**Hilla + React + SringBoot**

**Getting Started**



**Hilla**: es un framework completo que ofrece seguridad de tipos de un extremo a otro mediante la generación de código. El framework se compone de Java **Spring Boot** en el backend y **Typecript** en el frontend.

En este tutorial vamos a cubrir aspectos básicos: configuración del entorno de desarrollo, y agregar objetos de una clase con sus operaciones CRUD (Create, Read, Update, Delete).

Hilla requiere Java y NodeJS. Para Java, desde 17 en adelante y para NodeJS desde v18 en adelante.

Primero, cree una nueva aplicación Hilla e iníciela. En la carga inicial, puede llevar algún tiempo descargar todas las dependencias.

# generate project  
npx @hilla/cli init --react hilla-todo  
  
# start dev server  
cd hilla-todo/  
./mvnw

Nota: si acaso arroja un error, verificar que la versión de java verificada en línea de comandos es la misma que aparece en el pom.xml en la sección properties>java.version

A continuación, agregaremos Spring started for JPA y una base de datos Java en memoria que podemos usar para el desarrollo. También incluiremos Lombok para facilitar el desarrollo.

<dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <!-- in-memory Java Database -->

        <!-- https://mvnrepository.com/artifact/org.hsqldb/hsqldb -->

        <dependency>

            <groupId>org.hsqldb</groupId>

            <artifactId>hsqldb</artifactId>

            <version>2.7.2</version>

        </dependency>

        <!-- Lombok because no one wants to code Java without it -->

        <dependency>

            <groupId>org.projectlombok</groupId>

            <artifactId>lombok</artifactId>

            <version>1.18.24</version>

            <scope>provided</scope>

        </dependency>

Back-End

Una vez completada nuestra configuración, podemos usar Lombok para crear rápidamente nuestra primera clase, en la ruta siguiente: // src/main/java/com/example/application/model/Todo.java

package com.example.application.model;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.Id;

import jakarta.validation.constraints.NotBlank;

import lombok.Getter;

import lombok.Setter;

@Entity

public class Todo {

  @Getter

  @Setter

  @Id

  @GeneratedValue

  private Integer id;

  @Setter

  @Getter

  private boolean done = false;

  @Getter

  @Setter

  @NotBlank

  private String task;

  public Todo() {}

  public Todo(String task) {

    this.task = task;

  }

}

También necesitaremos un repositorio para nuestros objetos.

// src/main/java/com/example/application/repository/TodoRepository.java

package com.example.application.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.application.model.Todo;

public interface TodoRepository extends JpaRepository<Todo, Integer> {

}

Finalmente, necesitamos un punto final para manejar la creación, actualización y eliminación. El método ***save*** se encargará tanto de la creación como de la actualización de nuestro recurso.

// src/main/java/com/example/application/endpoint/TodoEndpoint.java

package com.example.application.endpoint;

import java.util.List;

import com.example.application.model.Todo;

import com.example.application.repository.TodoRepository;

import com.vaadin.flow.server.auth.AnonymousAllowed;

import dev.hilla.Endpoint;

import dev.hilla.Nonnull;

@Endpoint

@AnonymousAllowed

public class TodoEndpoint {

    private TodoRepository repository;

    public TodoEndpoint(TodoRepository repository) {

        this.repository = repository;

    }

    public @Nonnull List<@Nonnull Todo> findAll() {

        return repository.findAll();

    }

    public Todo save(Todo todo) {

        return repository.save(todo);

    }

    public Integer delete(Todo todo) {

        repository.delete(todo);

        return todo.getId();

    }

}

Front-End

Ahora tomemos un segundo para ver lo que Hilla está haciendo por nosotros. Ejecute la aplicación una vez más (mvnw) y observe que Hilla genera automáticamente el código de interfaz. El código se envía a la interfaz/generado/..

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

De particular interés para nosotros son los dos archivos siguientes:

* Todo.ts: proporciona el tipo del objeto Todo que definimos anteriormente.
* TodoEndpoint.ts: proporciona métodos para llamar al backend.

Vale la pena señalar aquí que el uso de este código generado no se limita a Hilla. Si su aplicación tiene un cliente React-Native, el código podría compartirse con su aplicación React-Native como una forma de mantener una única fuente de verdad para llamar a su backend.

Antes de comenzar a escribir cualquier código de interfaz, instalaremos **formik**.

npm install formik --save

Finalmente estamos listos para escribir una vista CRUD rápida para nuestros objetos Todo. Usaremos el **TodoEndpoint** generado automáticamente y los métodos que definimos anteriormente: findAll(), save() y delete().

// frontend/views/TodoView.tsx

import { EndpointValidationError } from '@hilla/frontend';

import { Button } from '@hilla/react-components/Button.js';

import { Checkbox } from '@hilla/react-components/Checkbox.js';

import { TextField } from '@hilla/react-components/TextField.js';

import { FormikErrors, useFormik } from 'formik';

import Todo from 'Frontend/generated/com/example/application/model/Todo';

import { TodoEndpoint } from 'Frontend/generated/endpoints';

import { useEffect, useState } from 'react';

export default function TodoView() {

  const empty: Todo = { task: '', done: false };

  const [todos, setTodos] = useState(Array<Todo>());

  useEffect(() => {

    (async () => {

      setTodos(await TodoEndpoint.findAll());

    })();

    return () => {};

  }, []);

  const createForm = useFormik({

    initialValues: empty,

    onSubmit: async (value: Todo, { setSubmitting, setErrors }) => {

      try {

        const saved = (await TodoEndpoint.save(value)) ?? value;

        setTodos([...todos, saved]);

        createForm.resetForm();

      } catch (e: unknown) {

        if (e instanceof EndpointValidationError) {

          const errors: FormikErrors<Todo> = {};

          for (const error of e.validationErrorData) {

            if (typeof error.parameterName === 'string' && error.parameterName in empty) {

              const key = error.parameterName as string & keyof Todo;

              errors[key] = error.message;

            }

          }

          setErrors(errors);

        }

      } finally {

        setSubmitting(false);

      }

    },

  });

  async function changeStatus(todo: Todo, done: Todo['done']) {

    const newTodo = { ...todo, done: done };

    const saved = (await TodoEndpoint.save(newTodo)) ?? newTodo;

    setTodos(todos.map((item) => (item.id === todo.id ? saved : item)));

  }

  async function updateTask(todo: Todo, task: Todo['task']) {

    if (todo.task == task) return;

    const newTodo = { ...todo, task };

    const saved = (await TodoEndpoint.save(newTodo)) ?? newTodo;

    setTodos(todos.map((item) => (item.id === todo.id ? saved : item)));

  }

  async function deleteTodo(todo: Todo) {

    const deletedTodoId = await TodoEndpoint.delete(todo);

    if (deletedTodoId) {

      setTodos(todos.filter((t) => t.id != deletedTodoId));

    }

  }

  return (

    <>

      <div className="m-m flex items-baseline gap-m">

        <TextField

          name="task"

          label="Task"

          value={createForm.values.task}

          onChange={createForm.handleChange}

          onBlur={createForm.handleChange}

        />

        <Button theme="primary" disabled={createForm.isSubmitting} onClick={createForm.submitForm}>

          Add

        </Button>

      </div>

      <div className="m-m flex flex-col items-stretch gap-s">

        {todos.map((todo) => (

          <div key={todo.id}>

            <Checkbox

              name="done"

              checked={todo.done}

              onCheckedChanged={({ detail: { value } }) => changeStatus(todo, value)}

            />

            <TextField

              name="task"

              value={todo.task}

              onBlur={(e: any) => updateTask(todo, e.target.value)}

            />

            <Button onClick={() => deleteTodo(todo)}>X</Button>

          </div>

        ))}

      </div>

    </>

  );

}

Por último, antes de que podamos usar nuestra vista, necesitaremos agregarla a MainLayout.tsx

<nav>

            <NavLink className={navLinkClasses} to="/">

              Hello World

            </NavLink>

            <NavLink className={navLinkClasses} to="/todo">

              Todo

            </NavLink>

            <NavLink className={navLinkClasses} to="/about">

              About

            </NavLink>

          </nav>

..y también en routes.tsx

import TodoView from './views/todo/TodoView';

..y en la sección **children** del arreglo routing:

{ path: '/todo', element: <TodoView />, handle: { title: 'Todo' } },

Después de ejecutar la aplicación y navegar a la página Todos, nos queda algo como esto.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

URL del tutorial

https://medium.com/@robert83sutherland/getting-started-hilla-react-c78c9dba38d8

Apendice

Una vez terminado nuestro desarrollo, podemos cambiar la base de datos de HSQL por una base postgreSQL. El procedimiento es muy sencillo: en primer lugar, debemos quitar o comentar del archivo pom.xml la dependencia de HSQL y agregar la dependencia de PostgreSQL.

<!--

        <dependency>

            <groupId>org.hsqldb</groupId>

            <artifactId>hsqldb</artifactId>

            <version>2.7.2</version>

        </dependency>

        -->

        <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->

        <dependency>

            <groupId>org.postgresql</groupId>

            <artifactId>postgresql</artifactId>

            <version>42.7.3</version>

        </dependency>

En segundo lugar, agregamos los parámetros de configuración en el archivo de configuración de nuestro proyecto.

//src/resources/application.properties

spring.datasource.url=jdbc:postgresql://localhost:5432/todo

spring.datasource.username=postgres

spring.datasource.password=xxxxxxxx

spring.jpa.hibernate.ddl-auto=create

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

Conviene describir el siguiente propiedad y sus posibles valores

**spring.jpa.hibernate.ddl-auto**=<**create** | **create-drop** | **update** | **validate** | **none**>

* create: al iniciar la aplicación, elimine todas las tablas administradas por Hibernate y luego las crea desde cero.
* create-drop: al iniciar la aplicación, crea todas las tablas administradas por Hibernate. Al apagarla, remueve todos.
* update: al iniciar la aplicación, actualiza las tablas existentes para que coincidan con el esquema que espera Hibernate si es necesario.
* validate: al iniciar la aplicación, verifique que las tablas existentes coincidan con el esquema que espera Hibernate y genere una excepción si no coinciden.
* none: no realiza ninguna gestión automática de esquemas.