18) Database

Agenda

- "Guess the secret number" homework - solution
- Problem with cookies
- SmartNinja NoSQL pip package
- Bonus: How to use pymongo

Here is the **solution** to the homework from the previous lesson: **Guess the secret number** game.

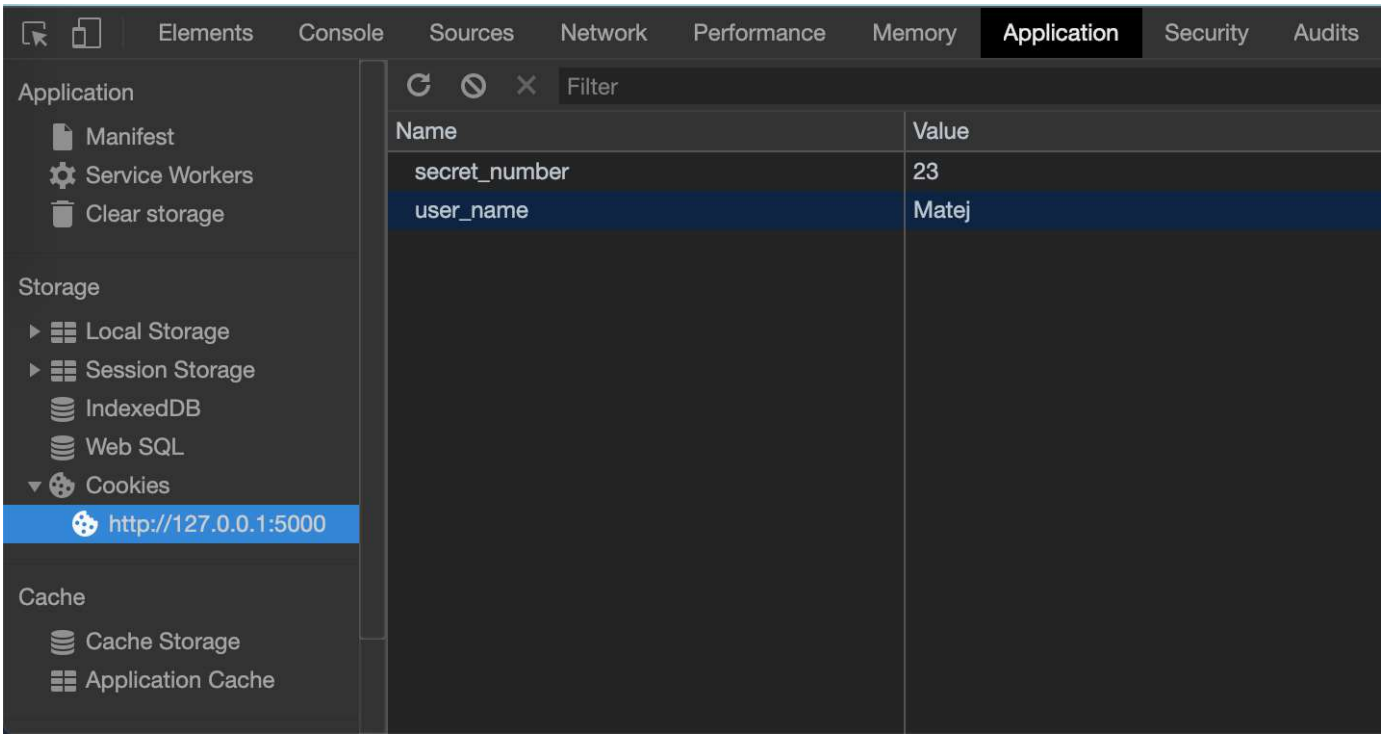
Let's go through the code:

```
@app.route("/", methods=["GET"])
def index():
    secret_number = request.cookies.get("secret_number")  # check if there is already a cookie

    response = make_response(render_template("index.html"))
    if not secret_number:  # if not, create a new cookie
        new_secret = random.randint(1, 30)
        response.set_cookie("secret_number", str(new_secret))

    return response


@app.route("/result", methods=["POST"])
def result():
    guess = int(request.form.get("guess"))
    secret_number = int(request.cookies.get("secret_number"))

    if guess == secret_number:
        message = "Correct! The secret number is {0}".format(str(secret_number))
        response = make_response(render_template("result.html", message=message))
        response.set_cookie("secret_number", str(random.randint(1, 30)))  # set the new secret
        return response
    elif guess > secret_number:
        message = "Your guess is not correct... try something smaller."
        return render_template("result.html", message=message)
    elif guess < secret_number:
        message = "Your guess is not correct... try something bigger."
        return render_template("result.html", message=message)
```

Problem with cookies

As you can see, we **stored** the secret number in a **cookie**. But a user can **see** what is stored in the cookie and then win the game easily.

Run the "Guess the secret number" web app, **open** the index page in your **browser** and then open Chrome **developer tools** (right click --> Inspect element). Then go to the **Application** tab and find **Cookies** in the left sidebar:



You can see the **secret number** there which can help users **cheat** and win the game in the **first** attempt.

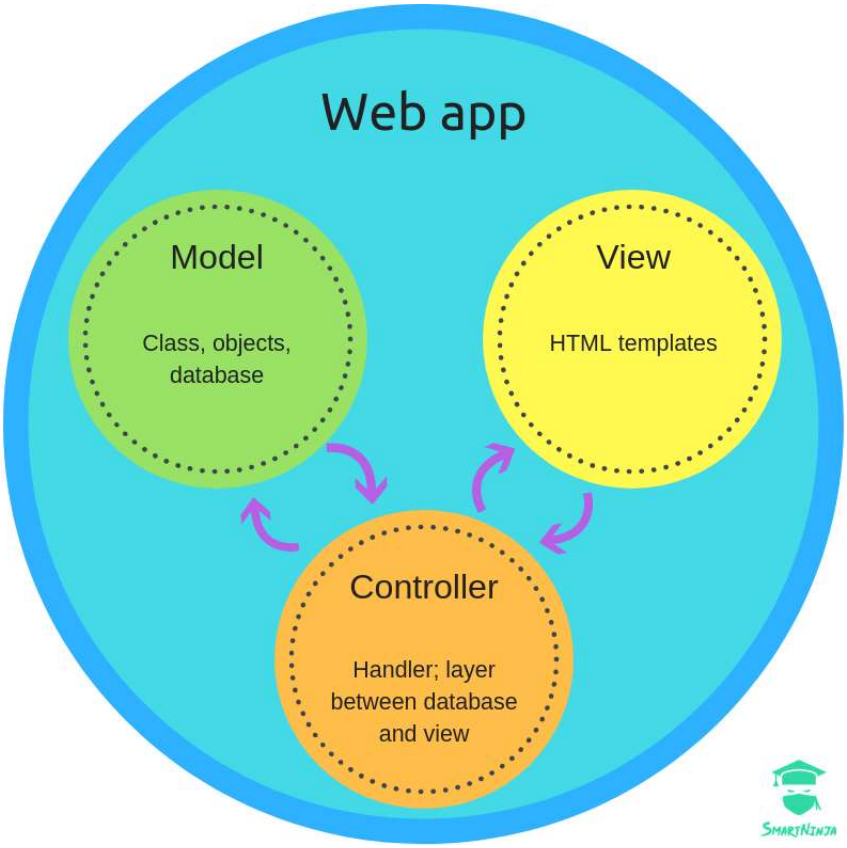
This is not good.

How can we solve this problem?

Database

Let's use a database instead!

Remember this image of the **Model-View-Controller** (MVC) relationship?



The **Model** part applies to a **database**. And as you can see, the **View** part (HTML, browser) **cannot access** the database **directly**. Everything goes through the **Controller** (handler) that controls what data goes into the View.

Long story short: user will **not** be able to see the contents of the database via Developer Tools. User will only see what we'll specifically **allow** her/him to see.

In this lesson we'll learn how to install and use a database. Then you will be able to use this knowledge to update your "Guess the secret number" web app.

Create a new Flask project

Create a fresh new Flask web app.

main.py:

```
from flask import Flask, render_template

app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")


if __name__ == '__main__':
    app.run()
```

templates/index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Simple database example</title>
</head>
<body>

    <h1>Simple database example</h1>

</body>
</html>
```

requirements.txt:

```
Flask==1.0.2
```

Install the requirements and run your app to see if everything works correctly.

Installing a database

Let's add a database into our Flask app. We will use a special `smartninja-nosql` pip package, that will allow us to use four different databases:

- TinyDB (used on localhost)
- MongoDB (used on Heroku)
- Firebase (used on Google App Engine)
- CosmosDB (used on Azure)

The `smartninja-nosql` package will automatically figure out where our web app is currently served (localhost, GAE, Heroku or Azure) and use the appropriate database for the environment.

Let's add this package into our `requirements.txt` file:

```
smartninja-nosql
```

Now run this command in the terminal (or install the package via PyCharm settings):

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

Next we'll need to create models for our database.

User model

Remember how we created models in the OOP lesson? We used a `class`.

Let's first create a `User` model. We will only store user's name and email in our database.

Create a **new file** called `models.py` :

```
from smartninja_nosql.odm import Model


class User(Model):
    def __init__(self, name, email, **kwargs):
        self.name = name
        self.email = email

        super().__init__(**kwargs)
```

Our `User` class will inherit some methods from the `Model` class (methods for working with a database).

> *The `**kwargs` part must be added, but don't worry about its meaning for now.*
>
> *Also, by `email` we mean an **email address**.*

View & Controller

Let's also adapt our View (HTML template) and Controller (`index()` handler) to accept user's name and email address.

index.html:

```
<form method="post" action="/login">
    <input type="text" name="user-name" placeholder="Enter your name"><br>
    <input type="email" name="user-email" placeholder="Enter your email address"><br>

    <button>Submit</button>
</form>
```

main.py:

```
from flask import Flask, render_template, request, redirect, url_for
from models import User

# ... (other code)

@app.route("/login", methods=["POST"])
def login():
    name = request.form.get("user-name")
    email = request.form.get("user-email")

    # create a User object
    user = User(name=name, email=email)
    user.create()  # save the object into a database

    return redirect(url_for('index'))

# ... (other code)
```

Let's try out our web app. Enter your name and email and press Submit.

Then take a look in your project folder (within PyCharm). A new file was created: `db.json`. This is your **localhost database**.

Open the `db.json` file and take a look at its contents. You will see that your name and email are stored there.

Saving data into a cookie

We will still need a cookie in order to be able to figure out which user is using our web app. Let's update the `login()` handler like this:

```
@app.route("/login", methods=["POST"])
def login():
    name = request.form.get("user-name")
    email = request.form.get("user-email")

    # create a User object
    user = User(name=name, email=email)
    user.create()  # save the object into a database

    # save user's email into a cookie
    response = make_response(redirect(url_for('index')))
    response.set_cookie("email", email)

    return response
```

Do another login on the web app and check if the cookie was successfully created.

Reading from a database

Now that we have **data in our database** and we have our **cookie**, we can try **reading** from the database. Let's do this in our `index()` handler:

```
@app.route("/")
def index():
    email_address = request.cookies.get("email")

    # get user from the database based on email address
    user = User.fetch_one(query=["email", "==", email_address])

    print(user.name)

    return render_template("index.html")
```

Reload `main.py` and index page and check your Terminal. You should see the user's name printed in the Terminal.

> *To reset the data just delete the `db.json` contents (and also the cookie).*

Showing data in the view (index.html)

Of course showing data **only** in the Terminal is **not** that much **useful**. We would want to show the user's data on the **website**. Or in case user doesn't have a cookie, show her/him the login form instead. Let's **adapt** our handler accordingly.

main.py:

```
@app.route("/")
def index():
    email_address = request.cookies.get("email")

    if email_address:
        user = User.fetch_one(query=["email", "==", email_address])
    else:
        user = None

    return render_template("index.html", user=user)
```

This code basically says:

- if there is a cookie with email address, find the user object in the database and send it to the `index.html`,
- but if there is no email cookie, set the `user` object as `None` (non-existing)

Let's now adapt the `index.html` template:

```
<h1>Simple database example</h1>

{% if user %}
    <p>Hello {{user.name}}</p>
    <p>This is your email address: {{user.email}}</p>
{% else %}
    <form method="post" action="/login">
        <input type="text" name="user-name" placeholder="Enter your name"><br>
        <input type="email" name="user-email" placeholder="Enter your email address"><br>

        <button>Submit</button>
    </form>
{% endif %}
```

So if `user` object exists, show a greeting (`Hello {{user.name}}`) and the user's email address.

But if there's no `user` , show the login form.

Cool, right?

> *See the complete example here (https://github.com/smartninja/wd1-py3-exercises/tree/master/lesson-18/smartninja-nosql-example).*
>
> *Learn more about how to use SmartNinja NoSQL here (https://github.com/smartninja/smartninja-nosql).*

Exercise 18.1: Update the Guess the secret number game

> *Instructor can do this exercise together with students (students telling him what code to write).*

Now that you've learned how to save an object into a database and read from it, you can update the **"Guess the secret number"** game.

You `User` model should have the following fields:

- name
- email
- secret_number

The `secret_number` field will hold the secret number that user has to guess.

> *Solution. (https://github.com/smartninja/wd1-py3-exercises/tree/master/lesson-18/guess-secret-number)*

Homework 18.2: Deploy your web app

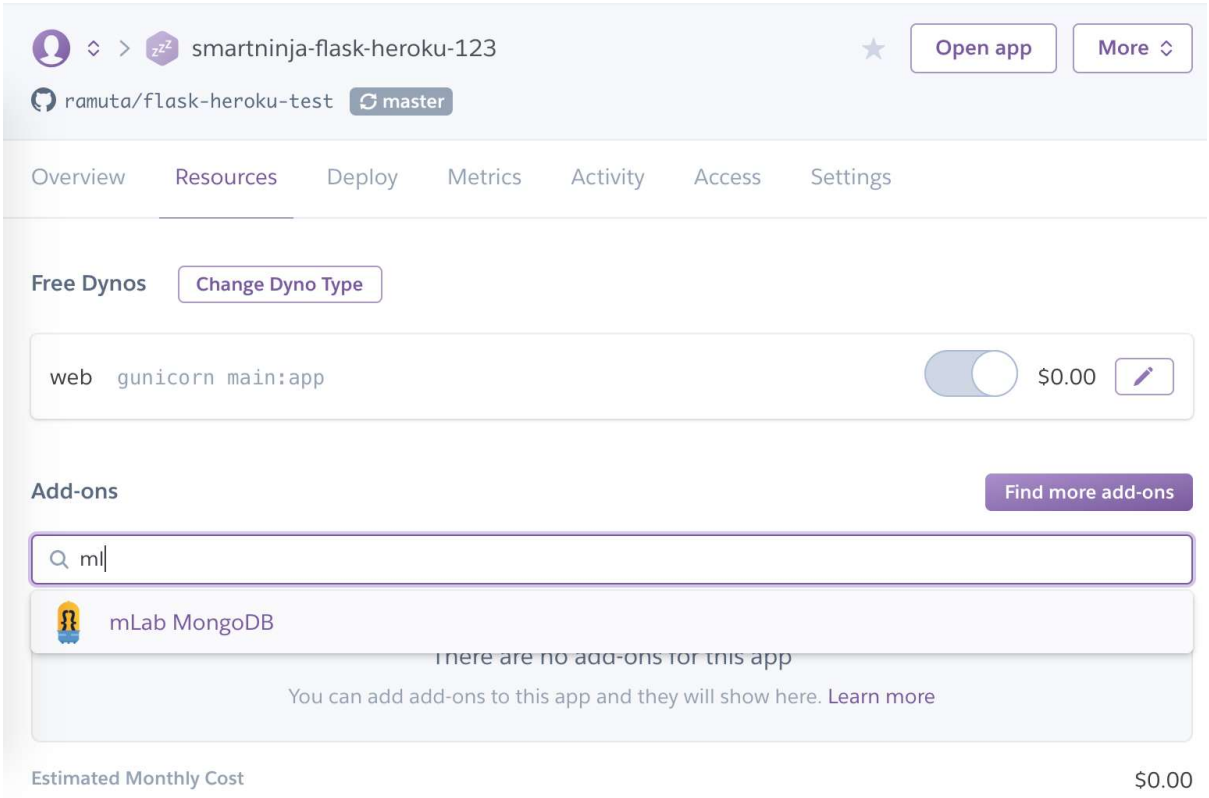
Deploy your web app to Heroku, Google App Engine and/or Azure.

There are a few more things to set up before you can make a successful deployment:

Heroku (MongoDB)

You will need to add `pymongo` to your **requirements.txt** file.

You will also need to add an **add-on** called **"mLab MongoDB"** on the Heroku Dashboard (click on your app, go to the Resources tab and "mLab" into the Add-Ons input box).



> ***Important:*** *You will need to enter your* ***credit card info*** *in Heroku in order to be able to use the Heroku add-ons. The credit card info is necessary for the* ***verification purposes****, so that Heroku can make sure you're not abusing their free tier with thousands of fake Heroku accounts. When you add your credit card into the Heroku system, you'll* ***still be able to use the free tier*** *and you won't be charged unless you choose to upgrade to a paid tier.*

Google App Engine (Datastore database)

You will need to add `google-cloud-datastore` in your **requirements.txt** file.

You will also need to add the following line in your `app.yaml` file:

```
env_variables:
  GAE_DATABASE: "datastore"
```

Google App Engine (Firestore database)

You will need to add `firebase-admin` and `google-cloud-firestore` in your **requirements.txt** file.

Make sure you created your web app in one of these two regions: `us-east1` or `europe-west3` (Firestore only runs on these two for now).

Azure App Service (Cosmos DB)

You will need to add `pymongo` to your **requirements.txt** file. This is because Azure allows you to use MongoDB API for its Cosmos DB (so they can make people easily switch from MongoDB to Cosmos DB).

You will also need to **enable Cosmos DB** for your App Service.

Bonus: Using pymongo library (without SmartNinja NoSQL)

▶ Click to see the bonus content.

Learn more about databases

If you'd like to learn more about databases, apply to either our **SQL and databases course** or to the **Web development 2** course.
