Eden ASP

Eden ASP Team

May 01, 2023

CONTENTS:

1 Introduction into Eden ASP **3**
- 1.1 Basic Concepts . . . 3

2 Building Applications **7**
- 2.1 Setting up for Development . . . 7
- 2.2 About Templates . . . 11
- 2.3 About Controllers . . . 13
- 2.4 Implementing Templates . . . 17
- 2.5 Advanced Topics . . . 17

3 Reference Guide **19**
- 3.1 The *current* Object . . . 19
- 3.2 Services . . . 19
- 3.3 Settings . . . 24
- 3.4 Built-in Data Models . . . 24
- 3.5 Standard CRUD Methods . . . 29
- 3.6 User Interface Elements . . . 39
- 3.7 Tools . . . 48

4 How to deploy Eden ASP applications **53**

5 Extending Eden ASP **55**
- 5.1 Implementing Controllers . . . 55
- 5.2 Implementing Data Models . . . 55
- 5.3 Adding new themes . . . 56
- 5.4 Adding new Form Widgets . . . 56

6 Indices and tables **57**

Index **59**

Eden ASP is a rapid application development (RAD) kit for web-based, database-driven humanitarian and emergency management applications, originally derived from the *Sahana Eden Humanitarian Management Platform*.

Eden ASP builds on the **web2py** web application framework, and is written in the **Python** programming language (version 3.6+). It also uses *HTML5*, *JavaScript*, and *SCSS* to generate web contents, as well as *XSLT* to handle certain data formats.

This documentation is aimed at application developers, and included in the source code.

CHAPTER

ONE

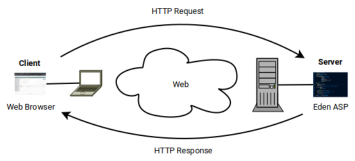
INTRODUCTION INTO EDEN ASP

1.1 Basic Concepts

This page explains the basic concepts, structure and operations of Eden ASP, and introduces the fundamental terminology used throughout this documentation.

1.1.1 Client and Server

Eden ASP is a **web application**, which means it is run as a **server** program and is accessed remotely by **client** programs connected over the network.



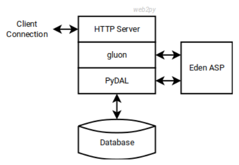
Most of the time, the client program will be a **web browser** - but it could also be a mobile app, or another type of program accessing web services. Many clients can be connected to the server at the same time.

Client and server communicate using the **HTTP** protocol, in which the client sends a **request** to the server, the server processes the request and produces a **response** (e.g. a HTML page) that is sent back to the client, and then the client processes the response (e.g. by rendering the HTML page on the screen).

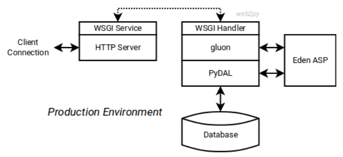
Note: Responding to HTTP requests is Eden ASP's fundamental mode of operation.

1.1.2 Web2Py and PyDAL

Eden ASP builds on the **web2py** web application framework, which consists of three basic components: a *HTTP server*, the *application runner* and various libraries, and a *database abstraction layer*.



The **HTTP server** (also commonly called "web server") manages client connections. Web2py comes with a built-in HTTP server (*Rocket*), but production environments typically deploy a separate front-end HTTP server (e.g. *nginx*) that connects to web2py through a WSGI plugin or service (e.g. *uWSGI*).



The **application runner** (*gluon*) decodes the HTTP request, then calls certain Python functions in the Eden ASP application with the request data as input, and from their output renders the HTTP response. Additionally, *gluon* provides a number of libraries to generate interactive web contents and process user input.

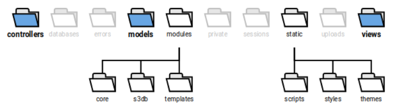
The **database abstraction layer** (*PyDAL*) provides a generic interface to the database, as well as a mapping between Python objects and the tables and records in the database (*ORM, object-relational mapping*). For production environments, the preferred database back-end is PostgreSQL with the PostGIS extension, but SQLite and MariaDB/MySQL are also supported.

1.1.3 Application Structure

Web2py applications like Eden ASP implement the MVC (model-view-controller) application model, meaning that the application code is separated in:

- **models** defining the data(base) structure,
- **views** implementing the user interface,
- **controllers** implementing the logic connecting models and views

This is somewhat reflected by the directory layout of Eden ASP:



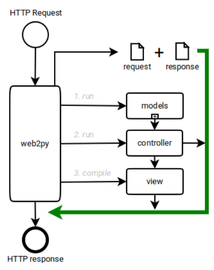
Note: This directory layout can be somewhat misleading about where certain functionality can be found in the code:

The *controllers* directory contains Python scripts implementing the logic of the application. In Eden ASP, these controllers delegate much of that logic to **core** modules.

The *models* directory contains Python scripts to configure the application and define the database structure. In Eden ASP, the former is largely delegated to configuration **templates**, and the latter is reduced to the instantiation of a model loader, which then loads the actual data models from **s3db** modules if and when they are actually needed.

1.1.4 The Request Cycle

Eden ASP runs in cycles triggered by incoming HTTP requests.



When an HTTP request is received, web2py parses and translates it into a global **request** object.

For instance, the request URI is translated like:

```
https://www.example.com/[application]/[controller]/[function]/[args]?[vars]
```

. . . and its elements stored as properties of the *request* object (e.g. *request.controller* and *request.function*). These values determine which function of the application is to be executed.

Web2py also generates a global **response** object, which can be written to in order to set parameters for the eventual HTTP response.

Web2py then runs the Eden ASP application:

1. executes all scripts in the *models/* directory in lexical (ASCII) order.
2. executes the script in the *controllers/* directory that corresponds to *request.controller*, and then calls the function defined by that script that corresponds to *request.function*.

   E.g. if *request.controller* is “dvr” and *request.function* is “person”, then the *controllers/dvr.py* script will be executed, and then the *person()* function defined in that script will be invoked.
3. takes the output of the function call to compile the view template configured as *response.view*.

These three steps are commonly referred to as the *request cycle*.

CHAPTER

TWO

BUILDING APPLICATIONS

2.1 Setting up for Development

This page describes how you can set up a local Eden ASP instance for application development on your computer.

Note: This guide assumes that you are working in a Linux environment (shell commands are for *bash*).

If you are working with another operating system, you can still take this as a general guideline, but commands may be different, and additional installation steps could be required.

Note: This guide further assumes that you have *Python* (version 3.6 or later) installed, which comes bundled with the *pip* package installer - and that you are familiar with the Python programming language.

Additionally, you will need to have git installed.

2.1.1 Prerequisites

Eden ASP requires a couple of Python libraries, which can be installed with the *pip* installer.

As a minimum, *lxml* and *python-dateutil* must be installed:

```
sudo pip install lxml python-dateutil
```

The following are also required for normal operation:

```
sudo pip install pyparsing requests xlrd xlwt openpyxl reportlab shapely geopy
```

Some specialist functionality may require additional libraries, e.g.:

```
sudo pip install qrcode docx-mailmerge
```

Tip: The above commands use *sudo pip* to install the libraries globally. If you want to install them only in your home directory, you can omit *sudo*.

2.1.2 Installing web2py

To install web2py, clone it directly from GitHub:

```
git clone --recursive https://github.com/web2py/web2py.git ~/web2py
```

Tip: You can of course choose any other target location than *~/web2py* for the clone - just remember to use the correct path in subsequent commands.

Change into the *web2py* directory, and reset the repository (including all submodules) to the supported stable version (currently 2.24.1):

```
cd ~/web2py
git reset --hard 7685d373
git submodule update --recursive
```

2.1.3 Installing Eden ASP

To install Eden ASP, clone it directly from GitHub:

```
git clone --recursive https://github.com/aqmaster/eden-asp.git ~/eden
```

Tip: You can of course choose any other target location than *~/eden* for the clone - just remember to use the correct path in subsequent commands.

Configure Eden ASP as a web2py application by adding a symbolic link to the *eden* directory under *web2py/applications*:

```
cd ~/web2py/applications
ln -s ~/eden eden
```

The name of this symbolic link (*eden*) becomes the web2py application name, and will later be used in URLs to access the application.

Tip: You can also clone Eden ASP into the *~/web2py/applications/eden* directory - then you will not need the symbolic link.

2.1.4 Configuring Eden ASP

Before running Eden ASP the first time, you need to create a configuration file. To do so, copy the *000_config.py* template into Eden ASP's *models* folder:

```
cd ~/eden
cp modules/templates/000_config.py models
```

Open the *~/eden/models/000_config.py* file in an editor and adjust any settings as needed.

For development, you do not normally need to change anything, except setting the following to *True* (or removing the line altogether):

Listing 1: Editing models/000_config.py

```
FINISHED_EDITING_CONFIG_FILE = True
```

That said, it normally makes sense to also turn on *debug* mode for development:

Listing 2: Editing models/000_config.py

```
settings.base.debug = True
```

2.1.5 First run

The first start of Eden ASP will set up the database, creating all tables and populating them with some data.

This is normally done by running the *noop.py* script in the web2py shell:

```
cd ~/web2py
python web2py.py -S eden -M -R applications/eden/static/scripts/tools/noop.py
```

This will give a console output similar to this:

Listing 3: Console output during first run

```
WARNING:  S3Msg unresolved dependency: pyserial required for Serial port modem usage
WARNING:  Setup unresolved dependency: ansible required for Setup Module
WARNING: Error when loading optional dependency: google-api-python-client
WARNING: Error when loading optional dependency: translate-toolkit

*** FIRST RUN - SETTING UP DATABASE ***

Setting Up System Roles...
Setting Up Scheduler Tasks...
Creating Database Tables (this can take a minute)...
Database Tables Created. (3.74 sec)

Please be patient whilst the database is populated...

Importing default/base...
Imports for default/base complete (1.99 sec)

Importing default...
Imports for default complete (5.20 sec)

Importing default/users...
Imports for default/users complete (0.04 sec)

Updating database...
Location Tree update completed (0.63 sec)
Demographic Data aggregation completed (0.01 sec)

Pre-populate complete (7.90 sec)

Creating indexes...
```

(continues on next page)

(continued from previous page)

```

*** FIRST RUN COMPLETE ***
```

You can ignore the *WARNING* messages here about unresolved, optional dependencies.

2.1.6 Starting the server

In a development environment, we normally use the built-in HTTP server (*Rocket*) of web2py, which can be launched with:

```
cd ~/web2py
python web2py.py --no_gui -a [password]
```

Replace *[password]* here with a password of your choosing - this password is needed to access web2py's application manager (e.g. to view error tickets).

Once the server is running, it will give you a localhost URL to access it:

Listing 4: Console output of web2py after launch

```
web2py Web Framework
Created by Massimo Di Pierro, Copyright 2007-2023
Version 2.24.1-stable+timestamp.2023.03.22.21.39.14
Database drivers available: sqlite3, MySQLdb, psycopg2, imaplib, pymysql, pyodbc

please visit:
        http://127.0.0.1:8000/
use "kill -SIGTERM 10921" to shutdown the web2py server
```

Append the application name *eden* to the URL (http://127.0.0.1:8000/eden), and open that address in your web browser to access Eden ASP.

The first run will have installed two demo user accounts, namely:

- *admin@example.com* (a user with the system administrator role)
- *normaluser@example.com* (an unprivileged user account)

. . . each with the password *testing*. So you can login and explore the functionality.

2.1.7 Using PostgreSQL

to be written

2.2 About Templates

2.2.1 Global Config

Many features and behaviors of Eden ASP can be controlled by settings.

These settings are stored in a global *S3Config* instance - which is accessible through *current* as *current.deployment_settings*.

```
from gluon import current

settings = current.deployment_settings
```

Note: In the models and controllers context, *current.deployment_settings* is accessible simply as *settings*.

2.2.2 Deployment Settings

S3Config comes with meaningful defaults where possible.

However, some settings will need to be adjusted to configure the application for a particular system environment - or to enable, disable, configure, customize or extend features in the specific context of the deployment.

This configuration happens in a machine-specific configuration file:

models/000_config.py

Note: *models/000_config.py is not part of the code base, and must be created before the application can be started. An annotated example can be found in the *modules/templates* directory.

The configuration file is a Python script that is executed for every request cycle:

Listing 5: models/000_config.py (partial example)

```
# -*- coding: utf-8 -*-

"""
    Machine-specific settings
"""

# Remove this line when this file is ready for 1st run
FINISHED_EDITING_CONFIG_FILE = True

# Select the Template
settings.base.template = "MYAPP"

# Database settings
settings.database.db_type = "postgres"
#settings.database.host = "localhost"
#settings.database.port = 3306
settings.database.database = "myapp"
#settings.database.username = "eden"
```

(continues on next page)

(continued from previous page)

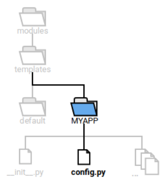
```
#settings.database.password = "password"

# Do we have a spatial DB available?
settings.gis.spatialdb = True

settings.base.migrate = True
#settings.base.fake_migrate = True

settings.base.debug = True
#settings.log.level = "WARNING"
#settings.log.console = False
#settings.log.logfile = None
#settings.log.caller_info = True

# =============================================================================
# Import the settings from the Template
#
settings.import_template()

# =============================================================================
# Over-rides to the Template may be done here
#
# After 1st_run, set this for Production
#settings.base.prepopulate = 0

# =============================================================================
VERSION = 1

# END =========================================================================
```

2.2.3 Templates

Deployment configurations use configuration **templates**, which provide pre-configured settings, customizations and extensions suitable for a concrete deployment scenario. The example above highlights how these templates are applied.

Important: Implementing configuration **templates** is the primary strategy to build applications with Eden ASP.

Templates are Python packages located in the *modules/templates* directory:



Each template package must contain a module *config.py* which defines a *config*-function :

Listing 6: modules/templates/MYAPP/config.py

```
def config(settings):

    T = current.T

    settings.base.system_name = T("My Application")
    settings.base.system_name_short = T("MyApp")

    ...
```

This *config* function is called from *models/000_config.py* (i.e. for every request cycle) with the *current.deployment_settings* instance as parameter, so that it can modify the global settings as needed.

Note: The template directory must also contain an *__init__.py* file (which can be empty) in order to become a Python package!

2.2.4 Cascading Templates

It is possible for a deployment configuration to apply multiple templates in a cascade, so that they complement each other:

Listing 7: Cascading templates (in models/000_config.py)

```
# Select the Template
settings.base.template = ("locations.DE", "MYAPP")
```

This is useful to separate e.g. locale-specific settings from use-case configurations, so that both can be reused independently across multiple deployments.

2.3 About Controllers

Controllers are functions defined inside Python scripts in the *controllers* directory, which handle HTTP requests and produce a response.

2.3.1 Basic Request Routing

Web2py maps the first three elements of the URL path to controllers as follows:

```
https:// server.domain.tld / application / controller / function
```

The *application* refers to the subdirectory in web2py's application directory, which in the case of Eden ASP is normally **eden** (it is possible to name it differently, however).

The **controller** refers to a Python script in the *controllers* directory inside the application, which is executed.

For instance:

```
https:// server.domain.tld / eden / my / page
```

executes the script:

```
controllers / my.py
```

The **function** refers to a *parameter-less* function defined in the controller script, which is subsequently called. In the example above, that would mean this function:

Listing 8: In controllers/my.py

```python
def page():
    ...
    return output
```

If the output format is HTML, the output of the controller function is further passed to the view compiler to render the HTML which is then returned to the client in the HTTP response.

Every controller having its own URL also means that every *page* in the web GUI has its own controller - and Eden ASP (like any web2py application) is a *multi-page application* (MPA). Therefore, in the context of the web GUI, the terms “controller function” and “page” are often used synonymously.

That said, not every controller function actually produces a web page. Some controllers exclusively serve non-interactive requests.

2.3.2 CRUD Controllers

The basic database functions **create**, **read**, **update** and **delete** (short: *CRUD*) are implemented in Eden ASP as one generic function:

Listing 9: In controllers/my.py

```python
def page():

    return crud_controller()
```

This single function call automatically generates web forms to create and update records, displays filterable tables, generates pivot table reports and more - including a generic RESTful API for non-interactive clients.

If called without parameters, *crud_controller* will interpret *controller* and *function* of the page URL as prefix and name of the database table which to provide the functionality for, i.e. in the above example, CRUD functions would be provided for the table *my_page*.

It is possible to override the default table, by passing prefix and name explicitly to *crud_controller*, e.g.:

Listing 10: In controllers/my.py

```
def page():

    return crud_controller("org", "organisation")
```

. . . will provide CRUD functions for the *org_organisation* table instead.

2.3.3 Resources and Components

As explained above, a *crud_controller* is a database end-point that maps to a certain table or - depending on the request - certain records in that table.

This *context data set* (consisting of a table and a query) is referred to as the **resource** addressed by the HTTP request and served by the controller.

Apart from the data set in the primary table (called *master*), a resource can also include data in related tables that reference the master (e.g. via foreign keys or link tables) and which have been *declared* (usually in the data model) as **components** in the context of the master table.

An example for this would be addresses (*component*) of a person (*master*).

2.3.4 CRUD URLs and Methods

The *crud_controller* extends web2py's URL schema with two additional path elements:

```
https:// server.domain.tld / a / c / f / record / method
```

Here, the **record** is the primary key (*id*) of a record in the table served by the crud_controller function - while the **method** specifies how to access that record, e.g. *read* or *update*.

For instance, the following URL:

```
https:// server.domain.tld / eden / org / organisation / 4 / update
```

. . . accesses the workflow to update the record #4 in the org_organisation table (with HTTP GET to retrieve the update-form, and POST to submit it and perform the update).

Without a *record* key, the URL accesses the table itself - as some methods, like *create*, only make sense in the table context:

```
https:// server.domain.tld / eden / org / organisation / create
```

The *crud_controller* comes pre-configured with a number of standard methods, including:

Method	Target	Description
create	*Table*	Create a new record (form)
read	*Record*	View a record (read-only representation)
update	*Record*	Update a record (form)
delete	*Record*	Delete a record
list	*Table*	A tabular view of records
report	*Table*	Pivot table report with charts
timeplot	*Table*	Statistics over a time axis
map	*Table*	Show location context of records on a map
summary	*Table*	Meta-method with list, report, map on the same page (tabs)
import	*Table*	Import records from spreadsheets
organize	*Table*	Calendar-based manipulation of records

Note: Both *models* and *templates* can extend the *crud_controller* by adding further methods, or overriding the standard methods with specific implementations.

2.3.5 Default REST API

If no *method* is specified in the URL, then the *crud_controller* will treat the request as **RESTful** - i.e. the HTTP verb (GET, PUT, POST or DELETE) determines the access method, e.g.:

```
GET https:// server.domain.tld / eden / org / organisation / 3.xml
```

…produces a XML representation of the record #3 in the org_organisation table. A *POST* request to the same URL, with XML data in the request body, will update the record.

This **REST API** is a simpler, lower-level interface that is primarily used by certain client-side scripts, e.g. the map viewer. It does not implement complete CRUD workflows, but rather each function individually (stateless).

Note: A data format extension in the URL is required for the REST API, as it can produce and process multiple data formats (extensible). Without extension, HTML format will be assumed and one of the interactive *read*, *update*, *delete* or *list* methods will be chosen to handle the request instead.

The default REST API *could* be used to integrate Eden ASP with other applications, but normally such integrations require process-specific end points (rather than just database end points) - which would be implemented as explicit methods instead.

2.3.6 Component URLs

URLs served by a *crud_controller* can also directly address a *component*. For that, the *record* parameter would be extended like:

```
https:// server.domain.tld / a / c / f / record / component / method
```

Here, the **component** is the *declared* name (*alias*) of the component in the context of the master table - usually the name of the component table without prefix, e.g.:

```
https:// server.domain.tld / eden / pr / person / 16 / address
```

…would produce a list of all addresses (*pr_address* table) that are related to the *pr_person* record #16. Similar, replacing *list* with *create* would access the workflow to create new addresses in the context of that person record.

Note: The */list* method can be omitted here - if the end-point is a table rather than a single record, then the *crud_controller* will automatically apply the *list* method for interactive data formats.

To access a particular record in a component, the primary key (id) of the component record can be appended, as in:

```
https:// server.domain.tld / eden / pr / person / 16 / address / 2 / read
```

…to read the *pr_address* record #2 in the context of the *pr_person* record #16 (if the specified component record does not reference that master record, the request will result in a HTTP 404 status).

Note: The *default REST API* *always* serves the master table, even if the URL addresses a component (however, the XML/JSON will include the component).

2.4 Implementing Templates

2.4.1 Settings

2.4.2 Customising resources

2.4.3 Customising controllers

2.4.4 Pre-populating data

2.4.5 Menus

2.4.6 Configuring Auth

2.5 Advanced Topics

2.5.1 Themes

2.5.2 Models in templates

2.5.3 Re-routing controllers

CHAPTER

THREE

REFERENCE GUIDE

3.1 The *current* Object

The *current* object holds thread-local global variables. It can be imported into any context:

```
from gluon import current
```

Table 1: Objects accessible through current

Attribute	Type	Explanation
current.db	DAL	the database
current.s3db	DataModel	the model loader
current.deployment_settings	S3Config	deployment settings
current.auth	AuthS3	global authentication/authorisation service
current.gis	GIS	global GIS service
current.msg	S3Msg	global messaging service
current.xml	S3XML	global XML decoder/encoder service
current.request	Request	web2py's global request object
current.response	Response	web2py's global response object
current.T	TranslatorFactory	String Translator (for i18n)
current.messages	Messages	Common labels (internationalised)
current.ERROR	Messages	Common error messages (internationalised)

3.2 Services

Services are thread-local global singleton objects, instantiated during the *models* run.

They can be accessed through *current* , e.g.:

```
from gluon import current

s3db = current.s3db
```

This section describes the services, and their most relevant functions.

3.2.1 Model Loader *s3db*

The **s3db** model loader provides access to database tables and other named objects defined in dynamically loaded models.

The model loader can be accessed through *current*:

```
from gluon import current

s3db = current.s3db
```

Accessing Tables and Objects

A table or other object defined in a dynamically loaded data model can be accessed by name either as attribute or as key of *current.s3db*:

Listing 1: Example: accessing the org_organisation table using attribute-pattern

```
table = s3db.org_organisation
```

Listing 2: Example: accessing the org_organisation table using key-pattern

```
tablename = "org_organisation"
table = s3db[tablename]
```

Either pattern will raise an *AttributeError* if the table or object is not defined, e.g. when the module is disabled.

Both access methods build on the lower-level *table()* method:

`s3db.`**`table`**(*tablename*, *default=None*, *db_only=False*)

Access a named object (usually a Table instance) defined in a dynamically loaded model.

Parameters

- **`tablename`** (*`str`*) – the name of the table (or object)
- **`default`** – the default to return if the table (or object) is not defined
- **`db_only`** (*`bool`*) – return only Table instances, not other objects with the given name

Note: If an *Exception* instance is passed as default, it will be raised rather than returned.

Table Settings

Table settings are used to configure entity-specific behaviors, e.g. forms, list fields, CRUD callbacks and access rules. The following functions can be used to manage table settings:

`s3db.`**`configure`**(*tablename*, ***attr*)

Add or modify table settings.

Parameters

- **`tablename`** (*`str`*) – the name of the table

- **attr** – table settings as key-value pairs

Listing 3: Example: configuring table settings

```
s3db.configure("org_organisation",
               insertable = False,
               list_fields = ["name", "acronym", "website"],
               )
```

`s3db.`**`get_config`**(*tablename*, *key*, *default=None*)

Inspect table settings.

Parameters

- **tablename** (*str*) – the name of the table
- **key** (*str*) – the settings-key
- **default** – the default value if setting is not defined for the table

Returns

the current value of the setting, or default

Listing 4: Example: inspecting table settings

```
if s3db.get_config("org_organisation", "insertable", True):
    # ...
else:
    # ...
```

`s3db.`**`clear_config`**(*tablename*, **keys*)

Remove table settings.

Parameters

- **tablename** (*str*) – the name of the table
- **keys** – the keys for the settings to remove

Listing 5: Example: removing table settings

```
s3db.clear_config("org_organisation", "list_fields")
```

Warning: If *clear_config* is called without keys, **all** settings for the table will be removed!

Declaring Components

The *add_components* method can be used to declare *components*.

`s3db.`**`add_components`**(*tablename*, ***links*)

Declare components for a table.

Parameters

- **tablename** (*str*) – the name of the table
- **links** – component links

Listing 6: Example: declaring components

```
s3db.add_components("org_organisation",

                    # A 1:n component with foreign key
                    org_office = "organisation_id",

                    # A 1:n component with foreign key, single entry
                    org_facility = {"joinby": "organisation_id",
                                    "multiple": False,
                                    },

                    # A m:n component with link table
                    project_project = {"link": "project_organisation",
                                       "joinby": "organisation_id",
                                       "key": "project_id",
                                       },
                    )
```

URL Method Handlers

`s3db.`**`set_method`**(*tablename*, *component=None*, *method=None*, *action=None*)

Configure a URL method for a table, or a component in the context of the table

Parameters

- **`tablename`** (*`str`*) – the name of the table
- **`component`** (*`str`*) – component alias
- **`method`** (*`str`*) – name of the method (to use in URLs)
- **`action`** – function or other callable to invoke for this method, receives the CRUDRequest instance and controller keyword parameters as arguments

Listing 7: Example: defining and configuring a handler for a URL method for a table

```
def check_in_func(r, **attr):
    """ Handler for check_in method """

    # Produce some output...

    # Return output to view
    return {}

# Configure check_in_func as handler for the "check_in" method
# (i.e. for URLs like /eden/pr/person/5/check_in):
s3db.set_method("pr_person", method="check_in", action=check_in_func)
```

Tip: If a CRUDMethod class is specified as action, it will be instantiated when the method is called (lazy instantiation).

`s3db.`**`get_method`**`(`*tablename*, *component=None*, *method=None*`)`

Get the handler for a URL method for a table, or a component in the context of the table

Parameters

- **`tablename`** (*`str`*) – the name of the table
- **`component`** (*`str`*) – component alias
- **`method`** (*`str`*) – name of the method

Returns
the handler configured for the method (or None)

CRUD Callbacks

to be written

3.2.2 Authentication and Authorisation *auth*

Global authentication/authorisation service, accessible through **current.auth**.

```
from gluon import current

auth = current.auth
```

User Status and Roles

`auth.`**`s3_logged_in`**`()`

Check whether the user is logged in; attempts a HTTP Basic Auth login if not.

Returns bool
whether the user is logged in or not

`auth.`**`s3_has_role`**`(`*role*, *for_pe=None*, *include_admin=True*`)`

Check whether the user has a certain role.

Parameters

- **`role`** (*`str`* | *`int`*) – the UID/ID of the role
- **`for_pe`** (*`int`*) – the *pe_id* of a realm entity
- **`include_admin`** (*`bool`*) – return True for ADMIN even if role is not explicitly assigned

Returns bool
whether the user has the role (for the realm)

Access Permissions

Access methods:

Method Name	Meaning
create	create new records
read	read records
update	update existing records
delete	delete records
review	review unapproved records
approve	approve records

auth.s3_has_permission(method, table, record_id=None, c=None, f=None):

Check whether the current user has permission to perform an action in the given context.

Parameters

- **method** (*str*) – the access method
- **table** (*str* | *Table*) – the table
- **record_id** (*int*) – the record ID
- **c** (*str*) – the controller name (if not specified, current.request.controller will be used)
- **f** (*str*) – the function name (if not specified, current.request.function will be used)

Returns bool

whether the intended action is permitted

3.2.3 Geospatial Information and Maps *gis*

3.2.4 Messaging *msg*

3.2.5 XML Encoder/Decoder *xml*

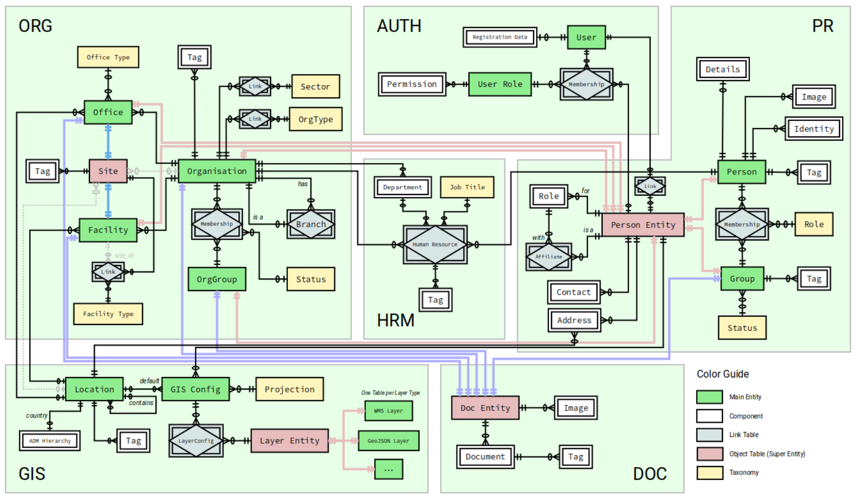
3.3 Settings

3.4 Built-in Data Models

3.4.1 Core Models

Core models form the basis of the Eden ASP database, defining base entities *Persons*, *Organisations* and *Locations* that represent the fundamental elements of the user world.

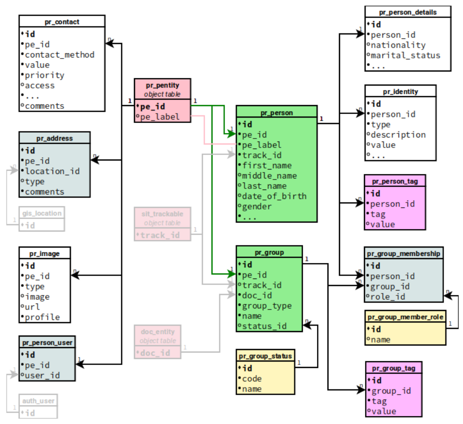
These models are required for essential system functionality, and therefore cannot be disabled.



Persons and Groups - *pr*

This data model describes individual persons and groups of persons.

Database Structure



Description

Table	Type	Description
pr_address	Object Component	Addresses
pr_contact	Object Component	Contact information (Email, Phone, …)
pr_group	Main Entity	Groups of persons
pr_group_member_role	Taxonomy	Role of the group member within the group
pr_group_membership	Relationship	Group membership
pr_group_status	Taxonomy	Status of the group
pr_group_tag	Key-Value	Tags for groups
pr_identity	Component	A person's identities (ID documents)
pr_image	Object Component	Images (e.g. Photos)
pr_pentity	Object Table (Super-Entity)	All entities representing persons
pr_person	Main Entity	Individual persons
pr_person_details	Subtable	Additional fields for pr_person
pr_person_tag	Key-Value	Tags for persons
pr_person_user	Link Table	Link between a person and a user account

Organisations and Sites - *org*

to be written

Human Resources

to be written

User Accounts and Roles - *auth*

to be written

Geospatial Information and Maps - *gis*

to be written

Document Management - *doc*

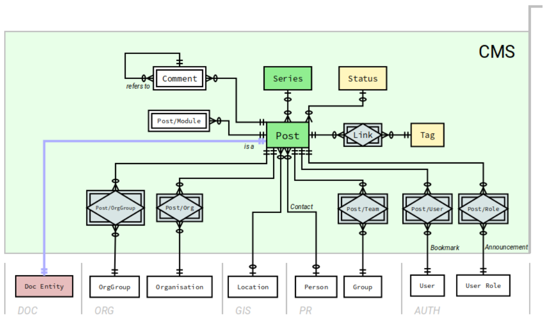
to be written

3.4.2 Extensions

Extension models implement data elements for non-essential system functionality.

Content Management

The Content Management System (*cms*) is a place to store all kinds of user-editable contents. Its main entity is the **Post** (=content item), which can be linked to various *core entities*. Posts are also *DocEntities*, i.e. can have attachments.



The CMS was originally designed for news and discussion feeds, but is more commonly used for informative page contents including, but not limited to:

- page intros
- legal, contact and privacy information pages
- guidance on forms or form elements
- group announcements
- online user guides

. . . as well as for notification templates.

Deployment Settings

to be written

Project Tracking

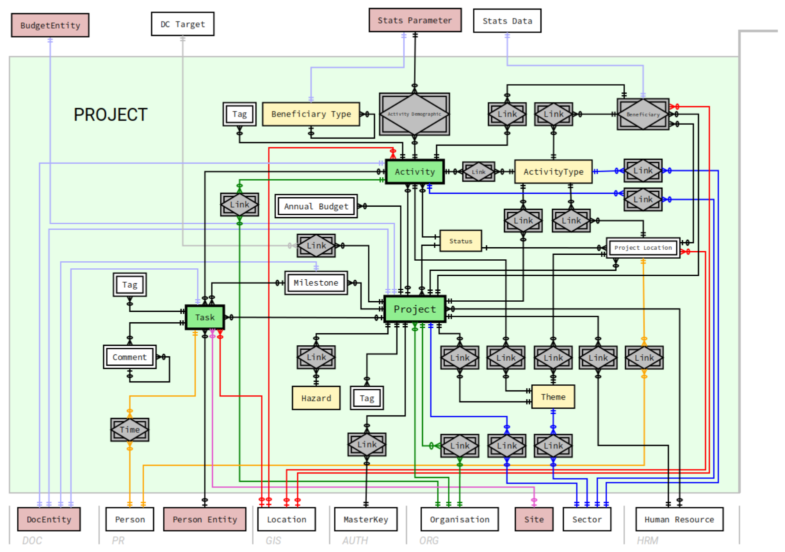
The main purpose of the *project* module is to track contexts of project-based business activities and collaboration.

Projects can be both multi-location and multi-organisation, through qualified links describing exactly how the respective location or organisation is involved.

Activities represent concrete actions within a project, with place, time and type.

Various categories are available for both activities and projects, e.g. themes, sectors, and hazards addressed.

Additionally, the module provides a basic **task** management, which can also be used standalone for simple TODO lists.



Deployment Settings

to be written

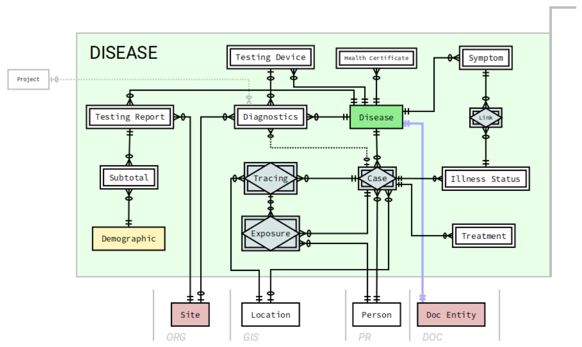
3.4.3 Business Data Models

The models implement data structures for specific business cases. Typically, they have been developed for actual deployments, and then (often only partially) generalized.

Note: Some of these models may be under active development, and thus this documentation not always fully up-to-date - please study the current code before planning your project.

Disease Tracking

This module implements data elements to track disease outbreaks, both on the individual case level, and in mass testing. It was originally developed for Ebola Virus Disease outbreaks, and has later been re-used during the COVID-19 pandemic.



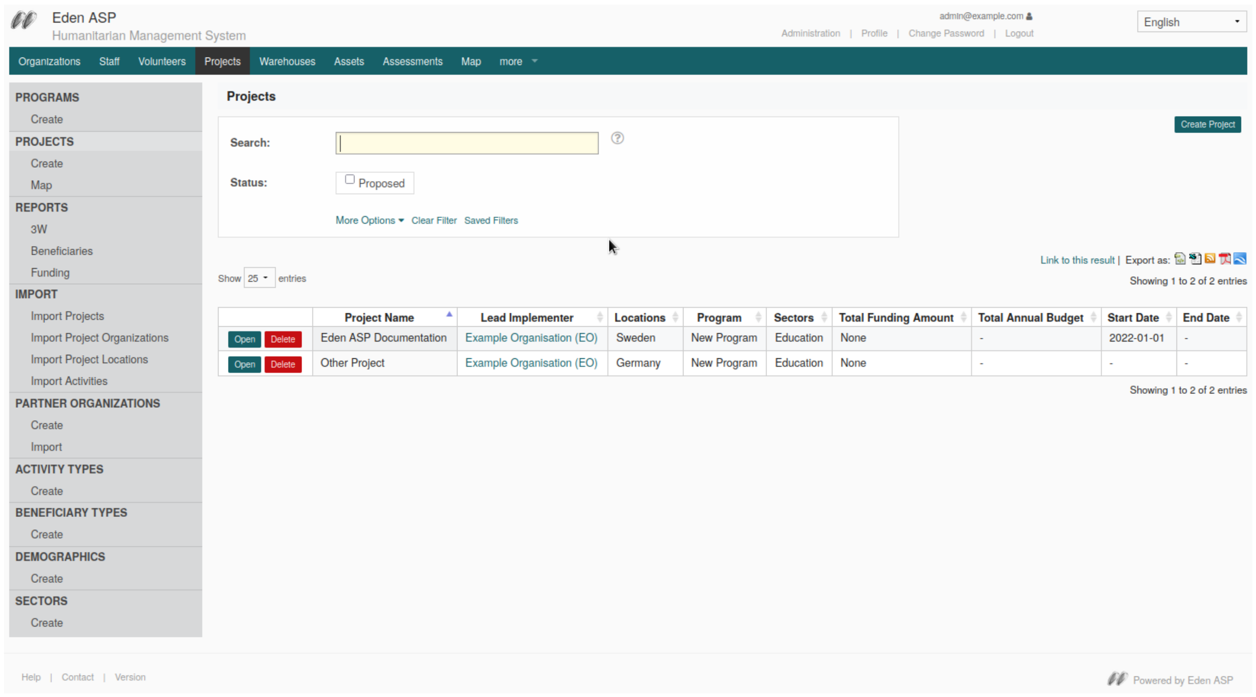
Training Courses and Events

to be written

3.5 Standard CRUD Methods

3.5.1 Data Tables

Tabular view of records (end-point: */list*, and default for table end-point without method and interactive data format).



Projects

Search:

Status: Proposed

Show 25 entries

Showing 1 to 2 of 2 entries

	Project Name	Lead Implementer	Locations	Program	Sectors	Total Funding Amount	Total Annual Budget	Start Date	End Date
Open Delete	Eden ASP Documentation	Example Organisation (EO)	Sweden	New Program	Education	None	-	2022-01-01	-
Open Delete	Other Project	Example Organisation (EO)	Germany	New Program	Education	None	-	-	-

Showing 1 to 2 of 2 entries

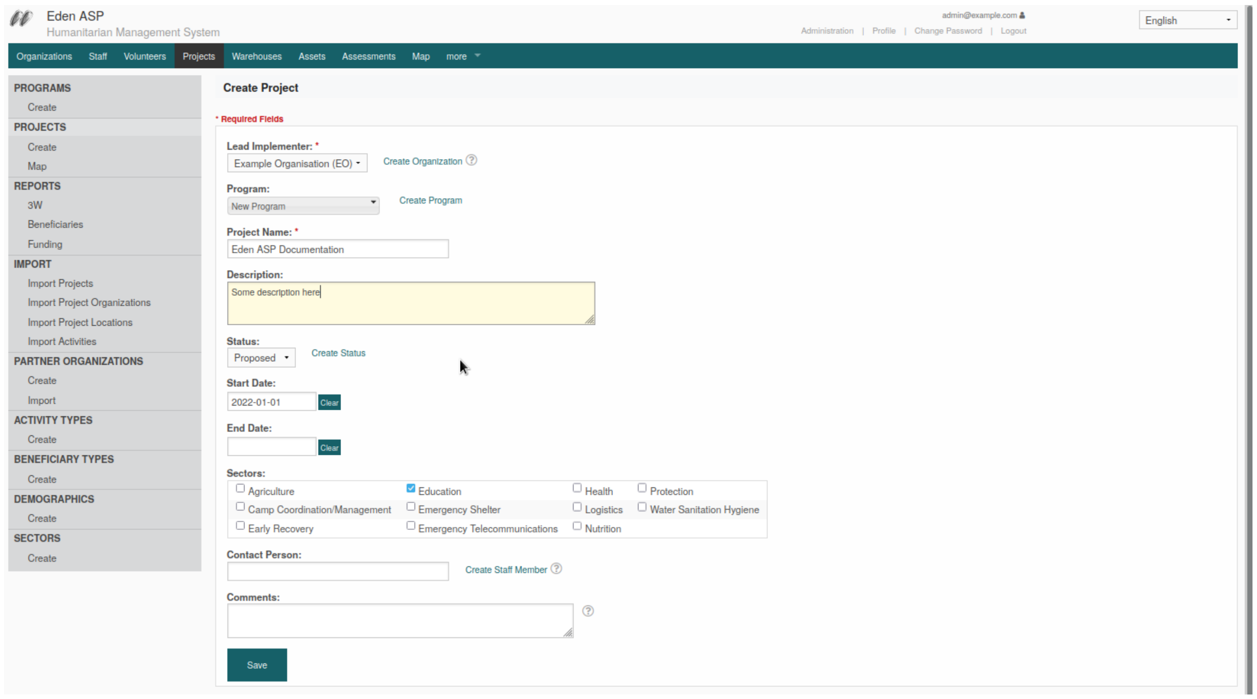
Fig. 1: Data Table View with Filter Form

3.5.2 Form-based CRUD

Simple, form-based Create, Read, Update and Delete functions.

Create

End-point: */create*



Create Project

* Required Fields

Lead Implementer: * Example Organisation (EO)

Program: New Program

Project Name: * Eden ASP Documentation

Description: Some description here

Status: Proposed

Start Date: 2022-01-01

End Date:

Sectors:

☐ Agriculture	☑ Education	☐ Health	☐ Protection
☐ Camp Coordination/Management	☐ Emergency Shelter	☐ Logistics	☐ Water Sanitation Hygiene
☐ Early Recovery	☐ Emergency Telecommunications	☐ Nutrition	

Contact Person:

Comments:

Fig. 2: Create-form

Read

End-point: *[id]/read*

Update

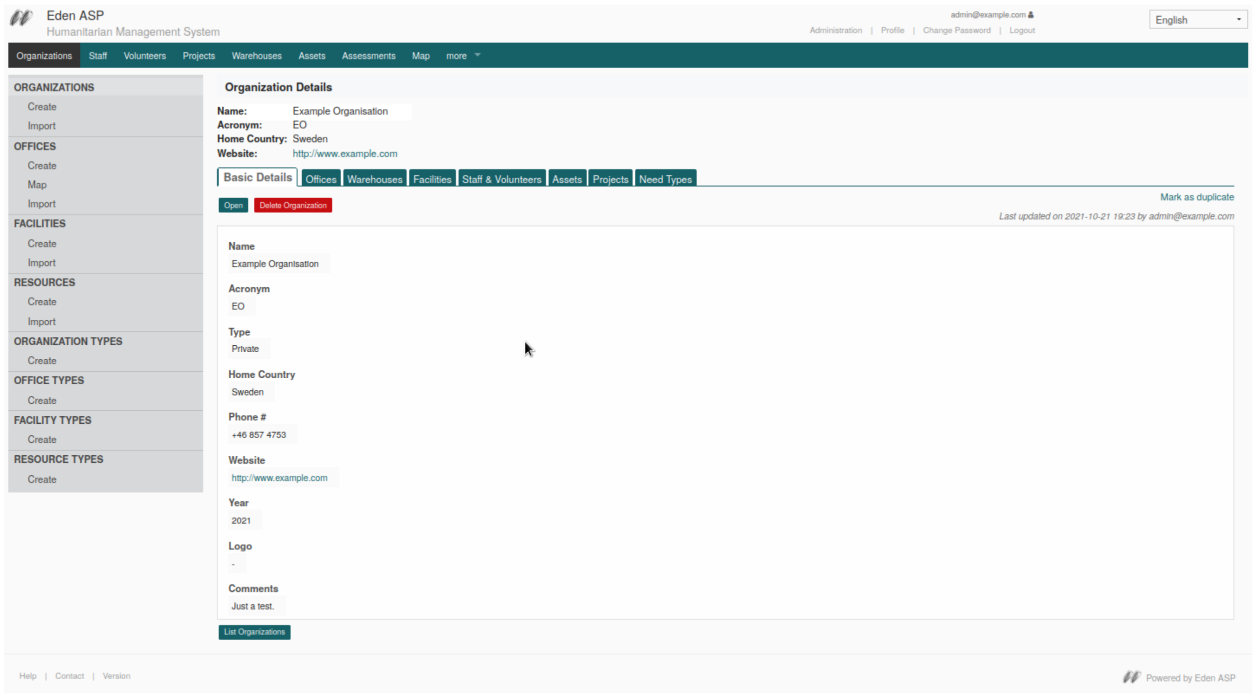
End-point: *[id]/update*

Delete

End-point: [id]/delete

3.5.3 Map

Filterable Map (end-point: */map*).



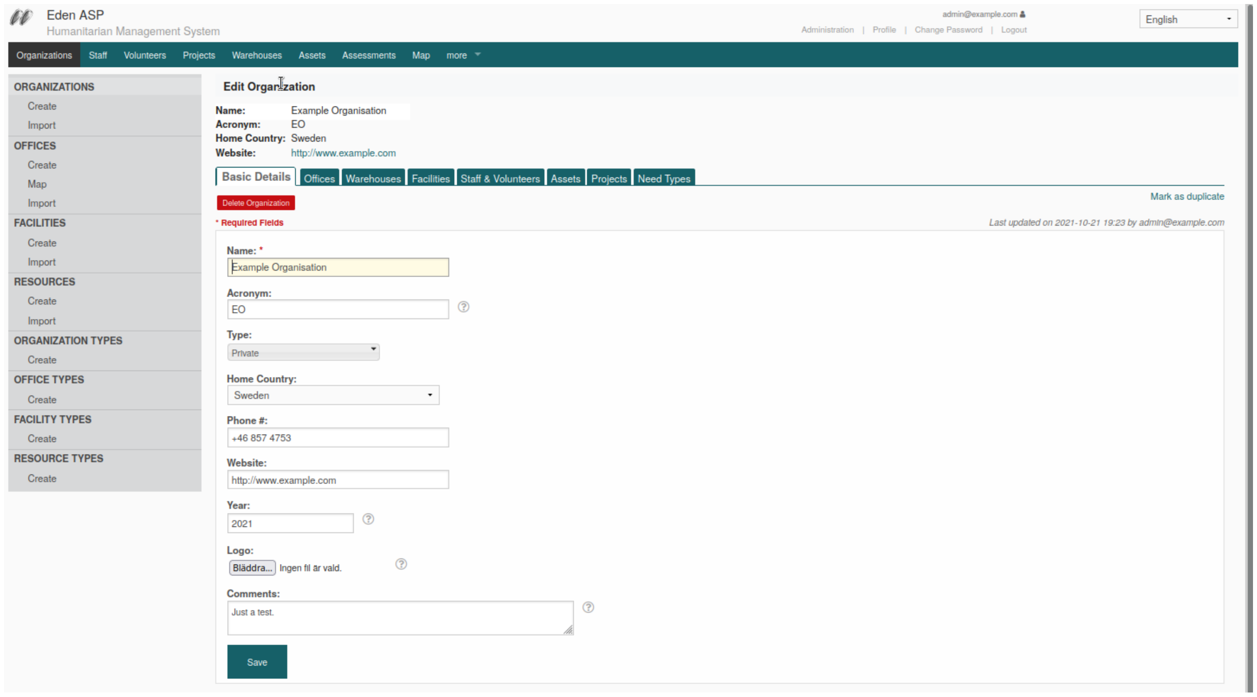
Organization Details

Name: Example Organisation
Acronym: EO
Home Country: Sweden
Website: http://www.example.com

Last updated on 2021-10-21 19:23 by admin@example.com

Name	Example Organisation
Acronym	EO
Type	Private
Home Country	Sweden
Phone #	+46 857 4753
Website	http://www.example.com
Year	2021
Logo	-
Comments	Just a test.

Fig. 3: Read view with component tabs



Edit Organization

Name: Example Organisation
Acronym: EO
Home Country: Sweden
Website: http://www.example.com

* Required Fields

Last updated on 2021-10-21 19:23 by admin@example.com

Name: *	Example Organisation
Acronym:	EO
Type:	Private
Home Country:	Sweden
Phone #:	+46 857 4753
Website:	http://www.example.com
Year:	2021
Logo:	Ingen fil är vald.
Comments:	Just a test.

Fig. 4: Update-form on tab

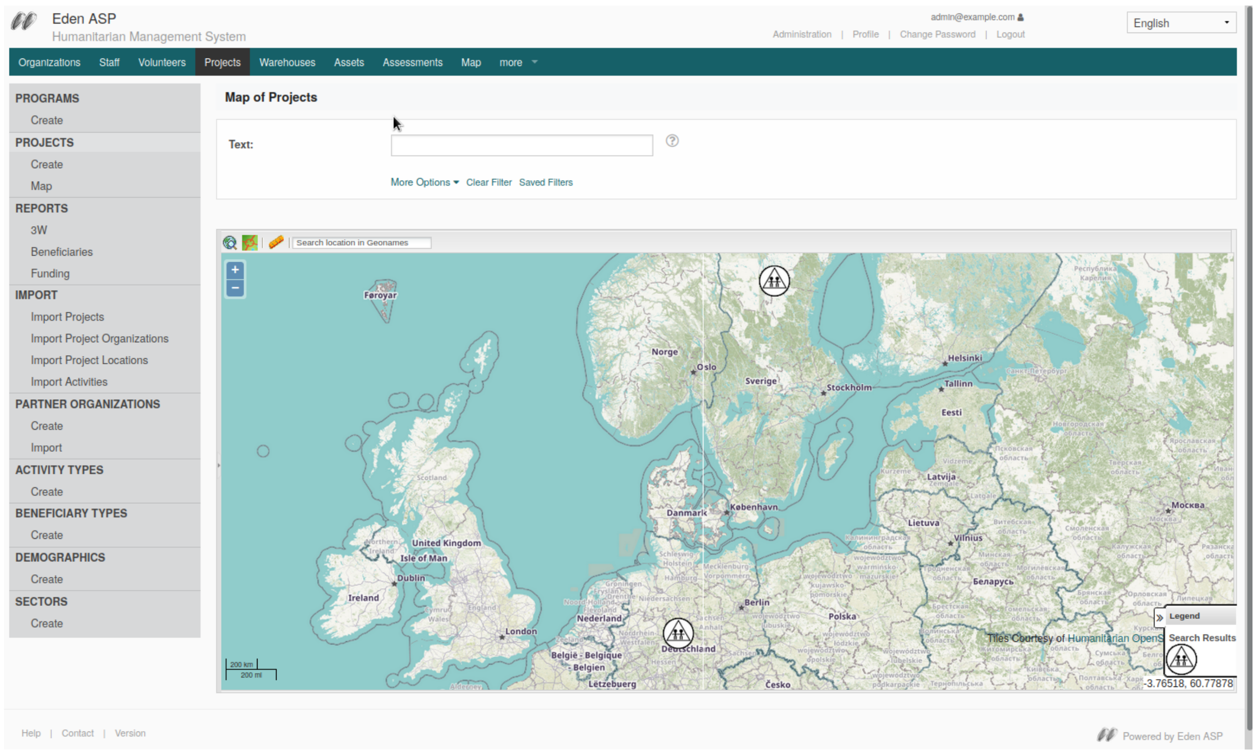


Fig. 5: Map with filter form

3.5.4 Pivottable Reports

User-definable pivot tables with chart option (end-point: */report*).

Note: This method requires configuration.

3.5.5 Timeplot

Aggregation and visualisation of one or more numeric facts over a time axis (endpoint: */timeplot*).

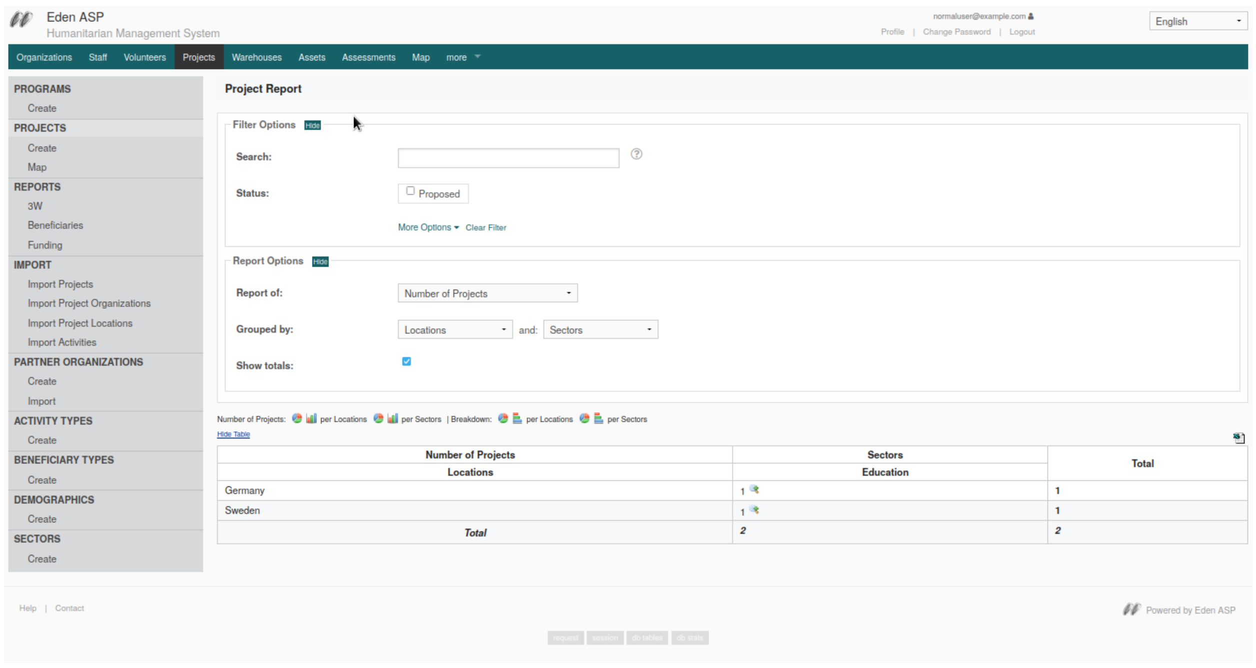
Configuration

The `timeplot_options` table setting is used to configure the report:

Listing 8: Example of timeplot_options configuration

```
facts = [(T("Number of Tests"), "sum(tests_total)"),
         (T("Number of Positive Test Results"), "sum(tests_positive)"),
         (T("Number of Reports"), "count(id)"),
         ]

timeframes = [("All up to now", "", "", ""),
```

(continues on next page)



Project Report

Filter Options

Search:

Status: ☐ Proposed

Report Options

Report of: Number of Projects

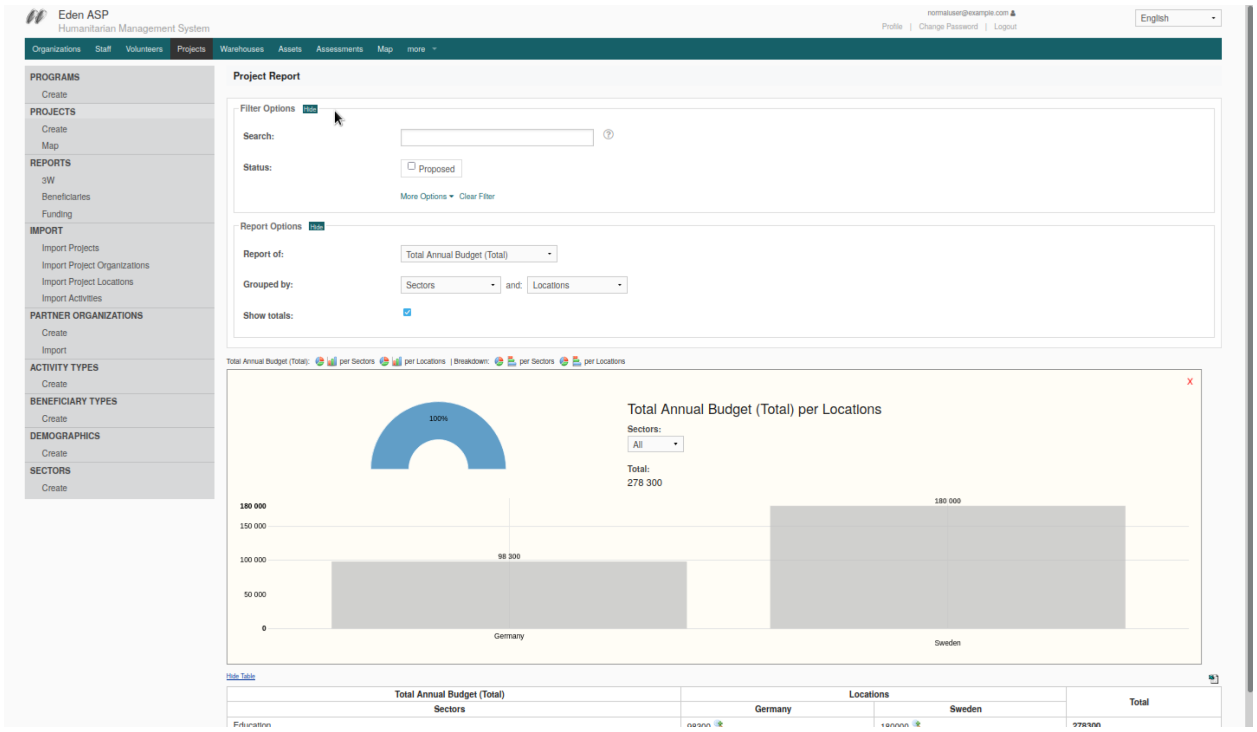
Grouped by: Locations and: Sectors

Show totals: ✓

Number of Projects: per Locations per Sectors | Breakdown: per Locations per Sectors

Number of Projects	Sectors	Total
Locations	Education	
Germany	1	1
Sweden	1	1
Total	*2*	*2*

Fig. 6: Pivot Table Report



Project Report

Filter Options

Search:

Status: ☐ Proposed

Report Options

Report of: Total Annual Budget (Total)

Grouped by: Sectors and: Locations

Show totals: ✓

Total Annual Budget (Total): per Sectors per Locations | Breakdown: per Sectors per Locations

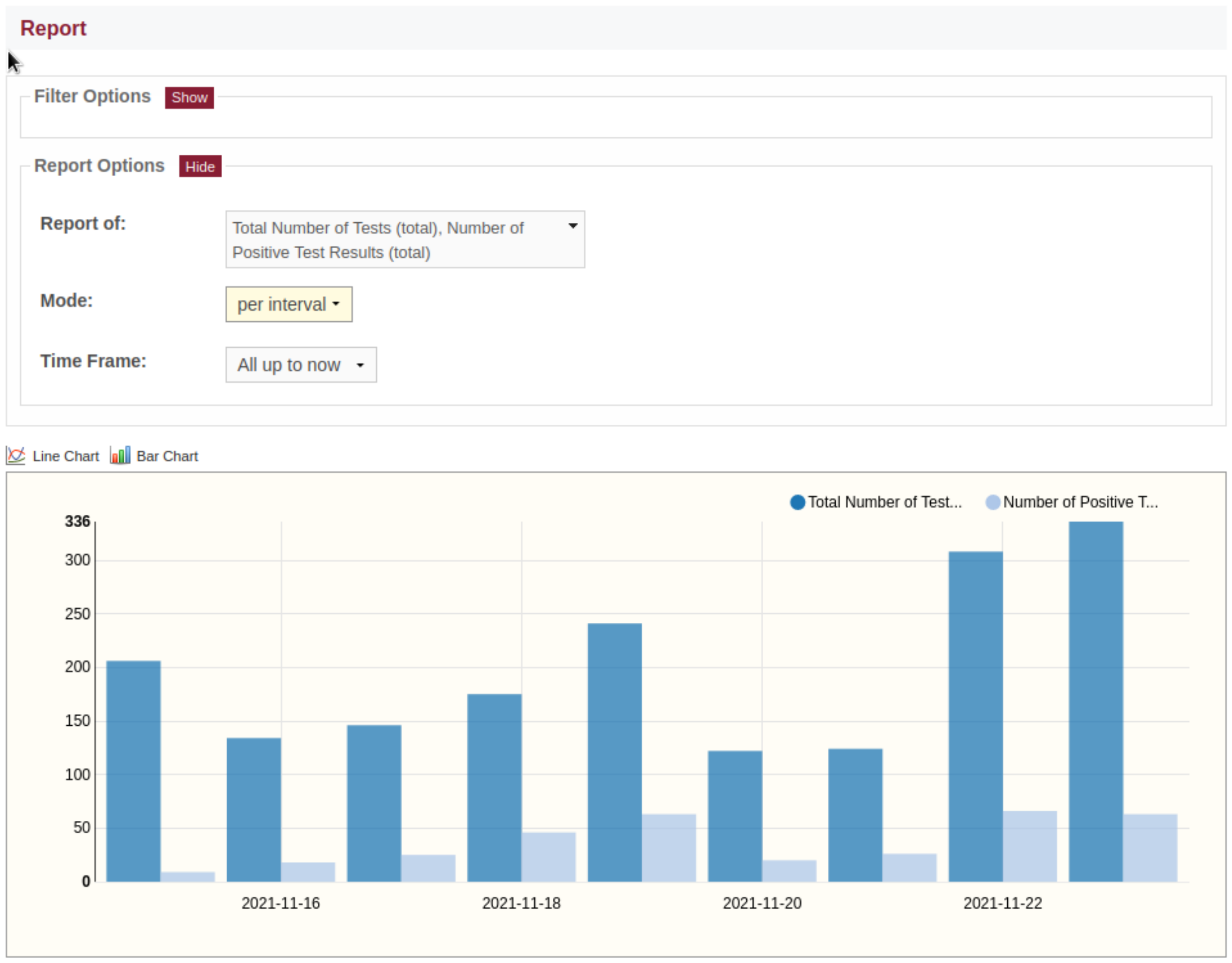
Total Annual Budget (Total) per Locations

Sectors: All

Total:
278 300

Total Annual Budget (Total)	Locations		Total
Sectors	Germany	Sweden	
Education	98300	180000	278300

Fig. 7: Pivot Table with Chart



Report

Filter Options Show

Report Options Hide

Report of: Total Number of Tests (total), Number of Positive Test Results (total)

Mode: per interval

Time Frame: All up to now

(continued from previous page)

```
              ("Last 6 Months", "-6months", "", "weeks"),
              ("Last 3 Months", "-3months", "", "weeks"),
              ("Last Month", "-1month", "", "days"),
              ("Last Week", "-1week", "", "days"),
              ]

timeplot_options = {
    "facts": facts,
    "timestamp": [(T("per interval"), "date,date"),
                  (T("cumulative"), "date"),
                  ],
    "time": timeframes,
    "defaults": {"fact": facts[:2],
                 "timestamp": "date,date",
                 "time": timeframes[-1],
                 },
    }

s3db.configure("disease_testing_report",
               timeplot_options = timeplot_options,
               )
```

The attributes of the `timeplot_options` setting are as follows:

Option	Type	Explanation
facts	list	The selectable facts as tuples (label, expression)
timestamp	list	Selectable time stamps as tuples *(label, expr)* If *expr* contains two comma-separated field selectors, it is interpreted as "start,end". If *expr* is a single field selector, it is interpreted as start date; in this case events are treated as open-ended, and hence facts cumulating over time.
time	list	List of time frames as tuples *(label, start, end, slots)* *start* and *end* can be either absolute dates (ISO-format), or *relative date expressions*, or "". A relative *start* is relative to now. A relative *end* is relative to *start*, or, if no *start* is specified, it is relative to now. *start* "" means the date of the earliest recorded event, *end* "" means now. The *slots* length is the default for the time frame, but can be overridden with an explicit slot-selector (see below).
slots	list	List of tuples *(label, expr)* A separate selector for the slot length is rendered only if this option is configured.
		Otherwise, the slot length is fixed to that specified by the selected time frame option.
defaults	dict	

Relative Time Expressions

The *start* and *end* parameters for the time frame of the report support relative expressions of the form `[<|>][+|-]{n}[year|month|week|day|hour]s`.

The *n* is an integer, e.g.:

```
"-1 year"    # one year back
"+2 weeks"   # two weeks onward
```

Additionally, the < and > markers can be added to indicate the start/end of the respective calendar period, e.g.:

```
"<-1 year"   # one year back, 1st of January
">+2 weeks"  # two weeks onward, Sunday
```

In this context, weeks go from Monday (first day) to Sunday (last day).

Note: Even when using < and > markers, the rule that *end* is relative to *start* still applies.

This can be confusing when using these markers for both interval ends, e.g. the time frame for January 1st to December 31st of last year is **not**:

```
("<-1 year", ">-1 year")
```

but actually:

```
("<-1 year", ">+0 years")
```

…namely, from the beginning of last year to the end of that **same** year.

More intuitive in this case is to specify: `("<-1 year", "+1 year")`.

3.5.6 Summary

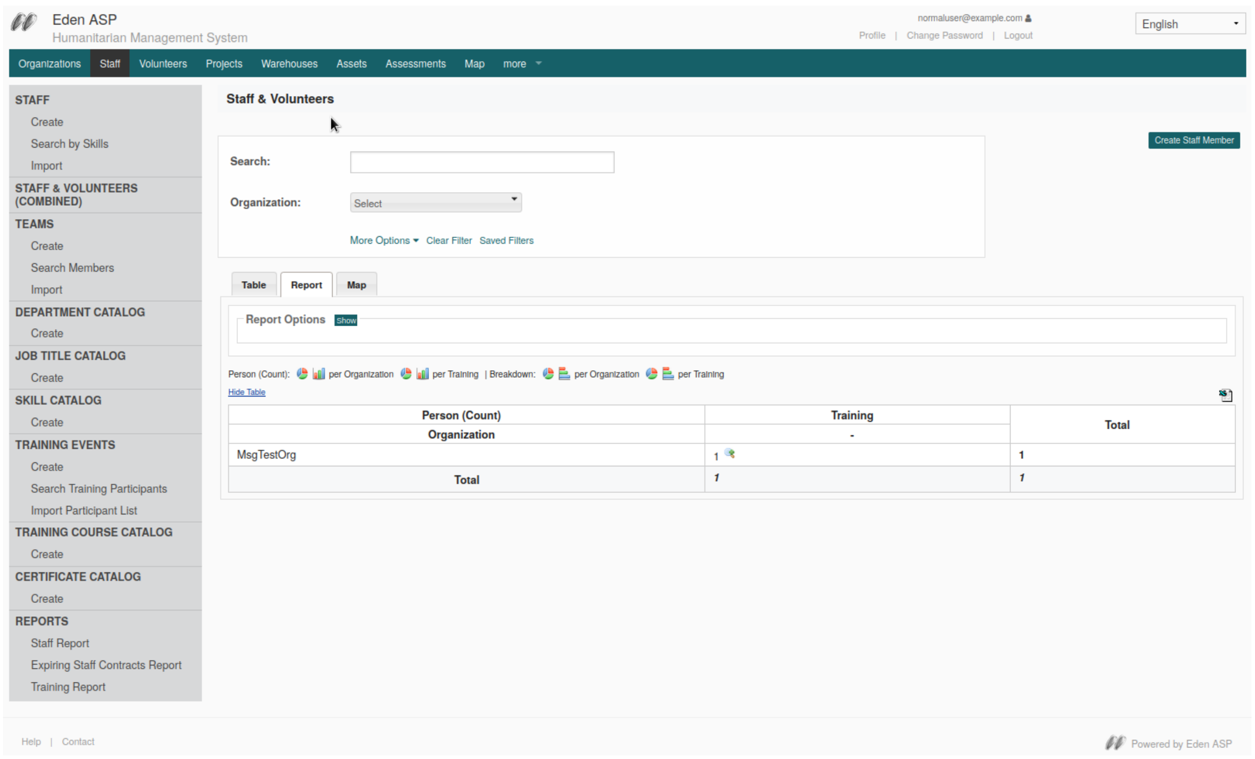
Meta-method with multiple other methods on the same page (on tabs), and a common filter form (end-point: */summary*).

Note: This method requires configuration.

3.5.7 Organizer

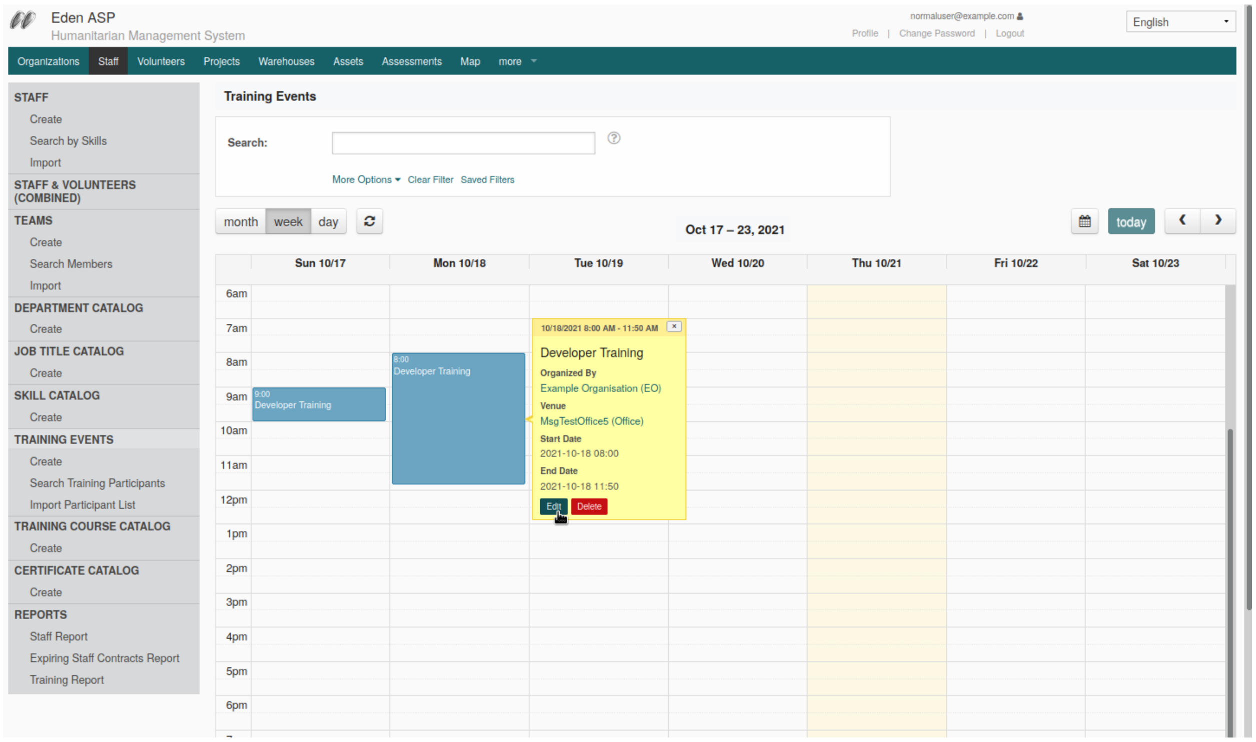
Calendar-based view and manipulation of records (end-point: */organize*)

Note: This method requires configuration of start and end date fields, as well as of popup contents.



Person (Count)	Training	Total
Organization	-	
MsgTestOrg	1	1
Total	1	1

Fig. 8: Summary view with table, report and map tabs, and common filter form.



10/18/2021 8:00 AM - 11:50 AM

Developer Training

Organized By
Example Organisation (EO)

Venue
MsgTestOffice5 (Office)

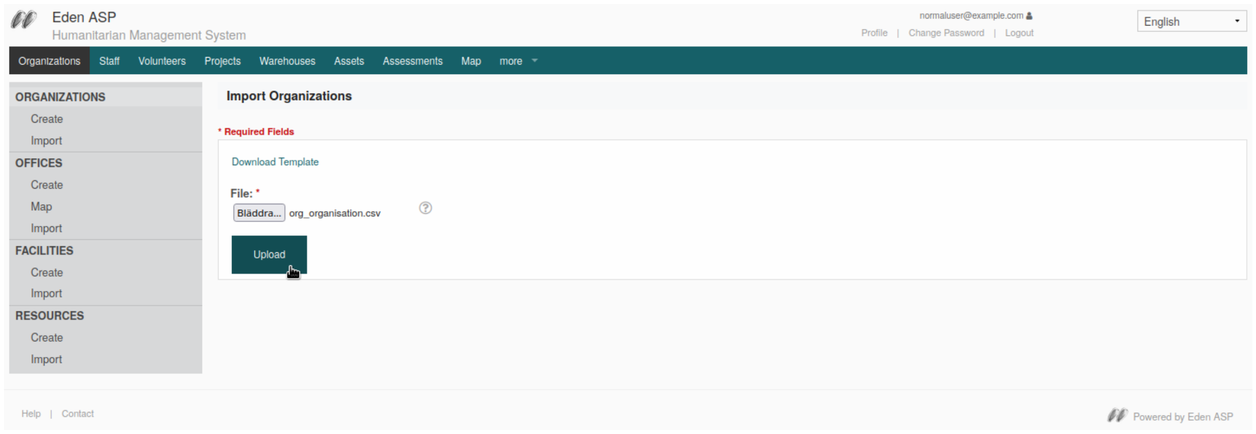
Start Date
2021-10-18 08:00

End Date
2021-10-18 11:50

Fig. 9: Organizer (Weekly Agenda View)

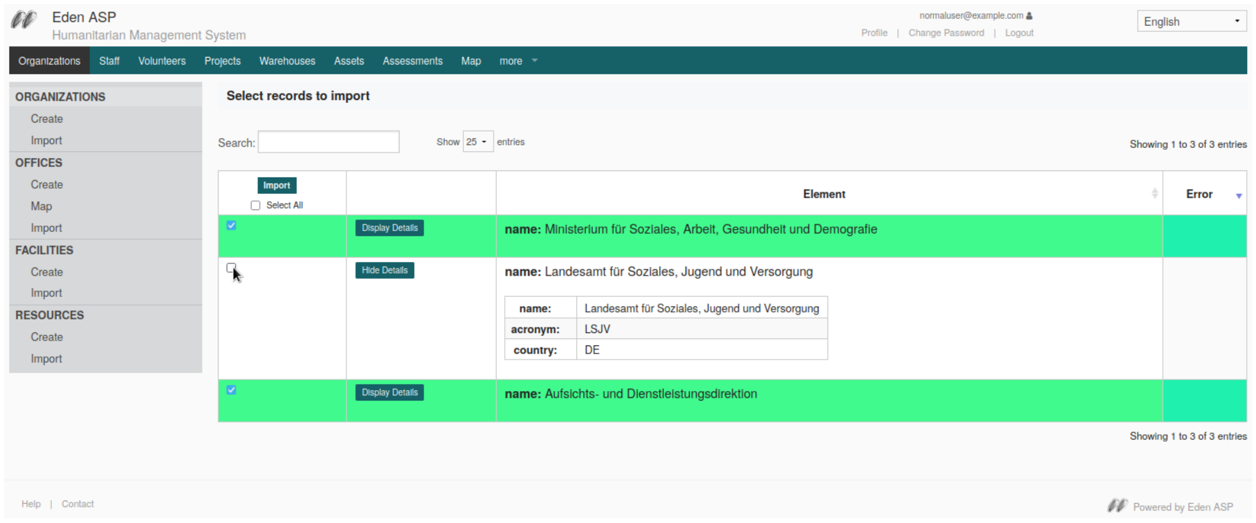
3.5.8 Spreadsheet Importer

Interactive Spreadsheet (CSV/XLS) Importer with review and record selection (end-point: */import*).



File: * org_organisation.csv

Fig. 10: Spreadsheet Importer, Upload Dialog



Import		Element	Error
✓	Display Details	**name:** Ministerium für Soziales, Arbeit, Gesundheit und Demografie	
☐	Hide Details	**name:** Landesamt für Soziales, Jugend und Versorgung **name:** Landesamt für Soziales, Jugend und Versorgung **acronym:** LSJV **country:** DE	
✓	Display Details	**name:** Aufsichts- und Dienstleistungsdirektion	

Showing 1 to 3 of 3 entries

Fig. 11: Spreadsheet Importer, Review and Record Selection

3.6 User Interface Elements

3.6.1 Form Widgets

3.6.2 Filter Widgets and Forms

3.6.3 DataTable

The **DataTable** widget represents a set of records as an interactive HTML table.

DataTables are one of the most common UI features in EdenASP, and a standard aspect of interactive CRUD.

The `DataTable` class implements the server-side functions to configure, build and update a DataTable. The client-side parts are implemented by the *s3.ui.datatable.js* script, using jQuery datatables.

Overview

`class DataTable`(*rfields*, *data*, *table_id=None*, *orderby=None*)

Parameters

- **`rfields`** – the table columns, [S3ResourceField, …]
- **`data`** – the data, [{colname: value, …}, …]
- **`table_id`** – the DOM ID for the <table> element
- **`orderby`** – the DAL orderby expression that was used to extract the data

Note: The first column should be the record ID.

`html`(*totalrows*, *filteredrows*, ***attr*)

Builds the data table HTML.

Parameters

- **`totalrows`** – total number of rows available
- **`filteredrows`** – total number of rows matching filters
- **`attr`** – *build parameters*

Returns
the HTML for the data table widget

Return type
FORM

`json`(*totalrows*, *filteredrows*, *draw*, ***attr*)

Builds a JSON object to update the data table.

Parameters

- **`totalrows`** – total number of rows available
- **`filteredrows`** – total number of rows matching filters
- **`draw`** – unaltered copy of “draw” parameter sent from the client
- **`attr`** – *build parameters*

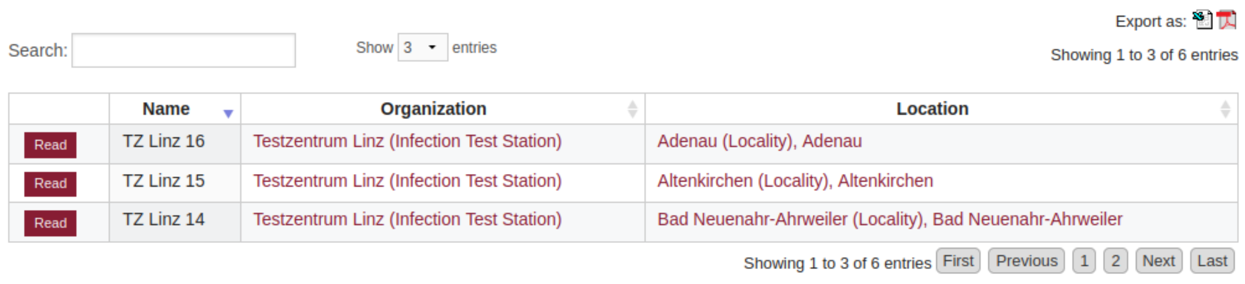
Returns
the JSON data

Return type
str

Example

Typically, *DataTable* views are implemented in CRUD methods.

The following example implements a *DataTable* view for the *org_facility* table, including server-side pagination and Ajax-filtering, like this:


Export as:
Search:
Show 3 entries
Showing 1 to 3 of 6 entries
Showing 1 to 3 of 6 entries First Previous 1 2 Next Last

	Name	Organization	Location
Read	TZ Linz 16	Testzentrum Linz (Infection Test Station)	Adenau (Locality), Adenau
Read	TZ Linz 15	Testzentrum Linz (Infection Test Station)	Altenkirchen (Locality), Altenkirchen
Read	TZ Linz 14	Testzentrum Linz (Infection Test Station)	Bad Neuenahr-Ahrweiler (Locality), Bad Neuenahr-Ahrweiler

```
class FacilityList(CRUDMethod):

    def apply_method(self, r, **attr):

        get_vars = r.get_vars

        # -----------------------------------------------------------------
        # Pagination

        page_length = 25
        if r.interactive:
            # Default limits when page is first loaded
            # - extracting twice the page length here to fill the cache,
            #   so no Ajax-request is required for the first two pages
            start, limit = 0, 2 * page_length
        else:
            # Dynamic limits for subsequent Ajax-requests
            start, limit = self._limits(get_vars, default_limit=page_length)

        # -----------------------------------------------------------------
        # Extract the data, applying client-side filters/sorting

        resource = current.s3db.resource("org_facility")
        fields = ["id", "name", "organisation_id", "location_id"]

        query, orderby, left = resource.datatable_filter(fields, get_vars)
        if query is not None:
            totalrows = resource.count()
            resource.add_filter(query)

        data = resource.select(fields,
                               start = start,
                               limit = limit,
                               left = left,
                               orderby = orderby,
```

(continues on next page)

(continued from previous page)

```
                              count = True,
                              represent = True,
                              )

        filteredrows = data.numrows
        if query is None:
            totalrows = filteredrows

        # -----------------------------------------------------------------
        # Set up the DataTable

        from core import DataTable
        dt = DataTable(data.rfields, data.rows, "facility_list")

        # -----------------------------------------------------------------
        # Configure row actions (before building the DataTable)

        current.response.s3.actions = [{"label": "Read",
                                         "url": URL(args = ["[id]", "read"]),
                                         "_class": "action-btn"
                                         },
                                        ]

        # -----------------------------------------------------------------
        # Build the DataTable

        # Rendering parameters to pass to .html() and .json()
        dtargs = {"dt_pagination": True,
                  "dt_pageLength": page_length,
                  "dt_base_url": URL(args=[], vars={}),
                  }

        if r.interactive:
            # This is the initial page load request
            # - build the HTML:
            dt_html = dt.html(totalrows, filteredrows, **dtargs)
            output = {"items": dt_html}

        elif r.representation == "aadata":
            # Client-side script uses the "aadata" extension to request updates
            # - generate a JSON response:
            draw = int(r.get_vars.get("draw", 1))
            output = dt.json(totalrows, filteredrows, draw, **dtargs)

        else:
            r.error(405, current.ERROR.BAD_FORMAT)

        # View template, includes dataTables.html
        current.response.view = "list.html"

        return output
```

Note: The view template must `include` the *dataTables.html* template to add the necessary JavaScript for the DataTable widget.

Build Parameters

Both build methods *html()* and *json()* accept the same set of keyword arguments to control the build of the DataTable. Most of these arguments are optional (see *example* above for a typical minimum set).

Basic configuration

Basic parameters for the data table.

Keyword	Type	Default	Explanation
dt_ajax_url	str	None	URL for Ajax requests
dt_base_url	str	None	Base URL for exports, usually the resource default URL without any method or query part
dt_dom	str	None	The jQuery datatable “dom” option, determines the order in which elements are displayed - see https://datatables.net/reference/option/dom
dt_formkey	str	None	A form key (XSRF protection for Ajax requests)

Pagination

Parameters for pagination (server-side pagination requires *dt_ajax_url*).

Keyword	Type	Default	Explanation
dt_pagination	bool	True	Enable/disable pagination
dt_pageLength	int	25	Default number of records that will be shown per page - the user can change this using the length menu
dt_lengthMenu	tuple	[[25, 50, -1], [25, 50, "All"]]	The menu options for the page length
dt_pagingType	str	deployment setting	How the pagination buttons are displayed - settings.ui.datatables_pagingType (default full_numbers) - see https://datatables.net/reference/option/pagingType

Searching

Parameters to control the search box.

Keyword	Type	Default	Explanation
dt_searching	bool	True	Enable/disable search-field

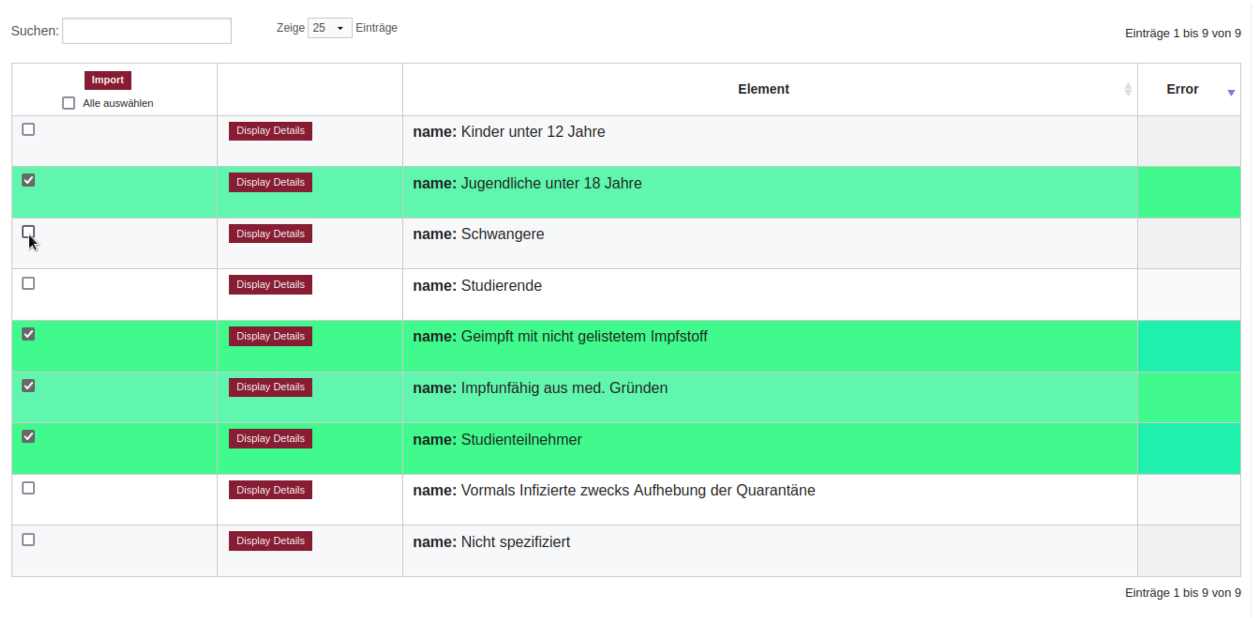
Note: The search box should normally be disabled when using separate filter forms.

Row Actions

Keyword	Type	Default	Explanation
dt_row_actions	list	None	list of actions (each a dict) - overrides current.response.s3.actions
dt_action_col	int	0	The column where the action buttons will be placed

Bulk Actions

Bulk-action DataTable views render an additional column with checkboxes to select rows and then perform actions “in bulk” for all selected rows with a single button click.



Suchen: Zeige 25 Einträge Einträge 1 bis 9 von 9

Import ☐ Alle auswählen		Element	Error
☐	Display Details	**name:** Kinder unter 12 Jahre	
☑	Display Details	**name:** Jugendliche unter 18 Jahre	
☐	Display Details	**name:** Schwangere	
☐	Display Details	**name:** Studierende	
☑	Display Details	**name:** Geimpft mit nicht gelistetem Impfstoff	
☑	Display Details	**name:** Impfunfähig aus med. Gründen	
☑	Display Details	**name:** Studienteilnehmer	
☐	Display Details	**name:** Vormals Infizierte zwecks Aufhebung der Quarantäne	
☐	Display Details	**name:** Nicht spezifiziert	

Einträge 1 bis 9 von 9

Fig. 12: Spreadsheet Importer: DataTable with bulk action column.

Keyword	Type	Default	Explanation
dt_bulk_actions	list	None	list of labels for the bulk actions
dt_bulk_col	int	0	The column in which the checkboxes will appear, - default: insert bulk actions as first column
dt_bulk_single	bool	False	allow only one row to be selected
dt_bulk_selected	list	None	list of (pre-)selected items

Note: Bulk-actions require server-side processing of the DataTable FORM upon submit.

Grouping

Group table rows by column values.

Keyword	Type	Default	Explanation
dt_group	list	None	The column(s) that is(are) used to group the data
dt_group_totals	list	None	The number of record in each group. - this will be displayed in parenthesis after the group title.
dt_group_titles	list	None	The titles to be used for each group. These are a list of lists with the inner list consisting of two values, the repr from the db and the label to display. This can be more than the actual number of groups (giving an empty group).
dt_group_space	bool	False	Insert a space between the group heading and the next group
dt_shrink_groups	str	None	If set then the rows within a group will be hidden two types are supported, 'individual' and 'accordion'
dt_group_types	str	None	The type of indicator for groups that can be 'shrunk' Permitted valies are: 'icon' (the default) 'text' and 'none'

Contents Rendering

Keyword	Type	Default	Explanation
dt_text_maximum_len	int	80	The maximum length of text before it is condensed
dt_text_condense_len	int	75	The length displayed text is condensed down to

Styles

Keyword	Type	Default	Explanation
dt_styles	dict	None	dictionary of styles to be applied to a list of ids - example: {"warning" : [1,3,6,,9], "alert" : [2,10,13]}
dt_col_widths	dict	None	dictionary of columns to apply a width to - example: {1 : 15, 2 : 20}

Other Features

Keyword	Type	Default	Explanation
dt_double_scroll	bool	False	Render double scroll bars (top+bottom), only available with settings.ui.datatables_responsive=False

Response Parameters

to be written

Deployment Settings

to be written

3.6.4 Card Lists

3.7 Tools

The core.tools library provides a number of tools for common application tasks, e.g. representing data, handling date and time, or importing data.

This section describes the tools, and their most relevant functions.

3.7.1 Bulk Importer

The **BulkImporter** is a tool to run a series of data import tasks from a configuration file. It is most commonly used during the first run of the application, to pre-populate the database with essential data (a process called *prepop*).

The individual import task handlers of the BulkImporter can also be used standalone, e.g. in upgrade/maintenance scripts, or for database administration from the CLI.

Configuration File

Configuration files for the BulkImporter are CSV-like files that must be named `task.cfg`, and are typically placed in the template directory to be picked up by the first-run script.

Listing 9: Example of tasks.cfg

```
# Roles
*,import_roles,auth_roles.csv
# GIS
gis,marker,gis_marker.csv,marker.xsl
gis,config,gis_config.csv,config.xsl
gis,hierarchy,gis_hierarchy.csv,hierarchy.xsl
gis,layer_feature,gis_layer_feature.csv,layer_feature.xsl
```

Tip: This file format differs from normal CSV in that it allows for comments, i.e. everything from # to the end of the line is ignored by the parser.

Each line in the file specifies a *task* for the BulkImporter. The general format of a task is:

```
<prefix>,<name>,<filename>,<xslt_path>
```

By default, tasks is the S3CSV import handler (*import_csv*). In this case, the task parameters are:

Parameter	Meaning
prefix	The module prefix of the table name (e.g. *org*)
name	The table name without module prefix (e.g. *organisation*)
filename	- the source file name (if located in the same directory as tasks.cfg), or - a file system path relative to *modules/templates*, or - an absolute file system path, or - a HTTP/HTTPs URL to fetch the file from
stylesheet	- the name of the transformation stylesheet (if located in *static/formats/s3csv/<prefix>*), or - a file system path relative to *static/formats/s3csv*, or - a file system path starting with `./` relative to the location of the CSV file

Import Handlers

It is possible to override the default handler for a task with a *prefix* `*`, and then specifying the import handler with the *name* parameter, i.e.:

```
*,<handler>,<filename>,<arg>,<arg>,...
```

In this case, the number and meaning of the further parameters depends on the respective handler:

Handler	Task Format, Action
import_xml	`*,import_xml,<filename>,<prefix>,<name>,<dataformat>,<source_type>` - import XML/JSON data using *static/formats/<dataformat>/import.xsl* - *source_type* can be `xml` or `json`
import_roles	`*,import_roles,<filename>` - import user roles and permissions from CSV with a special format
import_users	`*,import_roles,<filename>` - import user accounts with special pre-processing of the data
import_images	`*,import_images,<filename>,tablename,keyfield,imagefield` - import image files and store them in record of the specified table - source file is a CSV file with columns *id* and *file* - records are identified by *keyfield* matching the *id* in the source file
schedule_task	`*,schedule_task,,taskname,args,vars,params` - schedule a task with the scheduler - *args*, *vars* and *params* are JSON strings, but can use single quotes - *args* (list) and *vars* (dict) are passed to the task function - *params* (dict) specifies the task parameters, e.g. frequency of execution - second task parameter (filename) is empty here (not a typo)!

It is possible to run the import task handlers standalone, e.g.:

Listing 10: Running a task handler function standalone

```
from core import BulkImporter
path = os.path.join(current.request.folder, "modules", "templates", "MYTEMPLATE", "auth_
↪roles.csv")
error = BulkImporter.import_roles(path)
```

The arguments for the handler function are the same as for the task line in the tasks.cfg (except * and handler name of course). All handler functions return an error message upon failure (or a list of error messages, if there were multiple errors) - or None on success.

Note: When running task handlers standalone (e.g. in a script, or from the CLI), the import result will **not** automatically be committed - an explicit `db.commit()` is required.

Task Runner

The task runner is a BulkImporter **instance**. To run tasks, the `perform_tasks` method is called with the path where the *tasks.cfg* file is located:

```
from core import BulkImporter
bi = BulkImporter()

path = os.path.join(current.request.folder, "modules", "templates", "MYTEMPLATE")
bi.perform_tasks(path)
```

Important: The task runner automatically commits all imports - i.e. *perform_tasks* cannot be rolled back!

Template-specific Task Handlers

It is possible for templates to add further task handlers to the BulkImporter, e.g. to perform special (import or other) tasks during prepop.

Listing 11: Template-specific task handler for the BulkImporter, in config.py

```
# Define the task handler:
# - must take filename as first argument
# - further arguments are freely definable, but tasks must match
#   this signature
def special_import_handler(filename, arg1, arg2):
    ...do something with filename and args

# Configure a dict {name: function} for template-specific task handlers:
settings.base.import_handlers = {"import_special": special_import_handler}
```

This also allows to override existing task handlers with template-specific variants.

With this, tasks for the new handler can be added to tasks.cfg like:

```
*,import_special,<filename>,<arg1>,<arg2>
```

Note: When received by the handler, the *filename* will be completed with a path, (see interpretation of *filename* in *tasks.cfg*). All other parameters are passed-in unaltered.

However, the *filename* parameter can be left empty, and/or get ignored by the task handler, if a file name is not required for the task.

CHAPTER

FOUR

HOW TO DEPLOY EDEN ASP APPLICATIONS

Eden ASP is normally deployed behind a separate front-end web server (e.g. nginx) using WSGI/uWSGI to plugin web2py. This section describes how to setup a production instance of an Eden ASP application on a Debian server.

CHAPTER

FIVE

EXTENDING EDEN ASP

5.1 Implementing Controllers

5.1.1 Basic Concepts

CRUDRequest

5.1.2 Implementing CRUD Controllers

crud_controller

prep

postp

5.2 Implementing Data Models

5.2.1 Basic Concepts

Model Loader *s3db*

Resources

Components

Super-Entities

Field Selectors and Resource Queries

5.2.2 Defining Tables

Subclassing DataModel

model()

defaults()

mandatory()

Exposing names

Defining Tables

5.2.3 Table Configuration

CRUD Hooks

Linking to Super-Entities

CRUD Strings

Adding Components

Adding Methods

5.2.4 Reusable Fields

Common Field Functions

Meta-Fields

Implementing Reusable Fields

5.2.5 Field Representation

Common Representation Functions

Foreign Key Bulk Representation (S3Represent)

5.3 Adding new themes

5.4 Adding new Form Widgets

CHAPTER

SIX

INDICES AND TABLES

- genindex

INDEX

A

auth.s3_has_role()
    built-in function, 23
auth.s3_logged_in()
    built-in function, 23

B

built-in function
    auth.s3_has_role(), 23
    auth.s3_logged_in(), 23
    s3db.add_components(), 21
    s3db.clear_config(), 21
    s3db.configure(), 20
    s3db.get_config(), 21
    s3db.get_method(), 22
    s3db.set_method(), 22
    s3db.table(), 20

D

DataTable (*built-in class*), 40

H

html() (*DataTable method*), 40

J

json() (*DataTable method*), 40

S

s3db.add_components()
    built-in function, 21
s3db.clear_config()
    built-in function, 21
s3db.configure()
    built-in function, 20
s3db.get_config()
    built-in function, 21
s3db.get_method()
    built-in function, 22
s3db.set_method()
    built-in function, 22
s3db.table()
    built-in function, 20