y el servidor. El servidor hace un broadcast de un mensaje cada 5 segundos a todos los clientes conectados. Para realizarlo utilizaremos:

- Spring como servidor web y de aplicaciones,
- el browser con Js y ReactJs como cliente pesado
- Y WebSockets para establecer la conexión bidirectional




### Conceptos de Spring

Recuerde que spring implementa un framework de IoC con inyección de dependencias. El objetivo de este tipo de framework ha sido el facilitar el desarrollo de aplicaciones empresariales, disminuyendo el riesgo de errores y aumentando la eficiencia. Por esto, Spring provee una serie de librerías y un entorno de ejecución que proponen un modelo de programación por componentes.

La idea detrás de esto es que los desarrolladores deben preocuparse por construir componentes con responsabilidades concretas de negocio, y el framework se preocupa por proveer de manera transparente características como:

- Seguridad
- Concurrencia
- Distribución-comunicación
- Soporte multiusuario
- Persistencia
- Tolerancia a fallos
- Escalabilidad

En este contexto el programador provee objetos de negocio y la interfaz gráfica, y el framework IoC se encarga de orquestarlos en una aplicación robusta.

En este contexto los programadores pierden un poco de flexibilidad pero pueden ganar en tiempo de entrega, calidad, escalabilidad y mantenibilidad.

Sin embargo, recuerde que todo esto se ejecuta sobre una máquina virtual por lo tanto tiene todo el poder de esta a su alcance.

**Arquitectura basada en anotaciones**

Las anotaciones son el mecanismo usado para comunicar la intención de un componente dentro del framework. El tipo de componente determina el comportamiento y el ciclo de vida que le otorga el framework. Así, el programador crea un conjunto de componentes, les asigna unas responsabilidades y el framework orquesta la aplicación.

Vamos a revisar algunas de las anotaciones y sus roles respectivos

@Inject (Java Standard), @Autowire (Spring specific). Inyecta componentes por tipo. En un campo inyecta le componente correspondiente, en un constructor llama al constructor con los parámetros correspondientes, y en un setter llama al setter con los parámetros correspondientes. (@Resource es similar pero busca primero por nombre).

```
class Car {
    @Inject
    Engine engine;
}
```

@Bean. Marca un método factory que instancia beans de spring. Es llamado cuando se require un Bean de ese tipo

```
@Bean
Engine engine() {
    return new Engine();
}
```

@Scope. Define el alcance de un @Componente o @Bean. Puede ser singleton, prototype, request, session, globalSession o un alcance personalizado. El siguiente ejemplo define un singleton para todo el entorno IoC.

```
@Component
@Scope(“singleton")
class Engine {}
```

@Component. Marca una clase que se escanea al iniciar Spring y se utiliza para definir un bean. Si no tiene scope por defecto será singleton. En general los componentes deben ser “sin estado”, es decir stateless.

@Repository. Es usado para marcar el acceso a una base de datos, es también un componente escaneado al principio.

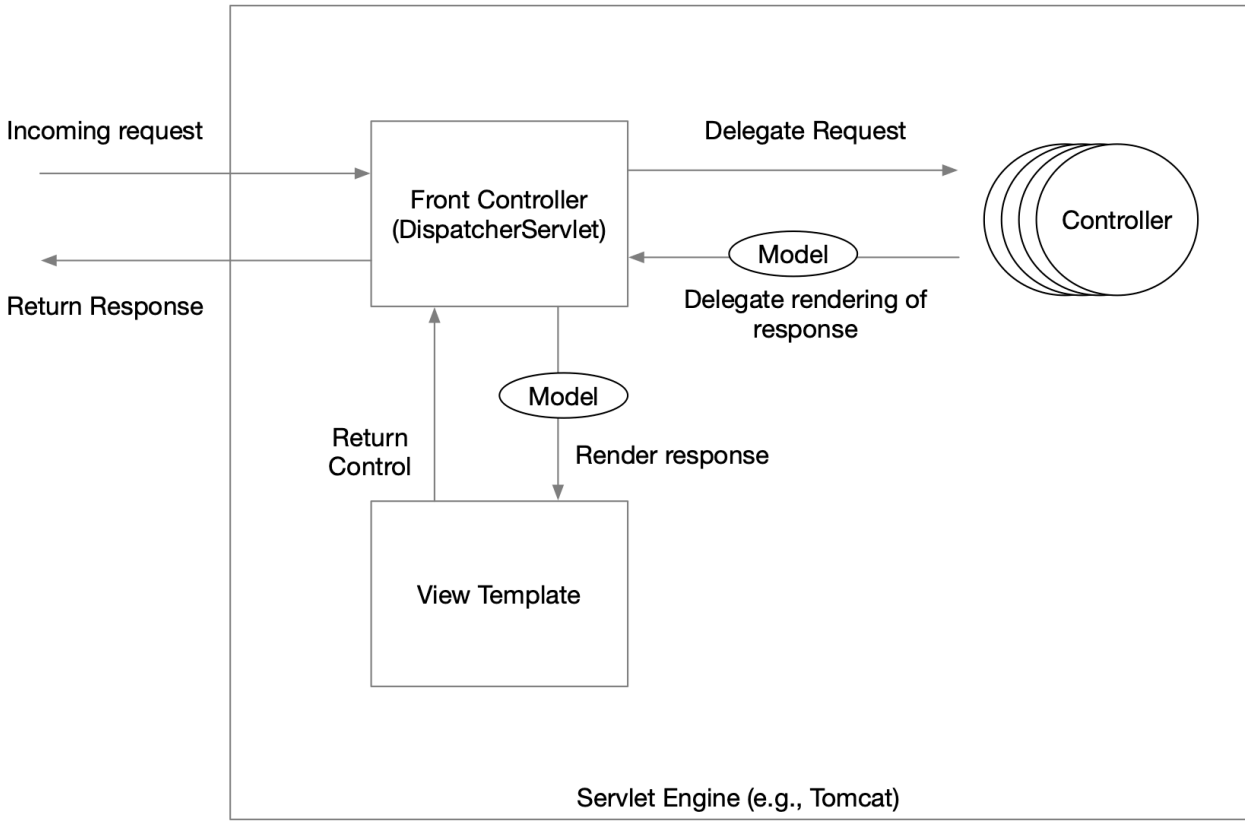
@Service. Un servicio de negocio,, es también un componente escaneado al principio.

Su habilidad para manejar la plataforma dependerá mucho de su conocimiento de los efectos de las anotaciones. Le recomiendo revisar en detalle estás responsabilidades:

- Spring Core Annotations
- Spring Web Annotations
- Spring Boot Annotations
- Spring Scheduling Annotations
- Spring Data Annotations
- Spring Bean Annotations

**Modelo Vista Controlador**

Cuando programamos una aplicación WEB generalmente pensamos en el patron de modelo vista controlador. En este patron el controlador controla el flujo de los mensajes, el modelo tiene el modelo de negocio y la vista presenta los resultados. Para entender el modelo y cómo funciona en spring podemos referirnos a la arquitectura de Dispatcher servlet. Esta arquitectura implementa el patrón Front Controller y se puede ever en la siguiente figura.




**WebSockets**

Vamos ahora a hablar un poco de los websockets. La API de WebSocket  permite abrir un canal de comunicación interactiva bidireccional entre el navegador del usuario y un servidor. Con esta API, puede enviar mensajes a un servidor y recibir respuestas controladas por eventos sin tener que sondear al servidor para obtener una respuesta. Es decir ya no programa con el modelo de solicitudes siempre realizadas por el browser.

WebSocket es funcionalmente similar a los sockets estándar de estilo Unix, pero no están técnicamente relacionados.

## 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
```

2. 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>2.3.1.RELEASE</version>
        </dependency>
    </dependencies>
```

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

```
package co.edu.escuelaing.websocketsprimer;

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

@SpringBootApplication
public class WSStartApp {

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

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

```
package co.edu.escuelaing.websocketsprimer;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class WebController {

    @GetMapping("/status")
    public String status() {
        return "{\"status\":\"Greetings from Spring Boot "
                + java.time.LocalDate.now() + ", "
                + java.time.LocalTime.now()
                + ". " + "The server is Runnig!\"}";
    }

}
```

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

```
localhost:8080/status
```

7. 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.

**Construyamos el EndPoint el servidor con Websockets**

El código de este tutorial está inspirado y adaptado del ejemplo encontrado en. https://docs.oracle.com/javaee/7/tutorial/websocket.htm

```
package co.edu.escuelaing.websocketsprimer.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.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import org.springframework.stereotype.Component;

@Component
@ServerEndpoint("/timer")
public class TimerEndpoint {

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


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


    @OnOpen
    public void openConnection(Session session) {


        /* Register this connection in the queue */
        queue.add(session);
        logger.log(Level.INFO, "Connection opened.");
        try {
            session.getBasicRemote().sendText("Connection established.");
        } catch (IOException ex) {
```

```
            Logger.getLogger(TimerEndpoint.class.getName()).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.");
    }
}
```

**Construyamos una clase que emita mensajes desde el servidor**

```
package co.edu.escuelaing.websocketsprimer.components;


import co.edu.escuelaing.websocketsprimer.endpoints.TimerEndpoint;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.springframework.context.annotation.Scope;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;


@Component
@Scope("singleton")
public class TimedMessageBroker {

    private static final SimpleDateFormat dateFormat = new
SimpleDateFormat("HH:mm:ss");
```

```
    private static final Logger logger =
Logger.getLogger(TimedMessageBroker.class.getName());


    @Scheduled(fixedRate = 5000)
    public void broadcast() {
        logger.log(Level.INFO, "broadcastingMessages");
        TimerEndpoint.send("The time is now " + dateFormat.format(new Date()));
    }
}
```

**Ahora construyamos un componente que nos ayude a configurar el contenedor IoC**

Es necesario construir esta clase porque el contenedor de Servlets en Spring, TOMCAT, tiene deshabilitado por defecto la detección de componentes Endpoints. Así, no carga los componentes si no se le indica explícitamente. Esto parece un error de diseño pero por el momento esta es la situación.

La clase ServerEndpointExporter detecta beans de tipo ServerEndpointConfig y los registra con con el motor standard de java de webSockets.  También detecta beans anotados con ServerEndpoint  y los registra igualmente.

```
package co.edu.escuelaing.websocketsprimer.configuration;

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 WSConfigurator {

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

**Ahora construimos el cliente Web**

El index.html sería

```
<!DOCTYPE html>
<html>
    <head>
        <title>Websockets Testing Client</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <hr/>
        <div id="timer"></div>
        <hr/>


        <!-- Load React. -->
        <!-- Note: when deploying, replace "development.js" with
"production.min.js". -->
        <script src="https://unpkg.com/react@16/umd/react.development.js"
crossorigin></script>
        <script src="https://unpkg.com/react-dom@16/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. -->
        <script src="js/WsComponent.jsx" type="text/babel"></script>
    </body>
</html>
```

**Construyamos el componente ReactJS**

```
class WSClient extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            error: null,
            isLoaded: false,
            msg: ""
        };
    }
```

```
    componentDidMount() {
        this.wsocket = new WebSocket("ws://localhost:8080/timer");
        this.wsocket.onmessage = (evt) => this.onMessage(evt);
        this.wsocket.onerror = (evt) => this.onError(evt);
    }
    onMessage(evt) {
        console.log("In onMessage", evt);
        this.setState({isLoaded:true,msg: evt.data});
    }
    onError(evt) {
        console.error("In onError", evt);
        this.setState({error: evt});
    }



    render() {
        console.log("Rendering...");
        const {error, isLoaded, msg} = this.state;
        if (error) {
            return <div>Error: {error.message}</div>;
        } else if (!isLoaded) {
            return <div>Loading...</div>;
        } else {
            return (
                    <div>
                        <h1>The server status is:</h1>
                        <p>
                            {msg}
                        </p>
                    </div>
                    );
        }
    }
}


ReactDOM.render(
  <WSClient />,
  document.getElementById('timer')
);
```

**¿Preguntas?**