Business Process Task Library

Release 0.1.0

Maykin Media

Aug 02, 2021

CONTENTS

Version 0.1.0

Source https://github.com/GemeenteUtrecht/bptl

Keywords bpmn, camunda, external tasks, process engine, VNG, Common Ground

PythonVersion 3.8

A webapplication to configure and run worker units to process tasks from external engines. Currently it supports Camunda external tasks .

Developed by Maykin Media B.V. for Gemeente Utrecht.

CHAPTER

ONE

INTRODUCTION

Common Ground zet in op een nieuwe, moderne gezamenlijke informatievoorziening. In het 5-lagen model van Common Ground worden gegevens gescheiden van Interactie en proces, waarbij gegevens via Services/APIs ontsloten worden.

BPTL zet hierbij in op de Integratielaag. Vaak leiden stappen in een proces (wat leeft in een proces-engine zoals Camunda) tot bepaalde taken die uitgevoerd dienen te worden tegen deze specifieke services/APIs.

In eerste instantie focust BPTL op de integratie met de API's voor zaakgericht werken - stappen in het Camunda proces leiden tot het aanmaken en bijwerken van Zaken, waarbij generieke bouwstenen opnieuw gebruikt kunnen worden voor verschillende processen.

Uitbreiding met nieuwe (typen) van taken wordt eenvoudig, en het invullen van de procestaken met Camunda is technologie-onafhankelijk door het gebruik van External Tasks.

Zie *Architecture* (EN) voor een overzicht van de architectuur.

CHAPTER

TWO

USAGE

2.1 Usage

2.1.1 Management commands

show_task_registry

A command to quickly see which tasks are registered in the project.

Example:

```
python src/manage.py show_task_registry

bptl.dummy.tasks.dummy

  A dummy task to demonstrate the registry machinery.

  The task receives the :class:`FetchedTask` instance and logs some information,
  after which it completes the task.
```

2.1.2 Python API

Execute tasks

When an external task for a certain topic is received, you can use `bptl.tasks.api.execute` to process it. Pass the `FetchedTask` instance and make sure that the required `WorkUnit` is added to the registry.

2.2 Deployment

For BPTL deployment, we recommend using the Docker images, available on Docker Hub.

The `docker-compose.yml` can provide a little insight in the required services.

2.2.1 Dependencies

BPTL is tested against Camunda. Support for Activiti is minimal but generic in the form of REST API endpoints.

If you're running against Camunda, you need:

- A Camunda instance with REST api, e.g. `https://camunda.example.com/engine-rest/`
- An API user with username/password credentials. The user needs at least the following permissions:
  - `READ`, `UPDATE`, `UPDATE_VARIABLE` on *Process Instance*, with wildcard Resource ID.

2.2.2 Services

BPTL requires the following services:

- PostgreSQL 9.6 or higher database
- Redis as message queue broker, result store and in-memory cache for the web interface
- Some form of reverse proxy (e.g. Nginx, Traefik…)

The BPTL docker image contains the following executables:

- web worker (`/start.sh`)
- celery beat to kick off periodic tasks (`/celery_beat.sh`)
- celery worker (`/celery_worker.sh`)
- celery monitoring (`/celery_flower.sh`)

Celery is the tooling used for asynchronous background tasks, which is *required* if you use Camunda.

Queues

BPTL makes use of two distinct Celery queues, which means you will need to have at least one worker running on each.

You can set the queue name via the `CELERY_WORKER_QUEUE` environment variable.

You can scale the parallel work-load by scaling the amount of workers.

Long-polling queue

This queue is intended for the long-polling tasks, which can run up to 30 minutes. Regular work may not be scheduled on this queue, as it might be blocked behind such a long-polling job.

We recommend running two workers for high-availability set-up, but one should work too.

```
export CELERY_WORKER_QUEUE=long-polling
/celery_worker.sh
```

Worker queue

The worker queue is intended for jobs that should run asynchronously, but still complete in a matter of seconds.

```
export CELERY_WORKER_QUEUE=celery
/celery_worker.sh
```

Celery beat

Beat is used to periodically kick off tasks, you can compare it a little to cronjobs. It ensures that the long-polling is initially started, and re-started in case a crash happens.

```
/celery_beat.sh
```

Celery monitoring

Flower is used for task monitoring. You should carefully protect the endpoint where Flower is hosted, as it gives insight into the app settings. It's meant for troubleshooting and should be developer/ops-only access.

```
/celery_flower.sh
```

2.2.3 Recap

If you're running 100% on Docker, for a single BPTL instance you would have:

- 1 PostgreSQL database container
- 1 Redis container
- 1 web worker
- 1 celery beat
- 2 celery workers, `long-polling` queue
- 3 celery workers, `celery` queue
- 1 celery flower
- nginx on the host system or a suitable Kubernetes Ingress solution

2.3 Third party documentation

BPTL integrates with third parties. Sometimes, these third parties need configuration on their end.

2.3.1 ValidSign

Configuration

BPTL needs to receive callbacks from ValidSign.

1. Navigate to the admin > **ValidSign configuration**
2. There is a generated authentication key, and the callback URL you will need in ValidSign
3. Navigate to the ValidSign dashboard. From there, navigate to the **admin**
4. Click **Integration**
5. Enter the callback URL and authentication key in the relevant fields
6. Select the **Transaction completed** event

You also need to configure the ValidSign API key with the service in BPTL:

1. See the integrator guide (page 10) on where you can find your API key
2. In BPTL, navigate to the admin > **Services**
3. Add a service, with the following fields:
    - Label: *ValidSign* (for example)
    - Type: ORC (Overige)
    - API root URL: `https://try.validsign.nl/` (sandbox) or `https://my.validsign.nl/` (production)
    - Authorization type: API key
    - Header key: `Authorization`
    - Header value: `Basic <api key>`
    - OAS: `https://apidocs.validsign.nl/validsign_openapi.yaml`

Connecting to a topic

When you connect a topic name and the valid sign task(s), you must add the ValidSign service with the alias `ValidSignAPI`.

Integration

BPTL can automate ValidSign package/transaction creation and configuration.

The *bptl.work_units.valid_sign.tasks.CreateValidSignPackageTask* takes documents and signer information as input, and performs the following actions:

1. A package is created. The signers specified in the task process variables are included in the package when it is created.
2. The documents are added to the package. All documents specified in the process variables are retrieved from their respective API. For each document, an 'approval' is created. This is a field where a signer will be able to sign. The approval is a field of dimensions 50x150 (px?) placed by the bottom left corner of the first occurrence of the string `Capture Signature`.
3. The package status is changed to SENT. This automatically sends an email to the signers with links to where they can sign the documents.
4. Once everyone has signed the package, ValidSign sends a callback to BPTL
5. BPTL processes the callback, and if configured, sends a BPMN message back to the process instance (Camunda only).

2.3.2 Xential

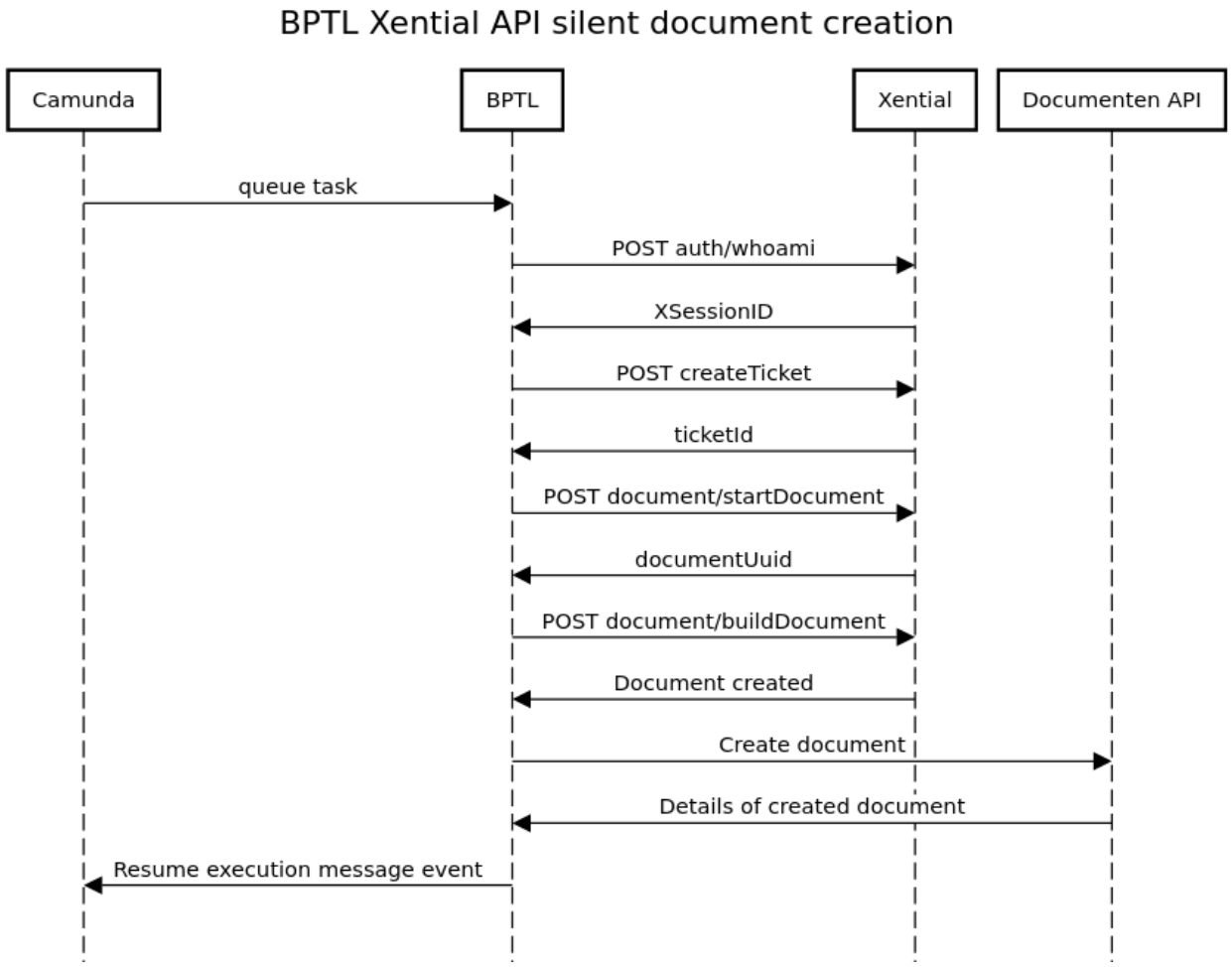
Integration

BPTL can automate document creation using Xential templates.

The *bptl.work_units.xential.tasks.start_xential_template()* work-unit needs to know the UUID of the template to use and whether to build the document interactively (the user fills the empty fields in the template) or 'silently'. In the latter case, values to fill the template fields also *need* to be provided to the work-unit. For interactive documents, they *may* be provided. Once the document is built, Xential sends it to BPTL, who then stores it in the Documenten API.

The workflow for both the interactive and silent document creation is explained in more details below.

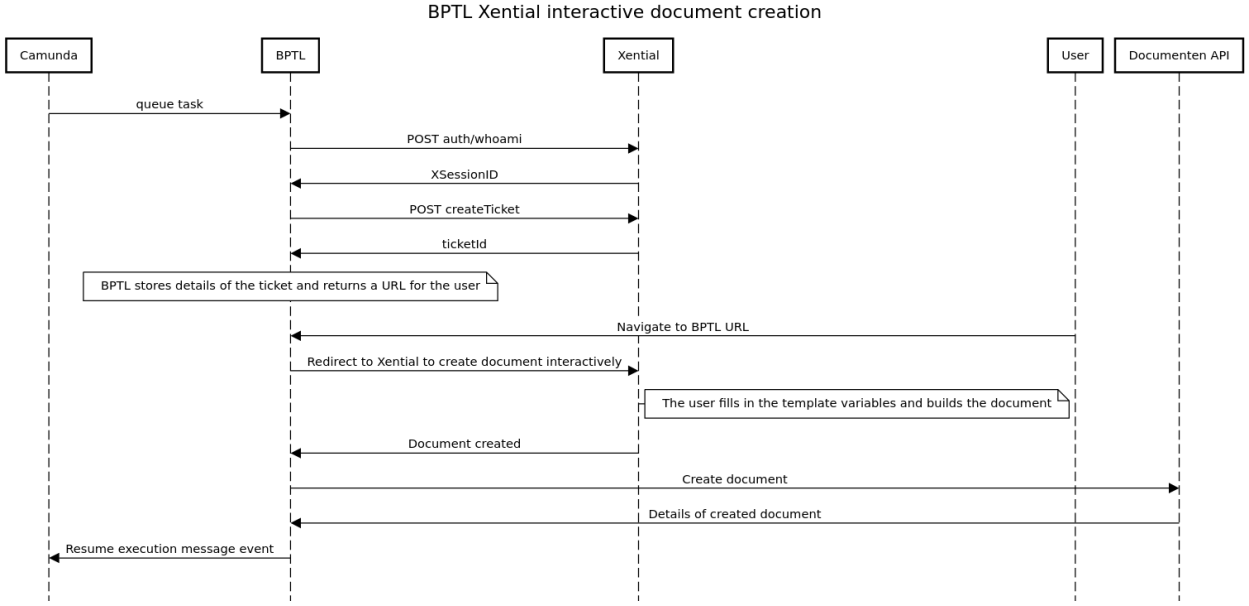
Silent document creation



The steps are as follows:

- BPTL requests the `XSessionId` from the Xential API. This is then included in every request.
- BPTL creates a ticket. The values to use to fill the template must be specified, as well as the template UUID and the URL of the webhook.
- BPTL starts the procedure to create a document. Xential returns the document UUID as well as a URL that can be used for interactively building the document (but this URL expires after 15 min).
- BPTL tells Xential to build the document. Once the document is successfully built, Xential sends it back to BPTL.
- BPTL sends the document to the Documenten API. Depending on the configuration, it can send a message to camunda to resume execution.

Interactive document creation



The interactive creation of a document involves more steps:

- BPTL requests the `XSessionId` from the Xential API. This is then included in every request.
- BPTL creates a ticket. The template UUID and the URL of the webhook need to be specified. BPTL stores the data related to this ticket and returns a BPTL URL to the user.
- When the user navigates to the BPTL URL, BPTL starts the procedure to create a document. Like in the silent case, Xential returns the document UUID as well as a URL that can be used for interactively building the document. This URL expires after 15 min. BPTL redirects the user to the Xential URL.
- Once the user has finished filling in the template and builds the document, Xential sends the document to the BPTL webhook.
- BPTL sends the document to the Documenten API. Depending on the configuration, it can send a message to camunda to resume execution.

Failures

A periodic task is configured to run every 12 hours to check for Xential errors.

Xential has an endpoint that can be queried to check the status of a particular document build. For both interactive and silent document creation, if an error occurs during the document build Xential changes the status of the document from `NONE` to `ERROR`.

The periodic task in BPTL looks for all open tickets with an associated document UUID. It then requests the status of each document from Xential. If any document has an `ERROR` status, the BPTL task is marked as failed.

2.4 Camunda support

2.4.1 Management commands

fetch_and_lock_tasks

This command fetches and locks a number of external tasks for futher processing, from the Camunda instance. The Camunda instance decides which tasks you get returned.

In its current form, only the topic `zaak-initialize` is recognized. Topic names are required input parameters for the Camunda API call, which will be made dynamic in future iterations.

The task is locked for 10 minutes in its current implementation, and fetched tasks are visible in the admin interface.

Example:

```
python src/manage.py fetch_and_lock_tasks 1
```

2.4.2 Python API

Complete tasks

Whenever an external task for a certain topic is done/performed, the task itself needs to be completed and updated with resulting process variables.

For this purpose, `bptl.camunda.utils.complete_task` exists. Pass the `FetchedTask` instance and a dict of `variable_name: value` to update process variables. If no process variables need to be updated, you can leave the `variables` off.

Note that this needs to happen within the expiry time for the tasks - when a task is fetched and locked, the lock expires after a while. You can verify this in the admin.

2.5 Public API

2.5.1 Tasks and task registry

Expose the public API to manage tasks.

exception bptl.tasks.api.**NoCallback**

exception bptl.tasks.api.**TaskExpired**

exception bptl.tasks.api.**TaskPerformed**

bptl.tasks.api.**execute**(*task: bptl.tasks.models.BaseTask*, *registry: bptl.tasks.registry.WorkUnitRegistry = <bptl.tasks.registry.WorkUnitRegistry object>*) → dict

Execute the appropriate task for a fetched external task.

This function takes care of looking up the appropriate handler for a task from the registry, and then calls it, passing the fetched task argument.

Parameters

- **task** – A `BaseTask` instance, that may not have expired yet.

- **registry** – A `bptl.tasks.registry.TaskRegistry` instance. This is the registry that will be used to find the corresponding callback for the topic name. Defaults to the default sentinel registry, mostly useful for tests.

Raises *TaskExpired* if the task is already expired, this exception is raised. You will need to re-fetch and lock the task before you can process it.

Raises *NoCallback* if no callback could be determined for the topic.

Raises *TaskPerformed* if the task is already completed, this exception is raised.

2.5.2 Work units

Work units are python callbacks which process tasks from external engines. They are engine independent and can be python functions or classes. Work units are registered in the registry.

API's voor Zaakgericht Werken

class `bptl.work_units.zgw.tasks.zaak.`**CloseZaakTask**(*task:* bptl.tasks.models.BaseTask)

Close the ZAAK by setting the final STATUS.

A ZAAK is required to have a RESULTAAT.

Required process variables

- `zaakUrl`: full URL of the ZAAK
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables

- `resultaattype`: full URL of the RESULTAATTYPE to set. If provided the RESULTAAT is created before the ZAAK is closed

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets the process variables

- `einddatum`: date of closing the zaak
- `archiefnominatie`: shows if the zaak should be destroyed or stored permanently
- `archiefactiedatum`: date when the archived zaak should be destroyed or transferred to the archive

class `bptl.work_units.zgw.tasks.zaak.`**CreateZaakTask**(*task:* bptl.tasks.models.BaseTask)

Create a ZAAK in the configured Zaken API and set the initial status.

The initial status is the STATUSTYPE with `volgnummer` equal to 1 for the ZAAKTYPE.

By default, the `registratiedatum` and `startdatum` are set to todays date.

Required process variables

- `zaaktype`: the full URL of the ZAAKTYPE
- `organisatieRSIN`: RSIN of the organisation
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.

- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables

- `NLXProcessId`: a process id for purpose registration ("doelbinding")
- `NLXSubjectIdentifier`: a subject identifier for purpose registration ("doelbinding")
- `zaakDetails`: a JSON object with extra properties for zaak creation. See https://zaken-api.vng.cloud/api/v1/schema/#operation/zaak_create for the available properties. Note that you can use these to override `zaaktype`, `bronorganisatie`, `verantwoordelijkeOrganisatie`, `registratiedatum` and `startdatum` if you'd require so.
- `initialStatusRemarks`: a text to use for the remarks field on the initial status. Must be maximum 1000 characters.
- **`initiator`: a JSON object with data used to create a rol for a particular zaak. See** https://zaken-api.vng.cloud/api/v1/schema/#operation/rol_create for the properties available.

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets the process variables

- `zaak`: the JSON response of the created ZAAK
- `zaakUrl`: the full URL of the created ZAAK
- `zaakIdentificatie`: the identificatie of the created ZAAK

class bptl.work_units.zgw.tasks.zaak.**LookupZaak**(*task:* bptl.tasks.models.BaseTask)

Look up a single ZAAK by identificatie and bronorganisatie.

This task looks up the referenced zaak, and if found sets the zaakUrl as a process variable. If not found, the variable will be empty.

You can use this to check if the referenced ZAAK does indeed exist, and relate it to other objects.

Required process variables

- `identificatie`: identification of the zaak, commonly known as "zaaknummer"
- **`bronorganisatie`: RSIN of the source organization for the zaak. The combination** of identificatie and bronorganisatie uniquely identifies a zaak.
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets the process variables

- **`zaakUrl`: the URL reference of the retrieved zaak, if retrieved at all. If the** zaak was not found, the value will be `null`

class bptl.work_units.zgw.tasks.status.**CreateStatusTask**(*task:* bptl.tasks.models.BaseTask)

Create a new STATUS for the ZAAK in the process.

Required process variables

- `zaakUrl`: full URL of the ZAAK to create a new status for
- `statusVolgnummer`: volgnummer of the status type as it occurs in the catalogus OR

- `statustype`: full URL of the STATUSTYPE to set
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Note that either `statusVolgnummer` or `statustype` are sufficient.

Optional process variables

- `toelichting`: description of the STATUS

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets the process variables

- `statusUrl`: the full URL of the created STATUS

class bptl.work_units.zgw.tasks.resultaat.**CreateResultaatTask**(*task:* bptl.tasks.models.BaseTask)

Set the RESULTAAT for the ZAAK in the process.

A resultaat is required to be able to close a zaak. A zaak can only have one resultaat.

Required process variables

- `zaakUrl`: full URL of the ZAAK to set the RESULTAAT for
- `resultaattype`: full URL of the RESULTAATTYPE to set
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables

- `toelichting`

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets the process variables

- `resultaatUrl`: the full URL of the created RESULTAAT

class bptl.work_units.zgw.tasks.zaak_relations.**CreateEigenschap**(*task:* bptl.tasks.models.BaseTask)

Set a particular EIGENSCHAP value for a given zaak.

Unique eigenschappen can be defined for a given zaaktype. This task looks up the eigenschap reference for the given zaak and will set the provided value.

Required process variables

- `zaakUrl`: URL reference to a ZAAK in a Zaken API. The eigenschap is created for this zaak.
- `eigenschap`: a JSON Object containing the name and value:

```
{
    "naam": "eigenschapnaam as in zaaktypecatalogus",
    "waarde": "<value to set>"
}
```

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables

- `NLXProcessId`: a process id for purpose registration ("doelbinding")
- `NLXSubjectIdentifier`: a subject identifier for purpose registration ("doelbinding")

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets no process variables

`class bptl.work_units.zgw.tasks.zaak_relations.CreateZaakObject(`*task:* bptl.tasks.models.BaseTask`)`

Create a new ZAAKOBJECT for the ZAAK in the process.

Required process variables

- `zaakUrl`: full URL of the ZAAK to create a new ZaakObject for
- `objectUrl`: full URL of the OBJECT to set
- `objectType`: type of the OBJECT
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

If *zaakUrl* is not given - returns empty dictionary.

Optional process variables

- `objectTypeOverige`: description of the OBJECT type if objectType = 'overige'
- `relatieomschrijving`: description of relationship between ZAAK and OBJECT

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets the process variables

- `zaakObjectUrl`: the full URL of the created ZAAKOBJECT

`class bptl.work_units.zgw.tasks.zaak_relations.RelateDocumentToZaakTask(`*task:* bptl.tasks.models.BaseTask`)`

Create relations between ZAAK and INFORMATIEOBJECT

Required process variables

- `zaakUrl`: full URL of the ZAAK
- **`informatieobject`: full URL of the INFORMATIEOBJECT. If empty, no relation** will be created.
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets the process variables

- `zaakinformatieobject`: full URL of ZAAKINFORMATIEOBJECT

class bptl.work_units.zgw.tasks.zaak_relations.**RelatePand**(*task:* bptl.tasks.models.BaseTask)
Relate Pand objects from the BAG to a ZAAK as ZAAKOBJECTs.

One or more PANDen are related to the ZAAK in the process as ZAAKOBJECT.

Required process variables

- `zaakUrl`: URL reference to a ZAAK in a Zaken API. The PANDen are related to this.
- `panden`: list of URL references to PANDen in BAG API.
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables

- `NLXProcessId`: a process id for purpose registration ("doelbinding")
- `NLXSubjectIdentifier`: a subject identifier for purpose registration ("doelbinding")

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets no process variables

class bptl.work_units.zgw.tasks.zaak_relations.**RelateerZaak**(*task:* bptl.tasks.models.BaseTask)
Relate a zaak to another zaak.

Different kinds of relations are possible, specifying the relation type will ensure this is done correctly. Existing relations are not affected - if there are any, they are retained and the new relation is added.

Required process variables

- `hoofdZaakUrl`: URL reference to a ZAAK in a Zaken API. This zaak receives the relations.
- `zaakUrl`: URL reference to another ZAAK in a Zaken API, to be related to `zaakUrl`.
- `bijdrageAard`: the type of relation. One of `vervolg`, `onderwerp` or `bijdrage`.
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Optional process variables

- `NLXProcessId`: a process id for purpose registration ("doelbinding")
- `NLXSubjectIdentifier`: a subject identifier for purpose registration ("doelbinding")
- `bijdrageAardOmgekeerdeRichting`: the type of reverse relation. One of `vervolg`, `onderwerp`, `bijdrage` or empty (`""`). Default is `onderwerp` if the process variable isn't given.

Optional process variables (Camunda exclusive)

- `callbackUrl`: send an empty POST request to this URL to signal completion

Sets no process variables

class bptl.work_units.zgw.tasks.documents.**GetDRCMixin**
Temp workaround to get credentials for the relevant DRC.

The services var should contain a DRC alias key with credentials, but that's currently a massive spaghetti. So, we'll allow for the time being that DRCs are all configured in BPTL, and we grab the right one from the document URL.

class bptl.work_units.zgw.tasks.documents.**LockDocument**(*task:* bptl.tasks.models.BaseTask)
Lock a Documenten API document.

A locked document cannot be mutated without having the lock ID.

Required process variables

- `informatieobject`: String, API URL of the document to lock. The API must comply with the Documenten API 1.0.x (https://vng-realisatie.github.io/gemma-zaken/standaard/documenten/index).
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Sets the process variables

- `lockId`: String, Lock ID for the locked document. Required to unlock or mutate the document.

class bptl.work_units.zgw.tasks.documents.**UnlockDocument**(*task:* bptl.tasks.models.BaseTask)
Unlock a Documenten API document.

Required process variables

- `informatieobject`: String, API URL of the document to lock. The API must comply with the Documenten API 1.0.x (https://vng-realisatie.github.io/gemma-zaken/standaard/documenten/index).
- `lockId`: String, Lock ID for the locked DRC document, obtained from locking the document.
- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls.
- `services`: DEPRECATED - support will be removed in 1.1

Sets no process variables

BRP

class bptl.work_units.brp.tasks.**DegreeOfKinship**(*task:* bptl.tasks.models.BaseTask)
Retrieve the degree of kinship from the BRP API.

Required process variables

- `burgerservicenummer1`: BSN of the first person
- `burgerservicenummer2`: BSN of the second person

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the process variables

- **`kinship`: integer, which represents the degree of kinship (blood relations). Values can be in** range [1..4] or `Null` if the BSNs are identical.

class bptl.work_units.brp.tasks.**IsAboveAge**(*task:* bptl.tasks.models.BaseTask)

Fetches BRP API and returns whether a person is exactly, or older than, a certain age.

Required process variables

- `burgerservicenummer`: BSN of the person
- `age`: integer, which represents the number of years

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

The task sets the process variables

- `isAboveAge`: boolean, which indicate if the requested person is equal or above a certain age. If the information about person's age is not found, `isAboveAge` will be set as `none`

Kadaster

bptl.work_units.kadaster.tasks.**retrieve_openbare_ruimten**(*task:* bptl.tasks.models.BaseTask) → Dict[str, Any]

Given a bounding box (or other polygon), retrieve the 'public space' objects contained/overlapping.

This consumes the BRT API to fetch relevant objects, which are returned so that they can be drawn/selected on maps as GeoJSON.

Checked resources:

- Wegdeel
- Terrein (in development)
- Inrichtingselement (in development)

Required process variables

- `geometry`: A GeoJSON geometry that is checked for overlap.

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the following return/process variables

- `features`: a list of GeoJSON features, in EPSG:4258 CRS. Properties contain feature-specific keys/values.

Note: The kadaster geo query APIs have long response times (up to 40s) - this work unit takes a considerable time to execute.

Camunda

class bptl.work_units.camunda_api.tasks.**CallActivity**(*task:* bptl.tasks.models.BaseTask)
Start subprocess in Camunda

Required process variables

- **`subprocessDefinition`: process definition key for the target subprocess to** start.

Optional process variables

- **`subprocessDefinitionVersion`: a specific version of the deployed subprocess.** defaults to `latest` if not set, which means the process will be kicked off by definition key.
- **`variablesMapping`: JSON object to map variables from the parent process** to be sent into the new subprocess. If renaming is not needed, use the same name as a key and a value. If `variablesMapping` is empty, the all parent variables are sent to subprocess unchanged.

```
{
    "<source variable name>": "<target variable name>",
}
```

Sets the process variables

- `processInstanceId`: instance id of the created subprocess

ValidSign

class bptl.work_units.valid_sign.tasks.**CreateValidSignPackageTask**(*task:* bptl.tasks.models.BaseTask)
Create a ValidSign package with signers and documents and send a signing request to the signers.

Required process variables

- **`documents`: List of strings. List of API URLs where the documents to be signed can be retrieved.** The API must comply with the Documenten API 1.0.x (https://vng-realisatie.github.io/gemma-zaken/standaard/documenten/index).
- **`signers`: JSON list with signers information. For ValidSign, the first name, the last name and the** email address of each signer are required. Example `signers`:

```
[{
    "email": "example.signer@example.com",
    "firstName": "ExampleFirstName",
    "lastName": "ExampleLastName"
},
{
    "email": "another.signer@example.com",
    "firstName": "AnotherFirstName",
    "lastName": "AnotherLastName"
}]
```

- **`packageName`: string. Name of the ValidSign package that contains the documents to sign and the signers.** This name appears in the notification-email that is sent to the signers.
- `services`: JSON Object of connection details for ZGW services:

```
{
    "<drc alias1>": {"jwt": "Bearer <JWT value>"},
    "<drc alias2>": {"jwt": "Bearer <JWT value>"}
}
```

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.
- **`messageId`: string. The message ID to send back into the process when the** package is signed by everyone. You can use this to continue process execution. If left empty, then no message will be sent.

Sets the process variables

- `packageId`: string. ID of the ValidSign package created by the task.

add_documents_and_approvals_to_package(*package: dict*) → List[dict]
Add documents and approvals to the package.

create_package() → dict
Create a ValidSign package with the name specified by the process variable and add the signers to it.

format_signers(*signers: List[dict]*) → List[dict]
Format the signer information into an array of JSON objects as needed by ValidSign.

send_package(*package: dict*)
Change the status of the package to 'SENT'

When the status of the package is changed, an email is automatically sent to all the signers with a link where they can sign the documents.

exception bptl.work_units.valid_sign.tasks.**DoesNotExist**

exception bptl.work_units.valid_sign.tasks.**NoAuth**

exception bptl.work_units.valid_sign.tasks.**NoService**

class bptl.work_units.valid_sign.tasks.**ValidSignReminderTask**(*task:* bptl.tasks.models.BaseTask)
Email a reminder (with links) to signers that they need to sign documents through ValidSign.

Required process variables

- `packageId`: string with the ValidSign Id of a package
- `email`: the email address of the signer who needs a reminder

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets no process variables

Email

class bptl.work_units.email.tasks.**SendEmailTask**(*task:* bptl.tasks.models.BaseTask)

This task sends an email to `receiver` signed by `sender`.

Required process variables

- `sender`: JSON with required fields email and name of sender.

```
{
  "email": "kees@example.com",
  "name": "Kees Koos"
}
```

- `receiver`: JSON with required fields email and name of receiver.

```
{
  "email": "jan@example.com",
  "name": "Jan Janssen"
}
```

- `email`: JSON with required fields email subject and email content:

```
{
  "subject": "This is an example subject.",
  "content": "This is an example body."
}
```

- `template`: string with template name. Valid choices are:

```
[
  "generiek",
  "accordering",
  "advies",
  "nen2580"
]
```

- `context`: JSON with optional fields:

```
{
  "kownslFrontendUrl": "https://kownsl.utrechtproeftuin.nl/kownsl/<uuid>/
↪",
  "deadline`": "2020-04-20"
}
```

Kownsl

bptl.work_units.kownsl.tasks.**get_approval_status**(*task:* bptl.tasks.models.BaseTask) → dict

Get the result of an approval review request.

Once all reviewers have submitted their approval or rejection, derive the end-result from the review session. If all reviewers approve, the result is positive. If any rejections are present, the result is negative.

In the task binding, the service with alias `kownsl` must be connected, so that this task knows which endpoints to contact.

Required process variables

- `kownslReviewRequestId`: the identifier of the Kownsl review request.

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the process variables

- `approvalResult`: a JSON-object containing meta-data about the result:

```
{
   "approved": true,
   "num_approved": 3,
   "num_rejected": 0,
   "approvers": ["mpet001", "will002", "jozz001"]
}
```

bptl.work_units.kownsl.tasks.**get_approval_toelichtingen**(*task:* bptl.tasks.models.BaseTask) → dict

Get the "toelichtingen" of all reviewers that responded to the review request.

Required process variables

- `kownslReviewRequestId`: the identifier of the Kownsl review request.

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the process variables

- `toelichtingen`: a string containing the "toelichtingen" of all reviewers.

bptl.work_units.kownsl.tasks.**get_email_details**(*task:* bptl.tasks.models.BaseTask) → dict

Get email details required to build the email that is sent from the accordeer/adviseer sub processes in Camunda.

Required process variables

- `kownslReviewRequestId`: the identifier of the Kownsl review request.
- `deadline`: deadline of the review request.
- `kownslFrontendUrl`: URL that takes you to the review request.

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the process variables

- `email`: a JSON that holds the email content and subject.

```
{
    "subject": "Email subject",
    "content": "Email content",
}
```

- `context`: a JSON that holds data relevant to the email:

```
{
    "deadline": "2020-12-31",
    "kownslFrontendUrl": "somekownslurl",
}
```

- `template`: a string that determines which template will be used for the email.
- `senderUsername`: a list that holds a string of the review requester's username. This is used to determine the email's sender's details.

`bptl.work_units.kownsl.tasks.`**`get_review_request_reminder_date`**(*task:* bptl.tasks.models.BaseTask) → dict

Get the reminder for the set of reviewers who are requested. The returned value is the deadline minus one day.

In the task binding, the service with alias `kownsl` must be connected, so that this task knows which endpoints to contact.

Required process variables

- `kownslReviewRequestId`: the identifier of the Kownsl review request.
- `kownslUsers`: list of usernames that have been configured in the review request configuration.

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the process variables

- `reminderDate`: a string containing the reminder date: "2020-02-29".
- `deadline`: a string containing the deadline date: "2020-03-01".

`bptl.work_units.kownsl.tasks.`**`get_review_response_status`**(*task:* bptl.tasks.models.BaseTask) → dict

Get the reviewers who have not yet responded to a review request so that a reminder email can be sent to them if they exist.

In the task binding, the service with alias `kownsl` must be connected, so that this task knows which endpoints to contact.

Required process variables

- `kownslReviewRequestId`: the identifier of the Kownsl review request.
- `kownslUsers`: list of usernames that have been configured in the review request configuration.

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the process variables

- `remindThese`: a JSON-object containing a list of usernames who need reminding:

```
[
    "user1",
    "user2",
]
```

`bptl.work_units.kownsl.tasks.`**`set_review_request_metadata`**(*task:* bptl.tasks.models.BaseTask) → dict

Set the metadata for a Kownsl review request.

Metadata is a set of arbitrary key-value labels, allowing you to attach extra data required for your process routing/handling.

Required process variables

- `kownslReviewRequestId`: the identifier of the Kownsl review request.
- `metadata`: a JSON structure holding key-values of the metadata. This will be set directly on the matching review request. Example:

```
{
    "processInstanceId": "aProcessInstanceId"
}
```

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets no process variables

Xential

bptl.work_units.xential.tasks.**start_xential_template**(*task:* bptl.tasks.models.BaseTask) → dict

Run Xential template with requested variables.

If the `interactive` task variable is:

- `True`: it returns a URL in `bptlDocumentUrl` for building a document interactively
- `False`: it returns an empty string in `bptlDocumentUrl`

In the task binding, the service with alias `xential` must be connected, so that this task knows which endpoints to contact.

Required process variables

- `bptlAppId`: the application ID in the BPTL credential store
- `templateUuid`: the id of the template which should be started
- `interactive`: bool, whether the process will be interactive or not
- `templateVariables`: a JSON-object containing the data to fill the template. In an interactive flow, this can be an empty object `{}`:

```
{
   "variable1": "String",
   "variable2": "String"
}
```

- `documentMetadata`: a JSON-object containing the fields required to create a document in the Documenten API. The fields shown below are required. The property 'creatiedatum' defaults to the day in which the document is sent to the Documenten API and the property 'taal' defaults to 'nld' (dutch).

```
{
   "bronorganisatie": "string",
   "titel": "string",
   "auteur": "string",
```

(continues on next page)

(continued from previous page)

```
    "informatieobjecttype": "url"
}
```

Optional process variable

- `messageId`: string. The message ID to send back into the process when the document is sent to the Documenten API. You can use this to continue process execution. If left empty, then no message will be sent.

Sets the process variable

- `bptlDocumentUrl`: BPTL specific URL for interactive documents. If the document creation is not interactive, this will be empty.

ZAC

class bptl.work_units.zac.tasks.**UserDetailsTask**(*task:* bptl.tasks.models.BaseTask)

Requests email and name data from usernames from the zac and feeds them back to the camunda process.

Required process variables

- `usernames`: JSON with usernames.

```
[
    "user1",
    "user2",
    "user3"
]
```

OR

- `emailaddresses`: JSON with email addresses.

```
[
    "user1@email",
    "user2@email"
]
```

Optional process variables

- `bptlAppId`: the application ID of the app that caused this task to be executed. The app-specific credentials will be used for the API calls, if provided.

Sets the process variables

- `userData`: a JSON-object containing a list of user names and emails:

```
[
    {
        "name": "FirstName LastName",
        "username": "username",
        "email": "test@test.nl"
    }
]
```

2.5.3 Camunda tasks

Module for Camunda API interaction.

bptl.camunda.utils.**complete_task**(*task: bptl.camunda.models.ExternalTask*, *variables: Optional[Dict[str, Union[str, int, bool]]] = None*) → None

Complete an External Task, while optionally setting process variables.

API reference: https://docs.camunda.org/manual/7.12/reference/rest/external-task/post-complete/

If a task variable `callbackUrl` is available, a post request is made to it.

Note that we currently only support setting process variables and not local task variables.

Camunda performs optimistic table locking, see the docs. This results in HTTP 500 exceptions being thrown when concurrent mutations to the process instance happen. The recommended way to deal with this by Camunda is to retry the operation to reach eventual consistency, which is why the `@retry` decorator applies.

bptl.camunda.utils.**fail_task**(*task: bptl.camunda.models.ExternalTask*, *reason: str = ''*) → None

Mark an external task as failed.

See https://docs.camunda.org/manual/7.11/reference/rest/external-task/post-failure/

When the number of retries becomes 0, an incident is created in Camunda.

bptl.camunda.utils.**fetch_and_lock**(*max_tasks: int*, *long_polling_timeout=None*) → Tuple[str, int, list]

Fetch and lock a number of external tasks. API reference: https://docs.camunda.org/manual/7.12/reference/rest/external-task/fetch/

CHAPTER

THREE

DEVELOPERS

3.1 Architecture

BPTL is a middle-man in your application landscape. It "talks" to APIs or performs task when asked to do so.

A typical layout of your application landscape would be the following set-up:

- a number of user-facing applications start process instances - they communicate with the API of your process engine (e.g. Camunda)
- process definitions can change as often as needed because of business needs
- processes require input or processing from certain data-sources that you wish to automate
- data needs to be stored in the appropriate locations

BPTL solves the last two items - it helps automating *very specific* tasks that are too complex for BPMN, but not complex enough to warrant an entire, dedicated application.

3.1.1 BPTL Components

BPTL consists of a number of components that make it work for various use cases.

Work units

Work units are logical units of work that can be performed. This can be a collection of API calls, for example to create a **Zaak**, or to check if someone's age is above a certain number, using the BRP API's. These are the steps that you want to "embed" in your process.

Work units are grouped around themes, such as the ZGW APIs, the BRP, Camunda API or the Kadaster APIs.

Work units are implemented in Python code.

Web interface

The web interface allows you to configure work-units to a certain topic. This way, you can use meaningful names in your process, or decide to only let BPTL handle a subset of topics relevant for you, and another solution for other specialized topics.

Additionally, the web interface provides you monitoring and debug-information for if/when something goes wrong.

Workers

Workers are responsible for performance of the work-units. Whenever a task is picked up from the task queue, a worker is assigned to execute it. Workers can be scaled independently from the web-interface, and they prevent the web-interface from locking up during long-running tasks.

Beat

Beat is used to periodically fire tasks that workers need to perform. Beat is essential to poll Camunda for new work to assign to the workers.

Task monitoring

The communication between web, workers and beat is monitored to see if tasks get dropped or investigating where scaling is needed.

Timeline

A typical timeline is the following:

1. Process execution is started
2. Process execution arrives at an external task
3. External task is put on the queue
4. BPTL polling picks up the queued task
5. BPTL assigns the task to a worker
6. BPTL worker performs the related work unit
7. BPTL worker marks the task as completed
8. Process execution continues to the next waiting point

3.1.2 Process engines

Currently, two process engines are supported to varying degrees:

- Camunda: arguably the most fleshed out, and the target architecture
- Activiti: a proof of concept showed promising results

3.1.3 Camunda architecture

The above *Timeline* describes Camunda architecture.

Camunda uses a service-task implementation called **External Task**. Whenever a process execution arrives at an external task, the task is put on a queue with its *topic name*.

BPTL periodically polls the Camunda queue for work, and it does so by only asking about topics that BPTL is configured to handle.

Whenever work is picked up, the task is locked and handled by BPTL. BPTL either completes it and sets the relevant process variables, or marks the task as failed if errors occur. The failure information is visible in BPTL monitoring and in the Camunda cockpit.

3.1.4 REST-full API architecture

Activiti does not use a queue to schedule work. Instead, you can include REST-call activities in the process definition. BPTL offers a REST-full API to call/execute work units, using a similar format to Camunda's external tasks.

The API endpoints can also be used by other applications who wish to re-use the building blocks offered by BPTL.

In this configuration, the workers, beat and task monitoring are not relevant.

3.2 Work units

Work units are the building blocks of BPTL. They are the smallest units that can be executed by themselves, while having sufficient meaning.

Work units typically require input variables, process these and do some work, and finally (optionally) return output variables.

3.2.1 Work unit interface

Work units have two possible interface: function or class based. Function based work units are easiest to reason about, while class-based units are suited to more complex units.

Function based

A function based unit follows the following pattern:

```
def some_work_unit(task):
    # ... extract relevant variables

    # ... perform work

    return {"foo": "bar"}  # return relevant result variables
```

See for example `bptl.dummy.tasks.dummy`.

Class based

Class based work units allow you to split up work into methods.

Example:

```
from bptl.tasks.base import WorkUnit


class MyWorkUnit(WorkUnit):

    def perform(self):
        # ... extract relevant variables

        # ... perform work

        return {"foo": "bar"}  # return relevant result variables
```

The unit constructor receives the task instance as sole argument.

3.2.2 Registering work units

Work units can be contributed to BPTL, or can be defined in third-party packages.

Autodiscover

Work units are auto-discovered for Django apps in the `tasks` module, so make sure to:

1. Add your app to `INSTALLED_APPS`
2. Define your units in `myapp.tasks`

or, alternatively, you can use the `ready` hook in your `AppConfig` to import the relevant tasks module.

Registration

Registering work units is done by decorating them with `bptl.tasks.registry.register`, which is the default registry:

```
from bptl.tasks.registry import register

@register
class SomeWorkUnit(WorkUnit):
    ...


@register
def another_work_unit(task):
    ...
```

Defining required services

Work units often interact with various external services, which require authentication. You can declare which type of services with which aliases are required for a work-unit, and then safely use those aliases in the code to build a client and retrieve credentials.

The forms to configure task mappings will validate that the declared required services are configured correctly.

Example:

```
from bptl.tasks.registry import register

@register
@register.require_service("zrc", "The Zaken API to use", alias="zrc")
def some_work_unit(task):
    service = DefaultService.objects.get(
        task_mapping__topic_name=task.topic_name,
        alias="zrc"
    ).service
    ...
```

The decorator is currently only used for form validation.

bptl.tasks.registry.register.**require_service**(*service_type: str*, *description: str = ''*, *alias: str = ''*)
Decorate a callback with the required service definitions.

Used to validate the task mappings to ensure the required services are present. This self-documents which service aliases must be used for the callback to be able to function.

Authenticating in a work unit

BPTL executes work units on behalf of another application, often through a process engine. For auditing purposes, you should not interface to external services with "blanket" BPTL credentials, but instead use application specific credentials.

BPTL has a credential store containing the credentials for a particular application (identified by an "App ID") for each service it needs to interact with. To use this, you must:

1. Extract the `bptlAppId` process variable from the task:

```
@register
def some_work_unit(task):
    app_id = check_variable(task.get_variables(), "bptlAppId")
```

2. Determine the required service(s):

```
@register
@register.require_service("zrc", "The Zaken API to use", alias="zrc")
@register.require_service("drc", "The Documenten API to use", alias="drc")
def some_work_unit(task):
    app_id = check_variable(task.get_variables(), "bptlAppId")
    default_services = DefaultService.objects.get(
        task_mapping__topic_name=task.topic_name,
        alias__in=["zrc", "drc"]
    )
    services = {
        default_service.alias: default_service.service
        for default_service in default_services
    }
```

3. Obtain the application-specific credentials:

```
@register
@register.require_service("zrc", "The Zaken API to use", alias="zrc")
@register.require_service("drc", "The Documenten API to use", alias="drc")
def some_work_unit(task):
    app_id = check_variable(task.get_variables(), "bptlAppId")
    default_services = DefaultService.objects.get(
        task_mapping__topic_name=task.topic_name,
        alias__in=["zrc", "drc"]
    )
    services = {
        default_service.alias: default_service.service
        for default_service in default_services
    }
    auth_headers = get_credentials(app_id, services["zrc"], services["drc"])
```

(continues on next page)

(continued from previous page)

```
zrc_client = services["zrc"].build_client()
zrc_client.set_auth_value(auth_headers[services["zrc"]])

drc_client = services["drc"].build_client()
drc_client.set_auth_value(auth_headers[services["drc"]])
```

The public api to get the credentials is:

bptl.credentials.api.**get_credentials**(*app_id: str*, **services: zgw_consumers.models.Service*) → Dict[zgw_consumers.models.Service, Dict[str, str]]

3.2.3 Task interface

Work units receive the `task` instance that they should execute. This is always a subclass of *bptl.tasks.models.BaseTask*:

class bptl.tasks.models.**BaseTask**(**args*, ***kwargs*)

An external task to be processed by work units.

Use this as the base class for process-engine specific task definitions.

exception DoesNotExist

exception MultipleObjectsReturned

get_variables() → dict

return input variables formatted for work_unit

Subclasses are aimed at particular process engines, and are expected to implement the *bptl.tasks.models.BaseTask.get_variables()* interface correctly.

3.2.4 Best practices

Documentation

Document your work unit extensively! You can use RST - the docstring is extracted into the task documentation and displayed in the web-interface, admin, and even command line output. The recommended format is:

```
def work_unit(task):
    """
    Describe a short summary of what the task does.

    **Required process variables**

    * ``var``: a string representing an example

    **Optional process variables**

    * ``foo``: if provided, will summon Chtulhu

    **Optional process variables (engine specific)**

    * ``bar``: complex JSON variable with the following structure:
```

(continues on next page)

(continued from previous page)

```
    .. code-block:: json

        {"ok": "I lied"}

**Sets the process variables**

* ``quux``: PI with all decimals, ever

"""
```

Variable extraction

Use the *bptl.tasks.models.BaseTask.get_variables()* to obtain the variables. This takes care of deserialization into the appropriate Python data-type, and is responsible for abstracting away the differences between process engines.

Use `bptl.tasks.base.check_variable` to retrieve (soft-)required process variables:

`bptl.tasks.base.`**`check_variable`**(*variables: dict*, *name: str*, *empty_allowed=False*)

It will raise a clear error when a process variable is missing, and shortcuts the unit execution.

3.3 General information

This section briefly describes the project structure and framework that was used to built this project.

3.3.1 CSS

CSS code is generated by SASS, we use the .scss syntax. We prefer a component based approach using the BEM methodology: https://en.bem.info/methodology/key-concepts/.

In addition to BEM we limit the scope of components to the "border-box" preventing components from defining a margin on itself. Parent components control the margins of its children.

The view is a component without parents and should be directly linked to a template. It's role is to "orchestrate" child components.

To compile SASS to CSS run:

```
$ gulp sass
```

To create a new component run

```
$ gulp create-component --name my-compoment-name --scss
```

To create a new view run

```
$ gulp create-view --name my-compoment-name --scss
```

3.3.2 JavaScript

JavaScript code is written in ECMAScript 2015 (ES6) and transpiled using webpack and babel. Therefore, all non-compiled code is placed outside the static directory into `src/bptl/js/`.

We write modules for every component/view matching the BEM structure provides by SASS.

Compiling ES6 to ES5:

```
$ gulp js
```

To create a new component run

```
$ gulp create-component --name my-compoment-name --js
```

To create a new view run

```
$ gulp create-view --name my-compoment-name --js
```

All third party libraries should be installed using npm:

```
$ npm install --save <package>
```

or:

```
$ npn install --save-dev <package>
```

After installing libraries can be included using ES6 imports:

```
import <package> from '<package>';
```

Exceptions

When you need to override third-party JavaScript you still need to manually place files into `src/bptl/static/`.

3.4 Coding style

Below you can find some best practices to maintain a good coding style. There are detailed coding style guides for the *frontend* <*coding_style_frontend*> and the *backend* <*coding_style_backend*>.

3.4.1 Backend coding style

The django coding style is the basis for this styleguide. Some sections dive a bit deeper or put extra emphasis.

Imports

In short: use isort to check your import ordering. The config file is in `setup.cfg`.

Order and group your imports

- Use relative imports for your django app
- **Ordering:**
  - future
  - standard libraries
  - Django components
  - third party libraries
  - project imports
  - local (app) imports

Example:

```
from __future__ import absolute_import, unicode_literals

import datetime
from datetime import timedelta

import django.contrib.admin

import bptl.other_app.models

from .models import SomeModel
```

Naming

- Use plural form for apps. E.g.: `accounts`, not `account`.
- Use singular form for model, view and form class

Example:

```
from bptl.accounts.models import Account

class Idea(models.Model):
    pass

class IdeaForm(forms.ModelForm):
    pass

class IdeaDetailView(views.DetailView):
    pass
```

3.4.2 Frontend coding style

Shortcuts:

- *HTML*
- *Sass*
- *JavaScript*

HTML

Common

- Inline style is evil

```
<p style="color: red;">
Inline style cannot be cached.<br />
Inline style is difficult to overwrite.<br />
Inline style makes HTML less readable.<br />
Inline style is harder to spot.<br />
</p>
```

- Inline script is evil (except Google Analytics)

```
<script>
    console.log('Inline script cannot be cached.');
    console.log('Inline script makes HTML less readable.');
    console.log('Inline script blocks loading of page.');
</script>
```

- Style your HTML, don't HTML your style (avoid adding divs for style)

```
<div class="wrapper">
    <div class="inner">
        <div class="content">
            <p class="text">
                All these tags have no acutual meaning.<br />
                Consider HTML as data model, it should represent data, not style␣
↪placeholders.<br />
                Good practice is to write you HTML first, based on the structure of␣
↪the content, then style.<br />
                It's almost never needed to add more tags, have you tried :before␣
↪and :after yet?<br />
            </p>
        </div>
    </div>
</div>
```

- Empty newline at the end of the file.

Indentation

- Indent with 4 spaces

```
<html>
    <body>
    </body>
</html>
```

- Indent HTML and template tags. (except `{% block %}` on root level).

```
{% block content %}
<article>
    {% if show_header %}
        {% block article__header %}
            <header>
            </header>
        {% endblock article__header %}
    {% endif %}
</article>
{% endblock content %}
```

Data-attributes

- (Meta)data should be stored in data- attributes.

```
<article data-article-id="1">...</article>
```

- Variables should be passed using data-attributes as well. They are no excuse for inline script.

```
<article data-some-variable="1">...</article>
```

Elements

- Avoid the `id` attribute, unless there's a good reason.

```
<article id="article-1" />  <!-- wrong -->
<article class="article" data-id="1" /> <!-- better -->

<!-- ok, since it's useful in unit tests with WebTest -->
<form id="submit-article">...</form>
```

- Use semantic tags like `<main>`, `<nav>`, `<article>`, `<section>`, `<aside>`, `<footer>` instead of meaningless `<div>` s.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
        <main>
            <nav>
```

(continues on next page)

(continued from previous page)

```
            </nav>

            <article>
                <header></header>
                <section></section>
                <section></section>
                <footer></footer>
            </article>

            <footer>
            </footer>
        </main>
    </body>
</html>
```

Sass

Common

- Readability comes first
- Annotate when useful

Globals

- Avoid global styling - leave that to the CSS reset.
- Limit global configuration to: - Grid - Breakpoints - Colors - Font definitions

Indentation

- Indent using two spaces

```
.block {
  width: 100%;
}
```

Nesting

- Namespace BEM blocks

```
.block {
  // Everything should be nested inside .block
  // This makes sure no elements "bleed" to the global scope
  .block__element {
    ...
  }
}
```

- Nest maximum 3 levels deep

```
.block {             // One
  .block__element {  // Two
    &:hover {        // Three
      color: #0000FF;
    }
  }
}
```

Newlines

- 1 empty newline after mixin/variable block

```
.block__element-one {
}

.block__element-two {
}
```

- Empty newline at the end of the file.

Order

- Block modifiers come before block elements, element modifier come after the element. Example:

```
.block {  // .block is the basic element
  // --active is the modifier for .block, and should be grouped with .block
  &.block--active {
  }

  // __element is a child element dependent on .block
  .block__element {
  }

  // --disabled is the modifier for .block__element, and should be grouped with .
↪block__element
  .block__element--disabled {
  }
}
```

- Mixins always come first, and then group attributes logically.

  Mixins come first so that their behaviour can still be overridden. Logical groups are for example text styling and borders.

```
.block {
  @include span-columns(4 of 12);

  font-size: 18px;
  color: #FFF;
```

(continues on next page)

(continued from previous page)

```
  border: solid 1px #FFFF00;
  border-radius: 5px;
}
```

Selectors

- Use BEM class naming.

```
// BEM (Block, Element, Modifier) is a structured naming convention for CSS classes
// A double underscore (__) separates the element from a block
// A double dash (--) separates the modifier from the block or element
// These fixed patters make it also possible to be parsed by (JavaScript) code

.block {  // A block describes a standalone component
  &.block--modifier {  // A modifier describes a state or theme for eithe a block␣
↪or an element
  }

  .block__element {  // An element is a component that depends on a block
  }

  .block__element--modifier {  // This modifier desrcibes the state or theme for an␣
↪element
  }
}
```

- Maximum one BEM block per file

```
// file src/bptl/sass/components/blocks/_block.scss

.block {  // That's it, no more blocks in this file
  // ...
}
```

- Only select using (BEM) class names (.block__element), not using tag/id.

```
div {  // Bad, tags may change an that would break our code
}

article { // Also bad, event semantic (descriptive) tags may change
}

h1 {  // Also bad, a marketeer may drop in and ask you to change it into an h2␣
↪(design will break and designer will be mad)
}

#content {  // Bad, we can't repeat this anymore because id's must be unique
}

.content {  // Better, content is our block
  .content__heading {  // Better, content__heading is a valid class name for an h1,␣
↪or h2 in block content
```

(continues on next page)

(continued from previous page)

```
  }
  .content__body {  // This could be a class name for a paragraph in block content
  }
}

.wysiwyg-content {
  h1 {  // Necessity breaks rule - WYSIWYG editors don't adhere to BEM.
  }
}
```

Variables

Privatize variables by assigning them on top of the module.

```
$article-color: $color;  // We copy the contents of a global variable into a private one
$article-font: $font;    // This allow us easily "fix" the values and reuse our component

.article {
  color: $article-color;  // We use private values here
  font-family: $article-font;
}
```

JavaScript

Common

- Readability first
- Annotate when useful - e.g. input for functions/methods and return values/types.

```
/**
 * Helper method to add an additional class name with a specific modifier (--
↪modifier) to a BEM (Block Element Modifier) element
 * A modifier class is created for each of the existing class names
 * Class names containing "--" (modifier pattern) are discarded
 * Double class names are prevented
 * @param {HTMLElement} node The block/element to append the class name to (block,␣
↪block__element)
 * @param {String} modifier The name of the modifier (--name)
 */
function addModifier(node, modifier) {
}
```

Indentation

- Indent using 4 spaces

Classes

- Use TitledCamelCase for class names

```
class Header {  // Bonus points: match class to BEM block name
}
```

Conditionals

- Put a space between the operator and brackets

```
if (foo === 'bar') {
    // ...
}
```

Constants

- Use the `const` keyword
- Use UPPERCASE
- Put constants at the top of the module, below the imports

```
import {Foo} from 'bar.js';

const MY_AWESOME_CONSTANT = 'foo';
```

Event binding

- Separate wiring events with event handlers from logic

```
class Handler {

  /**
   * We separate "wiring" from the main logic so we can resure the logic
   */
  setUpOpen() {
      BUTTON_OPEN.addEventListener('click', this.open.bind(this));
  }

  /**
   * We can now reuse `this`
   */
  open(event) {
      // `this` points to the `handler` instance
```

(continues on next page)

(continued from previous page)

```
  }

}
```

Functions

- use camelCase names
- no space between `function` and brackets
- opening bracket goes on the same line, closing bracket has its own line

Example:

```
function fooBar(arg1, arg2) {
    // ...
}
```

Line breaks/newlines

- watch the line length: soft limit on 79 characters, hard limit on 119
- no newline inside logical block:

```
function doFooBar() {

    // ^ Bad, keep related code together
    console.log('indent', 4, 'spaces');
}
```

- Empty newline after method/variable block.

```
function doFooBar() {
    let fooBar = 'foobar';

    console.log(fooBar);
}
```

- 2 empty lines after top level function/class/block

```
const FOO = 'foo';
const BAR = 'bar';


function doFooBaz() {  // 2 Empty newlines after a block of constants
    console.log('foobaz');
}


class Foo {  // 2 Empty newlines after a top level function
    constructor() {
        super();
```

(continues on next page)

(continued from previous page)

```
        this.doBar();
    }

    doBar() {  // 1 Empty newline after method
        let bar = new Bar();
    }
}


class Bar {   // 2 Empty newlines after a class
    constructor() {
        super();
        this.doBar();
    }

    doBar() {
        let bar = new Bar();
    }
}
```

- Empty newline at the end of the file

Variables

- Use the `let` keyword instead of `var`
- Group variable declarations together
- Use camelCase names

Example:

```
function doFooBar() {
    let foo = 'foo',
        bar = 'bar',
        fooBar = foo+bar;

    console.log(fooBar);
}
```

Tests

- Name the test files `foo.spec.js`. `.spec` indicates that it's a test file

3.4.3 Best practices

HTML

- Write semantic HTML before styling.
- Style your HTML, don't HTML your style.
- Don't put content in `master.html`, only put boilerplate/scaffolding here.
- Use inclusion tags for reusable components and blocks otherwise.
- Wrap components/logical page blocks/standalone sections in `{% block %}` tags.
- Respect the *coding style* <*coding_style_frontend*>.
- If it makes sense to divert, divert.

CSS/SASS

- Bootstrap is only allowed for quick prototyping (you discard it later).
- Adhere to the BEM naming standard.
- Match component (file)names to Django template blocks.
- Maximum 1 BEM block per sass file.
- Only select using (BEM) class names (`.block__element`), avoid using tag/id (Matching id's breaks reusability, matching tags breaks flexibility).
- WYSIWYG is an exception (customers don't type content__heading–primary).
- The Block (**B** in BEM) cannot set margin on itself, only on children. This avoids spacing issues.
- Use Neat mixins for (responsive) grids. Avoid complex overdoing mixins (e.g. Bourbon).
- Respect the *coding style* <*coding_style_frontend*>.
- Compile to CSS and keep the compiled css in version control.
- If it makes sense to divert, divert.

Javascript

- These libraries/tools are deprecated - better alternatives exist: - Bootstrap - Bower - Django Pipeline/Compressor - jQuery - RequireJS
- Keep the existing, working setup in older projects.
- Match component (file)names to Django template blocks.
- Write (object oriented) ES6 or newer.
- No dialects (typescript/coffeescript).
- Use a bundler (jspm or webpack) to manage dependencies/transpiling.
- gulp is our task runner (manage.py for frontend).
- Keep the JS source in the *static* folder per Django app.
- Respect the *coding style* <*coding_style_frontend*>.
- If it makes sense to divert, divert.

3.5 Testing

This document covers the tools to run tests and how to use them.

3.5.1 Django tests

Run the project tests by executing:

```
$ python src/manage.py test src --keepdb
```

To measure coverage, use `coverage run`:

```
$ coverage run src/manage.py test src --keepdb
```

It may be convenient to add some aliases:

```
$ alias runtests='python src/manage.py test --keepdb'
$ runtests src
```

and:

```
$ alias cov_runtests='coverage run src/manage.py test --keepdb'
$ cov_runtests src && chromium htmlcov/index.html
```

Jenkins

Run `./bin/jenkins_django.sh` to execute the tests for `develop` and `master`. This script runs the tests with `--keepdb`.

To run PR tests, run `./bin/jenkins_django_pr.sh`. This script drops the test database at the end, so it should be safe with different migrations between PR's.

3.5.2 SASS build - Jenkins

There is a simple `./bin/jenkins_sass.sh` script that checks if the sass compiles successfully.

3.5.3 Javascript tests

There are quite some options to run the Javascript tests. Karma is used as test-runner, and you need to install it globally if you have never done so:

```
$ sudo npm install -g karma
```

By default, the tests are run against PhantomJS and Chrome/Chromium. To run the tests, execute:

```
$ gulp test
```

If you want to target a single browser, you can run karma directly:

```
$ karma start karma.conf.js --single-run --browsers=PhantomJS
```

Coverage reports can be found in `build/reports/coverage`.

To trigger a test run on file change (source file or test file), run:

```
$ karma start karma.conf.js --single-run=false --browsers=PhantomJS
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX