Resumen **JUnit**

JUnit es un framework de testeo integrado en el IDE que permite testear unidades de código en JAVA, es decir, permite testear clases.Se ejecuta a través de líneas de comando y tiene una pequeña interfaz para seguir el testeo.

Quien escribe el test?
Metodología tradicional -> el equipo de testing
Metodología ágil -> el equipo de desarrollo.
Test Driven Design -> primero se escriben los test y luego se desarrolla y a medida que se implementa, se van pasando los test

Formas de testear: enfoques

- Meter código para testear en nuestro código (asserts)-> embebido
- Aislado-> Crear codigo de test aparte. Me ayuda a sacar dependencias y me doy cuenta que está de más o de qué depende verdaderamente mi clase. Ejemplo Test de Humo
- Uso de frameworks

Tipos de testeo

- Caja blanca: tengo el código que tengo que probar y al lado hago un test. Conozco el código
- Caja negra: solo conozco el nombre de las funciones y los parámetros. No conozco la estructura interna del código.

Cómo se hace un testeo

1. Escribir la clase:El Test va a ser una clase de Test con el mismo nombre + la palabra test.

2. Implementar método setUp() que es el punto de entrada
3. Implementar el método tearDown() es para cerrar el test y volver todo como estaba para que sea repetible
4. Definir los métodos de test
5. Implementar una clase suit
6. Implementar un Runner

Cómo se ve en Java el material de seguimiento

```
package tudai.tdv.junit.test;
@RunWith(Suite.class)
@Suite.SuiteClasses({
        DemoPrecedencia.class,

})
public class TestSuite {//CLASE VACIA}
package tudai.tdv.junit.test;
```

```
public class TestRunner {
	public static void main(String args[]) {
		System.out.println("Test runner -> comienzo del Runner");
		JUnitCore.runClasses(TestSuite.class);
		System.out.println("Test runner -> Fin del Runner");
	}
}

package tudai.tdv.junit.test;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
public class DemoPrecedencia {
		@BeforeClass
		public static void setUpBeforeClass() throws Exception {
			System.out.println("DemoPrecedencia -> BeforeClass ");
		}

		@AfterClass
		public static void tearDownAfterClass() throws Exception {
			System.out.println("DemoPrecedencia -> AfterClass ");
		}

		@Before
		public void setUp() throws Exception {
			System.out.println("DemoPrecedencia -> Before");
		}
		/**
		 * Metodo tearDown para instancias de test* @throws Exception */
		@After
		public void tearDown() throws Exception {
			System.out.println("DemoPrecedencia -> After");
		}

		@Test
		public void testFuncion1() {
			System.out.println("DemoPrecedencia -> testFuncion1");
		}

		@Test
		public void testFuncion2() {
			System.out.println("DemoPrecedencia -> TestFuncion2");
		}
		@Test
		public void testFuncion3() {
			System.out.println("DemoPrecedencia -> TestFuncion3");
		}

}
```

Explicación de cada clase

- **TestSuite**: Agrupa varias clases de prueba para ejecutarlas juntas.
- **TestRunner**: Ejecuta la suite de pruebas.
- **DemoPrecedencia**: Contiene las pruebas individuales que deseas realizar sobre tu código.

Aserciones

JUnit proporciona una serie de métodos de aserción que permiten verificar los resultados esperados. Por ejemplo, `assertEquals(expected, actual)` comprueba si dos valores son iguales. Dependiendo la versión de Junit, es como se escribe el assert.

Assert.equals()/ Assert.true()/ Assert.false

Assert.fail() se utiliza para marcar una prueba como fallida si se llega a un punto del código que no debería alcanzarse. Es útil en pruebas donde, por ejemplo, se espera que se lance una excepción y si no se lanza, se invoca `fail()` para indicar que la prueba no se comportó como se esperaba.

AssertThrows() es un método utilizado en las pruebas unitarias para verificar que una excepción específica se lanza durante la ejecución de un bloque de código. Es parte de JUnit 5, y su propósito es ayudar a garantizar que el comportamiento de un método se alinea con las expectativas, especialmente en el manejo de excepciones.

Anotaciones(etiquetas)

Before class- setUpBeforeClass() : Se configura el entorno de prueba por ejemplo se instancian las variables estáticas
Before- setUp(): Se configura el estado antes de cada prueba. Ejemplo: agrego un socio a la nómina
Test: Se realiza el test propiamente dicho con el assert necesario
After - tearDown(): Se restaura el estado despues de cada prueba, ejemplo remover un socio de la nómina
After class- tearDownAfterClass(): Se restaura el entorno de prueba, por ejemplo: se nullean las instancias de las variables declaradas.

@displayName es el nombre amigable del test (JUnit 5)
@RepeatedTest(2) le pido que ejecute el test 2 veces (JUnit 5)

—----------

Resumen TestNG

TestNG es un marco de pruebas (framework) para Java diseñado para facilitar la creación, organización y ejecución de pruebas automatizadas. Su nombre proviene de "Test Next

Generation" y ofrece características que lo diferencian de JUnit, haciéndolo más flexible y poderoso en ciertos aspectos.

TestNG se diseñó con el objetivo de superar algunas limitaciones de JUnit y proporcionar características adicionales, como soporte para pruebas parametrizadas y la ejecución de pruebas en paralelo.

Características Principales de TestNG:

1. Anotaciones Mejoradas: Similar a JUnit, TestNG utiliza anotaciones para definir métodos de prueba, pero ofrece una mayor variedad de anotaciones, como:
    - `@BeforeSuite`, `@AfterSuite`: Se ejecutan antes y después de todo el conjunto de pruebas.
    - `@BeforeTest`, `@AfterTest`: Se ejecutan antes y después de cada test.
    - `@DataProvider`: Permite pasar datos a las pruebas desde múltiples fuentes.
2. Soporte para Parámetros: TestNG permite pasar parámetros a las pruebas desde un archivo XML, lo que facilita la configuración de pruebas.
3. Grupos de Pruebas: Puedes agrupar pruebas para ejecutarlas juntas. Esto es útil para organizar grandes conjuntos de pruebas.
4. Pruebas Concurrentes: TestNG soporta la ejecución de pruebas en paralelo, lo que puede reducir significativamente el tiempo total de ejecución de las pruebas.
5. Informes Detallados: Proporciona informes de prueba detallados que son fáciles de leer, mostrando resultados, errores y excepciones.
6. Manejo de Excepciones: Permite especificar qué excepciones deberían lanzarse en pruebas, facilitando el manejo de pruebas que esperan fallos.
7. Integración con Herramientas: TestNG se integra fácilmente con herramientas de construcción como Maven y Gradle, y también puede trabajar con IDEs como Eclipse e IntelliJ IDEA.

    @Before y @After (JUnit) vs @BeforeMethod y @AfterMethod (TestNG)

—----------

Test Dinámicos

Los tests dinámicos son pruebas automatizadas que se generan o modifican durante el tiempo de ejecución de una aplicación. A diferencia de los tests estáticos, que son predefinidos y ejecutados tal como están escritos, los tests dinámicos pueden adaptarse a condiciones específicas o a datos de entrada en el momento en que se ejecutan.

Características de los Tests Dinámicos:

1. Generación en Tiempo de Ejecución: Los tests pueden ser creados o modificados sobre la marcha, dependiendo de los resultados de pruebas anteriores o de entradas específicas.
2. Flexibilidad: Permiten una mayor flexibilidad al adaptar las pruebas a situaciones cambiantes o a configuraciones del entorno en tiempo real.

3. Parametrización: A menudo, los tests dinámicos pueden usar diferentes conjuntos de datos sin necesidad de duplicar el código de prueba, lo que facilita la cobertura de múltiples escenarios con un solo test.
4. Ejecución Condicional: Los tests pueden ser ejecutados o saltados según condiciones específicas, lo que puede optimizar el proceso de prueba y hacerlo más eficiente.
5. Uso de Datos Externos: Pueden obtener datos de entrada de fuentes externas como bases de datos, archivos, o servicios web, adaptándose a esos datos durante la ejecución.

La diferencia principal entre las pruebas estáticas y dinámicas en JUnit radica en cómo se generan y ejecutan los casos de prueba.

Pruebas Estáticas

1. **Definición**: Los métodos de prueba estáticos se definen en la clase de prueba con la anotación `@Test`. Suelen estar fijos en el momento de la compilación y se ejecutan todos al mismo tiempo.

Ejemplo:
java
Copiar código

```
@Test

public void testCorreoValido() {
for (Usuario u : usuarios) {
        assertTrue(poseeCorreoValido(u));
    }
}
```

2. En este ejemplo, se ejecuta una sola prueba que verifica todos los usuarios a la vez. Si alguno de ellos falla, el test completo falla.
3. **Ventajas**:
    - Más simple de implementar.
    - Más fácil de entender para pruebas con un número fijo de casos.
4. **Desventajas**:
    - No se pueden agregar o eliminar casos de prueba en tiempo de ejecución.
    - Si uno falla, todos los resultados se agrupan bajo un solo test, lo que puede dificultar la identificación del problema.

Pruebas Dinámicas

1. **Definición**: Usando la anotación `@TestFactory`, las pruebas dinámicas generan casos de prueba en tiempo de ejecución. Cada caso de prueba se crea a partir de datos o condiciones que se determinan en el momento de la ejecución.

Ejemplo:
java

Copiar código

```
@TestFactory
public Collection<DynamicTest> testCorreoValido() {
    List<DynamicTest> dynamicTests = new ArrayList<>();
    for (Usuario u : usuarios) {
        dynamicTests.add(DynamicTest.dynamicTest("Verificando correo
de: " + u.getNombreYApellido(), () -> {
            assertTrue(poseeCorreoValido(u));
        }));
    }
    return dynamicTests;
}
```

2. En este ejemplo, se crea un test separado para cada usuario. Si un usuario tiene un correo inválido, solo esa prueba falla, dejando que las demás continúen.
3. **Ventajas**:
    - Permite una mayor flexibilidad, ya que los casos de prueba se pueden generar dinámicamente basados en datos en tiempo de ejecución.
    - Mejora la legibilidad de los resultados, ya que cada prueba se ejecuta de forma independiente.
    - Es más adecuado para escenarios donde hay una cantidad variable de datos o pruebas.
4. **Desventajas**:
    - Más complejo de implementar.
    - Puede requerir más configuración y comprensión de cómo funciona JUnit 5.

Los tests dinámicos no están disponibles en JUnit 4. Esta funcionalidad fue introducida en JUnit 5 con la adición de la anotación `@TestFactory`, que permite crear tests dinámicos en tiempo de ejecución.

En JUnit 4, todos los tests deben estar definidos de manera estática y no puedes generar tests dinámicamente. Si necesitas realizar pruebas dinámicas en JUnit 4, tendrías que implementar una lógica de iteración en un único método de prueba, pero eso no te dará la misma granularidad y claridad en los resultados que los tests dinámicos de JUnit 5.

Resumen

- **Ejecución**: Estáticas = todas juntas; Dinámicas = cada una por separado.
- **Flexibilidad**: Estáticas = fijas; Dinámicas = generadas en tiempo de ejecución.
- **Resultado**: Estáticas = un solo resultado; Dinámicas = resultados individuales para cada caso.

En resumen, si tienes un conjunto de datos que puede cambiar o que es variable, las pruebas dinámicas son más adecuadas. Para un conjunto fijo de pruebas, las estáticas son más sencillas.

—----------

Parcial 2021 para practicar

Comprobar que no pueden agregarse socios repetidos
Comprobar que no es posible agregar actividades repetidas

Comprobar que el encargado de una actividad es un socio de la nómina

Comprobar que el método toString() de Actividad devuelve una cadena con el formato "<NombreActividad> a cargo de <Apellido>"

Comprobar que el método toString() devuelve una cadena con el formato <Apellido>, <Nombre>

Crear un generador de datos de Socios para comprobar que el método toString() devuelve una cadena en formato <Nombre><Apellido>

Comprobar que al intentar inscribir más usuarios del cupo permitido, se dispara la excepción CupoExcedidoException

Comprobar que al intentar inscribir un socio con una edad no permitida se dispara la excepción EdadInsuficieneException

Crear un generador de datos de Actividades aleatorias para comprobar que el cupo es siempre > 0

Crear un test dinámico que pemita comprobar que todos los inscriptos de una actividad cumplen con la edad mínima

Crear un test dinámico que pemita comprobar que ninguna actividad excede su cupo máximo

Generar un script de prueba que permita parametrizar la cantidad de elementos Socios generados aleatoriamente

Generar un script de prueba que permita parametrizar la cantidad de Socios generados para inscribir a una actividad totalmente aleatoria

—----------

Parcial 2020 con respuestas (nota 8)

1) Indicar V o F. En caso de Falso, justificar

a) TestNG es anterior a Junit. FALSO. TestNG está basado en JUnit OK

b) Mediante un XML puede configurarse y ejecutar una suite de Junit. FALSO. Mediante un XML puede configurarse y ejecutar una suite de TestNG. OK

c) Los datos de un test JUnit pueden cargarse desde un archivo XML. FALSO. Junit no maneja datos en sus tests, sus test son tests de unidad en clases.

d) Un test para JUnit puede ejecutarse en TestNG y viceversa. FALSO. Un test para JUnit puede ejecutarse en TestNG pero no viceversa ya que TestNG abarca tests que no soporta Junit.

e) JUnit está basado en el patrón composite y layers. FALSO, Junit está basado en el patrón composite y command (comandos)

f) Un Mock es una unidad de resfrío. FALSO. Un mock son pruebas unitarias que utilizan objetos simulados

Un **mock** es un tipo de objeto utilizado en pruebas de software, específicamente en pruebas unitarias, que simula el comportamiento de objetos reales. Los mocks permiten verificar interacciones, controlar el comportamiento de las dependencias y aislar la unidad de código que se está probando.

g) TestNG puede ser utilizado en test de Unidad exclusivamente. FALSO.TestNG puede ser utilizado en test de unidad y como test de datos/integracio, funcionales y de sistema.

h) Un Mock de resultado realiza la comparación del resultado de salida con un objeto de idénticas características, previamente testeado. FALSO. Un mock es una copia de un objeto; el resultado del test se ralizara utilizando este mock.

En este contexto, el mock simplemente simula la respuesta esperada cuando se invoca un método. La comparación se realiza en el test mediante aserciones, donde el resultado obtenido se compara con el resultado esperado definido por el programador.

2) Dado un sistema de reservas de vuelos que aún se encuentra en desarrollo, se desea comenzar con la escritura de los test. Resolver indicando si es posible solucionarlo en JUnit, TestNG o ambos, justificando la respuesta

a) Escribir un test que partiendo de que un vuelo con una capacidad de 200 asientos, está completo, no se pueden asignar más asientos y en tal caso se arroje fullFlightException.

Es posible resolver este caso tanto en **JUnit** como en **TestNG**.

JUnit

```
public class VueloTest {
        private int capacidad;
        private int asientosAsignados;
```

```
        public Vuelo(int capacidad) {
                this.capacidad = capacidad;
                this.asientosAsignados = 0;
        }

        public void asignarAsiento(int asiento) throws FullFlightException {
                if (asientosAsignados >= capacidad) {
                throw new FullFlightException("El vuelo está completo.");
        }
        asientosAsignados++; }

@Test
public void testNoSePuedenAsignarMasAsientos() {
        Vuelo vuelo = new Vuelo(200); // Vuelo con 200 asientos
        vuelo.asignarAsiento(200); // Asignar todos los asientos
        // Intentar asignar otro asiento
        assertThrows(FullFlightException.class, () -> {
        vuelo.asignarAsiento(201); // Esto debería lanzar la excepción });
        }
}
```

b) Escribir un test para el agregado de pasajeros que se ejecute exactamente 25 veces. (nota, No utilizar iteraciones)

Para escribir un test que agregue pasajeros exactamente 25 veces sin usar iteraciones, puedes utilizar tanto JUnit como TestNG. Ambas bibliotecas permiten crear múltiples métodos de prueba, y puedes definir uno para cada caso que deseas probar.

JUnit

```
public class VueloTest {
        @RepeatedTest(25)
        public void testAgregarPasajero() {
        // Lógica para agregar un pasajero// Verifica que se haya agregado correctamente
        }}
```

TestNG

```
public class VueloTest {
        @Test(invocationCount = 25)
        public void testAgregarPasajero()
{ // Lógica para agregar un pasajero // Verifica que se haya agregado correctamente } }
```

c) Por cambios en los requerimientos, los test de carga de pasajeros deben ejecutarse 200 veces. ¿Qué cambios o acciones se deberían tomar?

Se debera cambiar unicamente el @RepeatedTest(200) o @Test(invocationCount = 200)

d) Escribir un test para una función “cerrarVuelo” que retorna verdadero sólo si están todos los asientos asignados

-Se puede implementar utilizando Junit o TestNG

```
public class VueloTest {
    private Vuelo vuelo;
    @Before
    public void setUp() {
        vuelo = new Vuelo(200); // Crea un vuelo con 200 asientos }
    @Test
    public void testCerrarVueloConTodosLosAsientosAsignados() { // Asignar todos los asientos
        for (int i = 0; i < 200; i++) {
        vuelo.asignarAsiento(i); } // Verifica que cerrar el vuelo retorna verdadero
        assertTrue(vuelo.cerrarVuelo()); }
```

El método cerrarVuelo debería devolver true en el caso de que todos los asientos están asignados

e) Suponer que se realizarán pruebas con información de 20 pasajeros. Implementar los esquemas para carga que creas posible

JUnit:

- Carga directa en el test (iterando o usando @RepeatedTest)
- @ParameterizedTest

Cómo funciona @ParameterizedTest**:**

1. **Definición del Método de Prueba**: Se define el método de prueba anotado con `@ParameterizedTest`.
2. **Fuente de Datos**: Se especifica cómo se obtendrán los datos de prueba. Esto puede hacerse utilizando una fuente de datos como `@MethodSource`, `@CsvSource`, `@ValueSource`, etc.
3. **Ejecución**: JUnit ejecuta el método de prueba una vez por cada conjunto de datos proporcionado.
-

TestNG

- Carga directa en el test
- @DataProvider

Cómo funciona:

1. **Definición del Método DataProvider**: Se define un método que devuelve un arreglo de objetos. Este método se anota con `@DataProvider`.

2. **Uso del DataProvider**: En el método de prueba, se utiliza la anotación `@Test` junto con el atributo `dataProvider` para especificar qué método se utilizará como proveedor de datos.
3. **Ejecución**: TestNG ejecuta el método de prueba una vez por cada conjunto de datos devuelto por el `DataProvider`.
-

Existen 2 formas de hacer este test, de manera dinamica utilizando @DataProvider en TestNG o de manera estática en Junit. Por cuestiones de practicidad, conviene obviamente utilizar el @dataProvider que aporta TestNG, donde cargaremos los 20 datos de manera dinámica.

—----------

API- Application Program Interface

Hasta acá veníamos testeando unidades de código, ahora se testeará más de un componente y cómo se integran a través de las API. La API va a posibilitar a terceros acceder a mi lógica de negocio a través de componentes de mi sistema con determinadas restricciones y vistas.

Tipos de API basadas en sistemas WEB

SOAP: intercambio de mensajes a mediante objetos XML

RPC: Se usaba mucho y es más para comunicación interna. Enfoque Streaming, el cliente envía datos al servidor hasta que alguien cierre conexión y el servidor va dando respuestas esporádicas. Ejemplo: cámara de video que envía la grabación al servidor. Ejemplo: Netflix (al revés). Se clickea en una peli y la info se envía hasta que el cliente pausa o cierra la peli. Otro enfoque: Duplex: ambos envían datos de manera indiscriminada hasta que alguno cierra la conexión.

WebSocket: comunicación bidireccional en formato JSON.

REST: Se le envían peticiones al servidor y este devuelve los resultados

*Existen otras API que no están basadas en web. Ejemplo de software para desarrollar juego. Mediante una API. El creador del juego define una API y como programador podrías sumar funcionalidades a partir de esa API (librería o script). Hoy esto se conoce más como SDK, por ejemplo para instalar Visual Studio o un Plugin.

REST: Estilo arquitectónico y enfoque de comunicación utilizado en el desarrollo de Web Services y APIs. Son simples, escalables y flexibles. Se usan en app, en web, microservicios.

Características:

- Stateless: No tienen estado, se pierde. Un llamado no tiene nada que ver con otro, si lo necesito mantener el estado ya se trabaja con sesiones y un backend que gestione los datos para eso.
- Basada en recursos: cada recurso es identificado por su URI y accedido mediante métodos estándar HTTP. La URI se compone de:
    - Esquema: indica el protocolo utilizado (http, https, ftp,etc)
    - Host: dominio
    - Puerto (opcional)
    - Ruta: ubicación del recurso en el servidor. Esto generalmente incluye nombres de recursos y puede tener parámetros.
    - Consulta (opcional): parámetros que proporcionan info adicional
- Interfaz uniforme: métodos y formatos estandarizados
- Cacheable: lo maneja el servidor. Permite tener respuestas comunes pre guardadas para no tener que consultarlo siempre.

Verbos:

- GET/HEAD: métodos de consulta
- PUT/POST/PATCH: generan cambios sobre el recurso que esté accediendo.
    - PUT se utiliza para actualizar un recurso completo o para crear uno nuevo en una ubicación específica si no existe. Cuando se envía una solicitud PUT, el cliente debe enviar la representación completa del recurso.
    - POST se utiliza para generar nuevos recursos
    - PATCH se utiliza para aplicar cambios parciales a un recurso existente.
- TRACE/OPTIONS/DELETE:
    - TRACE: se utiliza para realizar un "loop-back" a través del servidor. Esto permite al cliente ver lo que el servidor ha recibido en la solicitud. Es útil para la depuración y diagnóstico de problemas en la comunicación HTTP.
    - OPTIONS: se utiliza para describir las opciones de comunicación disponibles para un recurso específico. Esto incluye los métodos HTTP que son soportados por el recurso (GET/PUT,ETC) así como otras opciones relacionadas
    - DELETE: para eliminar recursos.

<u>Relacion entre DyV y API</u>

Se va a crear una especificación de qué contrato va a haber entre cliente y servidor para intercambiar datos, es decir, de qué forma van a interactuar ambos.

Generación de documentos formales que se puedan consultar en cualquier momento que sirvan de base de cómo consumir la API.

<u>OpenApi 3:</u> Es una especificación para describir APIs RESTful de manera estandarizada. Su objetivo es permitir que tanto humanos como máquinas puedan entender las capacidades de un servicio web sin necesidad de acceder a su código fuente.

Lo interesante de OpenAPI 3 es que a partir de la documentación que se va a generar se puede hacer un pre testeo y se va generando el código de la misma api (cascarón), que luego hay que completar con la lógica del negocio.. OpenAPi nació como Swagger.

La estructura de cómo describir una API consta de tres partes:

- Meta Información:Información general sobre la API (título, versión, descripción, términos de servicio, contacto, etc.).
- Path de los ítems, se definen los endpoints disponibles, junto con sus métodos HTTP (GET, POST, etc.) y sus parámetros.
    - Parámetros: Se definen entre {}.
        1. Están los de ruta /socios/123;
        2. Consulta(flexibles) /socios?idSocio=123; Para concatenar varios usar &
        3. Encabezado (no visible en la consulta) X-idSocio:123 y;
        4. De cookie idSocio=123 (no visible).

      Se pueden definir mínimos, máximos, valores por defecto, requeridos, filtros y modificadores

      Los "modificadores" en los paths se refieren a los métodos HTTP que se utilizan para interactuar con los recursos definidos en la API. Estos métodos indican la acción que se desea realizar sobre un recurso específico (GET/PUT,POST,DELETE,ETC)

    - Cuerpo de consulta: cuando hago envíos (PUT,POST Y PATCH) envío un body-> requestBody.
    - Respuestas
- Componentes reusables: Elementos reutilizables, como esquemas de datos (modelos), parámetros, respuestas, y más.

  (si para varios recursos necesito el objeto persona, lo defino una sola vez y lo reutilizo.)

<u>Ejemplo de documentación subida por la cátedra</u>

```
openapi: 3.0.0
info:
  title: Sports Activities API
  version: 1.1.0
  description: API for managing users, sports activities, and enrollments

servers:
  - url: http://localhost:8080
```

```
paths:
  /users:
    get:
      summary: Get all users
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'
    post:
      summary: Create a new user
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/UserInput'
      responses:
        '200':
          description: User created successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'

  /users/{id}:
    get:
      summary: Get a specific user
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
        '404':
```

```
        description: User not found

/users/{id}/age:
  get:
    summary: Get the age of a specific user
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: integer
    responses:
      '200':
        description: Successful response
        content:
          application/json:
            schema:
              type: object
              properties:
                age:
                  type: integer
      '404':
        description: User not found

/users/{id}/activities:
  get:
    summary: Get activities for a specific user
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: integer
    responses:
      '200':
        description: Successful response
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Activity'

/activities:
  get:
    summary: Get all activities
```

```
    responses:
      '200':
        description: Successful response
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Activity'
  post:
    summary: Create a new activity
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ActivityInput'
    responses:
      '200':
        description: Activity created successfully
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Activity'

/activities/{id}:
  get:
    summary: Get a specific activity
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: integer
    responses:
      '200':
        description: Successful response
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Activity'
      '404':
        description: Activity not found
  delete:
    summary: Delete a specific activity
    parameters:
```

```
      - name: id
        in: path
        required: true
        schema:
          type: integer
    responses:
      '204':
        description: Activity deleted successfully
      '404':
        description: Activity not found

/activities/{id}/users:
  get:
    summary: Get users enrolled in a specific activity
    parameters:
      - name: id
        in: path
        required: true
        schema:
          type: integer
    responses:
      '200':
        description: Successful response
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/User'

/enroll:
  post:
    summary: Enroll a user in an activity
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/EnrollmentInput'
    responses:
      '200':
        description: Enrollment successful
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Enrollment'
```

```
        '400':
          description: Activity has reached maximum enrollment

components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        gender:
          type: string
        birth_date:
          type: string
          format: date

    UserInput:
      type: object
      properties:
        name:
          type: string
        gender:
          type: string
        birth_date:
          type: string
          format: date

    Activity:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        max_enrollees:
          type: integer

    ActivityInput:
      type: object
      properties:
        name:
          type: string
        max_enrollees:
          type: integer
```

```
Enrollment:
  type: object
  properties:
    id:
      type: integer
    user_id:
      type: integer
    activity_id:
      type: integer

EnrollmentInput:
  type: object
  properties:
    user_id:
      type: integer
    activity_id:
      type: integer
```