TSVV-K regular VC#4 - ModCR status

Development status - Fortran DevOps and JSONFortranSerialisation

June 19, 2026 | Michael Gordon, Dr. Dmitriy Borodin, Pieter Willem Groen | IFN-1

Overview

Fortran DevOps

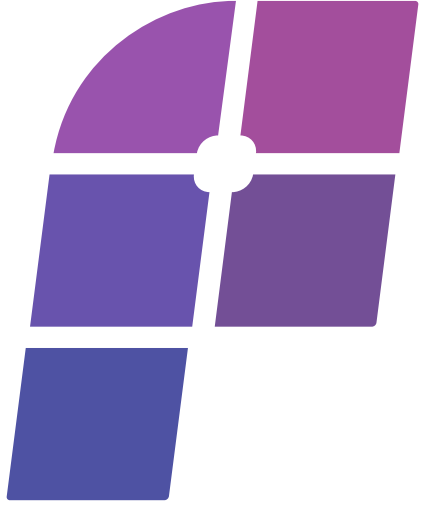
- Fortran Package Manager (FPM)
- Deployment with FPM in HPC
- Projects employing FPM

JSONFortranSerialisation

- Introduction
- Motivation
- How to use JSONFortranSerialisation

Fortran-DevOps

Fortran Package Manager



Source: ”https://fpm.fortran-lang.org/_images/fpm-logo-color.svg”

Fortran Package Manager (FPM)

Overview of Core Features

Quote from fpm.fortran-lang.org[1]

*Fortran Package Manager (fpm) is a **package manager** and **build system** for Fortran. Its key goal is to **improve the user experience** of Fortran programmers. It does so by making it **easier to build your Fortran program or library**, **run the executables, tests**, and examples, and **distribute it as a dependency to other Fortran projects**. Fpm's user interface is modeled after Rust's Cargo. Its long term vision is to nurture and grow the ecosystem of modern Fortran applications and libraries.*

Easy to operate Fortran development tool

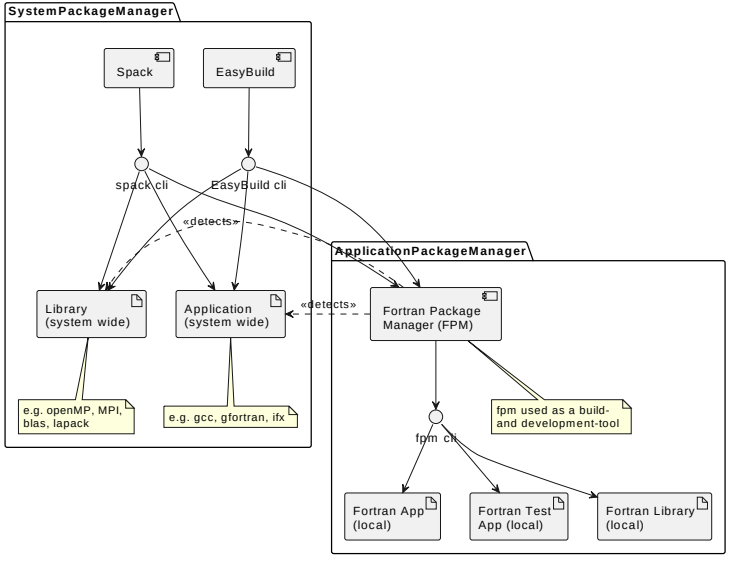
Spend less time configuring your project.
Spend more time doing physics!

[1] https://fpm.fortran-lang.org/design/index.html

FPM vs CMake

Feature	Fortran Package Manager (fpm)	CMake
Primary Focus	Fortran-specific	Multi-language
Config File	`fpm.toml`	`CMakeLists.txt`
Dependencies	Git-based	FetchContent
Build Backend	Ninja or Make.	Generator Based (Ninja, Make, Xcode, Visual Studio Solutions etc)
Learning Curve	**Low.** Intuitive CLI commands (`fpm build`, `fpm test`).	**High.** Steep learning curve.
Adoption	**Growing.** The standard for new, pure-Fortran scientific projects (e.g., Fortran-stdlib).	**Ubiquitous.** The industry standard for C/C++ and HPC. Most legacy Fortran projects still use it.

Deployment with FPM in HPC



JÜLICH Forschungszentrum

Projects employing FPM

Popular Open Source Projects

- jsonfortran
- testdrive
- fortran stdlib

Projects in the scope of ModCR development

- ModCR
- JSONFortranSerialisation
- fortran_solvers

JSONFortranSerialisation

a FPM project in the scope of ModCR development

What is Serialisation?

A quote from isocpp.org[1]

*It lets you take an **object or group of objects**, **put them on a disk** or send them through a wire or wireless transport mechanism, then later, perhaps on another computer, **reverse the process**: **resurrect the original object(s)**. The basic mechanisms are to flatten object(s) into a one-dimensional stream of bits, and to turn that stream of bits back into the original object(s).*

Key Feature in many object oriented serialisation libraries

- support complex data types (derived types in Fortran)
- maps structure of data type to structure of file
  - IO routines get derived from data type definition
  - automatically detect if files does not match to data type

[1] https://isocpp.org/wiki/faq/serialization#serialize-overview

Key Advantages of JSON for serialisation

- native serialisation file format of the JavaScript language
- widespread adoption into many programming languages
- human readible format for machines
  - mapping of keys to values (duplicate keys are forbidden!)
  - minimalistic
    - string, int, real, boolean as primitive types
    - object and list as complex types
- reduce maintenance for IO code
  - data definition is coupled to IO logic
  - reduced need for additional structures and conversions
  - less possibility for mistakes
  - validate semantic correctness with schemata

Difficulties of using JSON in Fortran with previous approaches

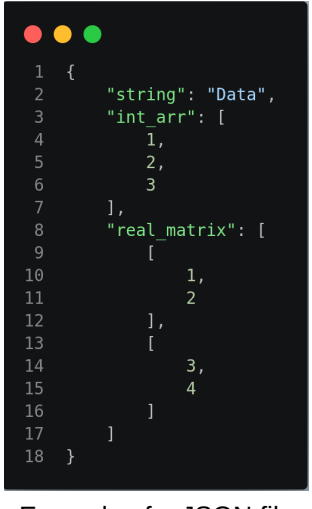
Motivation

- ModCR uses complex nested JSON input
- old ModCR routines are long, complicated, hard to maintain

Easier Access with JSONFortranSerialisation

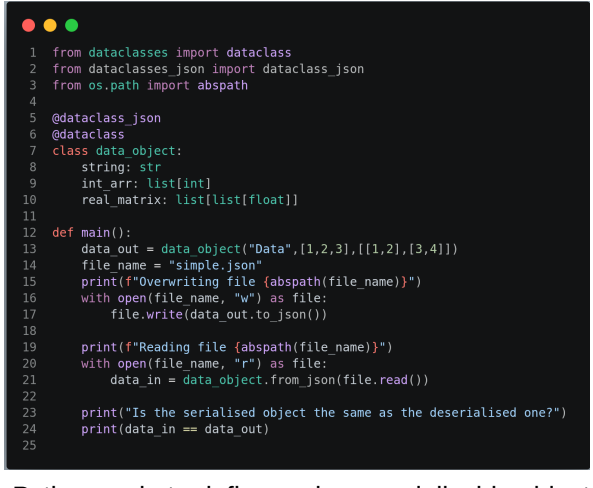
- built on top of jsonfortran by Jacob Williams (backend)
- JSONFortranSerialisation makes the json generation and reading more accessible

Inspiration from other programming languages



```
{
    "string": "Data",
    "int_arr": [
        1,
        2,
        3
    ],
    "real_matrix": [
        [
            1,
            2
        ],
        [
            3,
            4
        ]
    ]
}
```

Example of a JSON file



```python
from dataclasses import dataclass
from dataclasses_json import dataclass_json
from os.path import abspath

@dataclass_json
@dataclass
class data_object:
    string: str
    int_arr: list[int]
    real_matrix: list[list[float]]

def main():
    data_out = data_object("Data",[1,2,3],[[1,2],[3,4]])
    file_name = "simple.json"
    print(f"Overwriting file {abspath(file_name)}")
    with open(file_name, "w") as file:
        file.write(data_out.to_json())

    print(f"Reading file {abspath(file_name)}")
    with open(file_name, "r") as file:
        data_in = data_object.from_json(file.read())

    print("Is the serialised object the same as the deserialised one?")
    print(data_in == data_out)

```

Python code to define and use serialisable object

Intuitive IO in Fortran

```fortran
module data_module
    use JSer_member_type, only: StringMember_t, &
    IntegerArrayMember_t, RealMatrixMember_t
    use JSer_json_serialisable, only: JSONSerialisable_t
    use JSer_data_kinds, only: rk
    implicit none

    type, extends(JSONSerialisable_t), public :: Data_t
        type(StringMember_t) :: string
        type(IntegerArrayMember_t) :: int_arr
        type(RealMatrixMember_t) :: real_matrix

    contains
        procedure :: user_init => user_init_Data_t
        procedure :: set_default_values => set_default_values_Data_t
        procedure :: equals => equals_Data_t
    end type

contains

    subroutine user_init_Data_t(this)
        CLASS(Data_t), INTENT(INOUT) :: this
        call this%register_member(this%string, "string")
        call this%register_member(this%int_arr, "int_arr")
        call this%register_member(this%real_matrix, "real_matrix")
    end subroutine
```

Definition of serialisable object

Intuitive IO in Fortran

```fortran
subroutine main()
        ! this is a simple routine that demonstrates the possibilities of this library

        ! in this routine a json controller object is used to create a json file
        ! and to serialise to it a an object of type JSONSerialisable_t

        ! then the controller is used to reopen the file for reading in order
        ! to deserialise the object
        type(JSONController_t) :: json_controller
        type(Data_t) :: data_in, data_out

        call data_in%init()
        ! set default values after initialization
        call data_in%set_default_values()

        call json_controller%openFile("example/simple_example/simple.json", overwrite=.true.)
        call data_in%to_json_controller(json_controller, "data_object")
        call json_controller%closeFile(.true.)

        call data_out%init()
        ! don't set default values after initialization
        ! values will be retrieved from json_controller
        call json_controller%openFile("example/simple_example/simple.json", overwrite=.false.)
        call data_out%from_json_controller(json_controller, "data_object")
        call json_controller%closeFile(.true.)

        write (*, *) "Is the serialised object the same as the deserialised one?"
        write (*, *) data_in%equals(data_out)
    end subroutine
```

Driver Program code

JSONFortranSerialisation is an example of an FPM project

These FPM features are used

- all dependencies are installed with fpm
- code is compiled against ifx and gfortran
- Docker container setup for different compilers and for native fpm and spack
- example code, library code and test code is discovered, compiled and run

Usage of FPM in ModCR

- DVB_JW_play branch of ModCR used FPM
- it has JSONFortranSerialisation and fortran_solvers as a dependency

Applications in the scope of TSVV-K

- STYX 2.0
- COLRAD / ModCR

Many thanks for your attention!!!