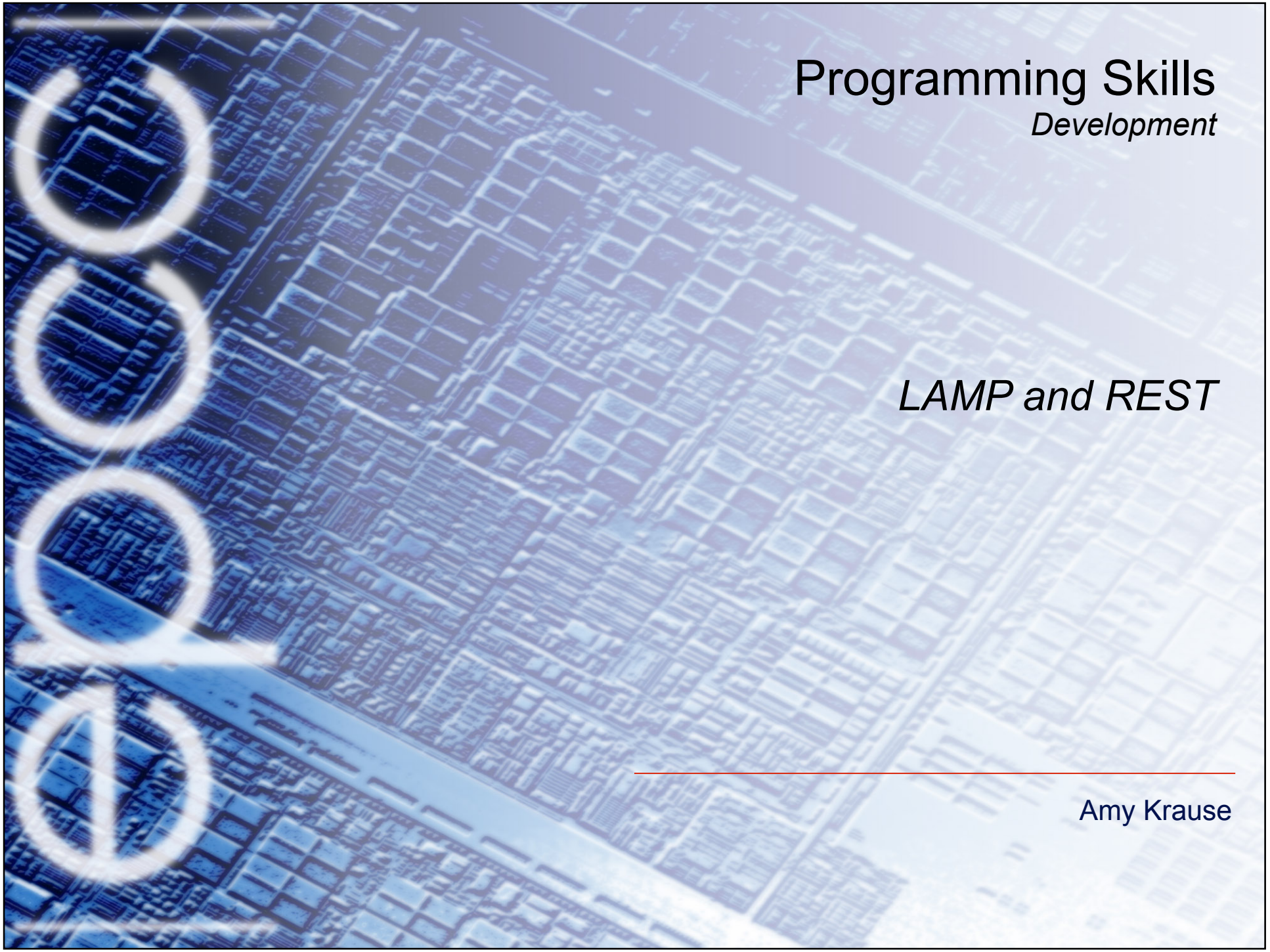

Programming Skills
Development
LAMP and REST
Amy Krause

Outline

- What is LAMP?
- What is REST?
- Introduction to REST
- An example REST application
- Building the LAMP stack

What is LAMP

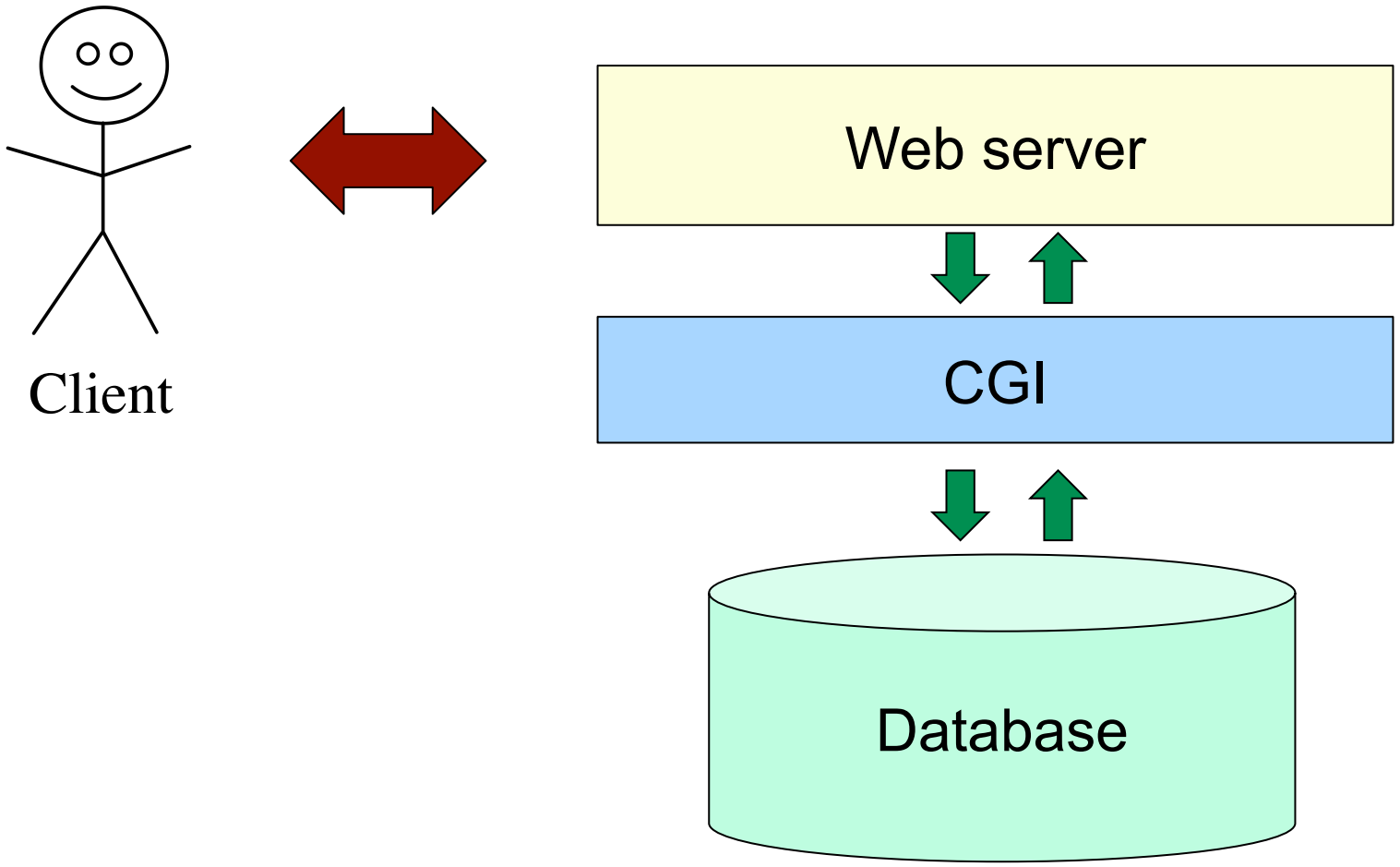
- Originally

  LAMP = Linux Apache MySQL PHP

- Now refers to a software stack model for building dynamic websites and web applications
- Components can vary
- Now

  Web Server + Database + Scripting Language

LAMP in a picture



“LAMP” software stacks

- Web server, e.g.
  - Apache
  - Tomcat
  - NGINX
- CGI: Common gateway interface
  - PHP
  - Python: Django/Flask/CherryPy
  - Java: Java EE/JSP/Grails
  - Ruby/Rails
- Database:
  - SQL: MySQL/PostgreSQL/Oracle/SQLServer/SQLite/…
  - NoSQL: MongoDB/…
  - Cassandra/HBase/…

Linux

- Easy to install a LAMP stack on most Linux distributions
  - Providing setup through packaging manager
- Choose a server-side scripting language
  - Processes input from the client (browser) and generates resulting web pages
  - Should provide good text processing facilities
  - Hence the popularity of Perl and Python
- Choose a database
  - SQL or NoSQL? Vendor?
- Choose a web server
  - Support for your scripting language
  - Should handle authentication and load balancing

What is REST

- **RE**presentational **S**tate **T**ransfer
- Interfaces defined with HTTP commands
  - GET, POST, PUT, DELETE
- Common architecture
  - Stateless
  - Resource representation
  - State transitions

REST

A **resource** is an object with a set of operations that can be applied to it:

- GET: show information about a resource
- POST: create a new resource
  - Returns the URL of the resource
- PUT: update or create a named resource
- DELETE: delete a resource

Resources can be grouped into collections.

A RESTful example

- Task list with RESTful interface
- Tasks are resources
- What kinds of methods do we need?
  - Create a new task
  - List all tasks
  - Read task details
  - Tick off a task when it's done

A RESTful example

- How does this translate to a REST interface?

- Create a new task: POST
- List all tasks: GET
- Read task details: GET
- Change a task: PUT
- Mark task as done: DELETE

Example of a client interaction

	Method	Activity
1	GET	List all tasks
2	POST	Create a new task
3	GET	Check the task was created
4	GET <id>	Get the task info
5	PUT <id>	Update the task
6	GET <id>	Check the task was updated
7	DELETE <id>	Remove the task
8	GET	Check the task was removed

Example: Implementation in Python

- Now we're going to build a RESTful web service in Python
- Using flask (http://flask.pocoo.org/)

Hello World!

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

@app.route("/")
def hello():
    return "Hello World!"
```

RESTful Python using flask

GET

```
@app.route("/<task_id>")
def get_task(task_id):
    return tasks[task_id]
```

POST

```
@app.route("/", methods=["POST"])
def add_task():
    task_id = uuid.uuid4()
    tasks[task_id] = request.form
```

RESTful Python using flask

```
@app.route("/<task_id>",
           methods=["GET",
                    "PUT",
                    "DELETE"])
def task_resource(task_id):
    task = tasks[task_id]

    if request.method == "PUT":
        task = json.loads(request.data)
        tasks[task_id] = task

    elif request.method == "DELETE":
        del tasks[task_id]

    return task
```

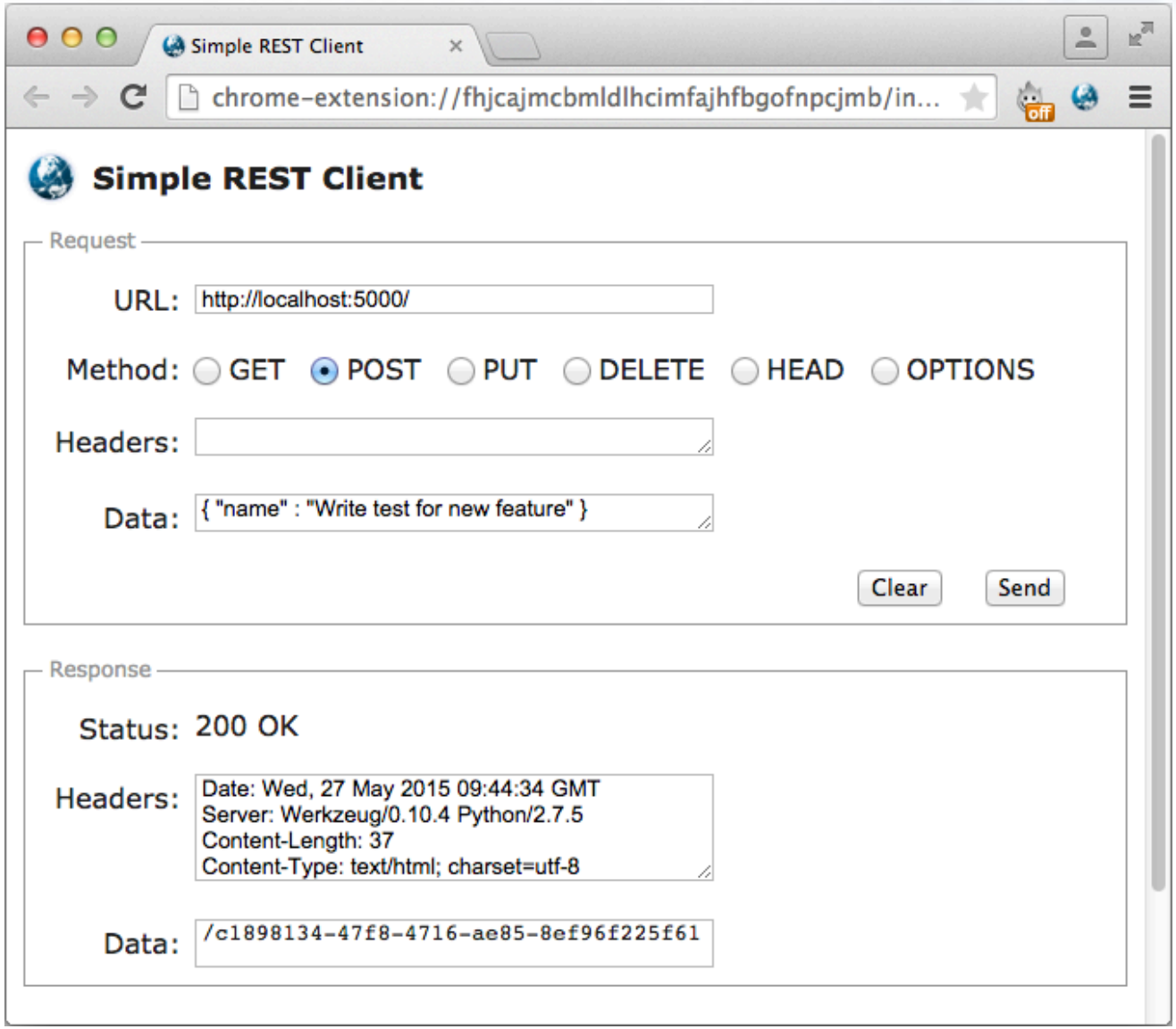
Deploying to a web server

- Flask includes a simple web server for testing
  - Debug mode available

```
$ python hello.py
 * Running on http://127.0.0.1:5000/
```

- Can be deployed to a “proper” web server (e.g. Apache, NGINX) that supports WSGI
  - WSGI: **W**eb **S**erver **G**ateway **I**nterface for Python web applications
  - Web server handles authentication, load balancing, etc.

Client example



Another client example

- Using the command line tool **cURL**(http://curl.haxx.se/)

```
curl -X POST
     -d '{"name":
          "Write test for new feature"}'
     http://localhost:5000/
```

Database

- In the task example we didn't use a persistent data storage
- Tasks are stored in memory
  - Not robust (data is lost if the server goes down)
  - Not thread-safe (clients writing to the same task)
  - Not scalable or distributed (many clients access at the same time)
- Add database access to address issues
  - Query the database to retrieve a list of tasks or information on a specific task
  - Update the database to create or remove a task
  - Database access is usually thread-safe and transactional

Adding a database: Requirements

- Task archive: Store and retrieve tasks
- Query: Find a task by name or other features
- Persistent storage: Tasks are not lost when the server is down
- Concurrent access: Write operations must be atomic on the level of a task

Database example: MongoDB

- There are many database vendors and products that can be used as storage solutions
  - MySQL, PostgreSQL, SQLite, MongoDB, HBase, Cassandra, …
- Choose the database solution for your use case and the queries you will run
  - NoSQL or SQL, distributed or single server, …
- For our example we will use MongoDB
- MongoDB is available for many platforms; easy to install
  - Unpack and start up

MongoDB clients

- Example: Python MongoDB client

```
import pymongo

client = MongoClient()
tasks = client.db.collection
```

Database example: MongoDB

- Add a task:

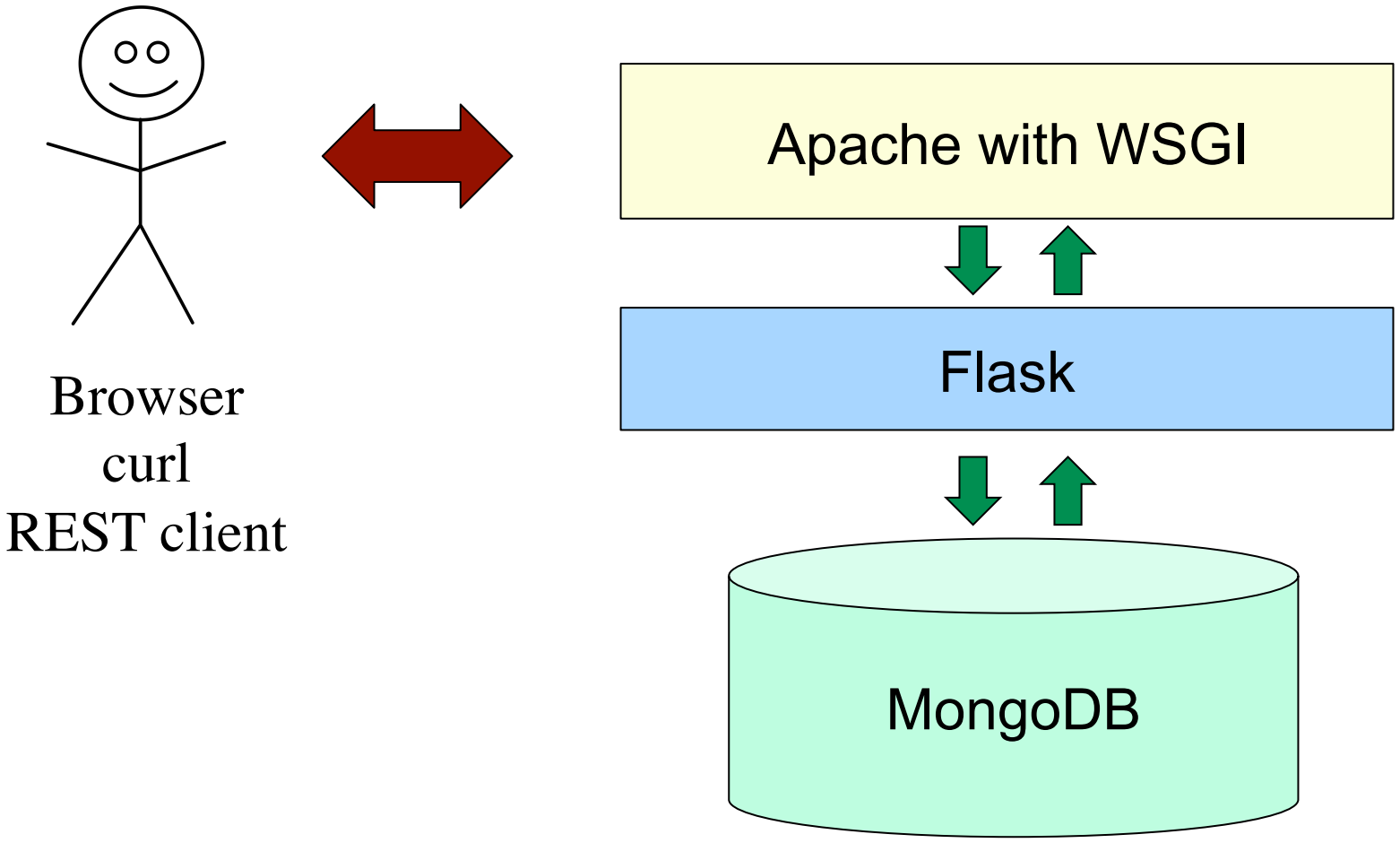
```
tasks.insert_one(task)
```

- Retrieve a task:

```
tasks.find_one({“name”: task_id})
```

- And more complicated queries!

Our example in a picture



Conclusion

- Web services are everywhere
- LAMP is a standard software stack
- Based on REST
- Defines server interfaces with a standard set of methods
- Easy to build applications