David

Release 0.1.0

Raphael Pinto

Feb 21, 2020

CONTENTS:

David is a powerfull engine dialogue engine with two main parts: `NLU` and `dialogue`. The main objetive of this project is too use existing chatbots projects and uses it to develop your solution.

The name was inspired by David Ausubel because the main objective of this project was to build a collaborative platform to maintain Bots for learning.

Inspired by Watson Assistant and Rasa.

CHAPTER

ONE

GETTING STARTED

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes. See deployment for notes on how to deploy the project on a live system.

- Docker:

```
docker-compose up --build
```

- Editable mode:

```
pip install -e .
cd examples/my-first-bot
david train
david run
```

- After that you should see this output:

```
david run
 * Serving Flask app "david.__main__" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

CHAPTER

TWO

PREREQUISITES

- To build, first install all necessary dependencies:

All the dependencies can be find in `requirements.txt` and development in `requirements-dev.txt`.

CHAPTER

THREE

INSTALLING DEVELOPMENT ENVIRONMENT

- A step by step installation guide:

1. Run these commands to install `david` in your python virtual env:

```
pip install -r requirements-dev.txt
pip install -e .
```

1. Go to examples folder and start the project:

```
cd examples/my-first-bot
david train
david run
```

1. Have fun :rocket:

CHAPTER

FOUR

CODE STYLE

To ensure a standardized code style we use the formatter black. To ensure our type annotations are correct we use the type checker pytype. If your code is not formatted properly or doesn't type check, travis will fail to build.

4.1 Formatting

If you want to automatically format your code on every commit, you can use pre-commit. Just install it via `pip install pre-commit` and execute `pre-commit install` in the root folder. This will add a hook to the repository, which reformats files on every commit.

If you want to set it up manually, install black via `pip install -r requirements-dev.txt`. To reformat files execute

```
make formatter
```

4.2 Type Checking

If you want to check types on the codebase, install `pytype` using `pip install -r requirements-dev.txt`. To check the types execute

```
make types
```

CHAPTER

FIVE

RUNNING THE TESTS

Still needed

CHAPTER

SIX

DEPLOYMENT

Still needed

CHAPTER

SEVEN

BUILT WITH

- Python - The main programing language used

CHAPTER

EIGHT

CONTRIBUTING

Please read CONTRIBUTING.md for details on our code of conduct, and the process for submitting pull requests to us.

CHAPTER

NINE

VERSIONING

We use SemVer for versioning. For the versions available, see the tags on this repository.

CHAPTER

TEN

AUTHORS

- **Raphael Pinto** - *Creator* - ralphg6

See also the list of contributors who participated in this project.

CHAPTER

ELEVEN

LICENSE

This project is licensed under the Apache-2.0 - see the LICENSE file for details

CHAPTER

TWELVE

ACKNOWLEDGMENTS

- **Arthur Temporim** - arthurTemporim

CHAPTER

THIRTEEN

WELCOME TO DAVID'S DOCUMENTATION!

13.1 david package

13.1.1 Subpackages

david.adapters package

Subpackages

david.adapters.adapters package

Submodules

david.adapters.adapters.google module

class david.adapters.adapters.google.**GoogleAdapter**
Bases: *david.adapters.adapter.Adapter*

classmethod input(*payload: Dict*) → david.typing.message.Message

classmethod name()
The name property is a function of the class - its __name__.

classmethod output(*message: david.typing.message.Message*) → Dict

classmethod validate_data(*payload: Dict*) → bool
validate_data method documentation

Module contents

Submodules

david.adapters.adapter module

class david.adapters.adapter.**Adapter**
Bases: *david.typing.module.Module*

Adapter Class documentation

abstract classmethod input(*payload: Dict*) → david.typing.message.Message

abstract classmethod output(*message: david.typing.message.Message*) → Dict

classmethod validate_data(*payload: Dict*) → bool
validate_data method documentation

class david.adapters.adapter.**MessageAdapter**
Bases: *david.adapters.adapter.Adapter*

classmethod input(*payload: Dict*) → david.typing.message.Message

classmethod name()
The name property is a function of the class - its __name__.

classmethod output(*message: david.typing.message.Message*) → Dict

classmethod validate_data(*payload: Dict*) → bool
validate_data method documentation

Module contents

david.components package

Subpackages

david.components.dialogue package

Submodules

david.components.dialogue.simple module

class david.components.dialogue.simple.**SimpleDialogue**(*component_config: Optional[Dict[str, Any]] = None*, *dialog_nodes: Dict = None*)
Bases: *david.components.component.Component*

classmethod load(*meta: Dict[str, Any], model_dir: Optional[str] = None, model_metadata: Optional[Metadata] = None, cached_component: Optional[Component] = None, **kwargs: Any*) → david.components.component.Component
Load this component from file. After a component has been trained, it will be persisted by calling *persist*. When the pipeline gets loaded again, this component needs to be able to restore itself. Components can rely on any context attributes that are created by `components.Component.create()` calls to components previous to this one.

classmethod name()
The name property is a function of the class - its __name__.

persist(*file_name: str*, *model_dir: str*) → Optional[Dict[str, Any]]
Persist this component to disk for future loading.

process(*message: david.typing.message.Message*, ***kwargs: Any*) → None
Process an incoming message. This is the components chance to process an incoming message. The component can rely on any context attribute to be present, that gets created by a call to `rasa.nlu.components.Component.create()` of ANY component and on any context attributes created by a call to `rasa.nlu.components.Component.process()` of components previous to this one.

train(*training_data: david.typing.training_data.TrainingData*, *cfg: david.config.DavidConfig*, ***kwargs: Any*) → None
Train this component. This is the components chance to train itself provided with the training data. The component can rely on any context attribute to be present, that gets created by a call to `rasa.nlu.components.Component.create()` of ANY component and on any context attributes created by a call to `rasa.nlu.components.Component.train()` of components previous to this one.

david.components.dialogue.simple.**evalCondition**(*condition*, *context*, *intent*, *entities*)

david.components.dialogue.simple.**get_dialog_anythinelse**(*dialog_nodes*)

david.components.dialogue.simple.**get_dialog_welcome**(*dialog_nodes*)

Module contents

david.components.nlu package

Submodules

david.components.nlu.simple module

class david.components.nlu.simple.**SimpleNLU**(*component_config: Optional[Dict[str, Any]] = None*, *intent_model: Dict = None*)
Bases: *david.components.component.Component*

classmethod load(*meta: Dict[str, Any]*, *model_dir: Optional[str] = None*, *model_metadata: Optional[Metadata] = None*, *cached_component: Optional[Component] = None*, ***kwargs: Any*) → david.components.component.Component
Load this component from file. After a component has been trained, it will be persisted by calling *persist*. When the pipeline gets loaded again, this component needs to be able to restore itself. Components can rely on any context attributes that are created by `components.Component.create()` calls to components previous to this one.

classmethod name()
The name property is a function of the class - its __name__.

persist(*file_name: str*, *model_dir: str*) → Optional[Dict[str, Any]]
Persist this component to disk for future loading.

process(*message: david.typing.message.Message*, ***kwargs: Any*) → None
Process an incoming message. This is the components chance to process an incoming message. The component can rely on any context attribute to be present, that gets created by a call to `rasa.nlu.components.Component.create()` of ANY component and on any context attributes created by a call to `rasa.nlu.components.Component.process()` of components previous to this one.

train(*training_data: david.typing.training_data.TrainingData*, *cfg: david.config.DavidConfig*, ***kwargs: Any*) → None
Train this component. This is the components chance to train itself provided with the training data. The component can rely on any context attribute to be present, that gets created by a call to `rasa.nlu.components.Component.create()` of ANY component and on any context attributes created by a call to `rasa.nlu.components.Component.train()` of components previous to this one.

david.components.nlu.simple.**simmilarity**(*a*, *b*)

david.components.nlu.simple.**tokenize**(*stopwords: List[str]*, *sentence: str*)

Module contents

Submodules

david.components.component module

class david.components.component.**Component**(*component_config: Optional[Dict[str, Any]] = None*)

Bases: *david.typing.module.Module*

A component is a message processing unit in a pipeline. Components are collected sequentially in a pipeline. Each component is called one after another. This holds for initialization, training, persisting and loading the components. If a component comes first in a pipeline, its methods will be called first. E.g. to process an incoming message, the `process` method of each component will be called. During the processing (as well as the training, persisting and initialization) components can pass information to other components. The information is passed to other components by providing attributes to the so called pipeline context. The pipeline context contains all the information of the previous components a component can use to do its own processing. For example, a featurizer component can provide features that are used by another component down the pipeline to do intent classification.

classmethod cache_key(*component_meta: Dict[str, Any], model_metadata: david.typing.model.Metadata*) → Optional[str]

This key is used to cache components. If a component is unique to a model it should return None. Otherwise, an instantiation of the component will be reused for all models where the metadata creates the same key.

classmethod can_handle_language(*language: Hashable*) → bool

Check if component supports a specific language. This method can be overwritten when needed. (e.g. dynamically determine which language is supported.)

classmethod create(*component_config: Dict[str, Any], config: david.config.DavidConfig*) → david.components.component.Component

Creates this component (e.g. before a training is started). Method can access all configuration parameters.

defaults = {}

language_list = None

classmethod load(*component_config: Dict[str, Any], model_dir: Optional[str] = None, model_metadata: Optional[Metadata] = None, cached_component: Optional[Component] = None, **kwargs: Any*) → david.components.component.Component

Load this component from file. After a component has been trained, it will be persisted by calling *persist*. When the pipeline gets loaded again, this component needs to be able to restore itself. Components can rely on any context attributes that are created by `components.Component.create()` calls to components previous to this one.

partially_process(*message: david.typing.message.Message*) → david.typing.message.Message

Allows the component to process messages during training (e.g. external training data). The passed message will be processed by all components previous to this one in the pipeline.

persist(*file_name: str, model_dir: str*) → Optional[Dict[str, Any]]

Persist this component to disk for future loading.

prepare_partial_processing(*pipeline: List[Component], context: Dict[str, Any]*) → None

Sets the pipeline and context used for partial processing. The pipeline should be a list of components that are previous to this one in the pipeline and have already finished their training (and can therefore be safely used to process messages).

process(*message: david.typing.message.Message*, ***kwargs: Any*) → None
Process an incoming message. This is the components chance to process an incoming message. The component can rely on any context attribute to be present, that gets created by a call to `rasa.nlu.components.Component.create()` of ANY component and on any context attributes created by a call to `rasa.nlu.components.Component.process()` of components previous to this one.

provide_context() → Optional[Dict[str, Any]]
Initialize this component for a new pipeline This function will be called before the training is started and before the first message is processed using the interpreter. The component gets the opportunity to add information to the context that is passed through the pipeline during training and message parsing. Most components do not need to implement this method. It's mostly used to initialize framework environments like MITIE and spacy (e.g. loading word vectors for the pipeline).

provides = []

classmethod required_packages() → List[str]
Specify which python packages need to be installed to use this component, e.g. `["spacy"]`. More specifically, these should be importable python package names e.g. *sklearn* and not package names in the dependencies sense e.g. *scikit-learn* This list of requirements allows us to fail early during training if a required package is not installed.

requires = []

train(*training_data: david.typing.training_data.TrainingData*, *cfg: david.config.DavidConfig*, ***kwargs: Any*) → None
Train this component. This is the components chance to train itself provided with the training data. The component can rely on any context attribute to be present, that gets created by a call to `rasa.nlu.components.Component.create()` of ANY component and on any context attributes created by a call to `rasa.nlu.components.Component.train()` of components previous to this one.

exception david.components.component.**UnsupportedLanguageError**(*component: str*, *language: str*)
Bases: `Exception`

Raised when a component is created but the language is not supported.

Parameters

- **component** – component name
- **language** – language that component doesn't support

david.components.engine module

class david.components.engine.**Engine**(*config: david.config.DavidConfig*)
Bases: `object`

components: List[david.components.component.Component] = []

pipeline: List[Type[david.components.component.Component]] = []

respond(*message*, *context={}*)

train()

david.components.engine.**build_model_filename**(*idx*, *componentCls: Type[david.components.component.Component]*)

Module contents

david.training_data package

Subpackages

david.training_data.formats package

Submodules

david.training_data.formats.json module

class david.training_data.formats.json.**JsonReader**
Bases: *david.training_data.reader.Reader*

classmethod load(*config: david.config.DavidConfig*) → david.typing.training_data.TrainingData

Module contents

Submodules

david.training_data.reader module

class david.training_data.reader.**Reader**
Bases: *david.typing.module.Module*

abstract classmethod load(*config: david.config.DavidConfig*) → david.typing.training_data.TrainingData

classmethod validate_data(*config: david.config.DavidConfig*, *data: Dict*) → bool

Module contents

david.typing package

Submodules

david.typing.message module

class david.typing.message.**Message**(*text: str*, *context=None*, *data={}*, *time=None*)
Bases: object

classmethod build(*text=None*, *intents=None*, *entities=None*, *context=None*, *payload=None*) → david.typing.message.Message

get(*prop*, *default=None*) → Any

set(*prop*, *info*) → None

david.typing.model module

class david.typing.model.**Metadata**(*data*)
Bases: object

class david.typing.model.**Model**(*data*)
Bases: object

david.typing.module module

class david.typing.module.**Module**
Bases: object

Metaclass with *name* class property

classmethod name()
The name property is a function of the class - its __name__.

david.typing.training_data module

class david.typing.training_data.**TrainingData**(*data*)
Bases: object

Module contents

david.utils package

Submodules

david.utils.io module

david.utils.io.**create_directory**(*directory_path: str*) → None
Creates a directory and its super paths. Succeeds even if the path already exists.

david.utils.io.**create_directory_for_file**(*file_path: str*) → None
Creates any missing parent directories of this file path.

david.utils.io.**create_path**(*file_path: str*) → None
Makes sure all directories in the 'file_path' exists.

david.utils.io.**create_temporary_file**(*data: Any*, *suffix: str = ''*, *mode: str = 'w+'*) → str
Creates a tempfile.NamedTemporaryFile object for data. mode defines NamedTemporaryFile's mode parameter in py3.

david.utils.io.**dump_obj_as_json_to_file**(*filename: str*, *obj: Any*) → None
Dump an object as a json string to a file.

david.utils.io.**enable_async_loop_debugging**(*event_loop: asyncio.events.AbstractEventLoop*, *slow_callback_duration: float = 0.1*) → asyncio.events.AbstractEventLoop

david.utils.io.**fix_yaml_loader**() → None
Ensure that any string read by yaml is represented as unicode.

david.utils.io.**is_subdirectory**(*path: str*, *potential_parent_directory: str*) → bool

david.utils.io.**list_directory**(*path: str*) → List[str]
Returns all files and folders excluding hidden files. If the path points to a file, returns the file. This is a recursive implementation returning files in any depth of the path.

david.utils.io.**list_files**(*path: str*) → List[str]
Returns all files excluding hidden files. If the path points to a file, returns the file.

david.utils.io.**list_subdirectories**(*path: str*) → List[str]
Returns all folders excluding hidden files. If the path points to a file, returns an empty list.

david.utils.io.**read_config_file**(*filename: str*) → Dict[str, Any]
Parses a yaml configuration file. Content needs to be a dictionary :param filename: The path to the file which should be read.

david.utils.io.**read_file**(*filename: str*, *encoding: str = 'utf-8'*) → Any
Read text from a file.

david.utils.io.**read_json_file**(*filename: str*) → Any
Read json from a file.

david.utils.io.**read_yaml**(*content: str*) → Union[List[Any], Dict[str, Any]]
Parses yaml from a text. :param content: A text containing yaml content.

david.utils.io.**read_yaml_file**(*filename: str*) → Union[List[Any], Dict[str, Any]]
Parses a yaml file. :param filename: The path to the file which should be read.

david.utils.io.**replace_environment_variables**() → None
Enable yaml loader to process the environment variables in the yaml.

david.utils.io.**unarchive**(*byte_array: bytes*, *directory: str*) → str
Tries to unpack a byte array interpreting it as an archive. Tries to use tar first to unpack, if that fails, zip will be used.

david.utils.io.**write_text_file**(*content: str, file_path: Union[str, pathlib.Path], encoding: str = 'utf-8', append: bool = False*) → None
Writes text to a file. :param content: The content to write. :param file_path: The path to which the content should be written. :param encoding: The encoding which should be used. :param append: Whether to append to the file or to truncate the file.

david.utils.io.**write_yaml_file**(*data: Dict, filename: Union[str, pathlib.Path]*) → None
Writes a yaml file. :param data: The data to write. :param filename: The path to the file which should be written.

david.utils.io.**zip_folder**(*folder: str*) → str
Create an archive from a folder.

Module contents

13.1.2 Submodules

13.1.3 david.config module

class david.config.**DavidConfig**(*configuration_values: Optional[Dict[str, Any]] = None*)
Bases: object

as_dict() → Dict[str, Any]

property component_names

for_component(*index*, *defaults=None*) → Dict[str, Any]

get(*key: str*, *default: Any = None*) → Any

items() → List[Any]

override(*config*) → None

set_component_attr(*index*, ***kwargs*) → None

view() → str

exception david.config.**InvalidConfigError**(*message: str*)
Bases: ValueError

Raised if an invalid configuration is encountered.

david.config.**component_config_from_pipeline**(*index: int, pipeline: List[Dict[str, Any]], defaults: Optional[Dict[str, Any]] = None*) → Dict[str, Any]

david.config.**load**(*config: Union[str, Dict, None] = None*, ***kwargs: Any*) → david.config.DavidConfig

david.config.**override_defaults**(*defaults: Optional[Dict[str, Any]], custom: Optional[Dict[str, Any]]*) → Dict[str, Any]

13.1.4 david.constants module

13.1.5 david.registry module

class david.registry.**Registry**
Bases: object

classmethod get(*moduleName: str = None*, *moduleCls: Type[david.typing.module.Module] = None*, *prefix: str = ''*, *default=None*) → Any

classmethod getAdapter(*config: david.config.DavidConfig*, *adapterName: str = None*)

modules = {'dialogue_simple': <class 'david.components.dialogue.simple.SimpleDialogue

classmethod registry(*module: Type[david.typing.module.Module], prefix: str = ''*)

classmethod registryAdapter(*adapter: Type[david.typing.module.Module]*)

david.registry.**pipeline_template**(*s: str*) → Optional[List[Dict[str, Any]]]

13.1.6 david.util module

david.util.**json_to_string**(*obj: Any*, ***kwargs: Any*) → str

13.1.7 david.version module

13.1.8 Module contents

PYTHON MODULE INDEX

d

INDEX

I

J

L

M

N

O

P

R

S

T

U

V

W

Z