

---

# **MasterServer Documentation**

***Release 1.0.0***

**José Domingo Álvarez Caba**

**May 31, 2020**



# CONTENTS

1	Enlaces a github	3
---	------------------	---



Este proyecto está implementado en el servidor de Efrén pero como los criterios de evaluación de ASIR y los de DAM son distintos hemos decidido separar la documentación de cada uno.



## **ENLACES A GITHUB**

### **1.1 Plugin**

<https://github.com/DomingoAlvarez99/MasterPluginRestApi/tree/master>

### **1.2 REST API**

<https://github.com/DomingoAlvarez99/MasterPlugin>

### **1.3 Vue SPA**

<https://github.com/DomingoAlvarez99/MasterWebApp>

### **1.4 Colecciones de postman**

<https://github.com/DomingoAlvarez99/MasterPluginRestApiPostman>

#### **1.4.1 Introducción**

Se pretende hacer un plugin de minecraft para añadir características extra al juego y para guardar información de los jugadores usando una REST API que se va a crear usando el Framework Spring. También se pretende hacer una SPA (Single Page Application) usando el Framework Vue.js para poder consultar información sobre los jugadores.

#### **1.4.2 Tecnologías y herramientas**

##### **Lenguajes de programación**

Un lenguaje de programación es una serie de instrucciones que le permite a un programador escribir un conjunto de órdenes, acciones consecutivas, datos y algoritmos para crear programas que controlen el comportamiento físico y lógico de la máquina.

## Tipos de lenguajes de programación

- Lenguajes compilados: Los lenguajes compilados requieren de un compilador. Un compilador es un programa que traduce un lenguaje en código máquina. Para traducirlo y crear la parte ejecutable.
- Lenguajes interpretados. Los lenguajes interpretados se ejecutan en tiempo real por un intérprete. Un intérprete es un software que recibe un programa que lo analiza y lo ejecuta.

## Java

Java es un lenguaje de programación orientado a objetos que fue diseñado para tener tan pocas dependencias de implementación como fuera posible permitiendo desarrollar y ejecutar aplicaciones en cualquier dispositivo.

## Programación orientada a objetos (POO)

La POO es una forma más cercana a como expresaríamos las cosas de la vida real en un lenguaje de programación, se basa en la estructuración del código en clases llamadas objetos, los cuales tienen propiedades y métodos.

## Principios de la POO

### Abstracción

La abstracción consiste en que un método o una clase diga lo que tiene que hacer pero no cómo hacerlo. Esto se consigue implementando clases abstractas e interfaces.

### Encapsulamiento

El encapsulamiento es la forma de hacer que los atributos de las clases se puedan editar sólo a través de métodos. De manera general se hace teniendo las propiedades como privadas y los métodos que las controlan públicos. Comúnmente, se crean un grupo de métodos llamados getters que se encargan de obtener el valor de la propiedad y setters que se encargan de dar valor a la propiedad.

Mantener las clases con este principio permite controlar el cambio de valor que pueda producirse en las variables añadiendo validaciones.

```
public class User {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name.startsWith("S")) {
            this.name = name;
        }
    }
}
```



## Herencia

La herencia es un mecanismo por el cual una clase puede heredar métodos y atributos de otra clase. Con esto se puede utilizar funcionalidad existente sin tener que volver a implementar dicha funcionalidad. Todas las clases por defecto heredan funcionalidad de la clase Object.

### Superclase

Es la clase que pasa la funcionalidad a otra clase para que esta pueda utilizar sus métodos. También es conocida como clase padre.

```
public class User {
    private String name;

    public User(String name){
        this.name = name;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

### Subclase

Es la clase que utiliza los métodos y atributos de otra clase para realizar alguna acción específica. También es conocida como clase hija.

Para implementar la herencia se usa la palabra reservada “extends” en la definición de la clase hija seguida del nombre de la clase padre. Poder hacer uso de los métodos y/o atributos que proporciona el padre usa la palabra reservada “super”.

```
public class Operator extends User {
    private String role;

    public Operator(String name, String role){
        super(name);
        this.role = role;
    }

    public String getRole(){
        return role;
    }

    public void setRole(String role){
```

(continues on next page)

(continued from previous page)

```
        this.role = role;
    }
}
```

## Polimorfismo

El polimorfismo es la capacidad de un método, variable u objeto de poseer varias formas.

### Polimorfismo de asignación

El polimorfismo de asignación es el que está más relacionado con el enlace dinámico. Una misma variable referenciada puede hacer referencia a más de un tipo de Clase. El conjunto de clase que pueden ser referenciadas está restringido por la herencia o la implementación.

La forma natural de instanciar un objeto de la clase Operator sería:

```
Operator operator = new Operator();
```

Sin embargo, el polimorfismo de asignación permite a una variable declarada como otro tipo usar otra forma, siempre y cuando haya una relación de herencia o implementación.

```
User user = new Operator();
```

### Polimorfismo de sobrecarga

En el polimorfismo de sobrecarga, dos o más métodos comparten el mismo identificador, pero distinta lista de de argumentos.

```
public void setUsername(String name) {
    this.username = name;
}

public void setUsername(String name, String lastname) {
    this.username = name + lastname;
}
```

### Polimorfismo de inclusión

El polimorfismo de inclusión es la capacidad de redefinir por completo un método.

Redefinir un método de una Superclase en una Subclase.

```
public abstract class User {  
    public abstract void doTheJob();  
  
    public class Operator extends User {  
  
        @Override  
        public void doTheJob() {  
  
        }  
    }  
}
```

Redefinir un método de una interfaz en una clase que lo implemente.

```
public interface Operation {  
    void doTheJob();  
}  
  
public class Operator implements Operation {  
  
    @Override  
    public void doTheJob() {  
  
    }  
}
```

## JavaScript

JavaScript es un lenguaje de programación que permite llevar a cabo actividades en páginas web. Es un lenguaje interpretado, el navegador web es el encargado de leer el código y llevar a cabo las acciones que este indica.

Para poder incrustar JavaScript en un documento HTML se tiene que usar la etiqueta `<script>` y dentro de ella se agrega el código que se va a ejecutar.

Para poder incrustar un archivo JavaScript se que indicar el archivo en el atributo `src` de la etiqueta `<script>`.

## Frameworks de desarrollo

Un Framework es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software.

## Spring

Spring es un framework para el desarrollo de aplicaciones JEE (Java Enterprise Edition). Es el encargado de generar beans a partir de objetos básicos de Java, o clases POJOs (Plain Old Java Objects). Para poder relacionar los beans, Spring usa el concepto de inyección de dependencia a través de la inversión de control. La configuración de los beans se describe mediante ficheros xml. Spring usa anotaciones para poder hacer uso de los propios.

## Plain Old Java Object (POJO)

Un POJO es una clase independiente de cualquier framework. Este tipo de clase no tiene ningún tipo de restricción especial, y se usa para simplificar la estructuración del desarrollo, reduciendo su complejidad, aumentando la legibilidad y facilitando la reutilización de código.

```
public class User {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if (name.startsWith("S")) {
            this.name = name;
        }
    }

    @Override
    public int compareTo(User user) {
        return name.compareTo(user.getName());
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        return prime + ((name == null) ? 0 : name.hashCode());
    }

    @Override
    public boolean equals(Object user) {
        User other = (User) user;
        return name.equals(other.getName());
    }
}
```

## Application.properties

Spring usa un fichero para la configuración de diferentes parámetros de la aplicación. Estos parámetros deben de ser pares clave-valor. Este fichero se debe de ubicar dentro de la carpeta de recurso src/main/resources con el nombre de application.properties.

```
#Postgres config
spring.datasource.url = jdbc:postgresql://192.168.1.165/masterserver
spring.datasource.username = postgres
spring.datasource.password = admin
#Hibernate config
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto = update
```

(continues on next page)

(continued from previous page)

```
spring.jpa.open-in-view = false
file.upload-dir = /home/dalvarez/Documents/uploads
```

Además se pueden definir parámetros propios para después usarlos en la aplicación. Cómo por ejemplo el parámetro file.upload-dir.

## Acceso a parámetros propios

### Usando la anotación @Value

La anotación @Value enlaza la propiedad que se indica en el atributo anotado.

```
@Value("${file.upload-dir}")
private String fileUploadDir;
```

### Usando una clase POJO

La anotación @ConfigurationProperties enlaza el prefijo de la propiedad que se indica con la clase POJO anotada.

```
@ConfigurationProperties(prefix = "file")
public class FileStorageProperties{
    private String uploadDir;

    public String getUploadDir(){
        return uploadDir;
    }

    public String setUploadDir(String uploadDir){
        this.uploadDir = uploadDir;
    }
}
```

Además en la clase Principal se tiene que usar la anotación @EnableConfigurationProperties indicando las clases de configuración que se quieran habilitar para usar esta característica.

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

## Modelo

Los modelos son los encargados de asociar la clase y sus atributos con la tabla y sus campos usando anotaciones que aporta la Java Persistence Api (JPA).

```
@Entity
@Table(name = "\"player\"")
public class PlayerModel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private long id;

    @Column(name = "uuid", length = 50)
    private String uuid;

    @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss", timezone = "Europe/Madrid")
    @Column(name = "first_login")
    private LocalDateTime firstLogin;

    @Column(name = "time_played")
    private long timePlayed;

    @OneToMany(mappedBy = "assasin", fetch = FetchType.EAGER)
    private Set<DeathModel> assasins = new HashSet<>();

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="rank_id")
    private RankModel rank;

    @OneToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="command_id")
    @JsonBackReference
    private CommandModel command;
}
```

- @Entity: Declara que la clase es una entidad de JPA.
- @Table: Indica la tabla a la cual se asocia la entidad.
- @Id: Indica que el atributo hace la función de clave primaria en la tabla.
- @GeneratedValue: Informa que el atributo es auto-incrementado automáticamente. Debe de ir conjuntamente con la anotación @Id.
- @Column: Informa de la tabla a la cual va asociada el atributo.
- @ManyToMany: Indica que esta tabla pertenece a una relación de muchos a muchos.
- @ManyToOne: Indica que la tabla pertenece a una relación de muchos a uno.
- @OneToMany: Indica que la tabla pertenece a una relación de uno a muchos.
- @OneToOne: Indica que la tabla pertenece a una relación de uno a uno.
- @JoinColumn: Indica el atributo de la relación al que hace referencia.
- @JsonBackReference: Indica la parte de la relación que no se va a Serializar.
- @JsonFormat: Indica el formato de la fecha.

Usando la anotación `@ManyToMany` podemos suprimir el modelo correspondiente de la tabla resultante en una relación de muchos a muchos.

Al hacer una relación bidireccional, para que la librería de JSON “Jackson” funcione correctamente, uno de los dos lados de la relación no debe ser serializado, para evitar un problema de recursión infinita.

## Repositorio

Los repositorios son los encargados de realizar las operaciones(Create, read, update and delete) CRUD de los datos de una entidad. Son interfaces que heredan de la interfaz `JpaRepository` que es la que contiene los métodos para realizar operaciones CRUD.

```
@Repository
public interface PlayerRepository extends JpaRepository<PlayerModel, Long> {
    Optional<PlayerModel> findById(String uuid);
    Optional<PlayerModel> findByName(String name);
    List<PlayerModel> findByPrefix(String prefix);
}
```

- `@Repository`: Indica que es un objeto de acceso a datos.

También se pueden declarar métodos propios en la interfaz del repositorio para poder realizar otro tipo de operaciones, cómo el método `findByName` que busca un jugador por su nombre.

Usando la anotación `@ManyToMany` podemos suprimir el modelo correspondiente de la tabla resultante en una relación de muchos a muchos.

Los repositorios se usan como intermediarios entre los servicios y la base de datos. Se basan en la utilización ORM (Object Relational mapping) para generar sentencias SQL que hacen las funciones básicas CRUD. Es un modelo de programación para transformar objetos en tablas de la base de datos.

## Servicio

Los servicios son los encargados de procesar los datos que reciben de los controladores y de acceder a los repositorios. Además se encargan de controlar los errores que se puedan llegar a producir.

```
@Service
public class PlayerServiceImpl implements PlayerService {
    @Autowired
    private PlayerRepository repository;

    @Override
    public List<PlayerModel> getAll() {
        List<PlayerModel> players = repository.findAll();
        if (players.isEmpty()) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND,
↵ "Players not found.");
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        }
        return repository.findAll();
    }

    @Override
    public PlayerModel getById(long id) {
        return repository.findById(id).orElseThrow(() -> new
↳ResponseStatusException(HttpStatus.NOT_FOUND,
        "Id {" + id + "} not found, couldn't get the player.
↳"));
    }

    @Override
    public PlayerModel create(PlayerModel player) {
        if (repository.findByName(player.getName()).isPresent()) {
            throw new ResponseStatusException(HttpStatus.CONFLICT,
↳"Name {" + player.getName() + "} exists,
↳couldn't create the player.");
        }
        if (repository.findById(player.getUuid()).isPresent()) {
            throw new ResponseStatusException(HttpStatus.CONFLICT,
↳"Uuid {" + player.getUuid() + "} exists,
↳couldn't create the player.");
        }
        return repository.save(player);
    }

    @Override
    public boolean delete(long id) {
        repository.findById(id).orElseThrow(() -> new
↳ResponseStatusException(HttpStatus.NOT_FOUND,
        "Id {" + id + "} not found, couldn't delete the
↳player."));
        repository.deleteById(id);
        return true;
    }
}

```

- @Service: Indica que es la capa lógica de negocio.

La clase ResponseStatusException permite lanzar excepciones indicando un código de estado Http, un mensaje y una causa. Esto permite crear excepciones de forma progresiva y además evita crear clases de excepciones propias.

## Controlador

Los controladores son los encargados de interpretar las datos de las peticiones que realiza el usuario dependiendo del método y de la ruta implementada.

```

@RestController
public class PlayerController {
    @Autowired
    private PlayerService service;
}

```

(continues on next page)



(continued from previous page)

```

    @GetMapping("/players")
    public ResponseEntity<List<PlayerModel>> getPlayers() {
        return new ResponseEntity<>(service.getAll(), HttpStatus.OK);
    }

    @GetMapping("/player/{id}")
    public ResponseEntity<PlayerModel> getPlayerById(@PathVariable("id") long id)
    ↪{
        return new ResponseEntity<>(service.getById(id), HttpStatus.OK);
    }

    @PostMapping(value = "/player")
    public ResponseEntity<PlayerModel> createPlayer(@RequestBody PlayerModel_
    ↪player) {
        return new ResponseEntity<>(service.create(player), HttpStatus.OK);
    }

    @PutMapping("/player/{id}")
    public ResponseEntity<PlayerModel> updatePlayer(@PathVariable("id") long id,
    ↪@RequestBody PlayerModel player) {
        return new ResponseEntity<>(service.update(id, player), HttpStatus.
    ↪OK);
    }

    @DeleteMapping("/player/{id}")
    public ResponseEntity<Boolean> deletePlayer(@PathVariable("id") long id) {
        return new ResponseEntity<>(service.delete(id), HttpStatus.OK);
    }
}

```

- **@RestController**: Combina las anotaciones **@ResponseBody** y **@Controller**. Indica la clase que hace de handler y convierte la respuesta de cada petición a JSON usando la librería Jackson.
- **@Autowired**: Permite inyectar los beans automáticamente.
- **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**: Indica la ruta y el método de la petición.
- **@PathVariable**: Indica que un parámetro debe de estar vinculado a un parámetro de la plantilla URI.
- **@RequestBody**: Indica que un parámetro debe de estar vinculado al valor del cuerpo de la petición.

La clase `ResponseEntity` maneja toda la respuesta HTTP incluyendo el cuerpo, cabecera y códigos de estado permitiendo configurar la respuesta que queremos que se envíe desde los endpoints.

## Pruebas unitarias

Los pruebas unitarias verifican el código comprobando si realmente funciona correctamente y asegurando que deban realizar todo lo que esperan que realicen.

```

public class PlayerServiceImplTests implements PlayerServiceTests {

    @Mock

```

(continues on next page)

(continued from previous page)

```

    private PlayerRepository repository;

    @InjectMocks
    private PlayerServiceImpl service;

    @BeforeEach
    public void setup() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    @Override
    public void getAll() {
        Mockito.when(repository.findAll()).thenReturn(new ArrayList
↪ <PlayerModel>());
        Assertions.assertThrows(ResponseStatusException.class, () -> {
            List<PlayerModel> players = service.getAll();
            Assertions.assertNull(players);
        });
        Mockito.verify(repository, Mockito.times(1)).findAll();
    }

    @Test
    @Override
    public void getById() {
        PlayerModel player = new PlayerModel(11, "8c898aa3-b0fb-4695-82fc-
↪ a816a7a3c3ec", "Federico",
            "[Dev]", "&3", "&1", "&4", "&1", CustomDate.getCurrentDate(),
↪ CustomDate.getCurrentDate(),
            12323121, "17.212.1");
        Mockito.when(repository.findById(Mockito.anyLong())).
↪ thenReturn(Optional.of(player));
        PlayerModel result = service.getById(Mockito.anyLong());
        Mockito.verify(repository, Mockito.times(1)).findById(Mockito.
↪ anyLong());
    }

    @Test
    @Override
    public void delete() {
        Assertions.assertThrows(ResponseStatusException.class, () -> {
            service.delete(Mockito.anyLong());
        });
        Mockito.verify(repository, Mockito.times(1)).findById(Mockito.
↪ anyLong());
        Mockito.verify(repository, Mockito.times(0)).deleteById(Mockito.
↪ anyLong());
    }
}

```

- @Mock: Permite convertir un objeto en un Mock.
- @InjectMocks: Permite inyectar al objeto anotado los Mocks creados en esa clase.
- @BeforeEach: Esta anotación permite ejecutar el método anotado antes de cada método anotado por la anotación @Test.
- @Test: Define el método que se puede ejecutar como un caso de prueba.

Los mock objects simulan parte del funcionamiento de una clase, esto se hace para evitar acceder a alguna capa de nuestra aplicación, en este caso la base de datos. Para iniciarlos se usa el método `MockitoAnnotations.initMocks`.

El método `Mockito.verify()` permite verificar cuántas veces se ha ejecutado un método de un objeto.

El método `Mockito.when().thenReturn()` indica qué debe de devolver el método que se indique cuando se ejecute.

Por otra parte se pueden usar arguments matchers para realizar llamadas a métodos mediante 'comodines', de forma que los parámetros a los mismos no se tengan que definir explícitamente. Por ejemplo `Mockito.anyString()`.

## Vue.js

Vue es un Framework progresivo para construir interfaces de usuario. A diferencia de otros Frameworks monolíticos, Vue está diseñado desde cero para ser utilizado incrementalmente. La librería central está enfocada solo en la capa de visualización, y es fácil de utilizar e integrar con otras librerías o proyectos existentes. Por otro lado, Vue también es perfectamente capaz de impulsar sofisticadas Single-Page Applications (SPA ) cuando se utiliza en combinación con herramientas modernas y librerías de apoyo.

Una de las principales características de este framework es la reactividad, eso quiere decir que si cambia una variable en una parte de la vista de la página, Vue actualizará su nuevo valor sin necesidad de que lo hagas manualmente.

## Instalación

Npm es el método de instalación recomendado para construir una SPA.

Para instalar Vue de esta manera, primero se necesita instalar NodeJS y NPM. Una vez instalado, bastaría con lanzar el siguiente comando en la terminal.

```
npm install vue
```

Vue nos ofrece una interfaz de línea de comandos (CLI) que está compuesta por muchas herramientas que nos permiten elegir qué dependencias o librerías queremos usar en nuestro proyecto.

```
npm i -g @vue/cli
```

Para crear un proyecto tenemos que ejecutar el siguiente comando.

```
vue create prueba
```

Este nos dará una opción a escoger entre una configuración manual o por defecto.

```
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
   Manually select features
```

## Componentes

Los componentes de Vue son instancias reutilizables con nombre único que se declaran mediante una etiqueta HTML, como por ejemplo, `<hello/>`.

Estos componentes son ficheros con extensión `.vue`. Están divididos en 3 bloques principales: `<template>`, `<script>` y `<style>`.

```
<template>
  <div>
    <p>{{hello}}</p>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        hello: 'hey'
      }
    }
  }
</script>

<style scoped>
  p {
    text-align: center;
  }
</style>
```

En la etiqueta `<template>` se declara el código HTML para diseñar la vista del componente, en la aplicación se ha usado la librería `bootstrap-vue` para hacer la aplicación responsive y `fontawesome` para usar los iconos que nos proporciona, en este bloque podremos añadir las variables del objeto `data`, métodos...

En la sección `<script>` podemos definir la lógica de la aplicación y determinar el código que queremos ejecutar durante los ciclos de vida del componente.

Finalmente, en el bloque `<style>` añadimos los estilos CSS para personalizar el componente.

## Sintaxis de una plantilla Vue

Vue utiliza una sintaxis de plantilla basada en HTML que permite renderizar declarativamente datos en el DOM. Una de sus principales características es su sistema de reactividad. Como los datos y el DOM están enlazados, al editar el valor de un objeto de JavaScript, la vista se modifica automáticamente.

## Interpolaciones

### Texto

Una forma de enlazar datos a la vista es mediante la sintaxis de Bigote (llaves dobles).

```
<p>{{hello}}</p>
```

La propiedad hello será reemplazada por su valor correspondiente, y si en algún momento este se modifica, se actualizará la vista con el nuevo valor.

### Expresiones JavaScript

Vue permite utilizar expresiones de Javascript para interpretar nuestros datos, como por ejemplo el operador ternario.

```
<p>{{id > 0 ? id : -1}}</p>
```

## Directivas

Las directivas son atributos especiales con el prefijo v-. Se espera que los valores de atributo de la directiva sean una única expresión de JavaScript (con la excepción de v-for).

Las dos directivas más usadas en Vue son: v-bind y v-on, estas tienen una abreviatura usando ":" y "@" respectivamente.

### V-bind

Para poder dar valor a un atributo se utiliza la directiva "v-bind".

#### Sin abreviar

```
<p v-bind:id="id">...</p>
```

#### Abreviado

```
<p :id="id">...</p>
```

### V-on

Para poder lanzar un evento se utiliza la directiva "v-on".

## Sin abreviar

```
<p v-on:click="doTheJob">...</p>
```

## Abreviado

```
<p @click="doTheJob">...</p>
```

## Propiedades computadas, observadores y métodos

Se usan para reutilizar código, la principal diferencia entre un método y una propiedad computada es que la propiedad computada se almacena en caché y un método no. Esto significa que la propiedad computada nunca se actualizará porque no es reactiva. Por otro lado, los observadores reaccionan a los cambios de una instancia de Vue.

```
<script>
  export default {
    data() {
      return {
        message: 'Hey',
        reverseMessage3: ''
      }
    },
    methods: {
      reverseMessage1() {
        this.message = this.message.split('').reverse().join('')
      }
    },
    computed: {
      reverseMessage2() {
        return this.message.split('').reverse().join('')
      }
    },
    watch: {
      reverseMessage3() {
        this.message = this.message.split('').reverse().join('')
      }
    }
  }
</script>
```

## Navegación

Para poder crear una SPA se necesita de un sistema de rutas. Para esto se va a usar la librería vue-router, que permite crear rutas modulares, haciendo que cada ruta sea un componente.

```
const router = new Router({
  mode: 'history',
  base: process.env.BASE_URL,
```

(continues on next page)

(continued from previous page)

```

routes: [
  {
    path: '*',
    redirect: '/',
  },
  {
    path: '/',
    name: 'home',
    component: () => import('../views/Home.vue'),
    meta: {
      title: 'Home'
    }
  },
  {
    path: '/players',
    name: 'players',
    component: () => import('../views/PlayersView.vue'),
    meta: {
      title: 'Players'
    }
  },
  {
    path: '/players/player/:name',
    name: 'player',
    component: () => import('../views/PlayerView.vue'),
    meta: {
      title: 'Player'
    }
  }
]
}))

router.beforeEach((to, from, next) => {
  if (to.name === 'player') {
    document.title = to.params.name
  } else {
    document.title = to.meta.title
  }
  next()
})

export default router

```

- Mode: Activa el history mode de HTML 5, además evita que aparezca una almohadilla en la url al navegar.
- Base: Usa la url base de la app.
- Routes: Contiene las rutas.
- Redirect: Reemplaza la url actual por la que se indique.
- Path: Ruta a dónde se va a dirigir. Cuando una ruta incluye ":" indica que se le va a pasar un parámetro al componente que se va a asociar.
- Name: Es el nombre que se va a asignar a la ruta.
- Component: Es el componente que se va a asociar a la ruta.

### Formas de realizar la navegación

Para mostrar el contenido de la página actual, se tiene que usar el elemento `<router-view>`. Permite renderizar componentes dinámicos en base a una url.

```
<router-view/>
```

La navegación se puede realizar de dos formas: al usar un elemento html, o al usar la instancia del router.

Usando un elemento html se usa la directiva “v-bind:to” dónde se tiene que indicar el nombre de la ruta.

### Usando un elemento html

```
<b-navbar-nav>
  <b-nav-item :to="{name: 'dashboard'}">Dashboard</b-nav-item>
</b-navbar-nav>
```

### Haciendo un push

```
this.router.push({ name: 'player'})
```

## Servicios y Vuex

Para centralizar información y funciones de la app que son accesibles desde cualquier componente se utiliza Vuex.

Los servicios se usan para realizar peticiones HTTP usando la librería Axios dando una capa de abstracción. Se van a usar junto a Vuex.

### Servicios

Los servicios son los encargados de realizar peticiones HTTP usando la librería Axios, cada servicio va a tener acceso a una instancia de axios.

### Estructura

1. Servicio
  - api.js
  - playerService.js



## Api.js

```
export const axiosInstance = axios.create({
  baseURL: process.env.VUE_APP_BASE_URL,
  withCredentials: false,
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  }
})
```

## PlayerService.js

```
export default {
  async retrievePlayersByRank (rankName) {
    const response = await axiosInstance.get('/players/getByRankName/' + rankName)
    return response.data
  },
  async retrievePlayerByName (name) {
    var response = {}
    try {
      response = await axiosInstance.get('/player/getByName/' + name)
    } catch (e) {
      response = e
    }
    return response.data
  },
  async retrievePlayersOnlineNumber () {
    var response = []
    try {
      response = await axiosInstance.get('/players/getByOnline/true')
    } catch (e) {
      response = []
    }
    return response.data
  }
}
```

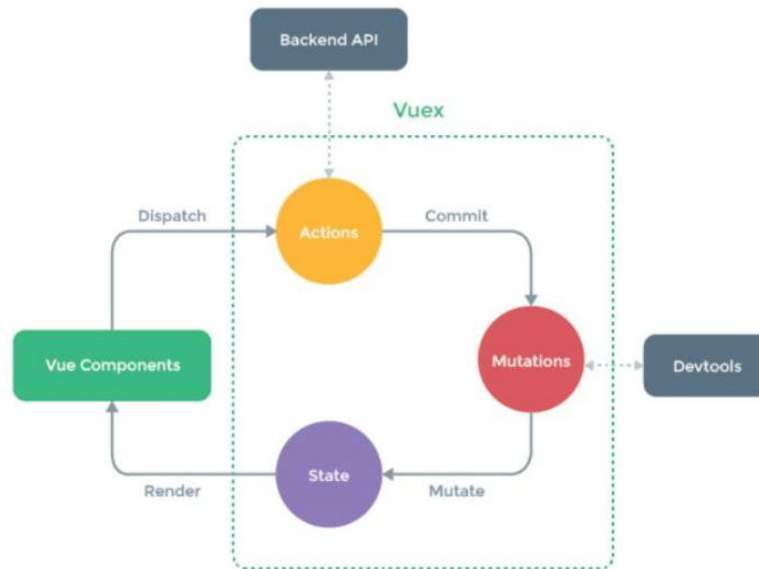
## Vuex

Vuex es un patrón de gestión de estado que nos permite tener los datos centralizados y accesibles en cualquier componente dentro de la aplicación.

Se puede instalar usando Cli o bien usando el comando `npm install vuex --save`.

## Arquitectura

Vuex proporciona 3 estados básicos los cuales se pueden apreciar en la imagen. Aunque tenemos que tener en cuenta uno más que son los getters.



- State: Es un objeto que puede contener cualquier tipo de información y está almacenado de forma centralizada en toda la app.
- Mutation: Son funciones que modifican el estado, son llamadas por las acciones.
- Mutation: Son funciones que modifican el estado, son llamadas por las acciones.
- Getters: Son funciones que devuelven estados. Se llaman desde un componente a través de getters.

## Estructura

### 1. Store

- **Modules**
  - player.js
- index.js

## Index.js

```
export default new Vuex.Store({
  modules: {
    player
  }
})
```

## Player.js

```
const state = () => ({
  players: [],
  player: {},
  playersOnline : 0
})
```

(continues on next page)

(continued from previous page)

```
const getters = {
  getPlayers (state) {
    return state.players
  },
  getPlayer (state) {
    return state.player
  },
  getPlayersOnline (state) {
    return state.playersOnline
  }
}

const actions = {
  async retrievePlayersByRank({ commit }, name) {
    const players = await playerService.retrievePlayersByRank(name)
    commit('setPlayers', players)
  },
  async retrievePlayerByName({ commit }, name) {
    const player = await playerService.retrievePlayerByName(name)
    commit('setPlayer', player)
  },
  async retrievePlayersOnlineNumber({ commit }) {
    const player = await playerService.retrievePlayersOnlineNumber()
    var playersOnline = 0
    if (typeof player === 'undefined') {
      playersOnline = 0
    } else {
      playersOnline = player.length
    }
    commit('setPlayersOnline', playersOnline)
  }
}

const mutations = {
  setPlayers (state, players) {
    state.players = players
  },
  setPlayer (state, player) {
    state.player = player
  },
  setPlayersOnline (state, playersOnline) {
    state.playersOnline = playersOnline
  }
}

export default {
  namespaced: true,
  state,
  getters,
  actions,
  mutations
}
```

## Cómo usar los estados desde un componente

```
await this.$store.dispatch('player/retrievePlayerByName', this.name)
await this.$store.getters['player/getPlayer']
```

- Acciones: Se llaman usando dispatch, al usar módulos se tiene que indicar el nombre del módulo seguido del nombre de la acción.
- Getters: Se llaman usando getter, al usar módulos se tiene que indicar el nombre del módulo seguido del nombre del getter.

## Internacionalización

La internacionalización o traducción de sitios Web, es una característica importante que nos facilita el Framework Vue.js.

### Vue-i18n

Vue I18n es el plugin de internacionalización de Vue.js. Integra fácilmente algunas características de localización en una aplicación de Vue.js.

### Instalación usando Cli

La ventaja de usar Cli es que añade todo lo necesario para que funcione el plugin automáticamente.

```
vue add i18n
```

### Añadir traducciones

- En la carpeta src/locales/\*.js con un conjunto de ficheros de traducción.
- En la sección <i18n> de un componente de archivo único.

Se pueden usar conjuntamente ambas opciones. La principal diferencia es los archivos JSON son accesibles de forma global mientras que las secciones <i18n> sólo están disponibles en ese componente.

### En archivos .vue

Se pueden añadir las traducciones en la sección <i18n>.

```
<i18n>
{
  "en": {
    "stats": "Statistics",
```

(continues on next page)

(continued from previous page)

```

    "achievements": "Achievements",
    "punishments": "Punishments"
  },
  "es": {
    "stats": "Estadísticas",
    "achievements": "Logros",
    "punishments": "Penalizaciones"
  }
}
</i18n>

```

## En archivos JSON

### En src/locales/en.json (Para las traducciones en Espa)

```

{
  "stats": "Statistics",
  "achievements": "Achievements",
  "punishments": "Punishments"
}

```

### En src/locales/es.json

```

{
  "stats": "Estadísticas",
  "achievements": "Logros",
  "punishments": "Penalizaciones"
}

```

## Usar las traducciones en las plantillas

### Texto simple

```
<p>{{t('hello')}}</p>
```

### Texto usando parámetros

```
<p>{{t('hello', { name: 'Federico' })}}</p>
```

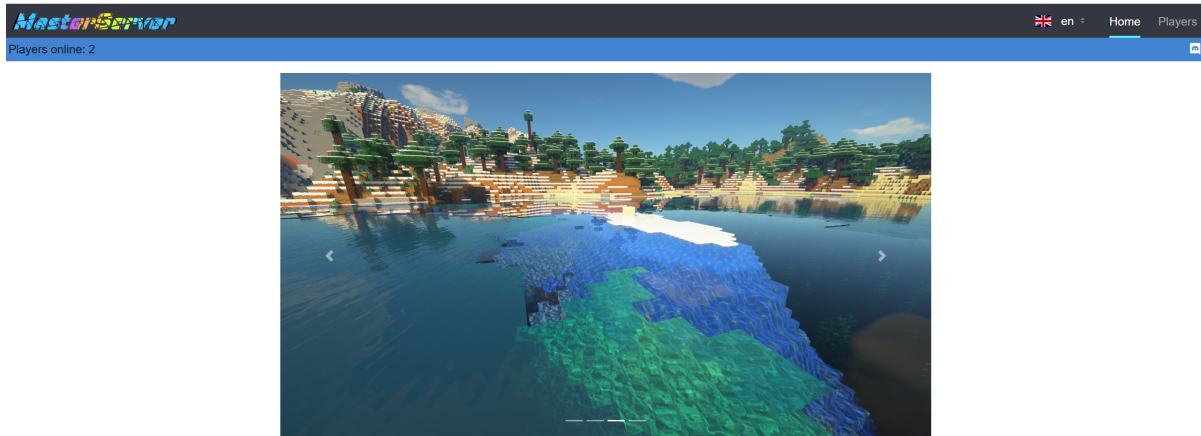
## Diseño de la aplicación

La función de la aplicación es mostrar estadísticas sobre los jugadores del servidor. Además está disponible en dos idiomas (Español e Inglés).

## Vistas

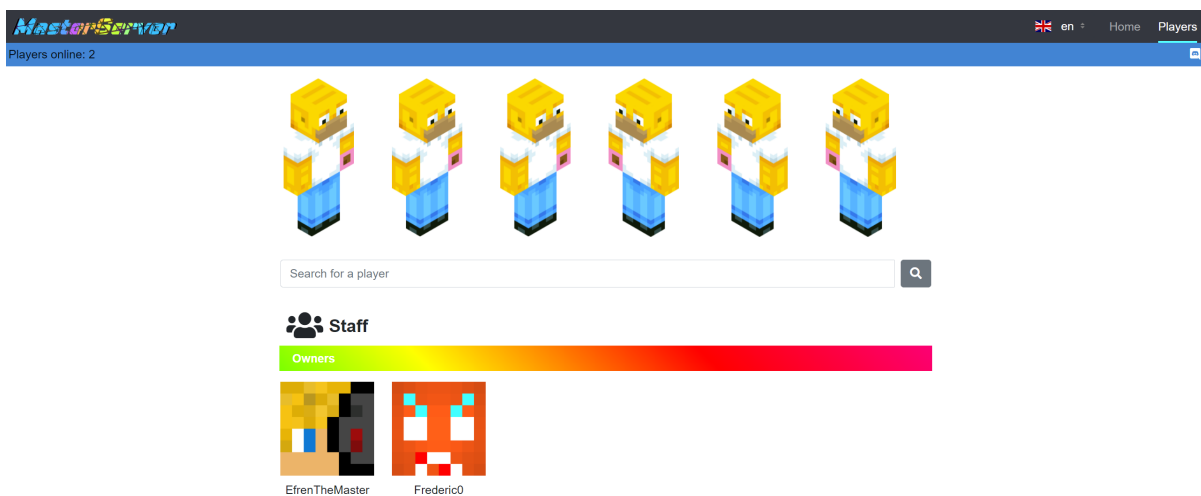
### Inicio

Muestra un slider con imágenes del servidor.



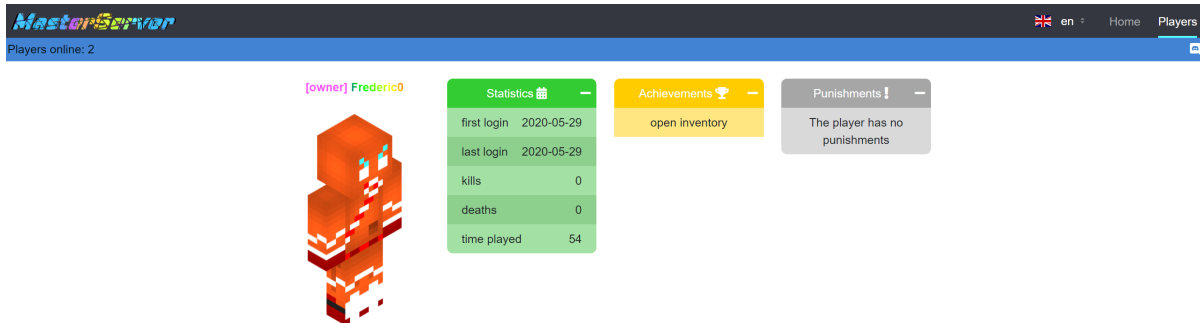
### Jugadores

Muestra la skin de los últimos 6 jugadores que han entrado en el servidor, además hay un input para buscar a los jugadores por su nombre y muestra los jugadores que pertenecen al staff del servidor.



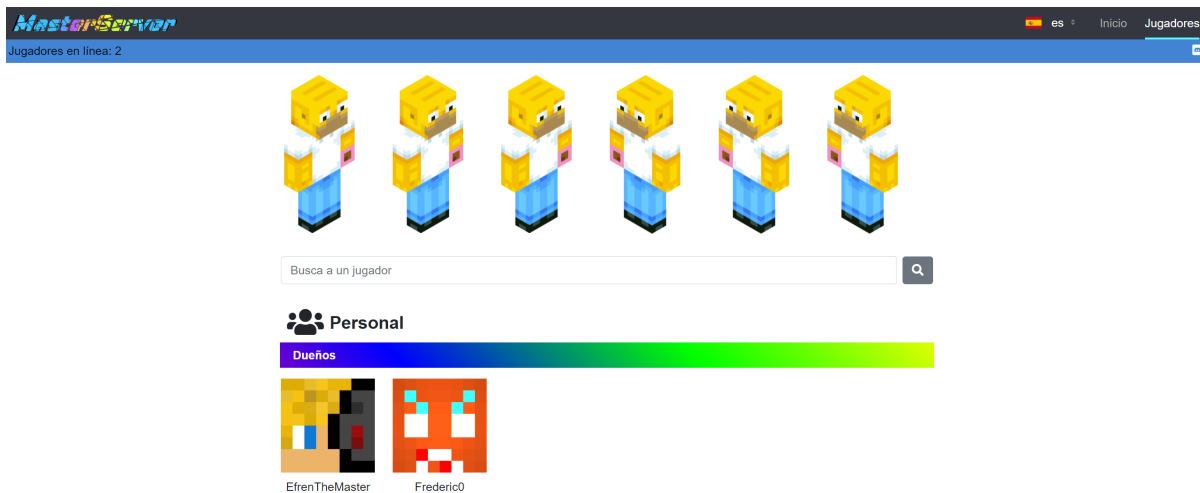
## Jugador actual

Muestra estadísticas y características sobre el jugador que se ha buscado en el input de la anterior vista.



## Jugadores traducida

Ejemplo de una vista traducida.



## Entornos de desarrollo integrados (IDE)

Un IDE es un sistema de software para el diseño de aplicaciones que combina herramientas del desarrollador comunes en una sola interfaz gráfica de usuario (GUI). Un IDE se compone de un editor de código fuente, un compilador y de un depurador.

### Spring Tool Suite (STS)

Es un IDE basado en la versión EE de Eclipse customizado para trabajar con Spring Framework. Está disponible para múltiples plataformas y está orientado al desarrollo de Java.

### Visual Studio Code

Visual Studio Code es un IDE completo para programar, depurar, probar e implementar soluciones en cualquier plataforma. En este caso se va a usar junto a Vue.js.

## REST API

El término REST (Representational State Transfer) se originó en el año 2000, descrito en la tesis de Roy Fielding, padre de la especificación HTTP. Un servicio REST no es una arquitectura software, sino un conjunto de restricciones con las que podemos crear un estilo de arquitectura software, la cual podremos usar para crear aplicaciones web respetando HTTP.

Las restricciones que definen a un sistema RESTful son:

- Cliente-servidor: esta restricción mantiene al cliente y al servidor acoplados. Esto quiere decir que el cliente no necesita conocer los detalles de implementación del servidor y el servidor se “despreocupa” de cómo son usados los datos que envía al cliente.
- Sin estado: cada petición que recibe el servidor debería ser independiente, es decir, no es necesario mantener sesiones.
- Cacheable: debe admitir un sistema de almacenamiento en caché. La infraestructura de red debe soportar una caché de varios niveles. Este almacenamiento evitará repetir varias conexiones entre el servidor y el cliente para recuperar un mismo recurso.
- Interfaz uniforme: define una interfaz genérica para administrar cada interacción que se produzca entre el cliente y el servidor de manera uniforme, lo cual simplifica y separa la arquitectura. Esta restricción indica que cada recurso del servicio REST debe tener una única dirección, “URI”.
- Sistema de capas: el servidor puede disponer de varias capas para su implementación. Esto ayuda a mejorar la escalabilidad, el rendimiento y la seguridad.

## PROTOCOLO HTTP

HTTP (hypertext Transfer Protocol) es el protocolo de comunicaciones usado en la Web para intercambiar documentos HTML, archivos CSS, Javascript, imágenes y otros recursos similares. El protocolo HTTP sigue un esquema petición-respuesta en donde un navegador web, el cliente del protocolo, envía un mensaje de petición a un servidor web y, en consecuencia el servidor devuelve un mensaje de respuesta.



En este protocolo, cada mensaje de petición y de respuesta se compone de un conjunto de líneas de texto. Por ejemplo, el mensaje de petición incluye una primera línea de texto que incluye la operación (conocida como método o verbo) que desea ejecutar el cliente, un conjunto de líneas adicionales con campos de encabezado, una línea vacía que representa el final del encabezado y, opcionalmente, el texto del mensaje. Esta simplicidad en el formato de los mensajes ha facilitado el desarrollo de una gran variedad de clientes y servidores web.

Algunos de los métodos de petición más usados por este protocolo son:

## **GET**

Éste método devuelve la información (en forma de entidad) asociada al recurso identificado con la URI solicitada.

Ejemplo: GET /players HTTP/1.1 → Obtiene todos los jugadores.

Ejemplo con parámetros: GET /player?name=Federico HTTP/1.1 → Obtiene los jugadores cuyo nombre es Federico.

## **POST**

Éste método es usado para que el servidor acepte la entidad enviada como parte de la petición, como un nuevo elemento del recurso asociado a la URI solicitada.

Ejemplo: POST /player HTTP/1.1 → Crea ese jugador.

El contenido va en el body del request, no aparece nada en la URL, aunque se envía en el mismo formato que con el método GET. Se debe de indicar el Content-Type. Se suele usar en formularios html porque es más seguro que el método GET, para crear un usuario en una base de datos, etc.

## **PUT**

Éste método es usado para que la entidad enviada como parte del request sea guardada bajo la URI solicitada. Si la entidad se refiere a un recurso ya existente, se procesa como una entidad actualizada.

Ejemplo: PUT /player/1 HTTP/1.1 → Actualiza ese jugador.

En el body del request se tiene que indicar el contenido que se va a actualizar.

## **DELETE**

Este método indica al servidor que el recurso identificado con la URI solicitada debe ser eliminado. Nunca lleva datos en el cuerpo.

Ejemplo: DELETE /player/1 HTTP/1.1 → Borra ese jugador.

## HEAD

Su funcionamiento es idéntico al de GET, con la excepción que no devuelve el cuerpo de la respuesta. Solo se reciben los datos del encabezado.

## Códigos HTTP

Los códigos de estado de respuesta HTTP indican si se ha completado satisfactoriamente una solicitud HTTP específica. Las respuestas se agrupan en cinco clases: respuestas informativas, respuestas satisfactorias, redirecciones, errores de los clientes y errores de los servidores. Nos vamos a centrar en los códigos de las respuestas satisfactorias y en los errores de los clientes más usados.

### Respuestas satisfactorias

#### 200 OK

La solicitud ha tenido éxito. El significado de un éxito varía dependiendo del método HTTP:

- GET: El recurso se ha obtenido y se transmite en el cuerpo del mensaje.
- HEAD: Los encabezados de entidad están en el cuerpo del mensaje.
- PUT o POST: El recurso que describe el resultado de la acción se transmite en el cuerpo

#### 201 CREATED

La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición PUT.

#### 400 NO CONTENT

La petición se ha completado con éxito pero su respuesta no tiene ningún contenido, aunque los encabezados pueden ser útiles. El agente de usuario puede actualizar sus encabezados en caché para este recurso con los nuevos valores.

### Errores de cliente

#### 400 BAD REQUEST

Esta respuesta significa que el servidor no pudo interpretar la solicitud dada una sintaxis inválida.

#### 401 UNAUTHORIZED

Es necesario autenticar para obtener la respuesta solicitada. Esta es similar a 403, pero en este caso, autenticación es posible.

## 403 FORBIDDEN

El cliente no posee los permisos necesarios para cierto contenido, por lo que el servidor está rechazando otorgar una respuesta apropiada.

## 404 NOT FOUND

El servidor no pudo encontrar el contenido solicitado. Este código de respuesta es uno de los más famosos dada su alta ocurrencia en la web.

## 405 METHOD NOT ALLOWED

El método solicitado es conocido por el servidor pero ha sido deshabilitado y no puede ser utilizado. Los dos métodos obligatorios, GET y HEAD, nunca deben ser deshabilitados y no debiesen retornar este código de error.

## PostgreSQL

PostgreSQL es un gestor de bases de datos relacional y orientado a objetos. Su licencia y desarrollo es de código abierto, siendo mantenida por una comunidad de desarrolladores, colaboradores y organizaciones comerciales de forma libre y desinteresadamente.

Características:

- Presenta un sistema de alta concurrencia: Presenta un sistema denominado MVCC, el cual permite que mientras un proceso escribe una tabla, otros puedan acceder a la misma tabla sin necesidad de verse bloqueados, y cada usuario obtiene una visión consistente.
- Notificaciones a tiempo real: A pesar de que PostgreSQL no fue diseñada para ser una BD que trabaje al 100% en tiempo real, si es posible mantener sincronizado en varios dispositivos un sistema de notificación para cuando se hacen cambios específicos en la base de datos, gracias a las funciones LISTEN, UNLISTEN y NOTIFY.
- Registro y guardado de transacciones: Es capaz de registrar cada transacción en un WAL (Write-Ahead-Log). Esto permite restaurar la base de datos a cualquier punto previamente guardado, una especie de “Checkpoint”. Esto permite que no sea necesario realizar respaldos completos de forma frecuente.
- Disparadores: En PostgreSQL, un disparador se define como la ejecución de un procedimiento almacenado, basado en una acción determinada sobre una tabla específica en la base de datos.

## Minecraft

Minecraft es un videojuego de construcción, de tipo mundo abierto, por lo que no posee un objetivo específico, permitiéndole al jugador una gran libertad en cuanto a la elección de su forma de jugar. A pesar de ello, el juego posee un sistema de logros. El modo de juego predeterminado es en primera persona, aunque los jugadores tienen la posibilidad de cambiarlo a tercera persona.

El juego se centra en la colocación y destrucción de bloques, siendo que este se compone de objetos tridimensionales cúbicos, colocados sobre un patrón de rejilla fija. Estos cubos o bloques representan principalmente distintos elementos de la naturaleza, como tierra, piedra, minerales, troncos, entre otros. Los jugadores son libres de

desplazarse por su entorno y modificarlo mediante la creación, recolección y transporte de los bloques que componen al juego, los cuales solo pueden ser colocados respetando la rejilla fija del juego.

### Servidor

Los servidores de Minecraft permiten a los usuarios jugar en línea con otra gente. Pueden estar alojados en un servidor dedicado, o ser temporales y estar en una máquina doméstica.

Para poder usar un servidor de minecraft lo que tenemos que hacer es descargar un jar del servidor desde el cliente de minecraft.

Una vez descargado lo ejecutamos y se creará un fichero llamado EULA.txt (acuerdo de licencia de usuario final) que primero debemos aceptar. Para aceptarlo abrimos el archivo y cambiamos el valor de la propiedad eula por verdadero.

```
eula = true
```

Lo siguiente es configurar el fichero server.properties. Dónde tenemos que asignar la ip del servidor de minecraft. Además de que podemos modificar otros parámetros del servidor.

```
#Minecraft server properties
#Sat May 10 15:46:06 CEST 2020
spawn-protection=16
generator-settings=
force-gamemode=false
allow-nether=true
gamemode=0
broadcast-console-to-ops=true
enable-query=false
player-idle-timeout=0
difficulty=1
spawn-monsters=true
op-permission-level=4
resource-pack-hash=
announce-player-achievements=true
pvp=true
snooper-enabled=true
level-type=DEFAULT
hardcore=false
enable-command-block=false
max-players=20
network-compression-threshold=256
max-world-size=29999984
server-port=25565
debug=false
server-ip=192.168.1.150
spawn-npcs=true
allow-flight=false
level-name=world
view-distance=10
```

(continues on next page)

(continued from previous page)

```
resource-pack=
spawn-animals=true
white-list=false
generate-structures=true
online-mode=true
max-build-height=256
level-seed=
enable-rcon=false
motd=A Minecraft Server
```

Para poder ver la consola y para poder añadir más memoria RAM al servidor, creamos un script con el siguiente contenido.

```
java -Xms8G -Xmx8G -jar server.jar
```

Una vez hecho esto ya podemos ejecutar el script y se iniciará el servidor.

```
D:\MasterServer>java -Xms8G -Xmx8G -jar server.jar
Loading libraries, please wait...
[15:46:06 INFO]: Starting minecraft server version 1.8.8
[15:46:06 INFO]: Loading properties
[15:46:06 INFO]: Default game type: SURVIVAL
[15:46:06 INFO]: This server is running CraftBukkit version git-Spigot-db6de12-
18fbb24 (MC: 1.8.8) (Implementing API version 1.8.8-R0.1-SNAPSHOT)
[15:46:06 INFO]: Debug logging is disabled
[15:46:06 INFO]: Server Ping Player Sample Count: 12
[15:46:06 INFO]: Using 4 threads for Netty based IO
[15:46:06 INFO]: Generating keypair
[15:46:06 INFO]: Starting Minecraft server on 192.168.1.150:25565
```

## Plugin

Un plugin de minecraft es un componente que usa la API Bukkit. Esta permite modificar la experiencia de multijugador de Minecraft mediante el uso de complementos.

## Cómo desarrollar un plugin de Minecraft

### Añadir la dependencia del servidor

Para poder acceder a las clases que proporciona Bukkit se tiene que añadir la dependencia del servidor de Bukkit que se vaya a implementar.

En este caso vamos a usar Maven para bajar la dependencia de un repositorio.

```
<repositories>
  <repository>
    <id>spigot-repo</id>
    <url>https://hub.spigotmc.org/nexus/content/repositories/snapshots/</url>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>org.spigotmc</groupId>
    <artifactId>spigot-api</artifactId>
    <version>1.15.2-R0.1-SNAPSHOT</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

## Clase principal

La clase principal es el punto de entrada para que Bukkit cargue e interactúe con su complemento. Es una clase que extiende de `JavaPlugin`.

```
public class Main extends JavaPlugin {
    private static final String ENABLED_MESSAGE = "-----MasterPlugin_
↪enabled-----";
    private static final String DISABLED_MESSAGE = "-----
↪MasterPluginDisabled disabled-----";

    @Override
    public void onEnable() {
        Bukkit.getConsoleSender().sendMessage(ENABLED_MESSAGE);
    }

    @Override
    public void onDisable() {
        super.onDisable();
        Bukkit.getConsoleSender().sendMessage(DISABLED_MESSAGE);
    }

    @Override
    public boolean onCommand(CommandSender sender, Command command, String label,
↪String[] args) {
        return super.onCommand(sender, command, label, args);
    }
}
```

- `onEnable`: Se ejecuta cuando el plugin se activa.
- `onDisable`: Se ejecuta cuando el plugin se desactiva.
- `onCommand`: Se ejecuta cuando un jugador escribe algo en el chat que comienza por “/”. Este método permite añadir comandos personalizados y/o modificar el funcionamiento de comandos existentes. Además permite acceder al objeto jugador que ha ejecutado ese comando.

## Eventos y listeners

Los listeners permiten escuchar acciones que se realizan en el servidor de minecraft. Por ejemplo, cuando un jugador entra en el servidor.

```
public class OnJoinListener implements Listener {

    public OnJoinListener(Main plugin) {
        plugin.getServer().getPluginManager().registerEvents(this, plugin);
    }

    @EventHandler
    public void onPlayerJoin(PlayerJoinEvent event) {
        Player player = event.getPlayer();
        player.sendMessage("Welcome: " + player.getName());
    }

}
```

Para registrar el listener se tiene que llamar al método registerEvents de la clase principal, y se tienen que pasar como argumentos la clase que va a hacer de listener y la clase del plugin principal.

- @EventHandler: Indica que se pueden vincular eventos a ese método. El tipo de evento se especifica con un argumento. En este caso el evento se ejecuta cuando un jugador entra en el servidor.

Además desde la clase principal en el método onEnable() se tiene instanciar la clase del listener.

```
@Override
public void onEnable() {
    new OnJoinListener(this);
}
```

## Plugin.yml

El archivo plugin.yml es necesario y se debe de ubicar en la raíz del proyecto. Proporciona información esencial a Bukkit para cargar su complemento.

```
name: "MasterPlugin"
version: "0.1.1"
main: "com.masterplugin.Main"
author: "Domingo Álvarez"
description: "This plugin does so much stuff it can't be contained!"
api-version: "1.8.8"
commands:
  change:
    usage0: "name color [color] [player_name?]"
    usage1: "prefix color [color] [player_name?]"
    usage2: "player rank [player_name]"
```

(continues on next page)

(continued from previous page)

```
list:
  usage0: "colors"
  usagel: "commands"
  usagel: "warps"
add:
  usage0: "rank [rank_name]"
  usagel: "rank permission [rank_name] [permission_name]"
  usage2: "chat player [chat_name] [player_name]"
delete:
  usage0: "rank permission [rank_name] [permission_name]"
  usagel: "chat player [chat_name] [player_name]"
  usage2: "chat [chat_name]"
chunksLoaded:
warp:
  usage0: "[warp_name]"
ping:
  usage0: "[player_name?]"
tpRandom:
  usage0: "[player_name?]"
create:
  usage0: "warp [warp_name]"
delete:
  usage0: "warp [warp_name]"
warp:
  usage0: "[warp_name]"
seen:
  usage0: "[player_name?]"
ping:
  usage0: "[player_name?]"
```

- Name: Indica el nombre del plugin.
- Version: Indica la versión del plugin.
- Main: Indica la clase principal del plugin.
- Author: Indica el autor del plugin.
- Description: Indica la descripción del plugin.
- Api-version: Indica la versión de la API que se ha usado.
- Commands: Contiene los comandos que se han creado.

## Cómo acceder a la REST API

### HttpClient

El `httpClient` es la clase que se encarga de realizar las llamadas HTTP en la API REST. Con esto lo que se hace es guardar y recibir datos de la base de datos.

```
public class HttpClient {
    private static final String URL = "http://localhost:8080/";

    private static void request(String path, final String method, Observer<String>
observer) {
```

(continues on next page)



(continued from previous page)

```

        CloseableHttpClient httpClient = null;
        try {
            httpClient = HttpClients.createDefault();
            HttpRequestBase requestBase = new HttpRequestBase() {
                @Override
                public String getMethod() {
                    return method;
                }
            };
            Header[] newHeaders = { new BasicHeader("Content-type",
↪ "application/json"),
                                new BasicHeader("Accept", "application/json") };
↪ };

            requestBase.setURI(new URI(path));
            requestBase.setHeaders(newHeaders);
            CloseableHttpResponse httpResponse = httpClient.
↪ execute(requestBase);

            HttpEntity entity = httpResponse.getEntity();
            String result = EntityUtils.toString(entity);
            if (httpResponse.getStatusLine().getStatusCode() >= 400) {
                observer.onFailure(new Throwable(result));
            } else {
                observer.onSuccess(result);
            }
            entity.getContent().close();
        } catch (URISyntaxException | IOException e) {
            observer.onFailure(e.getCause());
        } finally {
            try {
                httpClient.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    private static <T> void request(String path, final String method, T object,
↪ Observer<String> observer) {
        CloseableHttpClient httpClient = null;
        try {
            httpClient = HttpClients.createDefault();
            HttpEntityEnclosingRequestBase requestBase = new
↪ HttpEntityEnclosingRequestBase() {
                @Override
                public String getMethod() {
                    return method;
                }
            };
            Header[] newHeaders = { new BasicHeader("Content-type",
↪ "application/json"),
                                new BasicHeader("Accept", "application/json") };
↪ };

            requestBase.setURI(new URI(path));
            requestBase.setHeaders(newHeaders);
            String json = new Gson().toJson(object);
            requestBase.setEntity(new StringEntity(json));
            CloseableHttpResponse httpResponse = httpClient.
↪ execute(requestBase);

```

(continues on next page)

(continued from previous page)

```

        HttpEntity entity = httpResponse.getEntity();
        String result = EntityUtils.toString(entity);
        if (httpResponse.getStatusLine().getStatusCode() >= 400) {
            observer.onFailure(new Throwable(httpResponse.
↪getStatusLine().toString()));
        } else {
            observer.onSuccess(result);
        }
        entity.getContent().close();
    } catch (URISyntaxException | IOException e) {
        observer.onFailure(e.getCause());
    } finally {
        try {
            httpClient.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public static void get(String path, String route, Observer<String> observer) {
    request(path + route, "GET", observer);
}

public static void get(String route, Observer<String> observer) {
    request(URL + route, "GET", observer);
}

public static <T> void post(String path, String route, T object, Observer
↪<String> observer) {
    request(path + route, "POST", object, observer);
}

public static <T> void post(String route, T object, Observer<String>
↪observer) {
    request(URL + route, "POST", object, observer);
}

public static <T> void update(String path, String route, T object, Observer
↪<String> observer) {
    request(path + route, "PUT", object, observer);
}

public static <T> void update(String route, T object, Observer<String>
↪observer) {
    request(URL + route, "PUT", object, observer);
}

public static void delete(String path, String route, Observer<String>
↪observer) {
    request(path + route, "DELETE", observer);
}

public static void delete(String route, Observer<String> observer) {
    request(URL + route, "DELETE", observer);
}

```

(continues on next page)

(continued from previous page)

}

## Cómo ejecutar el plugin en el servidor

Para poder usar el plugin en el servidor necesitamos generar un jar del proyecto. Para eso se va a usar Maven. Lo que tenemos que hacer es añadir lo siguiente en el fichero POM.xml para que el jar incluya las dependencias.

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>com.masterplugin.Main</mainClass>
            </manifest>
          </archive>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

Para construir el jar tenemos que ejecutar el siguiente comando

```
mvn clean compile assembly:single
```

Esta sería la salida del comando.

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.masterplugin:plugin >-----
[INFO] Building plugin 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-clean-plugin:3.1.0:clean (default-clean) @ plugin ---
[INFO] Deleting C:\Users\DAlvarez\Documents\workspace-spring-tool-
↪suite\MasterPlugin\target
[INFO]
[INFO] --- maven-resources-plugin:3.0.2:resources (default-resources) @ plugin ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
```

(continues on next page)

(continued from previous page)

```
[INFO]
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ plugin ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 23 source files to C:\Users\DAlvarez\Documents\workspace-spring-tool-
suite\MasterPlugin\target\classes
[INFO]
[INFO] --- maven-assembly-plugin:2.2-beta-5:single (default-cli) @ plugin ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.985 s
[INFO] Finished at: 2020-05-30T18:16:16+02:00
[INFO] -----
```

Ahora tenemos que copiar ese jar en la carpeta plugins del propio servidor.

Si el servidor se estaba ejecutando basta con ejecutar el comando `/reload` en la terminal para cargar el plugin.

### Características implementadas

Algunas de las características que se han implementado son las siguientes.

### Mensaje de bienvenida

Al entrar en el servidor se muestra un mensaje de bienvenida mostrando el rango y el nombre del jugador.



A screenshot of a Minecraft chat message. The text is "Welcome Back To The Main Server [Member] Frederic0". The words "Welcome Back To The Main Server" are in a rainbow gradient, "[Member]" is in purple, and "Frederic0" is in green. The message is displayed on a black background with a red dashed border.

### Sistema de rangos

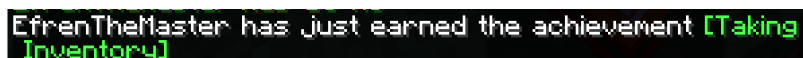
Se usa para autorizar a los jugadores a usar los recursos y/o comandos que ejecuten.

De momento los rangos que se usan son miembro y dueño.

### Guardar los logros de un jugador

Los logros son una forma de orientar a los nuevos jugadores de Minecraft y proponer desafíos a completar.

Cuando se realiza ese evento, el logro que ha conseguido el jugador se guarda en la base de datos.



A screenshot of a Minecraft chat message. The text is "EfrenTheMaster has just earned the achievement [Taking Inventory]". The text is in a green gradient. The message is displayed on a black background with a red dashed border.

## Comandos

Lo que está entre llaves son los parámetros del comando.

### **/ping**

Comprueba la latencia del jugador con el servidor.



### **/ping {nombre\_jugador}**

La diferencia con el comando anterior es que se comprueba la latencia del jugador que se indica.


El jugador requiere tener el rango dueño.



### **/create warp {nombre\_warp}**

Crea un checkpoint para que un jugador se pueda teletransportar en cualquier momento.

Los jugadores con el rango miembro sólo pueden crear 1 warp mientras que los jugadores con el rango dueño pueden crear 2 warps.



### **/warp {nombre\_warp}**

El jugador se transporta a las coordenadas de ese warp automáticamente.



### **/delete warp {nombre\_warp}**

Borra un warp creado anteriormente.



### **/list warps**


Muestra una lista con el nombre de los warps creados.



```
Warps: home
```

### **/change name color {nombre\_color}**

Cambia el color del nombre del jugador. Sólo el jugador con el rango dueño puede usar el color arcoiris.



```
The color has been changed correctly.
```

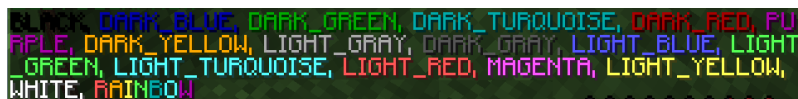


```
[Member] Frederic0
```

### **/change name color {nombre\_color} {nombre\_jugador}**

### **/list colors**

Muestra los colores disponibles.



```
BLACK, DARK_BLUE, DARK_GREEN, DARK_TURQUOISE, DARK_RED, PURPLE, DARK_YELLOW, LIGHT_GRAY, DARK_GRAY, LIGHT_BLUE, LIGHT_GREEN, LIGHT_TURQUOISE, LIGHT_RED, MAGENTA, LIGHT_YELLOW, WHITE, RAINBOW
```

### **/seen**

Muestra la fecha en la que el jugador se ha conectado al servidor por última vez.



```
Last seen: 2020-05-30T19:22:29
```

### **/seen {nombre\_jugador}**

La diferencia con el comando anterior es que muestra la fecha del jugador que se indica.

El jugador requiere tener el rango dueño.

## /chunksLoaded

Muestra la parte que está carga del mapa del servidor.

El jugador requiere tener el rango dueño.

## Errores personalizados al ejecutar un comando

### Comando con pocos argumentos



### Comando con demasiados argumentos



### Comando con argumentos incorrectos



### Si el jugador no está autorizado



## Otras herramientas

Se han usado otro tipo de herramientas para testear la API REST, para realizar un control de las versiones del proyecto, para manejar las dependencias y para poder construir el proyecto a través de comandos.

### Git

Git es un software de control de versiones, pensado en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones. Su propósito es llevar registro de lo cambios en archivos y coordinador el trabajo que varias personas realizan sobre archivos compartidos.

Para poder llevar un mejor control del código y de las versiones, se han de usar ramas. Una rama es un espacio independiente para trabajar en un proyecto.

**R**amas que se han usado:

- Master: Rama de producción.
- Develop: Después de realizar pruebas de integración, se integra en Master.
- Feature-XX: Por cada característica a conseguir se va a crear una. Cuando se ha probado, se integra en la rama Develop. Cada tarea se identifica con un código numérico.

### Comandos básicos de Git

#### Inicializar el directorio actual

```
git init
```

#### Vincular un repositorio remote con el repositorio local

```
git remote add origin url/ssh
```

#### Crear una rama

```
git checkout -b nombre_rama
```

#### Cambiar de rama

```
git checkout nombre_rama
```

#### Añadir todos ficheros nuevos o modificados a la rama actual

```
git checkout nombre_rama
```

#### Confirmar cambios

```
git commit -m "MasterServer-001: Proyecto creado"
```

#### Enviar los cambios a un repositorio remoto

```
git push
```

#### Fusionar otro rama a la rama activa

```
git merge nombre_rama
```



## Maven

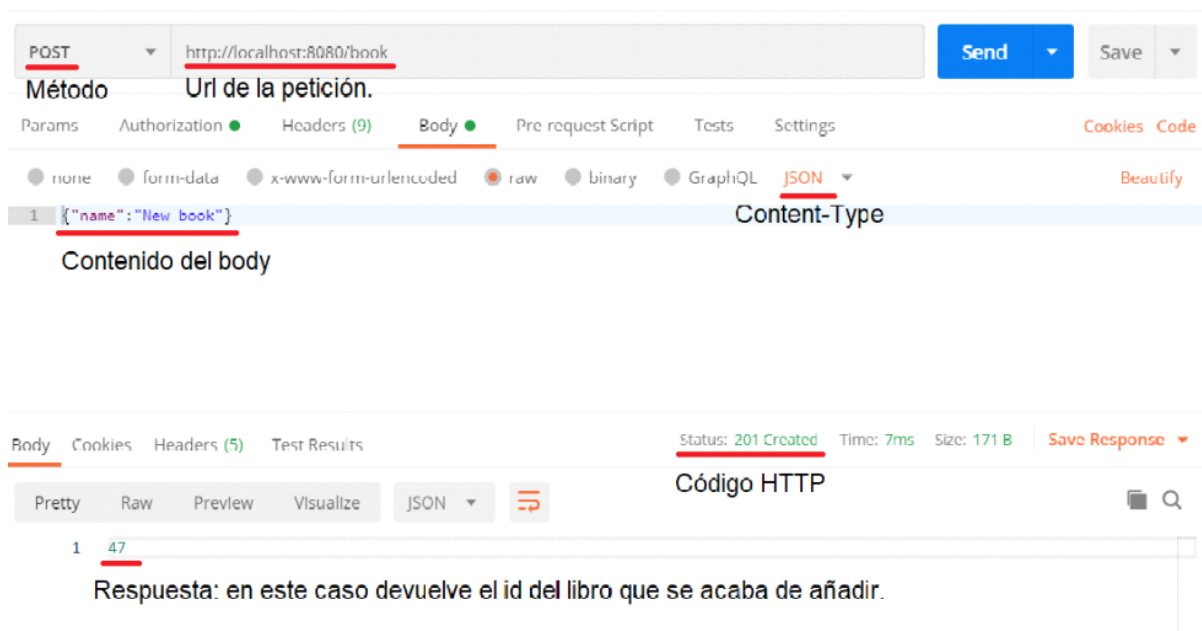
Maven es una herramienta para el manejo de dependencias automáticas. Basa su configuración en un archivo XML llamado POM (Project Object Model), el cuál permite guardar la información de todas las dependencias de la aplicación dentro del elemento XML dependencies.

En el siguiente ejemplo se muestra como incluir la dependencia con el módulo de Spring spring-boot-starter-data-jpa, para que, a la hora de compilar y ejecutar la aplicación, Maven la incluyese en el classpath.

Además Maven permite compilar, generar ejecutables ‘jar’, ejecutar pruebas unitarias y otro tipo de tareas a través de comandos.

## Postman

Postman es una herramienta que permite realizar peticiones HTTP a cualquier API. Es muy útil a la hora de realizar pruebas para comprobar el correcto funcionamiento de la API.



### 1.4.3 Bibliografía

#### Lenguajes de programación

[https://es.wikipedia.org/wiki/Lenguaje\\_de\\_programaci%C3%B3n](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n)

<https://concepto.de/lenguaje-de-programacion/>

[https://www.ecured.cu/Lenguaje\\_interpretado](https://www.ecured.cu/Lenguaje_interpretado)

<https://blog.makeitreal.camp/lenguajes-compilados-e-interpretados/>

[https://es.wikipedia.org/wiki/Java\\_\(lenguaje\\_de\\_programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n))

<https://definicion.mx/java/>

<https://mariocelis.com/java/herencia-y-polimorfismo/>

<https://www.luaces-novo.es/polimorfismo-en-java/>

<https://juanjavierrg.com/java-basico-encapsulamiento/>

<https://es.wikipedia.org/wiki/JavaScript>

## **Frameworks**

<https://es.wikipedia.org/wiki/Framework>

<https://spring.io/>

<https://vuejs.org/>

<https://www.baeldung.com/spring-boot>

<https://www.baeldung.com/spring-tutorial>

<https://www.callicoder.com/spring-boot-file-upload-download-rest-api-example/>

## **Entornos de desarrollo integrado**

[https://es.wikipedia.org/wiki/Entorno\\_de\\_desarrollo\\_integrado](https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado)

[https://www.ecured.cu/IDE\\_de\\_Programaci%C3%B3n](https://www.ecured.cu/IDE_de_Programaci%C3%B3n)

[https://es.wikipedia.org/wiki/Visual\\_Studio\\_Code](https://es.wikipedia.org/wiki/Visual_Studio_Code)

<https://spring.io/tools>

## **API REST**

[https://es.wikipedia.org/wiki/Transferencia\\_de\\_Estado\\_Representacional](https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional)

<https://bbvaopen4u.com/es/actualidad/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos>

## POSTGRESQL

<https://es.wikipedia.org/wiki/PostgreSQL>

<https://www.postgresql.org/>

<https://todopostgresql.com/ventajas-y-desventajas-de-postgresql/>

## MINECRAFT

<https://www.spigotmc.org/wiki/using-the-event-api/>

[https://minecraft-es.gamepedia.com/Minecraft\\_Wiki](https://minecraft-es.gamepedia.com/Minecraft_Wiki)

<https://riptutorial.com/es/bukkit>

[https://bukkit.gamepedia.com/Plugin\\_Tutorial/es#Creando\\_plugin.yml](https://bukkit.gamepedia.com/Plugin_Tutorial/es#Creando_plugin.yml)

## OTRAS HERRAMIENTAS

<https://git-scm.com/docs/gittutorial>

<http://maven.apache.org/guides/index.html>