H2020-EINFRA-2017

EINFRA-21-2017 - Platform-driven e-infrastructure innovation

DARE [777413] "Delivering Agile Research Excellence on European e-Infrastructures"

D3.6 DARE API II

Project Reference No	777413 — DARE — H2020-EINFRA-2017 / EINFRA-21-2017
Deliverable	D3.6 DARE API II
Work package	WP3: Large-scale Lineage and Process Management
Tasks involved	T3.3, T3.4
Type	DEM: Demonstrator, pilot, prototype
Dissemination Level	PU = Public
Due Date	31/12/2020
Submission Date	30/12/2020
Status	Draft
Editor(s)	Sissy Themeli (NCSRD), Iraklis Klampanos (NCSRD)
Contributor(s)	Sissy Themeli (NCSRD), Iraklis Klampanos (NCSRD), Alessandro Spinuso (KNMI), André Gemünd (Fraunhofer)
Reviewer(s)	Alessandro Spinuso (KNMI)
Document description	Implementation of the DARE software API.

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
1	09/09/2020	Initial version	Sissy Themeli (NCSRD)
2	4/12/2020	Major updates throughout	Iraklis Klampanos (NCSRD)
3	11/12/2020	DARE Login	André Gemünd (Fraunhofer)
4	12/12/2020	Internal review	Alessandro Spinuso (KNMI)
5	22/12/2020	Final version	Iraklis Klampanos (NCSRD)

Executive Summary

This deliverable is an incremental update of D3.5 and it reports the final state of the DARE platform's API. The DARE platform follows a "microservices" architectural approach, with multiple decoupled components exposing and communicating via RESTful APIs. In the following sections we provide a description of the current and final status of the overall DARE API, providing links to online documentation as appropriate.

Table of Contents

List of Terms and Abbreviations

Abbreviation	Definition
REST	Representational State Transfer
API	Application Program Interface
EPOS	European Plate Observing System
ESGF	Earth System Grid Federation
IS-ENES	Infrastructure for the European Network for Earth System
EOSC	European Open Science Cloud

1 Introduction

This deliverable reports the current version of the DARE platform's API. The DARE platform [1] follows the Microservices Architecture and consists of multiple individual components. Each component exposes its functionality via a RESTful API. In the following sections a thorough description of the DARE API is provided. The most recent documentation of the DARE platform's API is provided online as part of the DARE platform's dedicated website: https://project-dare.gitlab.io/dare-platform/api/

1.1 Approach and Relationship with other Work Packages and Deliverables

In accordance with the Architecture principles of D2.2 (which builds on D2.1 and ID2.2 [2]), the DARE API represents a modular set of APIs with the intention to move towards a common knowledge-base via federating over existing as well as new catalogues and registries. The DARE API consists of APIs to work with registries of workflows and processing elements as well as to interact with the WaaS, the shared file system, the provenance system, etc. The DARE API provided by this deliverable is the main interface of the DARE platform with the outside world, therefore also providing functionality to WP6 and WP7. It is being affected by work in WP2, as well as in WP3 and in WP4.

1.2 Methodology and Structure of the Deliverable

This Deliverable describes the individual RESTful APIs that consist of the DARE API. The components included are: the Dispel4py Information Registry, the Provenance API, the DARE Execution API, the CWL Workflow Registry and the Authentication API.

2 DARE API

The complete and latest version of the API documentation as well as deployment / installation instructions can be found in the DARE platform dedicated "microsite", as the gitlab source code repository, at: https://project-dare.gitlab.io/dare-platform/.

2.1 Dispel4py Information Registry

The main concepts of the dispel4py library are managed via the Dispel4py Information Registry [4], which is a RESTful Web Service implemented in Python Django. This component enables the efficient storage and retrieval of implemented Processing Elements (PEs), thus promoting workflow reusability. Users can create their own workspaces and register the Processing Elements (PEs) that they intend to execute or share. The Registry provides an API that enables creating, updating and deleting workspaces and PEs. Before a workflow can be executed, it needs to be registered in the Registry. In Table 1, a description of the Dispel4py Registry API is provided. The design approach and the main elements of the dispel4py registry are provided in [4].

HTTP method	Name/Endpoint	Description	Content type	Parameters
GET	/connections/	Retrieves all the available PE Connection resources. A PE Connection resource allows the addition and manipulation of PE connections. Connections are associated with PEs and are not themselves workspace items	application/json	No parameters

POST	/connections/	Creates a new PE Connection resource, which allows the addition and manipulation of PE connections. Connections are associated with PEs and are not themselves workspace items	application/json	data (body) example:
				{
				"comment" : "string" ,
				"kind" : "string" ,
				"modifiers" : "string" ,
				"name" : "string" ,
				"is_array" : true ,
				"s_type" : "string" ,
				"d_type" : "string" ,
				"pesig" : "string"
				}

GET	/connections/{id}/	Retrieves a specific PE Connection resource. A PE Connection resource allows the addition and manipulation of PE connections. Connections are associated with PEs and are not themselves workspace	application/json	id (integer)

		items.		

PUT	/connections/{id}/	Updates an existing PE Connection resource. A PE Connection resource allows the addition and manipulation of PE connections. Connections are associated with PEs and are not themselves workspace items.	application/json	-id (integer)
				-data (body)
				example:
				{
				"comment" : "string" ,
				"kind" : "string" ,
				"modifiers" : "string" ,
				"name" : "string" ,
				"is_array" : true ,
				"s_type" : "string" ,
				"d_type" : "string" ,
				"pesig" : "string"
				}

DELETE	/connections/{id}/	Deletes an existing PE Connection resource from the DB	application/json	id (integer)
GET	/fnimpls/	Retrieve all the available function implementation resources (Allows the creation and manipulation of function implementations. Function entities may have one or more implementations.)	application/json	No parameters
POST	/fnimpls/	Creates a new Function Implementation	application/json	data (body), example: { "code" : "string" , "parent_sig" : "string" ,

				"description" : "string" , "pckg" : "string" , "workspace" : "string" , "clone_of" : "string" , "name" : "string" }
GET	/fnimpls/{id}/	Retrieves a specific Function implementation resource	application/json	id (integer)
PUT	/fnimpls/{id}/	Updates an existing function implementation resource	application/json	-id (integer) -data (body), example: { "code" : "string" , "parent_sig" : "string" , "description" : "string" , "pckg" : "string" , "workspace" : "string" , "clone_of" : "string" , "name" : "string" }
DELETE	/fnimpls/{id}/	Deletes an existing Function Implementation resource from the DB	application/json	id (integer)
GET	/fnparams/	Retrieves all the available Function Parameters resource (Allows the	application/json	No parameters

		addition and manipulation of function parameters. Function parameters are associated with functions and are not themselves workspace items.)		
POST	/fnparams/	Creates a new Function Parameters resource in the DB	application/json	data (body), example: { "parent_function" : "string" , "param_name" : "string" , "param_type" : "string" }
GET	/fnparams/{id}/	Retrieves a specific Function Parameters resource	application/json	id (integer)
PUT	/fnparams/{id}/	Updates an existing Function Parameters entry in the DB	application/json	-id (integer) -data (body) example: { "parent_function" : "string" , "param_name" : "string" , "param_type" : "string" }
DELETE	/fnparams/{id}/	Deletes an existing Function Parameters resource from the DB	application/json	id (integer)
GET	/functions/	Retrieves all the Function resources from the DB (A	application/json	No parameters

		function allows addition and manipulation of dispel4py functions)		

POST	/functions/	Creates a new Function Resource in the DB	application/json	data (body), example:
				{
				"description" : "string" ,
				"parameters" : [
				"string"
				],
				"fnimpls" : [
				"string"
				],
				"pckg" : "string" ,
				"workspace" : "string" ,
				"return_type" : "string" ,
				"clone_of" : "string" ,
				"name" : "string"
				}

GET	/functions/{id}/	Retrieves an existing Function resource by id	application/json	id (integer)
PUT	/functions/{id}/	Updates an existing function resource	application/json	-id (integer) -data(body), example: { "description" : "string" , "parameters" : ["string"], "fnimpls" : ["string"], "pckg" : "string" , "workspace" :

				"string" , "return_type" : "string" , "clone_of" : "string" , "name" : "string" }
DELETE	/functions/{id}/	Deletes an existing function resource from the DB	application/json	id (integer)
GET	/groups/	Retrieves all the available groups from the DB. A group represents a basic user group resource	application/json	No parameters

POST	/groups/	Creates a new user group in the DB	application/json	data (body), example:
				{
				“name”: “string”
				}

GET	/groups/{id}/	Retrieves a user group based on its id	application/json	id (integer)

PUT	/groups/{id}/	Updates an existing user group	application/json	-id (integer)
				-data (body), example:
				{
				“name”: “string”
				}

DELETE	/groups/{id}/	Removes a user group from the DB	application/json	id (integer)
GET	/literals/	Retrieves all the literal entities resource from the DB	application/json	No parameters
POST	/literals/	Creates a new Literal Entities resource	application/json	data (body), example: { "description" : "string" ,

				"value" : "string" , "name" : "string" , "pckg" : "string" , "workspace" : "string" , "clone_of" : "string" }
GET	/literals/{id}/	Retrieves a Literal Entities resource based on its id	application/json	id (integer)
PUT	/literals/{id}/	Updates an existing Literal Entities resource	application/json	- id (integer) - data (body), example: { "description" : "string" , "value" : "string" , "name" : "string" , "pckg" : "string" , "workspace" : "string" , "clone_of" : "string" }
DELETE	/literals/{id}/	Deletes a Literal Entities resource from the DB	application/json	id (integer)
GET	/peimpls/	Retrieves all the available PE Implementation resources. A PE Implementation allows the creation and manipulation of PE implementations. PEs may have one or more implementations	application/json	No parameters

POST	/peimpls/	Creates a new PE Implementation	application/json	data (body), example:
				{
				"code" : "string"
				,
				"parent_sig" : "string" ,
				"description" : "string" ,
				"pckg" : "string"
				,
				"workspace" : "string" ,
				"clone_of" : "string" ,
				"name" : "string"
				}

GET	/peimpls/{id}/	Retrieves a specific PE Implementation from the DB based on its ID	application/json	id (integer)

PUT	/peimpls/{id}/	Updates an existing PE Implementation	application/json	-id (integer)
				- data(body), example:
				{
				"code" : "string"
				,
				"parent_sig" : "string" ,
				"description" : "string" ,
				"pckg" : "string"
				,
				"workspace" : "string" ,
				"clone_of" : "string" ,
				"name" : "string"
				}

DELETE	/peimpls/{id}/	Deletes an existing PE	application/json	id (integer)

		Implementation from the DB		
GET	/pes/	Retrieves all the available PE resources from the DB. A PE resource allows the addition and manipulation of dispel4py Processing Elements (PEs)	application/json	No parameters

POST	/pes/	Creates a new PE in the DB	application/json	data (body), example:
				{
				"description" : "string" ,
				"name" : "string" ,
				"connections" : [
				"string"
				],
				"pckg" : "string" ,
				"workspace" : "string" ,
				"clone_of" : "string" ,
				"peimpls" : [
				"string"
				]
				}

GET	/pes/{id}/	Retrieves a specific PE based on its ID	application/json	id (integer)
PUT	/pes/{id}/	Updates an existing PE	application/json	-id(integer) -data(body), example: { "description" : "string" , "name" : "string" , "connections" : [

				"string"], "pckg" : "string" , "workspace" : "string" , "clone_of" : "string" , "peimpls" : ["string"] }
DELETE	/pes/{id}/	Deletes an existing PE in the DB	application/json	id (integer)
GET	/registryusergroups/	Retrieves all the available registry user groups. Extends the functionality of the Django user groups	application/json	No Parameters

POST	/registryusergroups/	Creates a new Registry user group resource	application/json	data (body),
				example:
				{
				"description" :
				"string" ,
				"group_name" :
				"string"
				}

GET	/registryusergroups/{id}/	Retrieves a specific Registry user group based on its ID	application/json	id (integer)
PUT	/registryusergroups/{id}/	Updates an existing Registry user group	application/json	-id(integer) -data(body), example: { "owner" : "string" , "description" : "string" , "group_name" :

				"string" }
DELETE	/registryusergroups/{id}/	Deletes an existing Registry user group	application/json	id (integer)
GET	/users/	Retrieves all the existing users in the DB. Represents a Django Auth User entry	application/json	No parameters

POST	/users/	Creates a new user	application/json	data (body), example:
				{
				"username" : "string" ,
				"password" : "string" ,
				"first_name" : "string" ,
				"last_name" : "string" ,
				"email" : "string"
				}

GET	/users/{id}/	Retrieves a specific user	application/json	id (integer)

PUT	/users/{id}/	Updates a specific user	application/json	-id(integer)
				-data(body), example:
				{
				"username" : "string" ,
				"password" : "string" ,
				"first_name" : "string" ,
				"last_name" : "string" ,
				"email" : "string"
				}

DELETE	/users/{id}/	Deletes a specific user	application/json	id (integer)

GET	/workspaces/	Retrieves all the available workspaces	application/json	parameters: - name: name description: The name of the workspace we want to display paramType: query - name: username description: The username the workspace is associated with (workspaces are uniquely identiÕable for individual users) paramType: query - name: search description: perform a simple full-text on descriptions and names of workspaces. paramType: query
POST	/workspaces/	Create a new workspace, or clone an existing one. In the case of cloning only the Õeld name is taken into account.	application/json	parameters: name: name description: the name of the workspace. name: description description: a textual description of the workspace. name: clone_of description:

				indicates that a cloning operation is requested. paramType: query type: long
GET	/workspaces/{id}/	Retrieves a specific workspace	application/json	name: - ls: Lists the requested contents of the given workspace, as well as its packages, in short -kind: Lists details of the requested type of workspace item. Valid values are pes, functions, literals, peimpls, fnimpls and packages. -startswith: Optionally filters the displayed items depending on the string their package name starts with. startswith currently does not work if not ls is requested and not kind is provided. - fqn: Match the given 'fqn'

				within the workspace exactly. The fqn is in the form of package.name. -search: Perform a simple full-text search over the workspace's contents. The use of 'search' takes precedence over other parameters.

PUT	/workspaces/{id}/	Updates an existing workspace	application/json	-id (integer)
				-data (body), example:
				{
				"clone_of" : "string" ,
				"name" : "string" ,
				"description" : "string"
				}

DELETE	/workspaces/{id}/	Deletes an existing workspace	application/json	id (integer)

The User functionality (authentication, updates ,deletes etc) is handled via Keycloak. The Dispel4py and all the other DARE components use the dare-login component (described in the following section) to communicate with Keycloak and authenticate users. The successfully authenticated users are also stored in the local Dispel4py Registry DB, in order to facilitate the Registry's admin UI.

2.2 DARE Login

The DARE Login component is a simple Flask RESTful Web Service which exposes functionality for user authentication, token validation, internal token issuance etc. This is used internally by the DARE components since it just adds a layer between the components and the Keycloak service which is used

in the backend. The only endpoints used directly by the users is the /auth/ endpoint. In the following table, a description of the DARE Login API is provided.

HTTP method	Name/Endpoint	Description	Content type	Parameters
POST	/auth/	Authenticates a user performing HTTP call to the Keycloak service. After having successfully authenticated the user, Dispel4py Registry, CWL registry and Execution API are notified to check if the user already exists in their local DBs	application/json	data (body), example: { “username”: “string”, “password”: “string”, “requested_issuer”: “string” }
POST	/validate-token/	Validates a token using the Keycloak Service	application/json	data (body), example: { “access_token”: “string” }
POST	/delegation-token/	Issues a token for internal application use (from DARE component to DARE component)	application/json	data (body), example: { “access_token”: “string” }
POST	/refresh-token/	When the access token expires, uses the refresh token to issue a new token for a user	application/json	data (body), example: { “refresh_token”: “string”, “issuer”: “string” }

2.3 CWL Workflow Registry

Since v3.0 of the DARE platform, CWL support and execution is provided, in addition to the dispel4py workflow library. Therefore, a similar to the dispel4py Information Registry component is implemented. The component registers execution environments (dockers) and CWL workflows which can then be retrieved by name and version. An execution environment is represented by a Dockerfile and can be

associated with one or many scripts. The relevant Django models are the DockerEnv which is related to one or multiple DockerScript objects. On the other hand, the CWL workflows are divided into those of CWL class Workflow and those of class CommandLineTool. A CWL of class Workflow is associated with one or many CWLs of class CommandLineTool. In the CWL Workflow registry, the aforementioned workflows are represented by the Django models Workflow and Workflow part respectively. Below, in Table 3, a detailed list of the RESTful endpoints of this component is provided.

HTTP method	Name/Endpoint	Description	Content type	Parameters

POST	/docker/	Creates a new Docker Environment. The environment consist of a Dockerfile and can be associated with one or multiple DockerScript entries (which represent bash or python scripts)	application/json	data (body), example:
				{
				"docker_name": "name",
				"docker_tag": "tag",
				"script_names": ["script1.sh", "script2.sh"]
				"files": {
				"dockerfile": "string",
				"script1.sh": "string.",
				"script2.sh": "string"
				},
				"access_token": "token"
				}

POST	/docker/update_docker/	Updates an existing Docker Environment	application/json	data (body), example: { "docker_name": "name", "docker_tag"

				"tag", "update": {"tag": "v2.0"}, "files": {"dockerfile": "string"}, "access_token": "token" }

POST	/docker/provide_url/	Updates an existing Docker environment's url field. Once the docker image is built and pushed in a public repository, the relevant Docker entry should be updated with the URL	application/json	data (body), example:
				{
				"docker_name": "name",
				"docker_tag": "tag",
				"docker_url": "url",
				"access_token": "token"
				}

DELETE	/docker/delete_docker/		application/json	data (body), example:
				{
				"docker_name": "name",
				"docker_tag": "tag",
				"access_token": "token"
				}

GET	/docker/bynametag/	Retrieves a Docker Environment using its name and tag	application/json	-docker_name (string) - docker_tag(string)
GET	/docker/byuser/	Retrieves all the registered Docker environments by user	application/json	- requested_user(string) if exists, otherwise it uses the user that

				performed the request
GET	/docker/download/	Downloads in a zip file the Dockerfile and the relevant scripts of a Docker Environment	application/json	-docker_name (String) - docker_tag(string)

POST	/scripts/add/	Adds a new script in an existing Docker Environment	application/json	data (body), example:
				{
				"docker_name": "name",
				"docker_tag": "tag",
				"script_name": "entrypoint.sh",
				"files": {"entrypoint.sh": "string"},
				"access_token": "token"
				}

POST	/scripts/edit/	Edits an existing script of a Docker Environment	application/json	data (body), example:
				{
				"docker_name": "name",
				"docker_tag": "tag",
				"script_name": "entrypoint.sh",
				"files": {"entrypoint.sh": “string”},
				"access_token": "token"
				}

DELETE	/scripts/delete/	Deletes an existing script from a docker environment	application/json	data (body), example: { "docker_name":

				"name", "docker_tag": "tag", "script_name": "entrypoint.sh", "access_token": "token" }
GET	/scripts/download	Downloads a specific script from a Docker Environment	application/json	- docker_name(string) - docker_tag(string) - script_name(string)
GET	/scripts/byname	Retrieves a specific script based on the name & tag of the Docker Environment and on the name of the script	application/json	- docker_name(string) - docker_tag(string) - script_name(string)
POST	/workflows/	Creates a new CWL workflow of class Workflow	application/json	data (body), example: {"workflow_name": "demo_workflow.cwl", "workflow_version": "v1.0", "spec_file_name": "spec.yaml", "docker_name": "name", "docker_tag": "tag", "workflow_part_data":[{"name":

				arguments.cwl”, "version":"v1.0", "spec_name": "arguments.yaml"},{"name": "tar_param.cwl", "version" :"v1.0", "spec_name": "tar_param.yaml"}], "files": {"demo_workflow.cwl":"string", "spec.yaml": "string", "arguments.cwl": "string", "arguments.yaml": "string", "tar_param.cwl": "string", "tar_param.yaml": "string"}, "access_token": "token" }
POST	/workflows/update_workflow/	Updates an existing CWL workflow of class Workflow	application/json	data (body), example: { "workflow_name":"demo_workflow.cwl", "workflow_version": "v1.0", "files": {"workflow_file": "string", "spec_file": "string",}, "update": {"version":"v1.1"

				}, "access_token": "token" }
POST	/workflows/update_docker/	Associate a CWL workflow of class Workflow with a different Docker Environment	application/json	data (body), example: { "workflow_name":"demo_workflow.cwl", "workflow_version": "v1.0", "docker_name": "test", "docker_tag": "v1.0", "access_token": "token" }
DELETE	/workflows/delete_workflow/	Deletes an existing CWL workflow (class Workflow) and all the associated Workflow parts (class CommandLineTool)	application/json	data (body), example: { "workflow_name":"demo_workflow.cwl", "workflow_version": "v1.0", "access_token": "token" }
GET	/workflows/bynameversion/	Retrieve a CWL workflow of class Workflow and its associated workflow parts as well as the related docker environment, based on the workflow name and version	application/json	- workflow_name (string) - workflow_version(string)
GET	/workflows/download	Downloads in a zip file all the CWL files (Workflow and CommandLineTool) as well as the relevant	application/json	- workflow_name (string) -

		Dockerfile and scripts (if the parameter dockerized is provided)		workflow_version(string) - dockerized(boolean)

POST	/workflow_parts/add/	Adds a new CommandLineTool CWL in an existing CWL workflow	application/json	data (body), example:
				{
				"workflow_name":"demo_workflow.cwl",
				"workflow_version": "v1.0",
				"workflow_part_name":"arguments.cwl",
				"workflow_part_version": "v1.0",
				"spec_name": "arguments.yaml",
				"files": {"arguments.cwl": "string",
				"arguments.yaml": "string"},
				"access_token": "token"
				}

POST	/workflow_parts/edit/	Edits an existing CommandLineTool CWL workflow	application/json	data (body), example: { "workflow_name":"demo_workflow.cwl", "workflow_version": "v1.0", "workflow_part_name":"arguments.cwl", "workflow_part_version": "v1.0", "spec_name":

				"arguments.yaml", "files": {"arguments.cwl":"string", "arguments.yaml": "string"}, "update": {"version":"v1.1”}, "access_token": "token" }

DELETE	/workflow_parts/delete/	Deletes an existing CommandLIneTool CWL workflow	application/json	data (body), example:
				{
				"workflow_name":"demo_workflow.cwl",
				"workflow_version": "v1.0",
				"workflow_part_name": "arguments.cwl",
				"workflow_part_version": "v1.0",
				"access_token": "token"
				}

GET	/workflow_parts/bynameversion	Retrieves a specific CommandLineTool CWL based on its parent (name & version) and its own name and version	application/json	- workflow_name (string) - workflow_version(string) - workflow_part_name(string) - workflow_part_version(string)

GET	/workflow_parts/download	Downloads a specific CWL of class CommandLineTool	application/json	- workflow_name (string) - workflow_version(string) - workflow_part_name(string) - workflow_part_version(string)

POST	/accounts/login/	Authenticates a user (login) used by the dare-login component described in section 2.2 when a user calls the /auth/ endpoint of the dare-login. If the user does not exist in the CWL workflow registry's local DB, it creates a new user	application/json	data (body), example:
				{
				"username": "string",
				"password":"string",
				"access_token": "string",
				"email":"string",
				"given_name":"string",
				"family_name":"string"
				}

2.4 Execution API

The Execution API component is a Flask RESTful Web Service, used in order to instantiate new execution environments for dispel4py or CWL workflows, for folder and files handling/listing, for monitoring etc. In Table 4, all the Execution API endpoints are listed.

HTTP method	Name/Endpoint	Description	Content type	Parameters

POST	/create-folders/	Endpoint used by the /auth/ endpoint of dare-login. Checks if the user's workspace in the DARE platform is available, otherwise it creates the	application/json	data (body), example:
				{
				"username":"string"
				}

		necessary folder structure		

POST	/d4p-mpi-spec/	Used internally by the dispel4py execution environment in order to retrieve the respective PE Implementation and spec.yaml	application/json	data (body), example:
				{
				"pe_imple": "name",
				“nodes”: 3,
				“input_data”: {
				}
				}

POST	/run-d4p/	Creates a new dispel4py execution environment, using the Kubernetes API. Generates a new run directory, stored under the user’s “runs” folder(i.e. /<home>/<username>/runs/). All the execution results are stored in the generated run directory.	application/json	data (body), example:
				{
				“access_token: “string”,
				“workspace”: “string”,
				“pckg”: “string”,
				“pe_name”:”string”,
				“target”:”simple”,
				“nodes”:1
				}

POST	/run-specfem/	Endpoint dedicated to executing the Specfem3D application. Generates a new run directory, stored under the user’s “runs” folder(i.e. /<home>/<username>/runs/). All the execution results are stored in the generated run directory.	application/json	data (body), example:
				{
				“access_token: “string”,
				“folder_name”:”string”,
				“filename”:”string”,
				“n_nodes”:22
				}

POST	/run-cwl/	Endpoint to instantiate an execution environment for CWL workflow execution. The environment to be instantiated is retrieved from the CWL using the	application/json	data (body), example: { “access_token”: ”string”, “nodes”:12,

		CWL Workflow Registry. Generates a new run directory, stored under the user's "runs" folder(i.e. /<home>/<username>/runs/). All the execution results are stored in the generated run directory.		"workflow_name":"string", "workflow_version":"string", "input_data": { "example1":"string" } }
POST	/upload/	Endpoint used to upload files in the DARE platform. The files are stored under the user's home directory. The home directory is named after his/hers username and inside there are 3 folders, i.e. uploads, debug and runs. All the uploaded files are stored under the user's "uploads" directory	application/json	data (body), example: { "dataset_name":"string", "path":"string", "access_token":"string", "files": [<file list>] }
GET	/my-files/	Lists all the users' directories under the "uploads", "runs" and "debug" folders. If the parameter num_run_dirs is present, the response is limited to the most recent directories based on the number provided in the aforementioned parameter	application/json	- access_token(string) - num_run_dirs(integer)
GET	/list/	Lists all the files inside a specific directory. This directory could be retrieved from the previous endpoint	application/json	- access_token(string) -path(string)
GET	/download/	Downloads a specific file from the DARE platform. To find the file's full path use the two previous endpoints	application/json	- access_token(string) -path(string)

GET	/send2drop/	Uploads files from the dare platform to B2DROP	application/json	-access_token(string) -path(string)
GET	/cleanup/	Clears the user's folders (uploads, runs, debug)	application/json	-access_token(string) -runs(boolean) -uploads(boolean) -debug(boolean)

2.5 Provenance API

We report in this section the overview and description of the lineage methods characterising the API exposed by the S-ProvFlow system [3]. The methods return information in JSON-LD format and are interactively used by the tools described in D3.7 and D3.8. Full description is available in OpenAPI v3 format at https://platform.dare.scai.fraunhofer.de/prov/swagger/

Provenance acquisition				
HTTP method	Name/Endpoint	Description	Content type	Parameters
POST	**workflowexecutions/insert**	Bulk insert of bundle or lineage documents in JSON format	application/json	JSON with one of more provenance documents
POST	**workflowexecutions/<id>/edit**	Update of the description of a workflow execution. Users can improve this information in free text.	application/json	JSON with a description property with the updated text
GET	**workflowexecutions/<id>/delete**	Delete a workflow execution trace, including its bundle and all its lineage documents.	application/json	
POST	**workflowexecutions/import**	Import lineage traces	application/json	runId,

		from other workflow systems and maps them to S-PROV. The current implementation supports the import of traces in CWLProv. Clients can specify their own custom runId and the location of the resources. For instance, the hostname of a data catalogue where all the files will be stored.		resultLocation, format

Monitoring, validation and lineage queries				
HTTP method	Name/Endpoint	Description	Content type	Parameters
GET	**workflowexecutions(/<id> | ?<query string>)**	Provides a list of workflow runs performed by one or more users. Runs can be searched by specifying the parameters values used in the workflow and by the metadata associated with the data and the data-formats. Mode of the search can also be indicated (mode ::= (OR | AND). Boolean operators are applied metadata and parameters' values within each run.	application/json	usernames, wasAssociatedWith, terms, expressions, clusters, mode, functionNames, formats, types, start, limit
GET	**workflowexecutions/<runid>/showactivity?<query-string>**	Extract detailed information of the processes executed in each run. It shows progress, anomalies	application/json	level, start, limit

		(such as exceptions or systems' and users messages), count of the data produced and whether it is available for download. This method can also be used for runtime monitoring.		
GET	**instances/<id>** **invocations/<id>**	Extract details about the invocation of an instance of a workflow process.	application/json	wasAssociateFor, start, limit
GET	**data*(/<id>* | *?<query string>)***	Extract metadata associated with a *Data* item and its members, the *DataGranules*. The data is selected by specifying the id or a *query-string*. Query parameters allow searching by *attribution (*to a workflow or one of its processes/functions) or by specifying metadata *expressions* indicating single values, ranges, lists, and data formats. Mode of the search can also be indicated (mode ::= (OR | AND)).	application/json	usernames, wasAttributedTo, wasGeneratedBy, terms, expressions, clusters, mode, functionNames, formats, types, start, limit
POST	**data/filterOnAncestor*?<query string>***	Filter a list of data ids based on the existence of at least one ancestor in their data dependency graph, according to a matching metadata	application/json	terms, expressions, wasAssociatedWith, ids

		expression indicating single values, ranges or lists, and data formats. Maximum depth level and mode of the search can also be indicated.		
GET	**data/<*id*>/derivedData** **data/<*id*>/wasDerivedFrom**	Starting from a specific data entity of the data dependency is possible to navigate through the derived data (11) or backwards across the element's data dependencies (12). The number of traversal steps is provided as a parameter (*level*).	application/json	level
GET	**terms*?<query string>***	Return a list of discoverable metadata terms based on their appearance in a collection of runIds and usernames, passed as parameters, or for the whole provenance archive. Terms are returned indicating their type (when consistently used), min and max values and their number of occurrences within the scope of the search.	application/json	runIds, usernames, aggregationLevel

Comprehensive Summaries (*Experimental*): These methods are used to produce visual analytics within the BDV (Bulk Dependencies Visualiser) of the s-ProvFlow. They allow clients to cluster the returning data by specifying a particular property in the *groupBy* parameter.				
HTTP method	Name/Endpoint	Description	Content type	Parameters
GET	**summaries/workflowexecutions*?<query string>***	Returns an overview of the distribution of the computation within a single run. It reports the size of data movements between the workflow components, their instances or invocations depending on the specified granularity level.	application /json	runid, minidx, maxidx, groupby, mintime, maxtime, level
GET	**summaries/collaborative*?<query string>***	Extract information about the reuse and exchange of data between workflow executions based on terms' values-ranges and a group of users. The API method allows for inclusive or exclusive (mode ::= (OR | AND) queries on the terms' values.	application /json	usernames, terms, wasAssociatedWith, clusters, mode, groupby, expressions, level, functionNames, formats

2.6 Semantic-data API

DARE Semantic Data Discovery Service is an application to search through linked data. The main task is to make the metadata stored by the Data Catalogue available via search operations. But also external data catalogs can be added as data sources via configuration of the webservice and thus be made centrally searchable. The prerequisite is that the catalog data is stored in W3C's DCAT format.

HTTP method	Name/Endpoint	Description	Content type	Parameters
GET	/search/	Offers a full-text search query and filter fields to search within the index.	application/json	query, start,len, format, id, endpoint, type, text
GET	/index/	Lists all indexed datasets with source endpoint and unique URI.	application/json	
DELETE	/index/	Deletes the index of all datasets. Usefull to reindex everything after a schema change.	application/json	
PUT	/index/	Create/Updates index from all datasets found within the configured endpoints.	application/json	
GET	/index/endpoints/	Lists the exact name of all endpoints configured in this service.	application/json	
GET	/index/endpoints/<name>	Lists all indexed datasets on a specified endpoint with source endpoint and unique URI.	application/json	name
DELETE	/index/endpoints/<name>	Deletes the index for all datasets from a specified endpoint. Single dataset only is possible.	application/json	name
PUT	/index/endpoints/<name>	Create/Updates index from all datasets found in a specified endpoints. Single dataset only is possible.	application/json	name

GET	/monitor/heartbeat	Offers an easy-to-call resource to test the connection to Semantic Data Discovery.	application/json	
GET	/monitor/status	Checks connection to the external services solr and endpoints.	application/json	

2.7 Playground module

The DARE platform provides a testing execution environment to the users, in order to have immediate access to the generated files and logs during the workflow's development. Additionally, this module simulates a user's terminal, therefore the user can execute a dispel4py or CWL command using the playground's API. This component is a simple Flask RESTful Web Service and its endpoints are listed in Table 5.

HTTP method	Name/Endpoint	Description	Content type	Parameters
POST	/create-folders/	Endpoint used by the /auth/ endpoint of dare-login. Checks if the user's workspace in the DARE platform is available, otherwise it creates the necessary folder structure	application/json	data (body), example: { "username":"string" }
POST	/playground/	Simulates a dispel4py execution	application/json	data (body), example: { "access_token: "string", "workspace": "string", "pckg": "string", "pe_name":"string", "target":"simple", "nodes":1 }
POST	/run-command/	Simulate a user's terminal	application/json	data (body),

				example:
				{
				“access_token:
				“string”,
				“run_dir”:”string
				”(optional),
				“command”:”stri
				ng”
				}

POST	/upload/	Endpoint used to upload files in the DARE platform. The files are stored under the user’s home directory. The home directory is named after his/hers username and inside there are 3 folders, i.e. uploads, debug and runs. All the uploaded files are stored under the user’s “uploads” directory	application/json	data (body),
				example:
				{
				“dataset_name”
				:”string”,
				“path”:”string”,
				“access_token”:
				”string”,
				“files”: [<file
				list>]
				}

GET	/my-files/	Lists all the users’ directories under the “uploads”, “runs” and “debug” folders. If the parameter num_run_dirs is present, the response is limited to the most recent directories based on the number provided in the aforementioned parameter	application/json	- access_token(str ing) - num_run_dirs(in teger)
GET	/list/	Lists all the files inside a specific directory. This directory could be retrieved from the previous endpoint	application/json	- access_token(str ing) -path(string)
GET	/download/	Downloads a specific file from the DARE platform. To find the file’s full path use the two previous endpoints	application/json	- access_token(str ing) -path(string)

3 dispel4py and CWL-specific interfaces

The DARE platform aims to ease the orchestration of persistent computational services and the establishment of high-throughput data channels between them. This is achieved by creating stream-based workflows and establishing connection interfaces between the main operators through which data is either consumed or forwarded to output, called processing elements (PEs). In addition to streaming workflows, DARE is also capable of registering and executing CWL workflows.

In DARE, these PEs and workflows are provided by the dispel4py and CWL registries. The workflows and processing elements defined and implemented can also be seen as an API between users and the DARE platform via its WaaS endpoint. A basic collection of PEs available via the DARE platform has been provided in D3.5[1] and remains unchanged. Further development of workflows has taken place in order to implement the two DARE use-cases, details of which are provided in D6.4 and D7.4.

3.1 Provenance Controls in dispel4py

In D3.5 and in the literature [3] we have introduced the concept of provenance types to support users who may want to customise the extraction of metadata from a workflow PEs, or to tune the granularity and precision of the recorded data dependencies. Types are reusable across the workflow operators and can be assigned to the PEs by using the dispel4py *Provenance Configuration* framework. This consists in a JSON document, which is used by the workflow system to prepare the execution for customised provenance capturing. In the last phase of the project we simplified the information that had to be specified by the research developers in the JSON, and improved the flexibility of its adoption by DARE, as well as other systems, fostering automation. In the following example, we show the JSON document that configure the provenance for the MySplitMerge[2] workflow.

```
prov_config =  {
                's-prov:description' : "API demo",
                's-prov:workflowName': "splitMerge",
                's-prov:workflowType': "dare:Thing",
                's-prov:workflowId'  : "splitmerge",
                's-prov:save-mode'   : 'service',
                's-prov:WFExecutionInputs':  [],
# defines the Provenance Types and Provenance Clusters for the Workflow Components
                's-prov:componentsType' : {
                                'mergePE': {'s-prov:type':(AccumulateFlow,),
                                            's-prov:prov-cluster':'seis:Processor'},
                                'splitPE': {'s-prov:type':(DataInGranuleType,),
                                            's-prov:prov-cluster':'seis:Processor'}}
                        }
```

Compared to the version used at the beginning of DARE, we have removed the properties indicating the user and run ids. The API automatically identifies who started the workflow and assigns an *id* to the run. Both *ids* propagate to all the relevant components of the DARE platform, together with

[1] http://project-dare.eu/wp-content/uploads/2019/03/D3.5-DARE-API-I_final_draft.pdf

[2] https://gitlab.com/project-dare/exec-api/-/raw/master/examples/mySplitMerge/scripts/mySplitMerge_prov.py

authentication credentials. This fosters secure data access, delegation of operations to other integrated systems and eventually trusted traceability.

Currently, users still have to specify the *workflowName* of *workflowId*. However, these could be linked to the DARE Registry and set by the API. This is achievable thanks to the decoupling of the JSON configuration from the source code of the workflow, and to the implementation of additional dispel4py command line parameters that play a role in the automation of configuration, extraction and storage of provenance data. We describe these new parameters below.

```
-d <input-data>
--provenance-bearer-token = <user-OpenAuth-Token>
--provenance-config = inline (when provenance JSON in the workflow code)
                      file <filename> (when provenance JSON is provided as a file)
--provenance-repository-url = <s-ProvFlow ingestion URL>
--provenance-runid = <DARE generated run Id>
--provenance-userid = <subject>@<issuer>
```

4 Conclusions

The DARE platform aims to provide comprehensive and well-integrated functionality to research engineers and scientists to help them manage and manipulate large distributed data and remote systems with little knowledge of their underlying intricacies. The platform's API's role is to realise the aforementioned user/developer-friendly interface capable of encapsulating the diversity of data, methods and tools. This is achieved via providing functionality to describe different types of workflows, with the DARE platform taking care of its execution, provenance tracking, output file handling, etc.

5 References

[1] Klampanos et al., (2020). DARE Platform: a Developer-Friendly and Self-Optimising Workflows-as-a-Service Framework for e-Science on the Cloud. Journal of Open Source Software, 5(54), 2664, https://doi.org/10.21105/joss.02664

[2] Atkinson, M., Filgueira, R., Gemünd, A., Karkaletsis, V., Klampanos, I., Koukourikos, A., Levray, A., Lindner, M., Magnoni, F., Pagé, C., Rietbrock, A., Spinuso, A., Themeli, S., Tsilimparis, X., & Wolf, F. (2020). *DARE Architecture and Technology internal report*. https://doi.org/10.5281/ZENODO.3697898

[3] Spinuso et al., (2019). Active provenance for Data-Intensive workflows: engaging users and developers, 15th International Conference on eScience (eScience), IEEE, 2019.

[4] Klampanos I.A., Martin P., & Atkinson M.P. (2019, August 6). Consistency and Collaboration for Fine-Grained Scientific Workflow Development: The dispel4py Information Registry. Zenodo. http://doi.org/10.5281/zenodo.3361395