ezProbs Guide

Richard Pöttler 1530149

May 27, 2021

Contents

1 Introduction

1.1 Architecture

`ezProbs` is basically a web application where various problems can be listed. Numerical values can be configured using sliders.

1.1.1 Problems

A problem consists of the following sections:

Description general description of the problem

Plot optional plot of the current calculated result

Parameters configurable parameters for the problem

Solution description of the calculation

1.2 Requirements

On a Debian system the following packages must be installed:

- `python3-scipy`
- `python3-numpy`
- `python3-pandas`
- `python3-matplotlib`
- `python3-flask`
- `python3-svgwrite`

1.3 Development

The following commands are used to start the development server:

```
export FLASK_ENV=development
export FLASK_APP=ezprobs
flask run
```

1.4 Configuration

The following parameters can be set in the `config.ini`:

- `server.secret_key` a string to set the password with which cookies are enctrypted on the client
- `application.submit_on_change` a boolean to controll whether the calculation should be kicked off once a parameter slider is changed

2 Problems

In this section the steps needed to create a new problem will be discussed.

2.1 Skeleton

In this subsection the bare minimum of steps will be discussed to create the skeleton of a problem.

2.1.1 Template

The template controls how the problem is displayed on on the client. All templates are basically `HTML` processed by the Jinja template engine.

To create a new problem a new file with a descriptive name must be created under `ezprobs/templates/problems`. This file should have the extension `.html`. The content of the file might look like something like this:

```
{% extends "problem.html" %}

{% block title %}TEST{% endblock %}

{% block description %}
test description
{% endblock %}

{% block solution %}
test solution
{% endblock %}
```

The `extends` directive is mandatory and causes the generation of e.g. the parameter section. The `title`, `description` and `solution` blocks are also mandatory. The `title` block is used to set the title of the problem. The remaining blocks will be described in subsequent sections.

During this demonstration the file will be saved as `ezprobs/templates/problems/test.html`.

2.1.2 Python Code

All problems are implemented as Flask blueprints.

The problem python files should be saved under the `ezprobs/problems` directory with a representative name. The minimum code needed to implement a problem is as follows:

```
from flask import Blueprint, render_template

bp = Blueprint("test", __name__)

@bp.route("/", methods=["POST", "GET"])
def index():
    return render_template("problems/test.html")
```

The name provided to the `Blueprint` constructer must be unique across the whole application. The template provided to the `render_template` function must be the filename of the previously created template.

For this demonstration the file will be called `ezprobs/problems/test.py`.

2.1.3 Linking

To make the problem available in the Flask application it must be linked in it. This will be done by modifying `ezprobs/__init__.py` by adding the following lines to make the demonstration example available:

```
import ezprobs.problems.test
app.register_blueprint(
    problems.test.bp, url_prefix="/problems/test"
)
```

This loads the newly created module and makes the blueprint available under the given url. The problem should be acessible by pointing the browser to http://localhost:5000/problems/test/ if the development server is running.

The values must be changed to the names of the current problem to add.

By convention problems should be linked under `/problems`.

To make the problem available in the menu bar a new entry must be added to the the `app.config["problems"]` dictionary. To make The `test` problem available a new `Test Runs` section is added and the problem is named `Test`.

```
app.config["problems"] = {
    "Hydraulics": {
        "Flow Regime Transition": "flow_regime_transition_fit_3",
        "Pressure Pipe": "pressure_pipe",
    },
    "Mathematics": {
        "XY Problem": "xy",
    },
    "Test Runs": {
        "Test": "test",
    },
}
```

2.2 Description

The `description` block in the template is used to describe the problem at hand.

2.3 Parameters

It is possible to customize the problem's calculation with parameters which can be altered to the user's choosing. For this a instance of `ezprobs.problems.Parameter` must be created and passed to the `render_template` funciton.

The `Parameter` constructor takes the following arguments:

- `name` with which the value of the parameter should be read from the resulting request.
- `display` name of the value on the page
- `val_min` minimum value of the slider
- `val_max` maximum value of the slider
- `val_step` increment of the value per one slider tick
- `val_initial` initial value of the parameter on the slider
- `unit` optional parameter which denotes the unit used for the parameter
- `description` optional description of the parameter

To get the Parameter section generated with an example parameter the `index` function must be changed as follows:

```
@bp.route("/", methods=["POST", "GET"])
def index():
    from ezprobs.problems import Parameter
    param_a = Parameter(
        "a",
        "a_display",
        0,
        10,
        1,
        5,
```

```
        unit="kN",
        description="some description",)

    return render_template("problems/test.html", parameters=[param_a])
```

To get the submitted value from the request it has to be retrieved from the `POST` header of the request with the `name` set in the `Parameter` constructor. In the example code the new `index` function looks as follows:

```
@bp.route("/", methods=["POST", "GET"])
def index():
    a = 5

    from flask import request
    if request.method == 'POST':
        a = int(request.form['a'])

    from ezprobs.problems import Parameter
    param_a = Parameter(
        "a",
        "a_display",
        0,
        10,
        1,
        a,
        unit="kN",
        description="some description",)

    return render_template("problems/test.html", parameters=[param_a])
```

First the a variable is created with it's initial value. If the request is a `POST` request the value is fetched and cast to the appropriate datatype. It is good practice to initialize the parameter with the value from the request.

2.4 Solution

The solution section will be generated as soon as a `solution` parameter is passed to the `render_template` function. The `solution` parameter is a dictionary holding the names and values of variables which should be displayed in the solution section. In the solution section the dictionary can be accessed like any other Jinja variable.

The privious example can be altered as follows to pass the `a` and `result` variables:

```
@bp.route("/", methods=["POST", "GET"])
def index():
    a = 5

    from flask import request
    if request.method == 'POST':
        a = int(request.form['a'])

    from ezprobs.problems import Parameter
    param_a = Parameter(
        "a",
        "a_display",
        0,
        10,
        1,
        a,
        unit="kN",
        description="some description",)
```

```
    result = a + 5

    return render_template("problems/test.html",
                           parameters=[param_a],
                           solution={"a": a, "result": result})
```

The solution dictionary then can be used by changing the solution section as follows:

```
{% block solution %}
$$result = {{ solution.a }} + 5 = {{ solution.result }}$$
{% endblock %}
```

2.5 Images

2.5.1 Static Images

It is possible to use static images in the description or solution section. First the imagefile must be saved to the `ezprobs/static/images` directory. Afterwards the following snippet can be used to display the image:

```
<div class="container" align="center">
  <figure class="figure">
    <img src="{{ url_for('static', filename='images/test.png') }}"
         class="figure-img img-fluid rounded"
         alt="alt test">
    <figcaption class="figure-caption">test description</figcaption>
  </figure>
</div>
```

`test.png` must be adapted to the filename of the actual image which should be shown. The `alt="alt test"` is optional and `alt test` should be exchanged for the text which should be shown if the image could not be loaded correctly. The `figurecaption` tag is also optional and provides a caption fot the image. `test description` should be exchanged for the caption of the image.

2.5.2 Graphs

Graphs are created using Matplotlib. To do so all needed values must be saved to the `session` to preserve them over the HTTP requests. The easiest way to do so is by using the solution dictionary and add it to the session as the `solution` parameter using the following snippet:

```
s = {'name': value}

from flask import session
session["solution"] = s
```

Afterwards a new flask endpoint is created and the bytes of the image created by Matplotlib are streamed to the client. All needed values have to be retrieved from the `session`. In our example this could look like something like this:

```
@bp.route("/", methods=["POST", "GET"])
def index():
    a = 5

    from flask import request
    if request.method == 'POST':
        a = int(request.form['a'])

    from ezprobs.problems import Parameter
    param_a = Parameter(
        "a",
        "a_display",
        0,
```

```
        10,
        1,
        a,
        unit="kN",
        description="some description",)

    result = a + 5

    s = {"a": a, "result": result}

    from flask import session
    session["solution"] = s

    return render_template("problems/test.html",
                           parameters=[param_a],
                           solution=s)


@bp.route("/plot")
def plot_function():
    from flask import session
    a = session["solution"]["a"]

    import matplotlib.pyplot as plt
    fig, ax = plt.subplots()
    x = [0, 10]
    y = [i * a for i in x]
    ax.plot(x, y)

    from io import BytesIO
    buffer = BytesIO()
    fig.savefig(buffer, format="png")

    from flask import Response
    return Response(buffer.getvalue(), mimetype="image/png")
```

Here the new endpoint created is named `plot`. Once the problem was displayed the plot can be viewed under the URL http://localhost:5000/problems/test/plot.

It is generally recommended to reduce the amount of data saved to the session.

The plot can also be included in the solution section using the following snippet:

```
<div class="container" align="center">
  <figure class="figure">
    <img src="plot"
         class="figure-img img-fluid rounded"
         alt="alt plot">
    <figcaption class="figure-caption">plot description</figcaption>
  </figure>
</div>
```

The snippet is similar to the one for the static images but the `src` attribute of the `img` tag now points to the newly defined endpoint.

2.5.3 Vector Graphics

Vector graphics are created using svgwrite. The procedure is similar to the one when creating plots. First create the endpoint, then assemble the `Drawing` and stream it using the `image/svg+xml` mime type.

The example code can be changed to contain the following method:

```
@bp.route("/svg")
def display_svg():
```

```
    from flask import session
    a = session["solution"]["a"]

    from svgwrite import Drawing
    dwg = Drawing()
    dwg.add(dwg.circle(center=(a, a), r=a))

    from flask import Response
    return Response(dwg.tostring(), mimetype="image/svg+xml")
```

This creates a new endpoint called `svg` and can be accessed through pointing the browser to http://localhost:5000/probl
Displaying the image is done analog to displaying plots.

2.5.4 Plots

Optionally a Plot section can be rendered when displaying the problem. To do this a `ezprobs.problems.Plot` instance must be created and passed to the `render_template`function as `plot` parameter.

The `ezprobs.Plot` constructor takes the following parameters:

- `url` URL of the image or plot to display
- `alt` optional alterante text to display if the plot can't be shown
- `description` optional description text for the plot

The example code can be altered to show the resulting plot in the actual plot section.

```
@bp.route("/", methods=["POST", "GET"])
def index():
    a = 5

    from flask import request
    if request.method == 'POST':
        a = int(request.form['a'])

    from ezprobs.problems import Parameter
    param_a = Parameter(
        "a",
        "a_display",
        0,
        10,
        1,
        a,
        unit="kN",
        description="some description",)

    result = a + 5

    s = {"a": a, "result": result}

    from flask import session
    session["solution"] = s

    from ezprobs.problems import Plot
    p = Plot("plot", "plot alt", "plot description")

    return render_template("problems/test.html",
                           parameters=[param_a],
                           solution=s,
                           plot=p)
```

The `plot` endpoint has to be defined previously. Special care must be taken to sucessfully initialize and pass all needed variables to the endpoint function.

2.6 Mathematical Expressions

To render mathematical expressions MathJax is used. This enables the usage of LaTeX in the `description` and `solution` blocks.

To use the LaTeX math mode for inline expressions they have to be enclosed in `\(\)` like:

```
The discharge is given by \(Q\).
```

To render a single line equation it is enough to enclose it with `$$` like:

```
$$f(x) = a \cdot x + b$$
```

For multiline equations an align LaTeX environment should be used like this:

```
$$
\begin{align}
f(x) &= a \cdot x + b \\
g(x) &= c \cdot x + d \\
\end{align}
$$
```