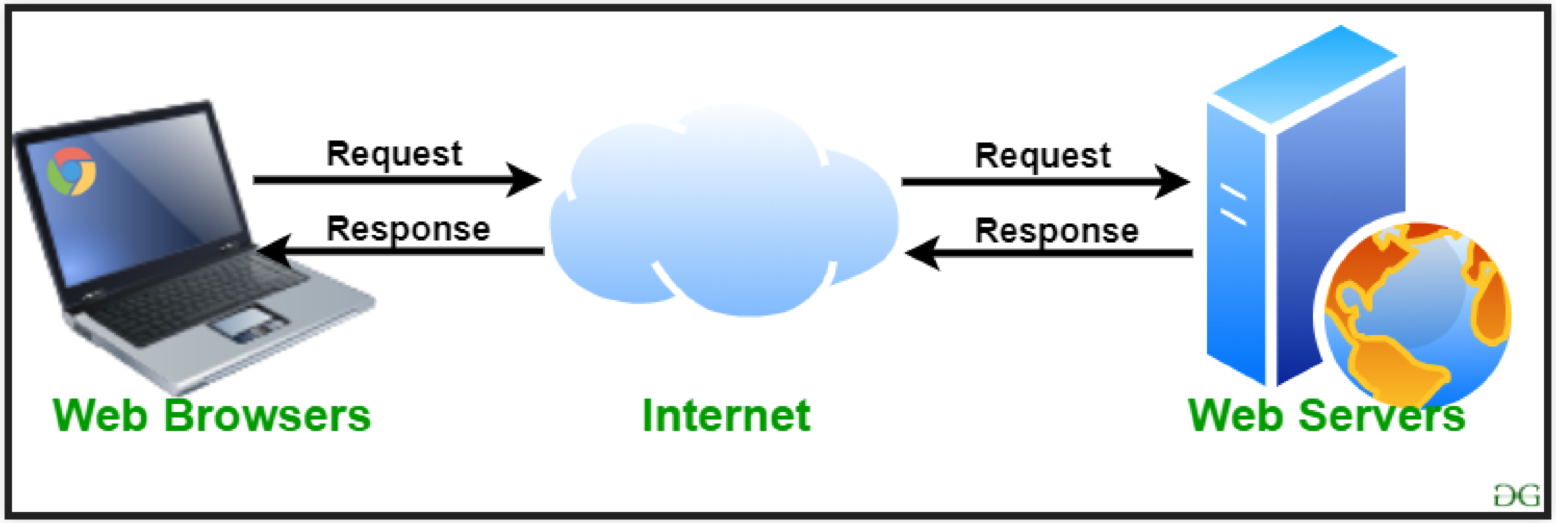
WEB SERVERS

ISD COURSE

Maura Pintor - maura.pintor@unica.it

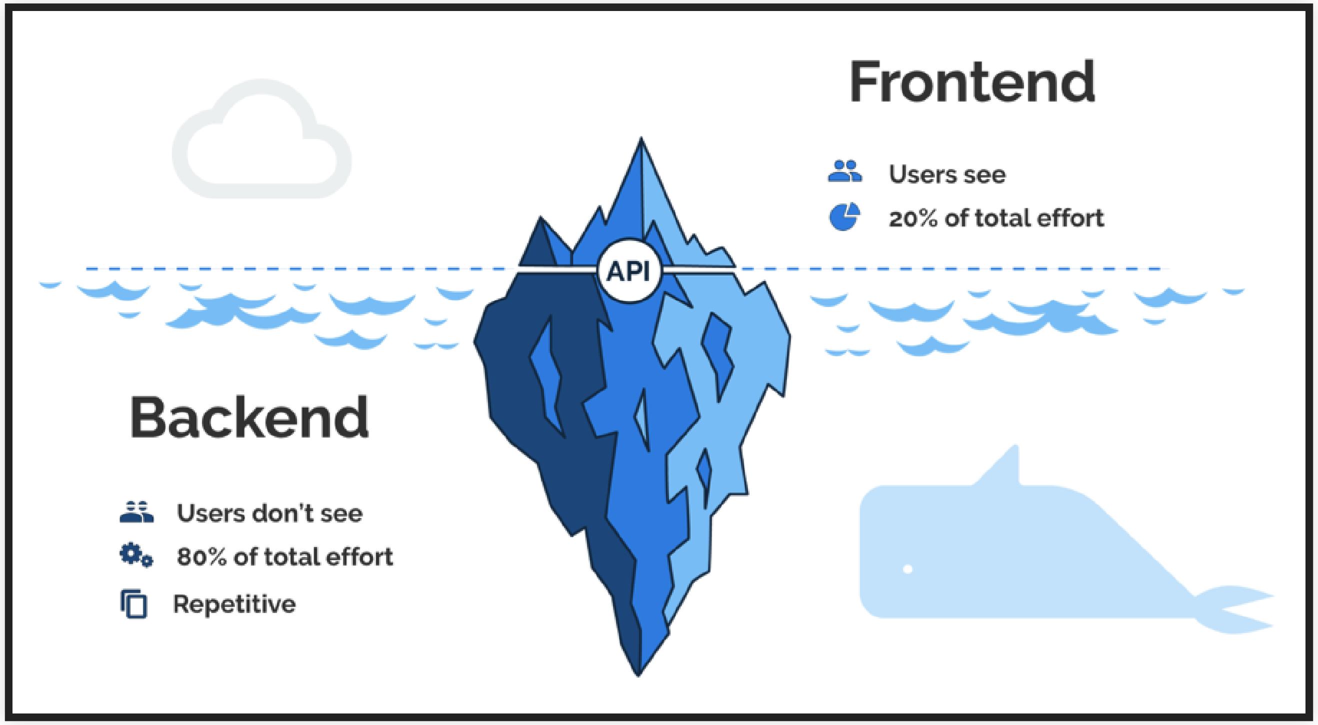
PART 0 : WEB SERVERS BASICS

WEB SERVER FOR THE USER



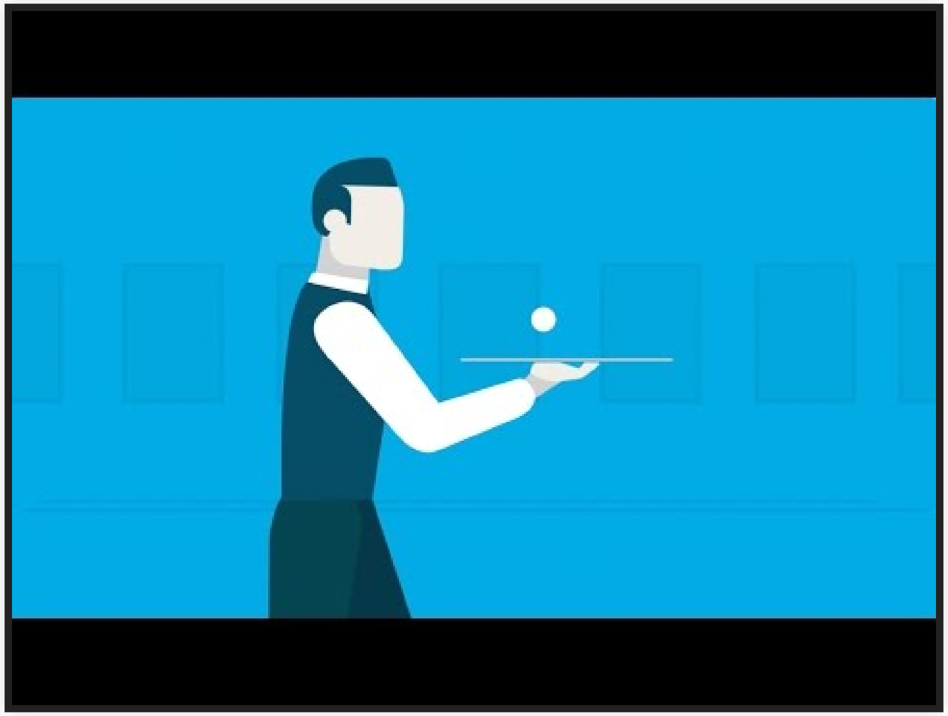
More info here.

WEB SERVER FOR THE DEVELOPER



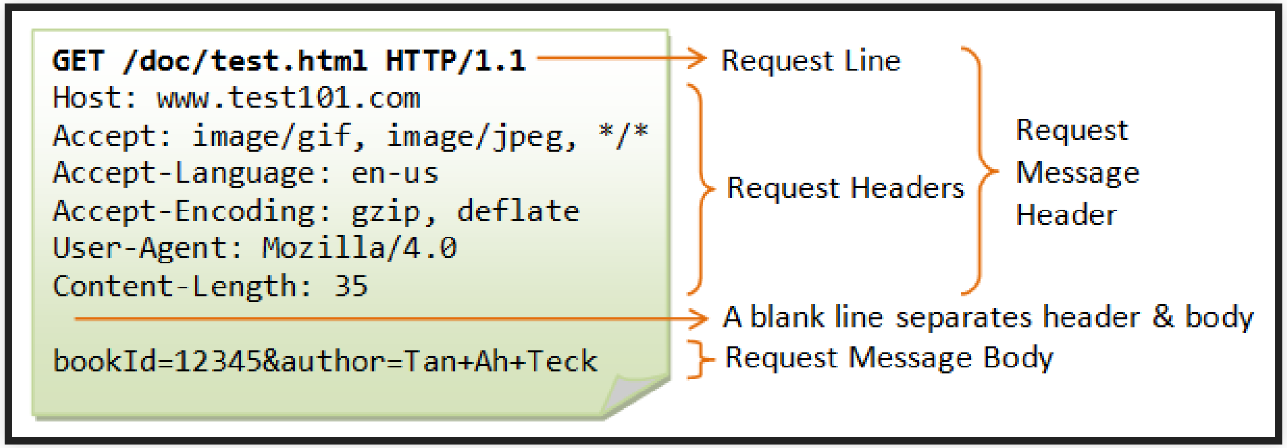
More info here.

API



More info here.

HTTP

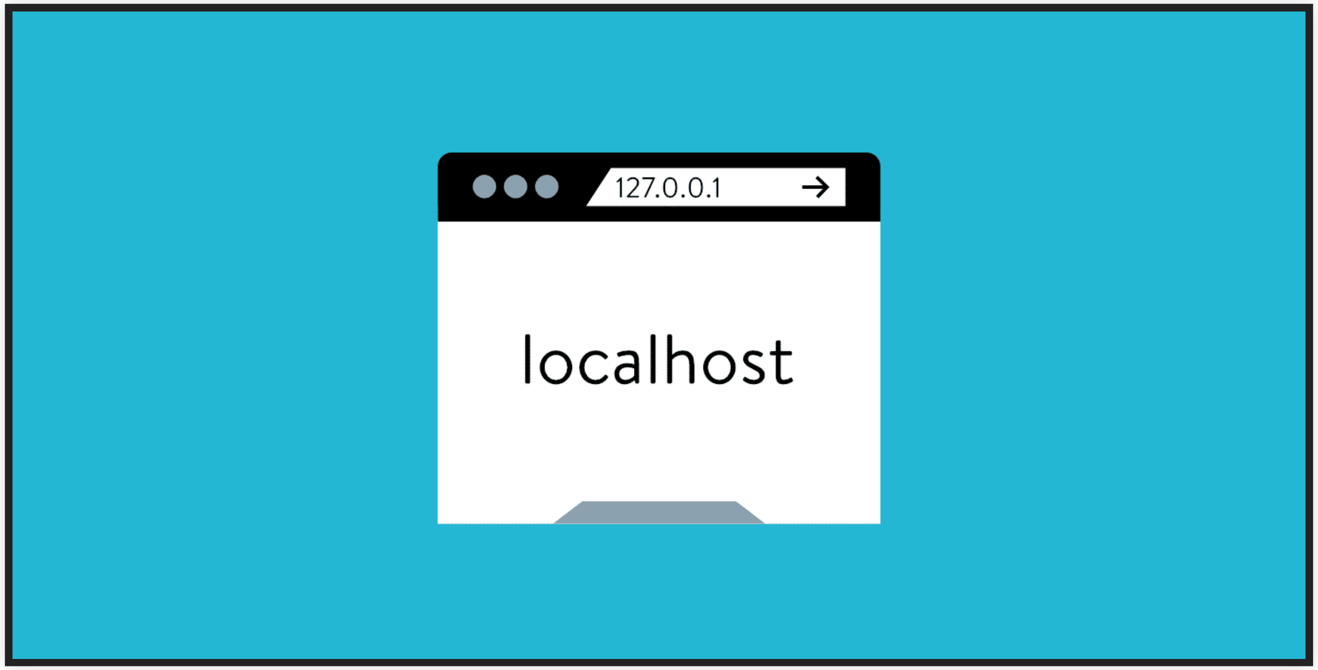


```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

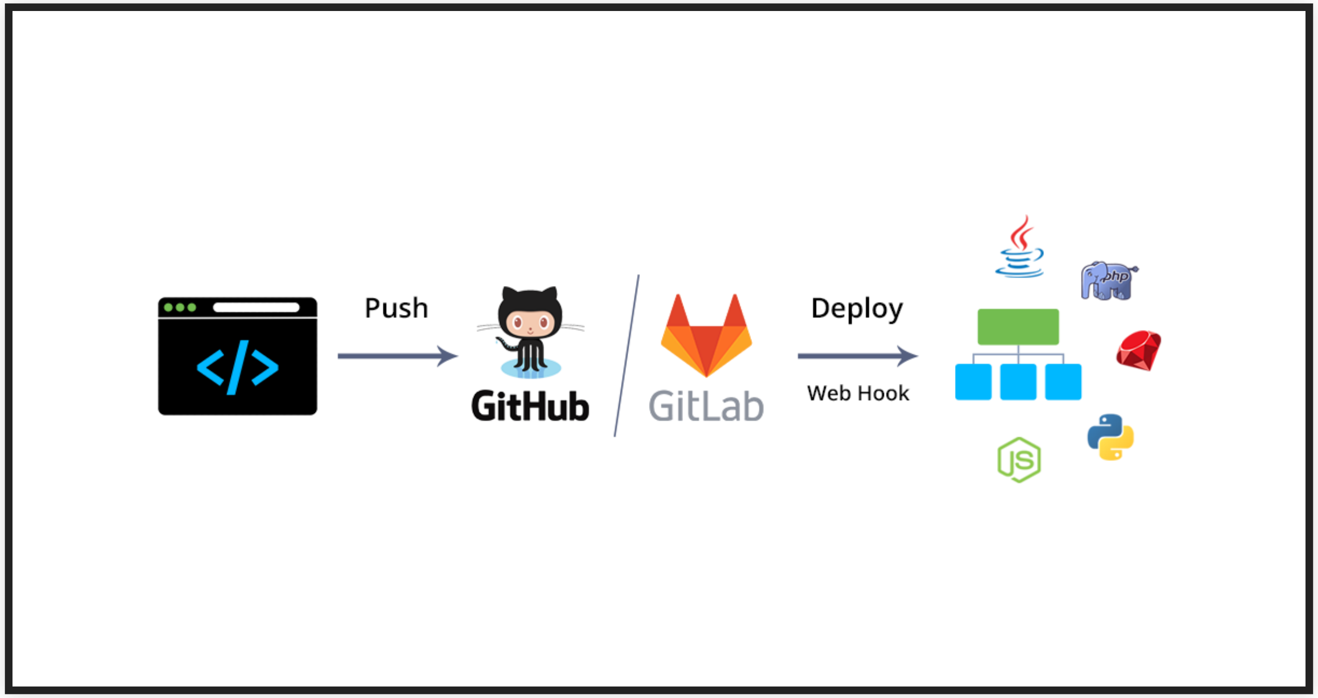
More info here.

LOCALHOST



More info here.

DEPLOY



deploy resources = make them ready to be used

PART I : GETTING STARTED

We will run a simple web server that will show to **logged-in users** a **webpage**.

We will cover:

- web server setup
- connection to database
- login logic

We will not cover:

- deployment
- bootstrap
- security enhancements

But you should have a look into those.

Prerequisites:

- install python3 (hope you already have it)
- install pip (same as above)
- install Flask

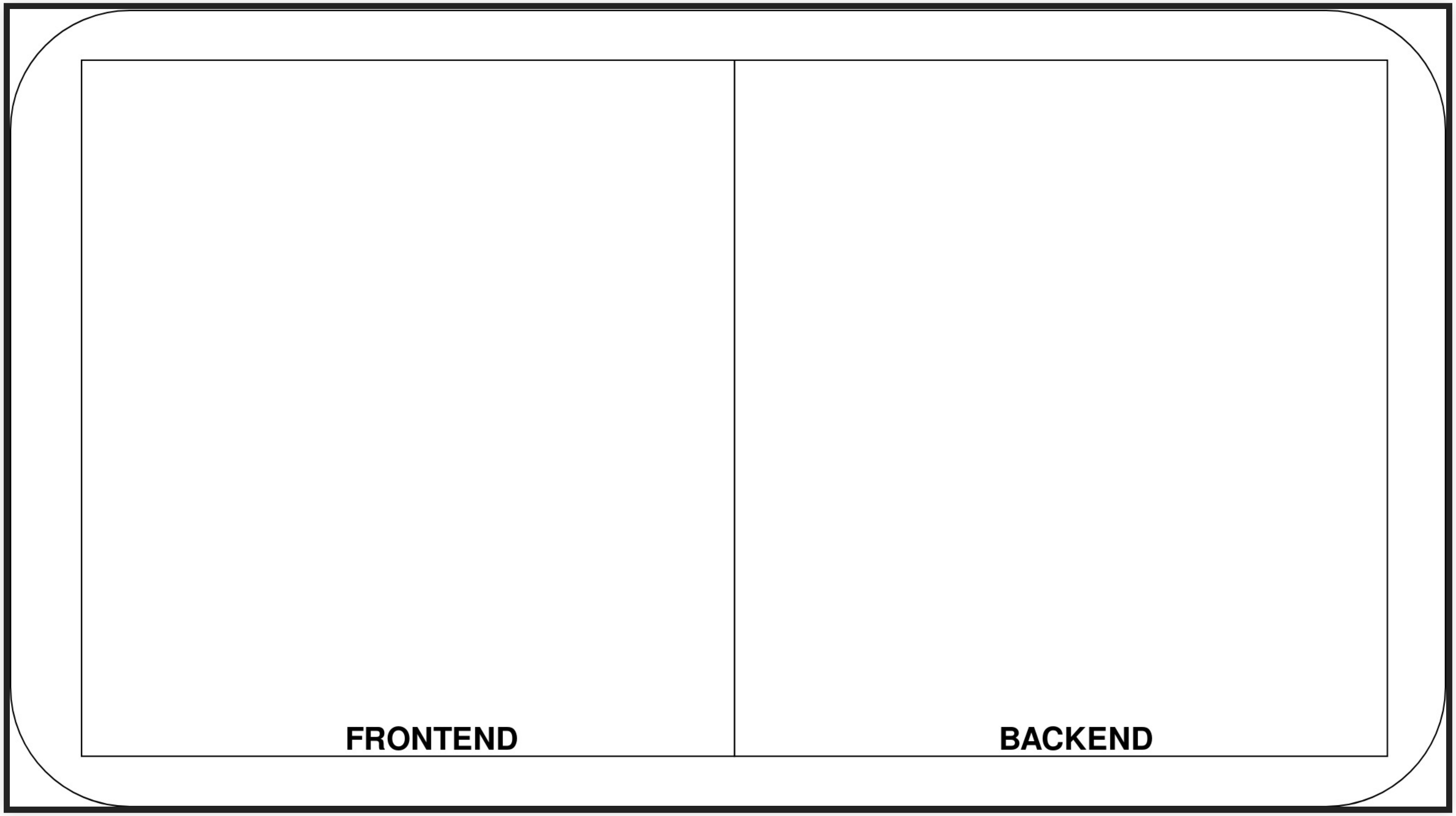
```
pip install Flask
```

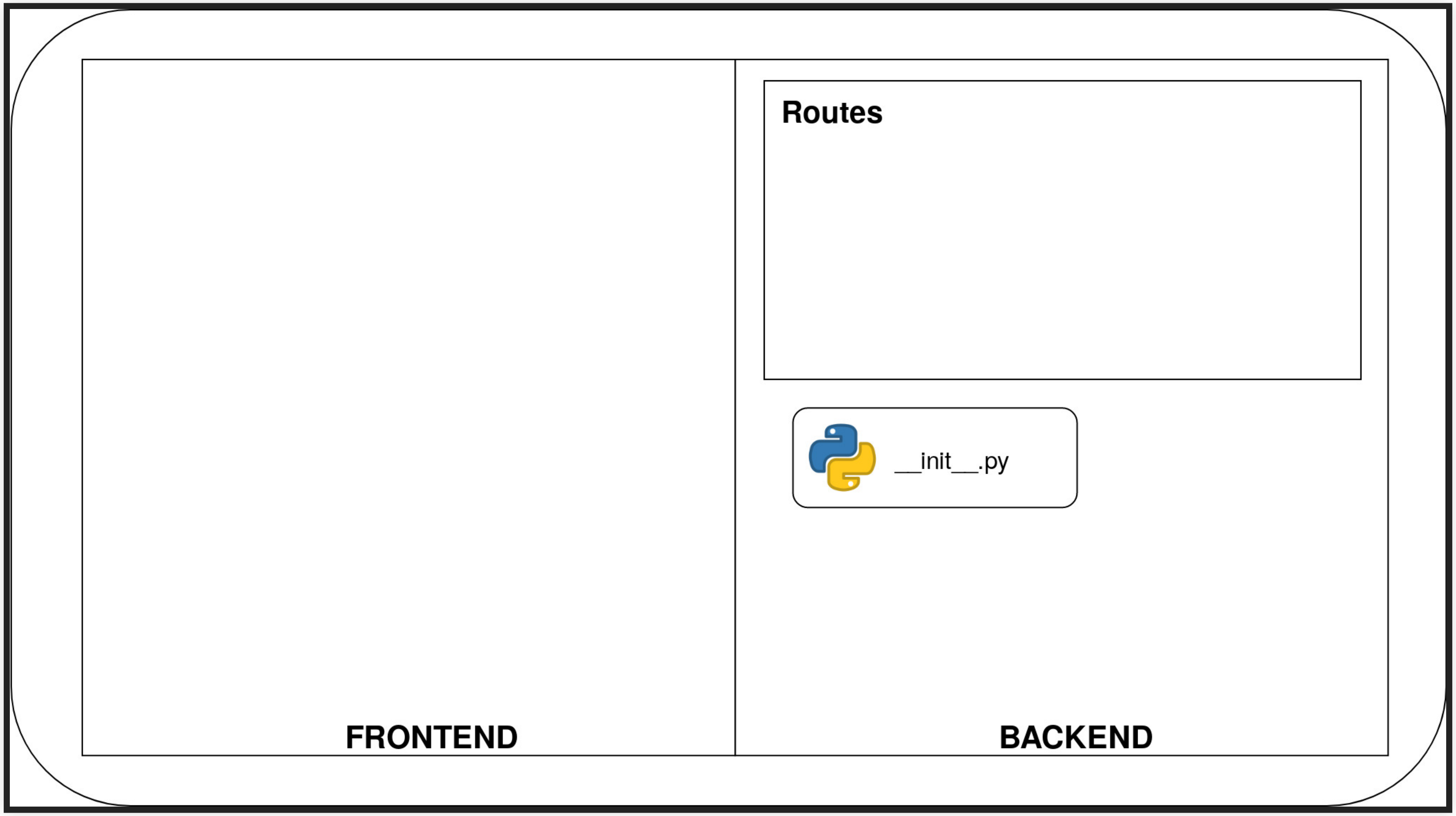
Now we are going to create the directory structure.

We need the following tree:

```
|* FlaskTutorial
|----* app
|---------* routes
```

Let's create a new file inside the `app` folder, named `__init__.py`. This is the file that is run when we import the `app` module of our application. At first we will only use it as a `python` file, but this is a common standard for deployment.


FRONTEND
BACKEND


Routes
__init__.py
FRONTEND
BACKEND

Inside the file, copy the following code:

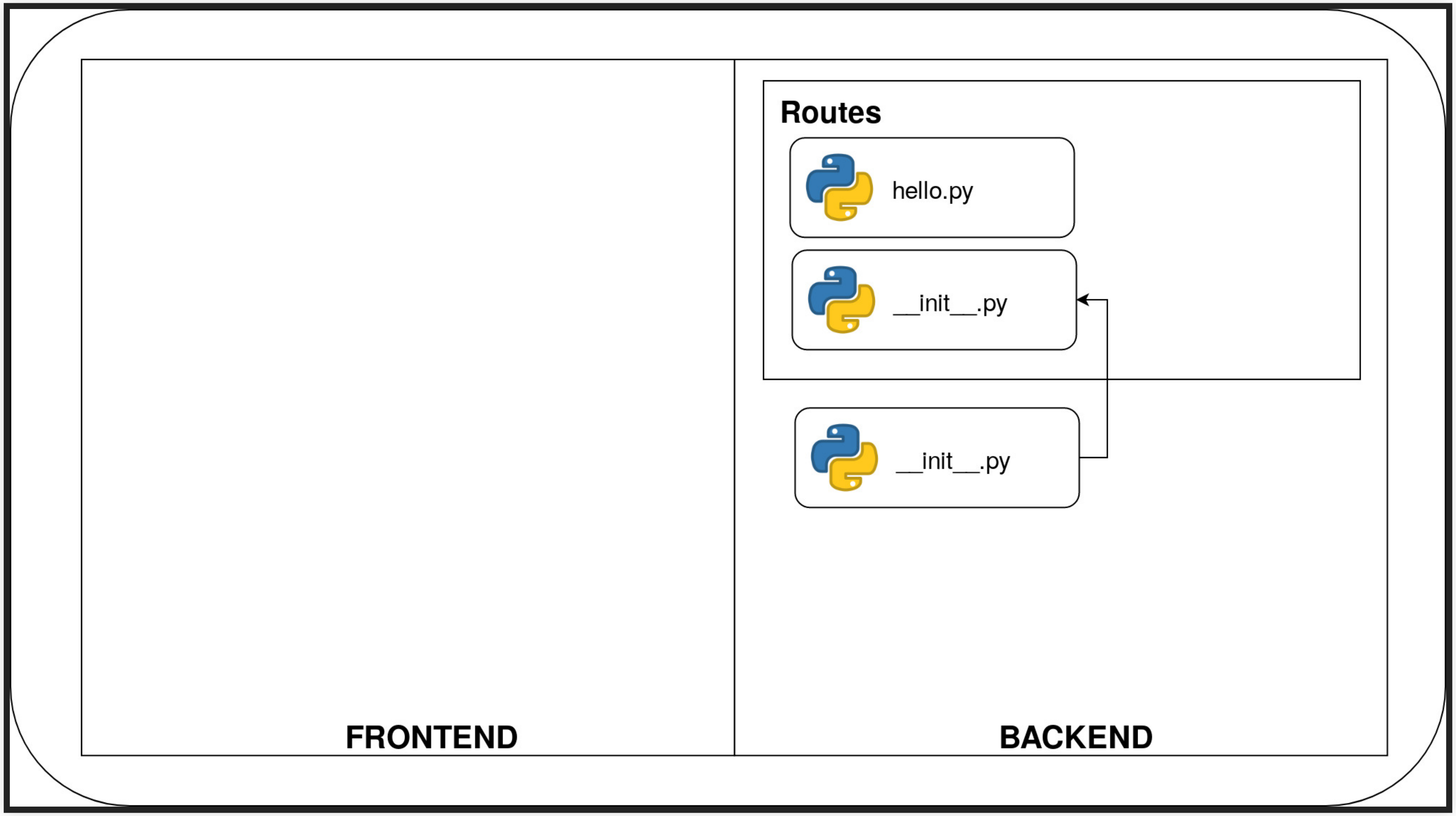
```
from flask import Flask
app = Flask(__name__)
from app import routes
```

The variable `__name__` is a Python predefined variable, which is set to the name of the module being used. Another important thing to notice is that the routes module is imported in the end of the script. We will see why in a minute.

The routes are the different **URLs** that the application provides.

```
URL      -->   mapped to -->   handler
handler -->   calls     -->   views (python functions)
```

The **route** defines the **backend** logic to be executed when a client requests a given URL through the **frontend**.


Routes
hello.py
__init__.py
__init__.py
FRONTEND
BACKEND

Now we can create a handler. Let's create `app/routes/hello.py`.

```python
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

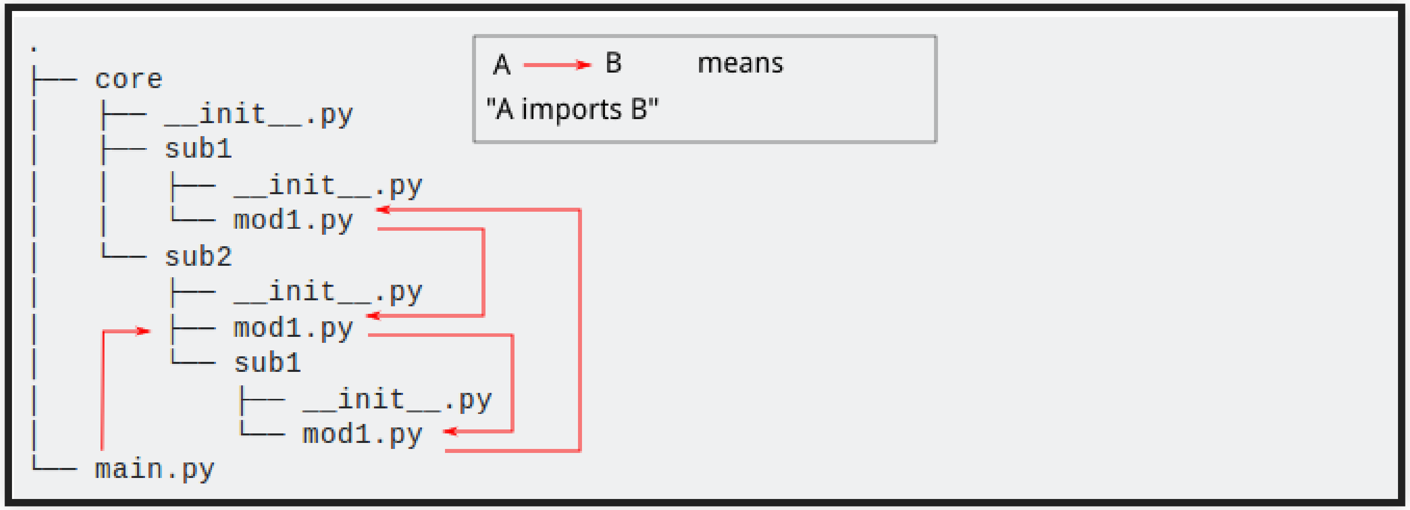
The two lines `@app.route('...')` are **decorators**, special modifiers for python functions. In this case they are used to associate the URL given as argument and the function.

Remember the line `from routes import *`? Here we need to create the `routes/__init__.py` file for allowing the import of our routes, from inside the files.

```
from .hello import *
```

The import that we issued at the end of the other file was there because we have to avoid cyclic imports.

CYCLIC IMPORTS



```
.
├── core
│   ├── __init__.py
│   ├── sub1
│   │   ├── __init__.py
│   │   └── mod1.py
│   └── sub2
│       ├── __init__.py
│       ├── mod1.py
│       └── sub1
│           ├── __init__.py
│           └── mod1.py
└── main.py
```

More info here.

We have almost completed the initial setup. We have to create the `main` module at the top level. Create and edit the file `runserver.py`, adding the following lines:

```python
from app import app
```

This will trigger the `__init__` that we defined inside the `app` module, running the server.

30 SECONDS BREAK

questions?

In the meanwhile, double check your directory structure:

```
|* FlaskTutorial
|----* app
|---------* routes
|-------------* __init__.py
|-------------* hello.py
|---------* __init__.py
|----* runserver.py
```

RUNNING THE SERVER

For now we have only one folder in the repo (`app`), which will be automatically loaded and run, but we want to make sure that Flask always loads the correct application in case we create a more complex app.

Open a terminal in the `FlaskTutorial` folder and issue the command:

```
export FLASK_APP=app
export FLASK_ENV=development
```

(see more on **environment variables** here)

From the same terminal, finally run the server:

```
flask run
```

Read the output.

```
* Serving Flask app "app" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 273-374-165
```

Let's comment this line by line.

```
 * Serving Flask app "app" (lazy loading)
```

Flask is telling us the running app. The lazy loading is due to the debug mode, it will load the resource only if they are requested.

```
 * Environment: development
 * Debug mode: on
```

Here we know the environment variable worked. The debug + lazy loading will refresh the app when we apply changes and the page is requested.

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

This is the address where we can find our running app (remember about **localhost**?).

Click on the link!

```
 * Restarting with stat
```

This line will appear every time we apply changes and save the application. It's a log for the server, which is restarting.

```
 * Debugger is active!
 * Debugger PIN: ........
```

Ignore these lines, no need to understand the meaning.

Notice that when we clicked on the link, a new line appeared:

```
127.0.0.1 - - [07/Oct/2019 12:15:41] "GET / HTTP/1.1" 200 -
```

This is a logging line, which tells that there's been a `GET` request, whith protocol `HTTP`, to the URL `/` in the localhost (`127.0.0.1`), and the response code is `200`.

Congratulations, you've issued your first request to a web server!

Actually.... Everything your browser does is HTTP requests

(try this at home)

```
firefox    -->    ctrl + shift + i    -->      Network
chrome     -->    ctrl + shift + j    -->      Network
```

And after opening the console, type www.google.it in your address bar.

PART II: TEMPLATES

Templates are used to share the aspect of the application across all pages of our application. This is important for the **user experience**.

Look at this code (DON'T COPY):

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'User'}
    return '''
<html>
    <head><title>Home Page - Microblog</title></head>
    <body>
        <h1>Hello, ''' + user['username'] + '''!</h1>
    </body>
</html>'''
```

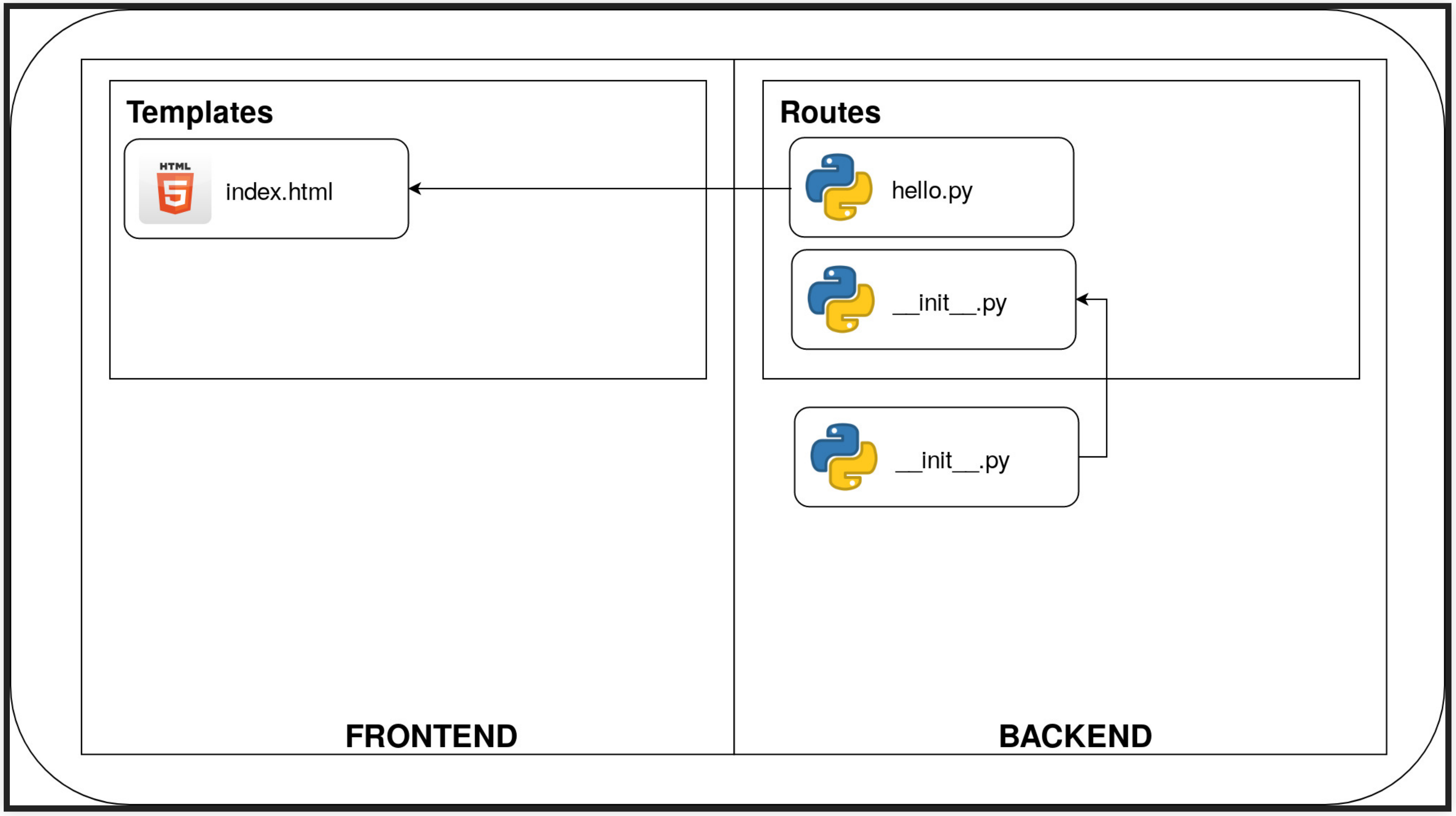
The lines:

```
<html>
    <head>
        <title>Home Page - Microblog</title>
    </head>
    <body>
        <h1>Hello, ''' + user['username'] + '''!</h1>
    </body>
</html>
```

Are written in HTML inside a string in Python. It will be rendered automatically by Flask. Of course we cannot expect to write our whole application like this.

As the application grows, we should prepare ourselves to organize our app in a smart way. We can use **templates** to separate the application **logic** from the **rendering** part.

We will write our **templates** in separate files, stored in a `templates` folder, inside the application package.

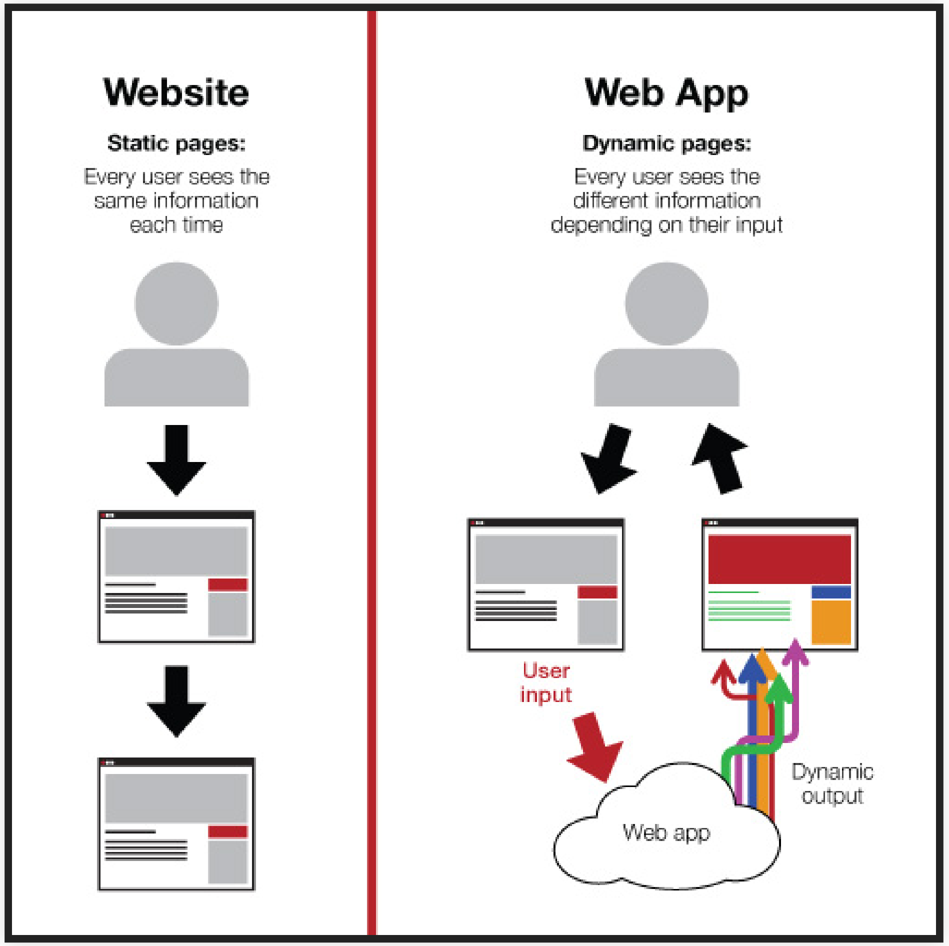

Templates
index.html
Routes
hello.py
__init__.py
__init__.py
FRONTEND
BACKEND

Create `templates/index.html`

```
<html>
    <head>
        <title>{{ title }} - Microblog</title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

You can see there is something weird with this html. Those `{{ ... }}` enclosed things are not really familiar.

Those are `variables`, that we can pass through the `flask` interface. They are now placeholders that will be filled with **dynamic** content at **runtime**.


Website
Static pages:
Every user sees the same information each time
Web App
Dynamic pages:
Every user sees the different information depending on their input
User input
Dynamic output
Web app

Get back to `routes/hello.py` and avoid that *ugly* html text.

```python
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Maura'}
    return render_template('index.html',
                           title='Home',
                           user=user)
```

The operation that converts the template in actual `html` is called **rendering**.

We use the function `render_template` for this, passing in the values for our **placeholders**. Note that we only need to provide the name of the file, since Flask automatically knows where to find the `templates` folder.

CONDITIONAL STATEMENTS AND LOOPS IN TEMPLATES

If we have to display a list of objects or display something with a condition, it would be better to avoid writing the html in the code.

We can add logic in our template with the following blocks:

```
{% if greeting %}
    <h1>{{ greeting }} {{user.username}}</h1>
{% else %}
    <h1>Hello {{user.username}}</h1>
{% endif %}
```

Remember the `endif` statement. This may not seem natural if you use Python

We can also use loops:

```
{% for student in students %}
  <div>
    <p>{{ students.name }} - ID: <b>{{ students.id }}</b></p>
  </div>
{% endfor %}
```

Let's add the lines in `templates/index.html`:

```html
<!DOCTYPE html>
<html lang="en">
<head><title>{{ title }} - Blog</title></head>
<body>
  {% if greeting %}
    <h1>{{ greeting }} {{ user.username }}</h1>
  {% else %}
    <h1>Hello {{ user.username }}</h1>
  {% endif %}
  {% for student in students %}
    <div>
      <p>{{ student.name }}: <b>{{ student.id }}</b></p>
    </div>
  {% endfor %}</body></html>
```

Of course, we should edit our `routes/hello.py` as well:

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'Maura'}
    students = [
        {'name': 'A', 'id': 1},
        {'name': 'C', 'id': 111},
    ]
    return render_template('index.html', title='Home',
                           user=user, students=students,
                           greeting="Yo")
```

IF YOU DON'T LIKE TO WASTE TIME LOOKING FOR AN INVISIBLE SYNTAX ERROR

Remember the single `{` in the logic control and the double `{{` in the variable getter.

TEMPLATE INHERITANCE

Would you imagine to create a website always rewriting your html?

What if we want all of our pages to have a bar in the top, and later we decide to change the color of the bar?

How much time would we waste replacing the code in every single `html` file?

What if we forget some?

THAT'S WHY WE LIKE TO REUSE OUR HTML.

We create a file called `templates/base.html`. This file will be structured as:

HEADER

CONTENT

FOOTER

Where the content is the only part that changes for each page. Let's first see how it works.

Here is the html `templated/base.html`.

```
<!DOCTYPE html>
<html lang="en">
    <head><title>{{ title }} - Blog</title></head>
    <body>
        {% if greeting %}
            <h1>{{ greeting }} {{ user.username }}</h1>
        {% else %}
            <h1>Hello {{ user.username }}</h1>
        {% endif %}
        {% block content %}{% endblock %}

    </body>
</html>
```

Here is `templates/index.html` updated:

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

{% block content %}
{% for student in students %}
  <div>
    <p>{{ student.name }} - ID: <b>{{ student.id }}</b></p>
    </div>
{% endfor %}
{% endblock %}
```

We have introduced another Template element, the `{% block ... %}{% endblock %}`. This control statement defines:

- in the **base**: the place where the derived template, inheriting from `base.html`, will place itself.
- in the **child**: the content to fit in the block in the base template. The `extends` statement will establish the inheritance link between the two templates.

Now it's time to add another route.

BUT YOU WILL DO IT YOURSELF

ADD ANOTHER ROUTE (1)

- add route file `routes/mypage.py`
  - give a path to your page: `/mypage`
  - remember to give a different name to the function (!=`index()`)
- add import in the route `routes/__init__.py`

ADD ANOTHER ROUTE (2)

- create `templates/mypage.html`
    - inherit from `templates/base.html`
    - fill in the block

ADD ANOTHER ROUTE (3)

- visit the page! --> localhost:5000/mypage
- bonus trick: add a link between the pages `<a href="http://localhost:5000/">Back to Homepage</a>`

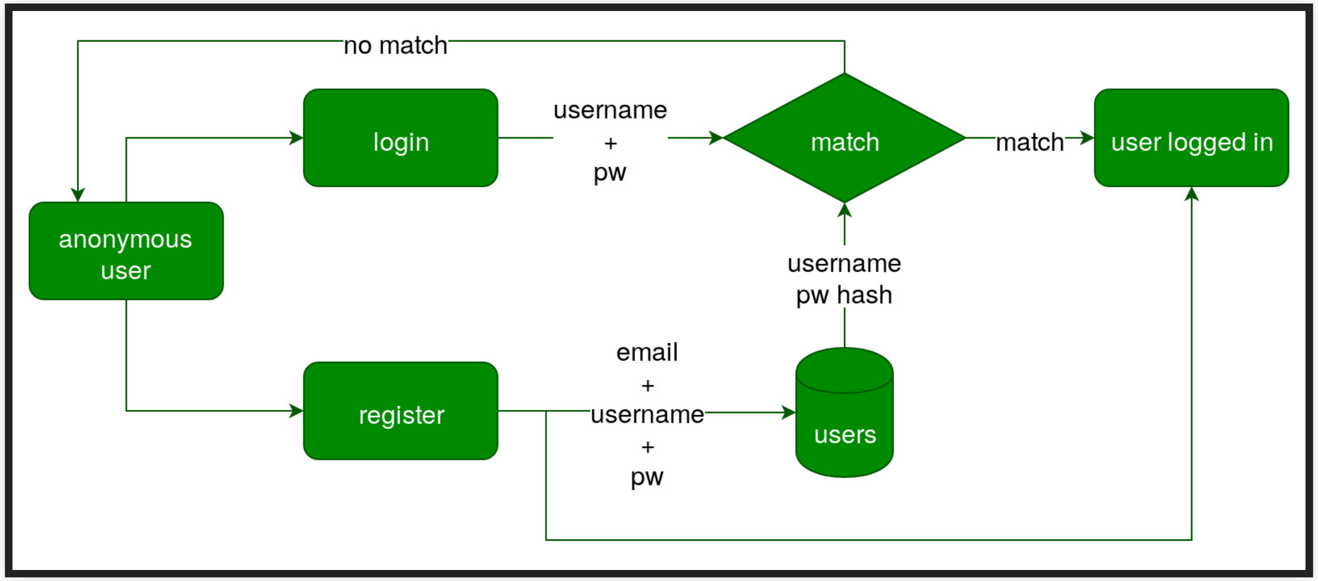
PART III: WEB FORMS

Ok, our application can now show us some content.

We want to add the possibility to accept input from the user.

For that, we will use **web forms**.

WE ARE GOING TO IMPLEMENT THE FOLLOWING LOGIC FOR LOGGING IN USERS



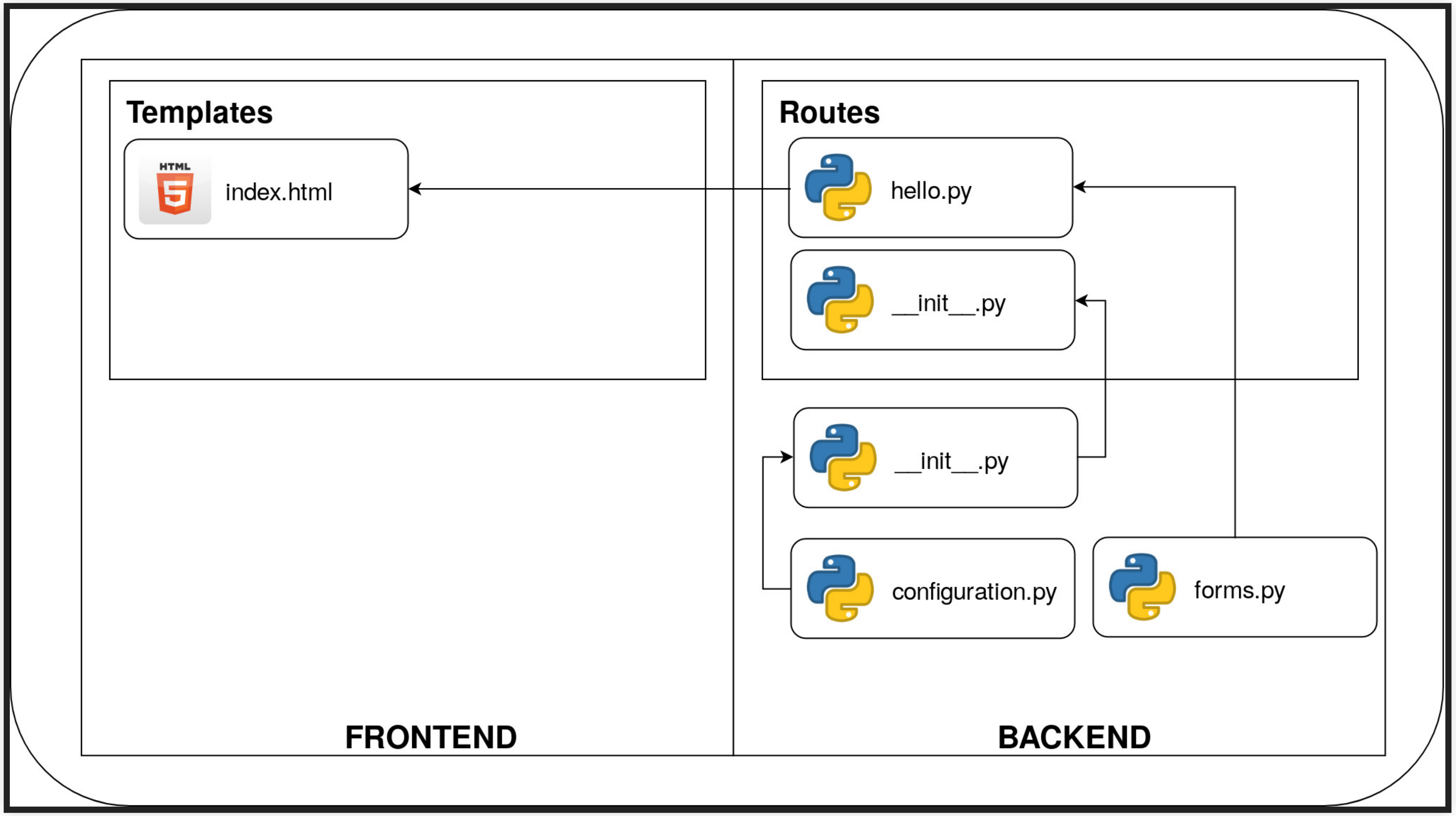
FLASK WTF

Flask-wtf is a Flask **extension**, that means that we can live without that but if we want to use this very useful functionality we will have to install it.

```
pip install flask-wtf
```

Before starting with forms, let's talk about **configuration**. We will have to set several configuration variables for our app. We could also just define them in our `runserver.py` script, but they can become hard to manage and change if our app becomes big.

That's why developers create **configuration files**, where all configuration variables can be collected and loaded by our app without need to search for them in the code.


Templates
index.html
Routes
hello.py
__init__.py
__init__.py
configuration.py
forms.py
FRONTEND
BACKEND

Create a configuration file, `config.py`, in the top-level directory of our app:

```
|* FlaskTutorial
|----* app
|---------* routes
|---------* templates
|----* runserver.py
|----* config.py                    (NEW!)
```

```
import os

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or "kkkkkkey!!"
```

As we need new configuration items, we can collect them all here so that they are easy to find and change.

Let's go back to this "secret key" thing... Flask uses this key as the **cryptographic key** for generating signatures and token. This extension uses the key to protect our app from an attack called **"Cross-Site Request Forgery"** (CSRF - pronounced sea surf).

In a CSRF attack, the attacker's goal is to cause an innocent victim to unknowingly submit a maliciously crafted web request to a website that the victim has privileged access to.

There pattern `os.environ.get('SECRET_KEY') or 'my very secret key'` can allow us to use as first choice an **environment variable** (remember them?), and if this is not defined we have a fallback option as an hard-coded string.

We can add the configuration class to our flask application just adding one line in our `__init__.py` script:

```python
from flask import Flask

from app.config import Config

app = Flask(__name__)
app.config.from_object(Config)

from app.routes import *
```

Now that we have our configuration done, we can finally head to the creation of a simple web form.

We can create a template form as a python class, just for keeping things structured. Create a file `forms.py` in the `app` directory.

```python
from flask_wtf import FlaskForm
from wtforms import (StringField, PasswordField,
                     BooleanField, SubmitField)
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Username',
                           validators=[DataRequired()])
    password = PasswordField('Password',
                           validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Login')
```

Note the particular classes used for each input space.

- `StringField`
- `PasswordField`
- `BooleanField`
- `SubmitField`

Another important thing to notice is the `validators` field, that optionally lets us define some validation method. The `DataRequired` validator only checks that some data is present at the moment of the submission.

Now we have to **render** this form as a webpage. The fields defined in our class already know how to render themselves, so we just have to include them in our html.

Go ahead and create a `templates/login.html`

```
{% extends "base.html" %}
{% block content %}
  <h1>Login</h1>
  <form action="" method="post" novalidate>
    {{ form.hidden_tag() }}
    <p>
      {{ form.username.label }}<br>
      {{ form.username(size=32) }}
    </p>
```

```
    <p>
      {{ form.password.label }}<br>
      {{ form.password(size=32) }}
    </p>
    <p>
      {{ form.remember_me() }} {{ form.remember_me.label }}
    </p>
    <p>{{ form.submit() }}</p>
  </form>
{% endblock %}
```

Remember we are extending the template `templates/base.html`. This template expects to receive a LoginForm object as argument, which we reference inside the html as `form`.

Pay attention to this line:

```
<form action="" method="post" novalidate>
```

- The `action` tells the browser to **which url** submit the data (we use this same url so the field is empty).
- The `method` field is to specify the **method** of the `HTTP request`
- The `novalidate` attribute is specified here because the validation will be performed by the web app, not by the browser.

Now, there is another weird line...

```
{{ form.hidden_tag() }}
```

This line is used for generating a hidden token that is used to protect the form against CRSF attacks. Flask will handle everything nicely as long as:

- there is a hidden tag in the form
- there is a specified secret key in the config

Now we have to link the form in our application. Create `routes/login.py`.

```python
from flask import render_template
from app import app
from app.forms import LoginForm

@app.route('/login')
def login():
    # instantiate a login form
    form = LoginForm()
    return render_template('login.html',
                           title='Login',
                           form=form)
```

We can include the login button in our navigation bar.

Edit `templates/base.html`:

```
<div>
    My Page:
    <a href="/index">Home</a>
    <a href="/login">Login</a>
</div>
```

Comment the user part for now ...

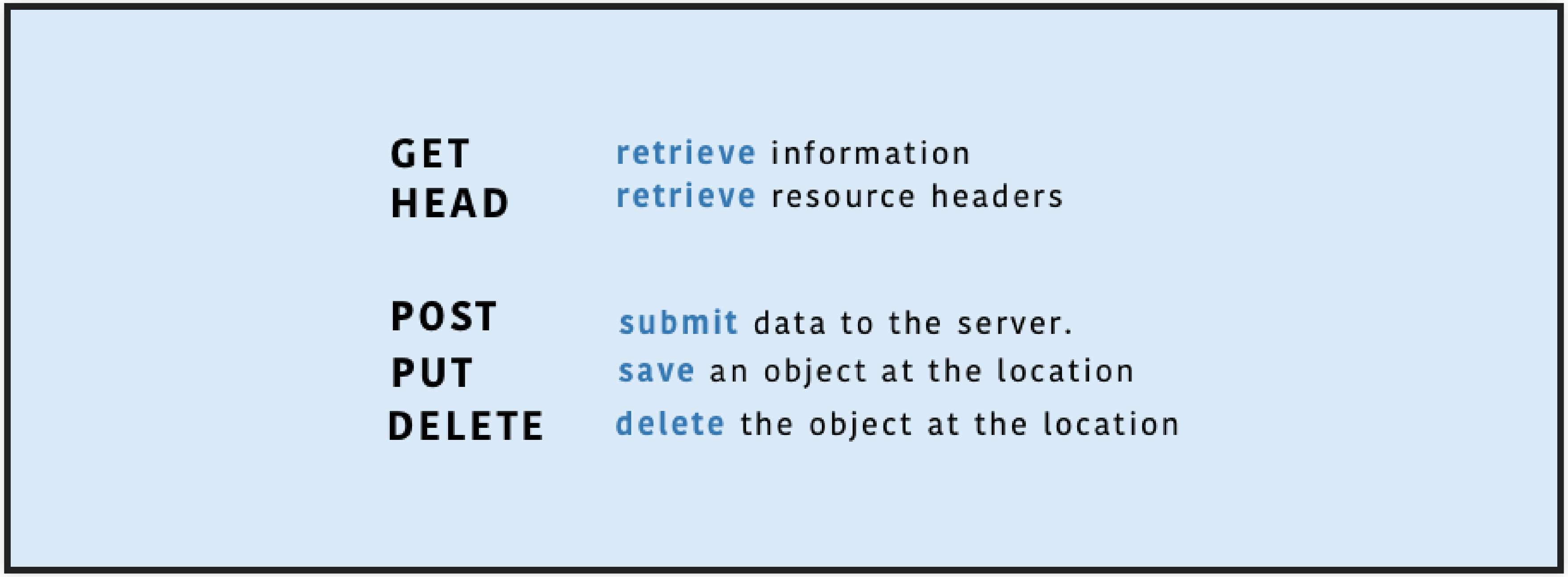
Go and try your login form. Of course you can't expect that everything runs smoothly :D

ACCEPTING DATA FROM FORMS

METHOD NOT ALLOWED ...

This is Flask telling us that we are trying to send information through a `POST` method, but we forgot to define it in the code!

HTTP METHODS



GET	retrieve information
HEAD	retrieve resource headers
POST	submit data to the server.
PUT	save an object at the location
DELETE	delete the object at the location

More info here

```python
from flask import render_template, flash, redirect

@app.route('/login', methods=['GET', 'POST']) # <<<<<<<<<<<<<
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # todo: we will handle this later
        return redirect('/index')
    return render_template('login.html',
                           title='Login',
                           form=form)
```

Now we can accept user data and **validate** them. First, let's have a look at the line:

```
@app.route('/login', methods=['GET', 'POST'])
...
```

The methods argument tells Flask that we want to use both `POST` and `GET` methods in this same url.

QUICK RECAP

GET: return information to the client (browser)

POST: client (browser) sends information to the server

We can always send information with GET, but that is probably a bad idea...

Do you really want to store in the server this line?

```
"GET /login HTTP/1.1" 200 - login?user=myUser&pass=MyPassword
```

This line **accepts** the input and **validates** it. This method returns `True` only if the browser sends a `POST` request **AND** if the validation methods in all the fields run smoothly.

```
if form.validate_on_submit():
    ...
```

If the validation fails, it will return `False`, so we will have to handle that later.

You see now that if we try to validate the form without filling the required fields, the actual behavior will be that the server re-display the form.

Of course we want to show some information to the user, so that she/he can understand what happened with the login.

The form validators already have some pre-defined error message, but the are not actually rendered in our form.

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

{% block content %}
    <p>
        {{ form.username.label }}<br>
        {{ form.username(size=32) }}<br>
        {% for error in form.username.errors %}
        <span style="color: red;">[{{ error }}]</span>
        {% endfor %}
    </p>
```

```
    <p>
        {{ form.password.label }}<br>
        {{ form.password(size=32) }}<br>
        {% for error in form.password.errors %}
        <span style="color: red;">[{{ error }}]</span>
        {% endfor %}
    </p>
{% endblock %}
```

The only change here is in the for loops that display all error messages added by the validators. The error messages in the form can be get through `form.<field_name>.errors`.

We are using a form because the errors are a list. This is because fields can have more than one validator.

Let's try to generate the errors!

http://127.0.0.1:5000/login

JUST SOME SMALL IMPROVEMENT ...

GENERATING LINKS INSIDE THE APPLICATION

Now we have used the line:

```
return redirect('/index')
```

Which is for redirecting the browser to the resource `/index`.

Many times you may want to change your links.

If you decide to do some refactoring, you will have to replace all links in your application.

One solution is to use a function that creates URLs inside Flask, with its **internal mapping** to view functions.

```
url_for('index')
```

will generate an URL for the view function `index`. The argument is the endpoint name, which is the name of the view function.

Another important aspect of this **separation** of URL and view function is the generation of dynamic urls.

Let's go and fix all the urls that we placed in our app:

- app/templates/base.html
- app/routes.py

Notice that in the html file we will have to use the double "{"

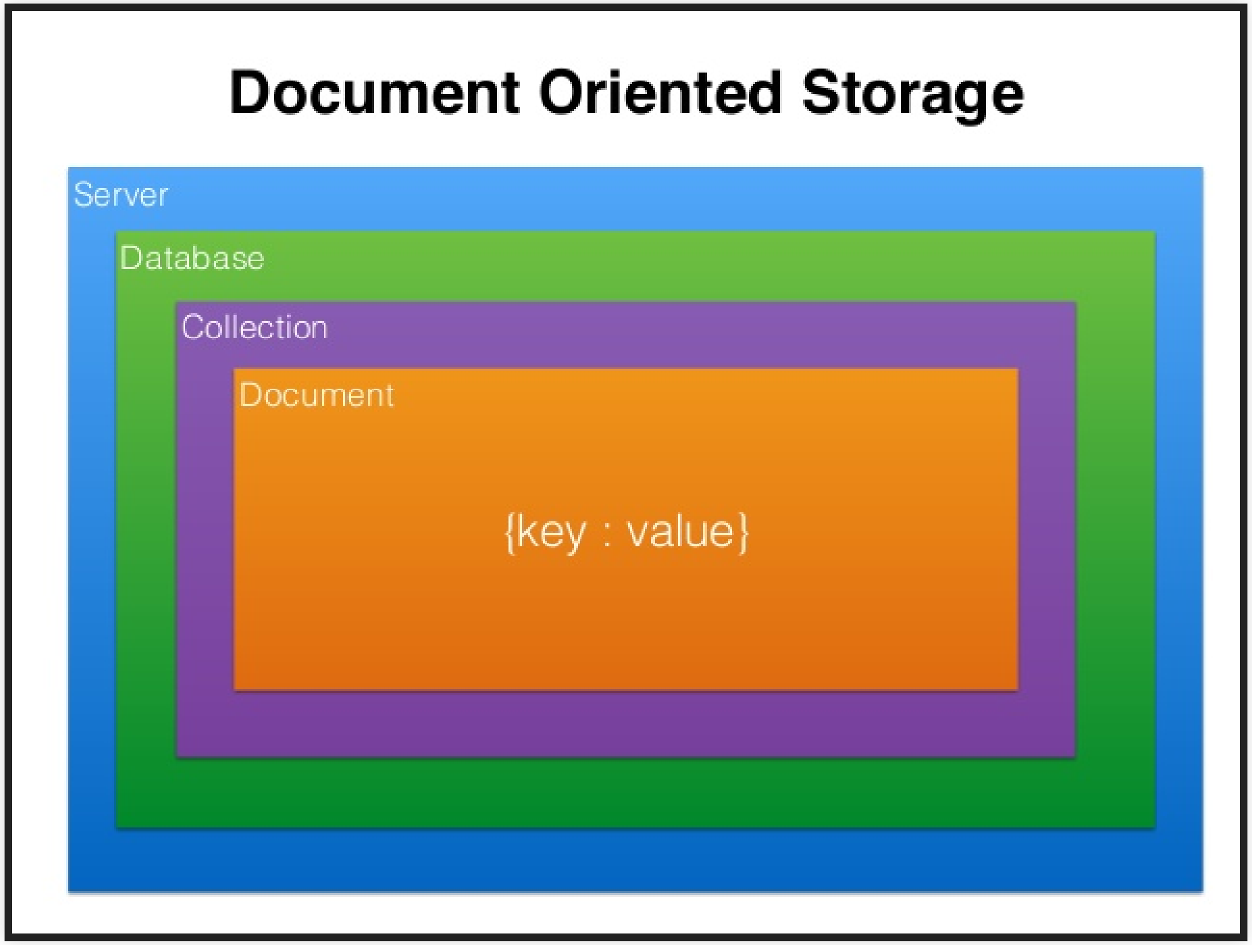
```
<div>
    My Page:
    <a href="{{ url_for('index') }}">Home</a>
    <a href="{{ url_for('login') }}">Login</a>
</div>
```

PART IV: DATABASES

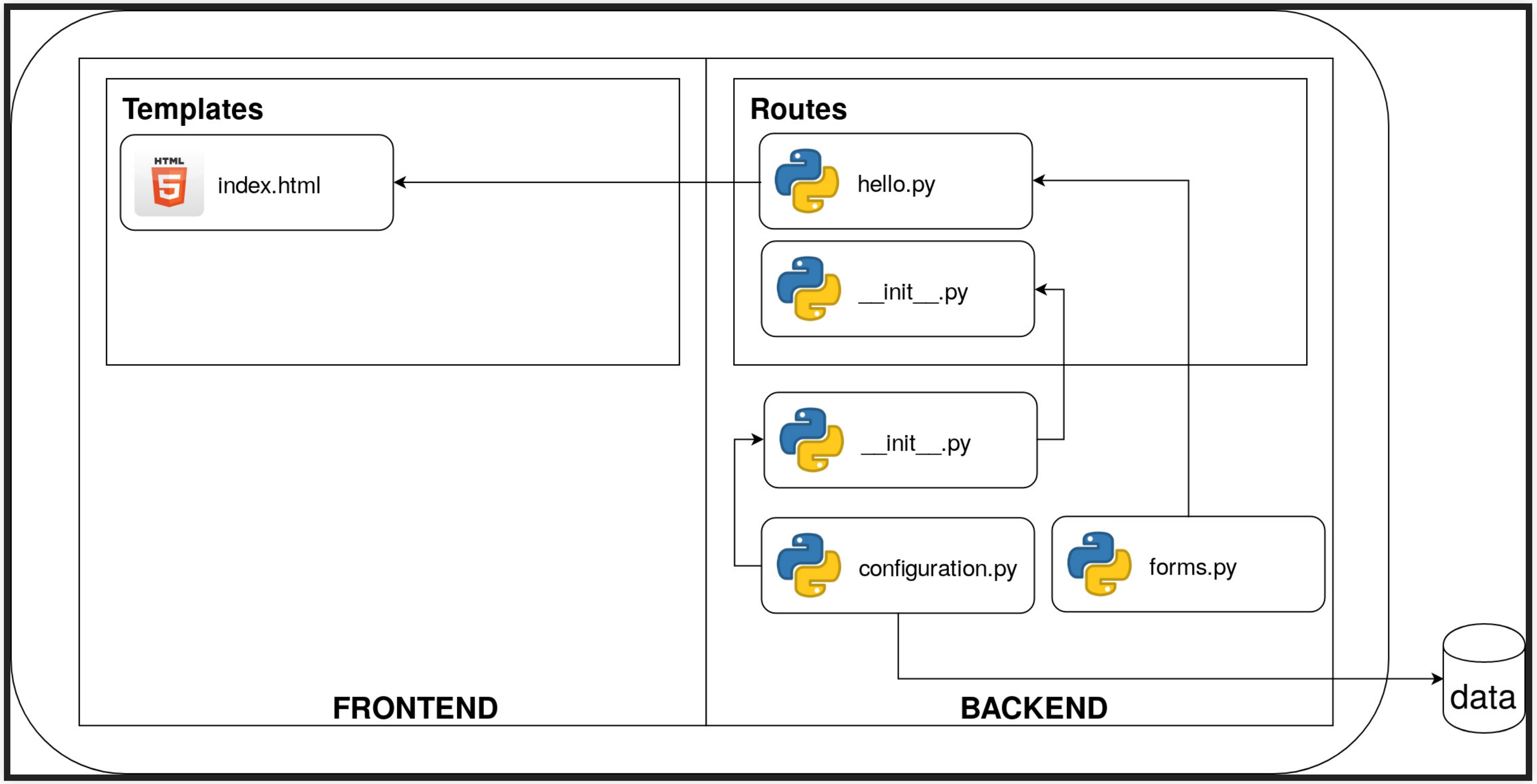
We want our server to remember the registered users. In order to do so we have to connect to a **database**.

For this application we will use a simple **non relational** database, called `Mongo`.

Mongo elements:



```
pip install flask_mongoengine flask_login
```


Templates
index.html
Routes
hello.py
__init__.py
__init__.py
configuration.py
forms.py
FRONTEND
BACKEND
data

Now we need to add our database settings to our config file:

```
# database configuration
MONGODB_HOST = "localhost"
MONGODB_PORT = 27017
MONGODB_USERNAME = "username"
MONGODB_CONNECT = True
MONGODB_DB = "my_app_db"
```

The complete db connection string is called db URI, and it is needed by the library for connecting to the right source.

It contains (at least):

- the driver name `mongodb`
- the host (server on which the db is hosted)
    - in this case our localhost
- the port where the db is accessible

```python
from flask import Flask
from flask_mongoengine import MongoEngine
from app.config import Config

app = Flask(__name__)
app.config.from_object(Config)
db = MongoEngine(app)

from app.routes import *
```

Mongoengine will connect our application to the specified database and expose the `db` attribute in the `app`. We can use it directly in views.

ORM = OBJECT-RELATIONAL MAPPERS

We have to create a model for storing our data in a structured way.

For example, all users will have a username, password and additional information such as the email for password recovery, phone number for two factor auth etc.

Let's create a file `models.py` in our source root, where `runserver.py` is located.

```python
from app import db


class User(db.Document):
    meta = {'collection': 'users'}
    email = db.StringField(max_length=30)
    password = db.StringField()
```

You probably noticed that the fields are the same that we have in our form. This will of course come in handy when we have to use the information in the form for logging in the user.

Ok, but we don't have users yet. We need to create a **Sign In** form.

EXERCISE: CREATE THE SIGN-IN FORM.

(HINT: START FROM THE LOGIN FORM)

- `forms.py`
- `routes/signin.py`
- `templates/signin.html`
- button in the navbar (`base.html`)
- `routes/__init__.py`

Other improvements:

Let's add the email field to the registration form. The email may be used for password recovery or to send nice emails with updates (we won't cover them in this course).

IMPORTANT: ALWAYS ADD THE (ANNOYING) EMAIL VALIDATION PROCESS IN THE WEBSITES.

It will prevent users to insert other people's mails in the field (and fill their inbox with unwanted spam).

```python
from wtforms.fields.html5 import EmailField
from wtforms.validators import Email

# wtforms email field and validator
email = EmailField('Email address',
                   validators=[DataRequired(), Email()])
```

It's time to try out the new path we added. (remember to try out also the validation of the email field!)

STORING THE USER DATA IN MONGO DB

Now we have to implement the following logic for signing-in:

- valid email + valid username + not in database: user info should be stored.
- valid email already registered: error (email already in use).
- valid username already registered: error (username already in use).

Inside `routes/signin.py`:

```
if form.validate_on_submit():
    user = User(email=form.email.data,
                username=form.username.data,
                password=form.password.data)
    user.save()
```

For now we are just adding a user, of course we cannot store the password as it is.

Let's verify that the user is correctly added. Connect to a mongodb shell using a terminal:

```
mongo
use my_app_db
coll = db.users
coll.find()
```

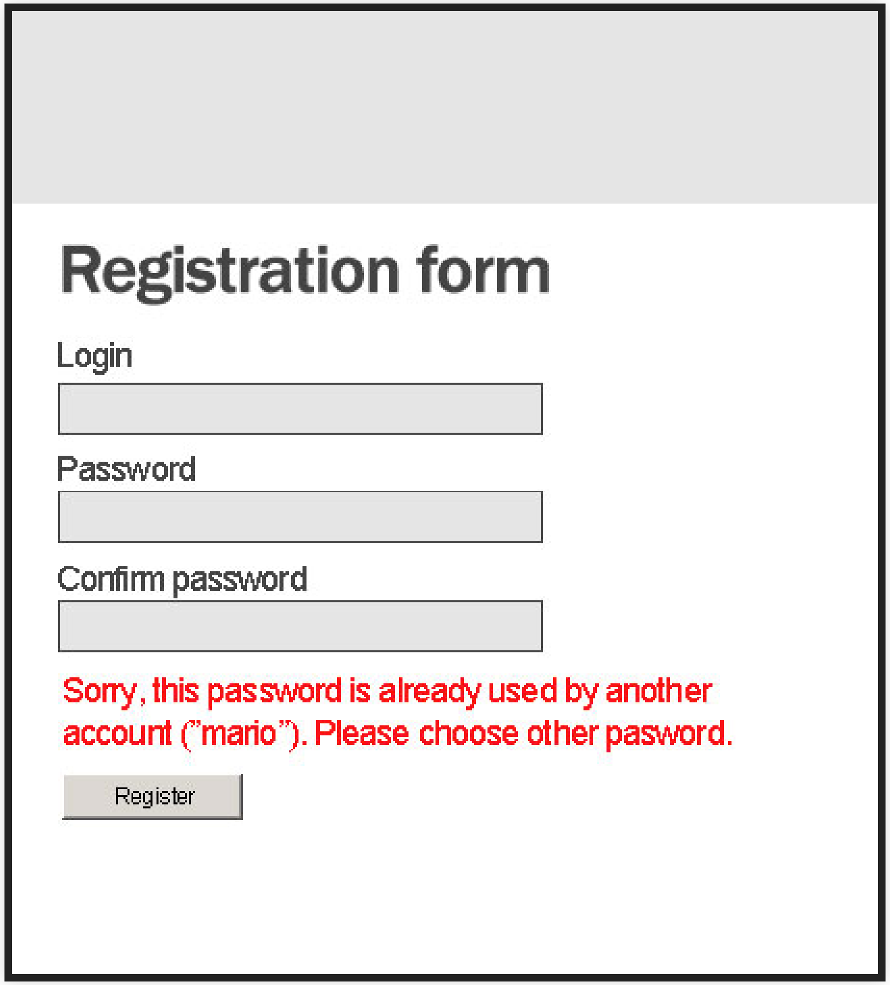
Now let's add the check for username and email. In `routes/signin.py`:

```python
from flask import abort

if form.validate_on_submit():
    same_email = User.objects(email=form.email.data)
    if len(same_email) > 0:
        abort(403, "Forbidden. Email already in use.")
    else:
        user = User(email=form.email.data,
                    username=form.username.data,
                    password=form.password.data)
        user.save()
```

EXERCISE: ADD THE SAME CHECK FOR "USERNAME ALREADY IN USE".

WHAT COULD GO WRONG?


Registration form

Login

Password

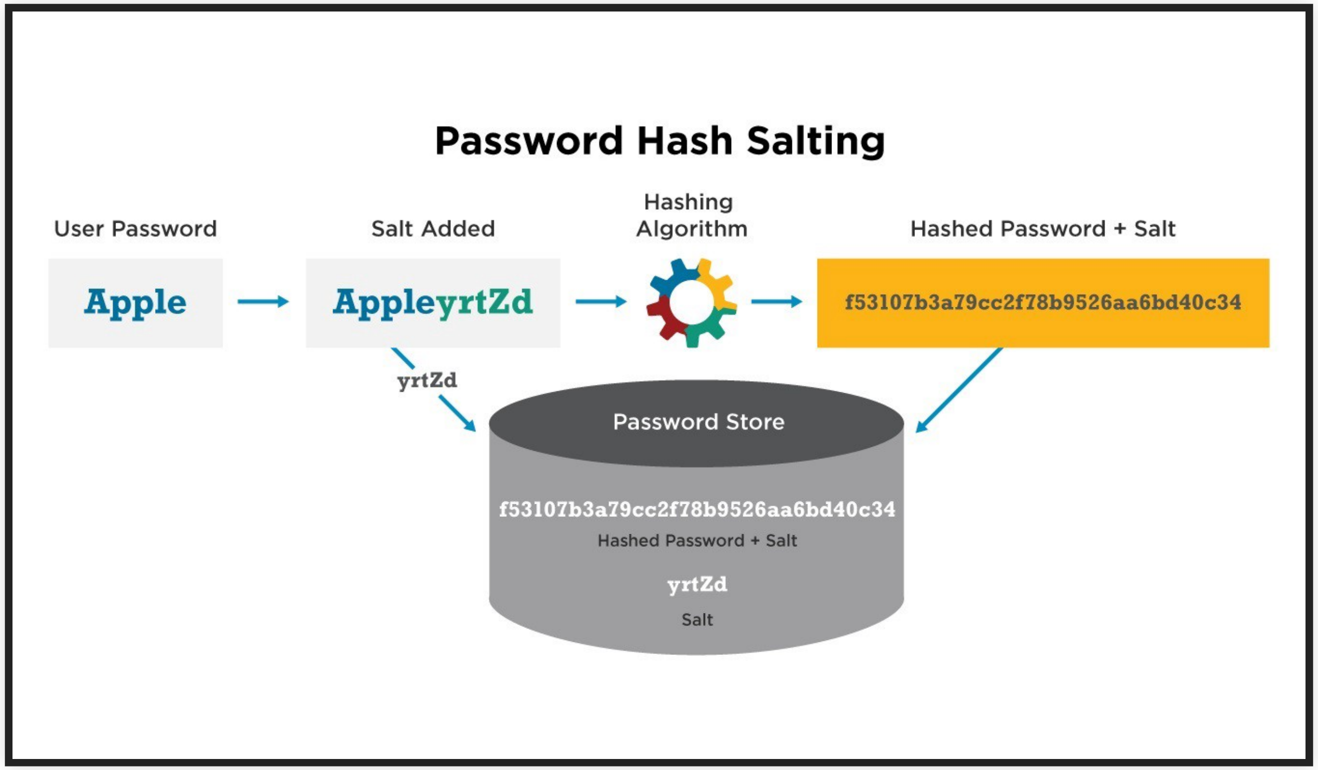
Confirm password

Sorry, this password is already used by another account ("mario"). Please choose other pasword.

Credits: www.codewell.com

We must also hide the password when we store it. It is not a good strategy to store it directly as it is inserted.

HASHING PASSWORDS



Credits: medium article

HASHING PASSWORDS

```python
from werkzeug.security import generate_password_hash
hashpass = generate_password_hash(form.password.data,
                                  method='sha256')
user = User(email=form.email.data,
            username=form.username.data,
            password=hashpass)
```

LET'S TEST THE PASSWORD HASHING

1) sign in a new account.

2) enter `mongo` shell and see the registered users.

3) you should see a user with the hashed password.

Now we can finally add the logic for logging in a registered user. We will edit `routes/login.py`

```
from werkzeug.security import check_password_hash
from flask_login import login_user

...
if form.validate_on_submit():
    user = User.objects(username=form.username.data)
    if len(user) > 1:
        if check_password_hash(user.password,
                               form.password.data):
            login_user(user)
            return redirect(url_for('index'))
    else:
        abort(404, "User not found. Please register.")
...
```

The method `check_password_hash` will match the hash of the password submitted by the user with the password stored at registration.

The method `login_user` will set the user logged in. Let's add the `login_user` method also after a user registers.

In order to use `login_user` we have to provide a method for flask to handle the user logins.

More information here.

We add the following to `__init__.py`:

```python
from flask_login import LoginManager
login_manager = LoginManager(app)
```

Does not work yet. This is because the `User` model that we defined does not have the attributes required for checking the login.

We have to add to the inheritances of `User` the mixin class `UserMixin`.

```
from flask_login import UserMixin
class User(db.Document, UserMixin):
    ...
```

Another small change to `models.py` and we are good to go. Here we provide a method for the Login Manager to load the object that contains the user.

```
from app import login_manager
@login_manager.user_loader
def load_user(user_id):
    return User.objects(pk=user_id).first()
```

Now let's remove all trash we have in the database ...

```
mongo
use my_app_db
db.dropDatabase()
```

Ok now we can log in. In the next sections we will create a view that is only accessible to logged in users and a logout button.

LOGIN REQUIRED

Edit `routes/mypage.py` and add the following decorator:

```
from flask_login import login_required
@login_required
@app.route('/mypage')
def mypage():
    ...
```

Now let's also use the user information.

```python
from flask_login import login_required, current_user
def mypage():
    user = current_user
    return render_template('mypage.html',
                           title='Home',
                           user=user,
                           greeting="Yo")
```

USER LOGOUT

Let's create a view `routes/logout.py` for logging out the user:

```python
from flask import url_for
from flask_login import login_required, logout_user
from app import app, redirect
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(url_for('index'))
```

We don't need to create an html file related to the logout, since we can directly render the index page again.

But we need to change `base.html` and add the button in our nav bar.

```
<div>
    <a href="{{ url_for('index') }}">Home</a>
    {% if not current_user.is_authenticated %}
        <a href="{{ url_for('login') }}">Login</a>
        <a href="{{ url_for('register') }}">Sign In</a>
    {% endif %}
    {% if current_user.is_authenticated %}
        <a href="{{ url_for('logout') }}">Logout</a>
    {% endif %}
</div>
```

ERROR HANDLING

Try to log out and visit

http://localhost:5000/does-not-exist

We want to add a custom template to our error pages. For doing so, create `routes/errors.py`

```python
from flask import render_template
from app import app
@app.errorhandler(404)
def page_not_found(e):
    # note that we set the 404 status explicitly
    return render_template('404.html'), 404
```

Now create `templates/404.html`

```
{% extends "base.html" %}
{% block content %}
  <h1>Page Not Found</h1>
  <p>What you were looking for is just not there.
  <p><a href="{{ url_for('index') }}">Go somewhere else.</a>
{% endblock %}
```

Finally, add the import in `routes/__init__.py`.

Navigate again to

http://localhost:5000/does-not-exist

EXERCISE: CREATE A BLOG

- create a page `/blog`
- redirect to that page after login and sign in
- show comments stored in mongo
- create `/comment` path (login required)
- create a button for sending new comments
- logged in users can post comment and they will be displayed in the blog

TO-REMEMBER (DO NOT ERASE AFTER EXAM) CHECKLIST:

- Web server
- Frontend vs. Backend
- API
- Deployment and Localhost
- URI and URL
- Requests
- Static vs. Dynamic website
- Rendering and Templates
- Connection to databases

THE END