

REAL TIME WEB APPS

with (just) python and postgres

A PRESENTATION IN TWO PARTS

1 - Why?

2 - How?

1 - WHY?

Real Time

Python

Postgres

REAL TIME === ?

- “hard” real time = Pacemakers. Car engines.
- “soft/near” real time = Chat rooms.
- “not at all” real time = Most web pages. (Hitting F5 repeatedly doesn't count.)

WHY “REAL TIME”?

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Jakob Nielsen - 10 Usability Heuristics for User Interface Design

WHY “REAL TIME”?

*The system should always keep users informed about what is going on, through **appropriate** feedback within **reasonable** time.*

Jakob Nielsen - 10 Usability Heuristics for User Interface Design

“APPROPRIATE”

- games
- monitoring (stocks. servers. sensors.)
- automation (batch jobs. builds. deployments.)
- collaboration (etherpad. trello.)

WHY PYTHON?

- Readability
- Maturity
- Productivity
- Fun

WHY PYTHON?

- Readability
- Maturity
- Productivity
- Fun**

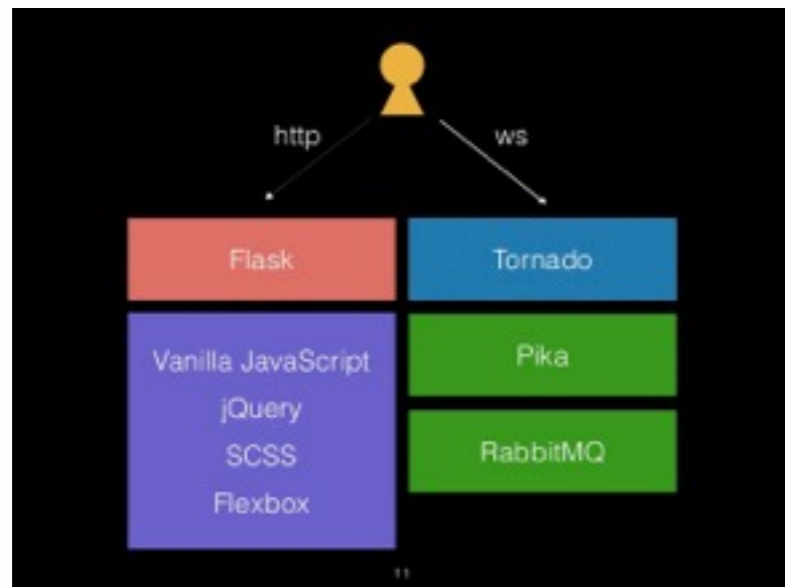
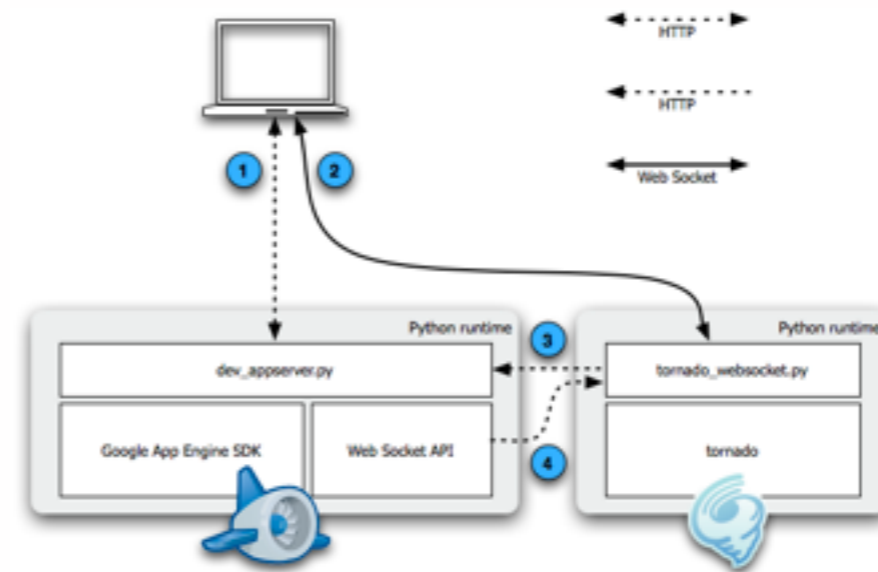
WHY POSTGRES?

- Maturity
- Interoperability
- Types!
- Data Integrity (Constraints, Foreign Keys)
- Flexibility/Extensibility (JSON types, triggers, foreign data wrappers)
- Fun

WHY POSTGRES?

- Maturity
- Interoperability
- Types!
- Data Integrity (Constraints, Foreign Keys)
- Flexibility/Extensibility (JSON types, triggers, foreign data wrappers)
- Fun**

WHY "JUST"?



LET'S SCALE DOWN

A solution that requires multiple server components or frameworks is unlikely to be considered for weekend hacks, internal tools, or unglamorous CRUD apps.

Let's make it so easy to build real time updates into our interfaces that there's no excuse *not* to.

LET'S SCALE DOWN



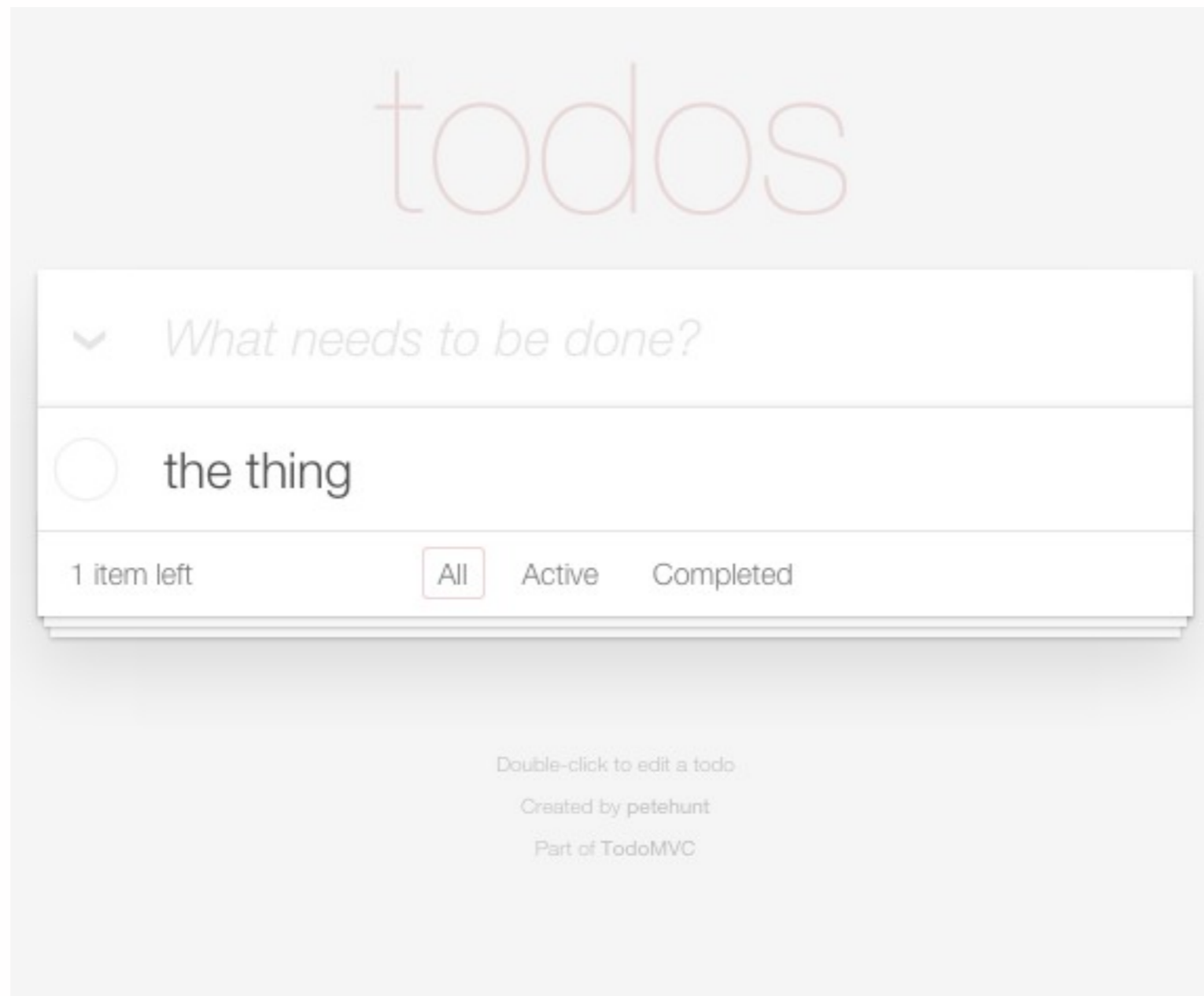
A PRESENTATION IN TWO PARTS

~~1 - Why?~~

2 - How?

OUR APPLICATION

<https://bitbucket.org/btubbs/todopy-pg>



WE NEED

- REST APIs
- WebSockets
- Javascript-based UI (Angular, Ember, React)
- Javascript-based toolchain (npm, bower, uglify.js)

WE DON'T NEED

- Server side templating
- Server-side form handling
- An ORM

NO FRAMEWORK

- Werkzeug + custom code for routing.
- Plain WSGI middleware for sessions, auth.
- Gevent + GWebSocket.

PERSONAL BIASES

- Routes all in one place (like Django), not spread around (Flask, CherryPy).
- RESTful routing (classes with `get()`, `post()`, etc. methods).
- No magical context-locals (`flask.request`, `cherrypy.response`).
- Implicit async (Gevent) > Explicit callbacks/yields/promises (Twisted, Tornado, Node.js).

2 - HOW?

... to get real time updates from Postgres.

... to create an API that cleanly combines REST and WebSockets.

... to build a real-time UI on top.

POSTGRES

Our schema:

```
CREATE EXTENSION IF NOT EXISTS "uuid-oss";
```

```
CREATE TABLE todos (  
  id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,  
  created_time TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,  
  title VARCHAR(512) NOT NULL,  
  completed BOOLEAN NOT NULL DEFAULT false  
);
```

POSTGRES

Has a pubsub!

PSQL

```
todos=# LISTEN some_channel_name;

todos=# SELECT 1;
...

Asynchronous notification
"some_channel_name" with payload "this is
a message" received from server process
with PID 22023.
todos=#
```

PSQL

```
todos=# NOTIFY some_channel_name, 'this
is a message';
```

*The "SELECT 1;" is to force psql to check back in with the server. When subscribing from Python we won't have to do that.

POSTGRES

Has JSON:

PSQL

```
todos=# INSERT INTO todos (description) VALUES ('the thing');
INSERT 0 1
todos=# SELECT row_to_json(todos) FROM todos;
           row_to_json
```

```
{"id":"ac51a46d-f905-4156-b0c7-251526bde8c4",
 "created_time":"2015-05-15T23:07:17.453143+00:00",
 "description":"the thing",
 "completed":false}
(1 row)
```


POSTGRES

Has JSON:

PSQL

```
todos=# INSERT INTO todos (description) VALUES ('the thing');
INSERT 0 1
todos=# SELECT row_to_json(todos) FROM todos;
           row_to_json
-----
{"id":"ac51a46d-f905-4156-b0c7-251526bde8c4",
 "created_time":"2015-05-15T23:07:17.453143+00:00",
 "description":"the thing",
 "completed":false}
(1 row)
```

POSTGRES

Has triggers:

```
CREATE OR REPLACE FUNCTION todos_notify_func() RETURNS trigger as $$  
BEGIN  
    PERFORM pg_notify('todos_updates', row_to_json(NEW));  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER todos_notify_trig AFTER INSERT OR UPDATE ON todos  
    FOR EACH ROW EXECUTE PROCEDURE todos_notify_func();
```

*See project repo for example that also handles DELETE operations.

POSTGRES

Has triggers:

```
CREATE OR REPLACE FUNCTION todos_notify_func() RETURNS trigger as $$
BEGIN
    PERFORM pg_notify('todos_updates', row_to_json(NEW));
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER todos_notify_trig AFTER INSERT OR UPDATE ON todos
    FOR EACH ROW EXECUTE PROCEDURE todos_notify_func();
```

*See project repo for an example that also handles DELETE operations.

GETTING BIG

- “Generally, PostgreSQL on good hardware can support a few hundred connections.” -- https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server
- Ubuntu 14.04 default max_connections=100
- For thousands of clients, forward messages through RabbitMQ (example at <https://bitbucket.org/yougov/mettle>)
- LISTEN/NOTIFY commands limit payloads to 8000 bytes.

PYTHON

Can subscribe:

```
import select

import psycopg2

conn = psycopg2.connect(user='postgres', database='todos')
conn.autocommit = True
cur = conn.cursor()
cur.execute('LISTEN test;');

while True:
    if select.select([conn], [], [], 5) != ([], [], []):
        conn.poll()
        while conn.notifies:
            notify = conn.notifies.pop(0)
            print notify.payload
```

PYTHON

Using pgpubsub:

```
import pgpubsub

pubsub = pgpubsub.connect(user='postgres', database='todos')

pubsub.listen('test');

for e in pubsub.events():
    print e.payload
```

<https://pypi.python.org/pypi/pgpubsub/>

PYTHON

Can subscribe:

Bash

```
$ python tmp/pglisten.py
```

```
Hello Python!
```

PSQL

```
todos=# NOTIFY test, 'Hello Python!';
```

2 - HOW?

~~... to get real time updates from Postgres.~~

... to create an API that cleanly combines REST and WebSockets.

... to integrate Python- and JS-based toolchains.

REST API

Create a todo	POST /api/todos/
List all todos	GET /api/todos/
Get one todo	GET /api/todos/<id>/
Update a todo	PUT /api/todos/<id>/
Delete a todo	DELETE /api/todos/<id>/
Stream changes to todos	???
Stream changes to a todo	???

RESTSockets

Create a todo	POST /api/todos/
List all todos	GET /api/todos/
Get one todo	GET /api/todos/<id>/
Update a todo	PUT /api/todos/<id>/
Delete a todo	DELETE /api/todos/<id>/
Stream changes to todos	WebSocket /api/todos/
Stream changes to a todo	WebSocket /api/todos/<id>/

A VIEW

```
class TodoDetail(ApiView):  
    def get(self, todo_id):  
        ...  
    def put(self, todo_id):  
        ...  
    def delete(self, todo_id):  
        ...  
    def websocket(self, todo_id):  
        ...
```

A VIEW

```
class TodoDetail(ApiView):
    ..

    def put(self, todo_id):
        todo = json.loads(self.request.data)
        query = "UPDATE todos SET title=%s, completed=%s WHERE id=%s RETURNING id;"
        self.db.execute(query, (todo['title'], todo['completed'], todo_id))
        updated = self.db.fetchone()
        if updated is None:
            return NotFound()
        url = reverse(self.app.map, 'todo_detail', {'todo_id': todo_id})
        return redirect(url)

    ...
```

A VIEW

```
class TodoDetail(ApiView):
    ..

    def websocket(self, todo_id):
        # first send out the data for this todo.
        todo = self.get_todo(todo_id)
        self.ws.send(json.dumps(todo))

        # Then stream out any updates.
        self.pubsub.listen('todos_updates')
        for e in self.pubsub.events(yield_timeouts=True):
            if e is None:
                self.ws.send_frame('', self.ws.OPCODE_PING)
            else:
                # Only publish this payload if it has our ID.
                parsed = json.loads(e.payload)
                if parsed.get('id') == todo_id:
                    self.ws.send(e.payload)
                else:
                    # No match. Just send a ping.
                    self.ws.send_frame('', self.ws.OPCODE_PING)
```

API DEMO

2 - HOW?

~~... to get real time updates from Postgres.~~

~~... to create an API that cleanly combines REST and
WebSockets.~~

... to build a real-time UI on top.

CHOICES CHOICES

Source Languages	Frameworks	Package Managers	Task Runners
Plain JS	jQuery(+UI)	npm	Grunt
JSX	Angular	bower	Gulp
CoffeeScript	React	component	Paver
ES6	Ember	browserify	Make
Elm	Backbone	webpack	
TypeScript	Knockout		
PureScript			

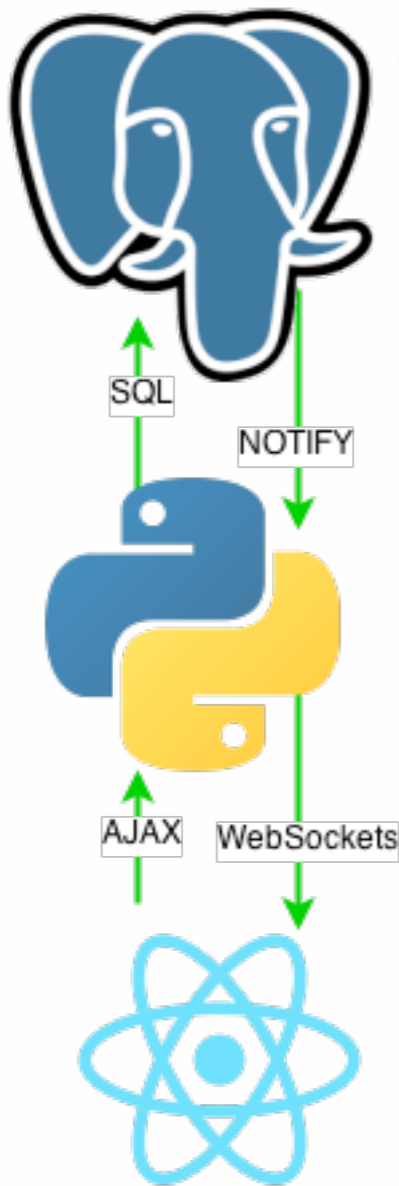
CHOICES CHOICES

Source Languages	Frameworks	Package Managers	Task Runners
Plain JS JSX CoffeeScript ES6 Elm TypeScript PureScript	jQuery(+UI) Angular React Ember Backbone Knockout	npm bower component browserify webpack	Grunt Gulp Paver Make

CHOICES CHOICES

Source Languages	Frameworks	Package Managers	Task Runners
Plain JS	jQuery(+UI)	npm	Grunt
JSX	Angular	bower	Gulp
CoffeeScript	React	component	Paver
ES6	Ember	browserify	Make
Elm	Backbone	webpack	
TypeScript	Knockout		
PureScript			

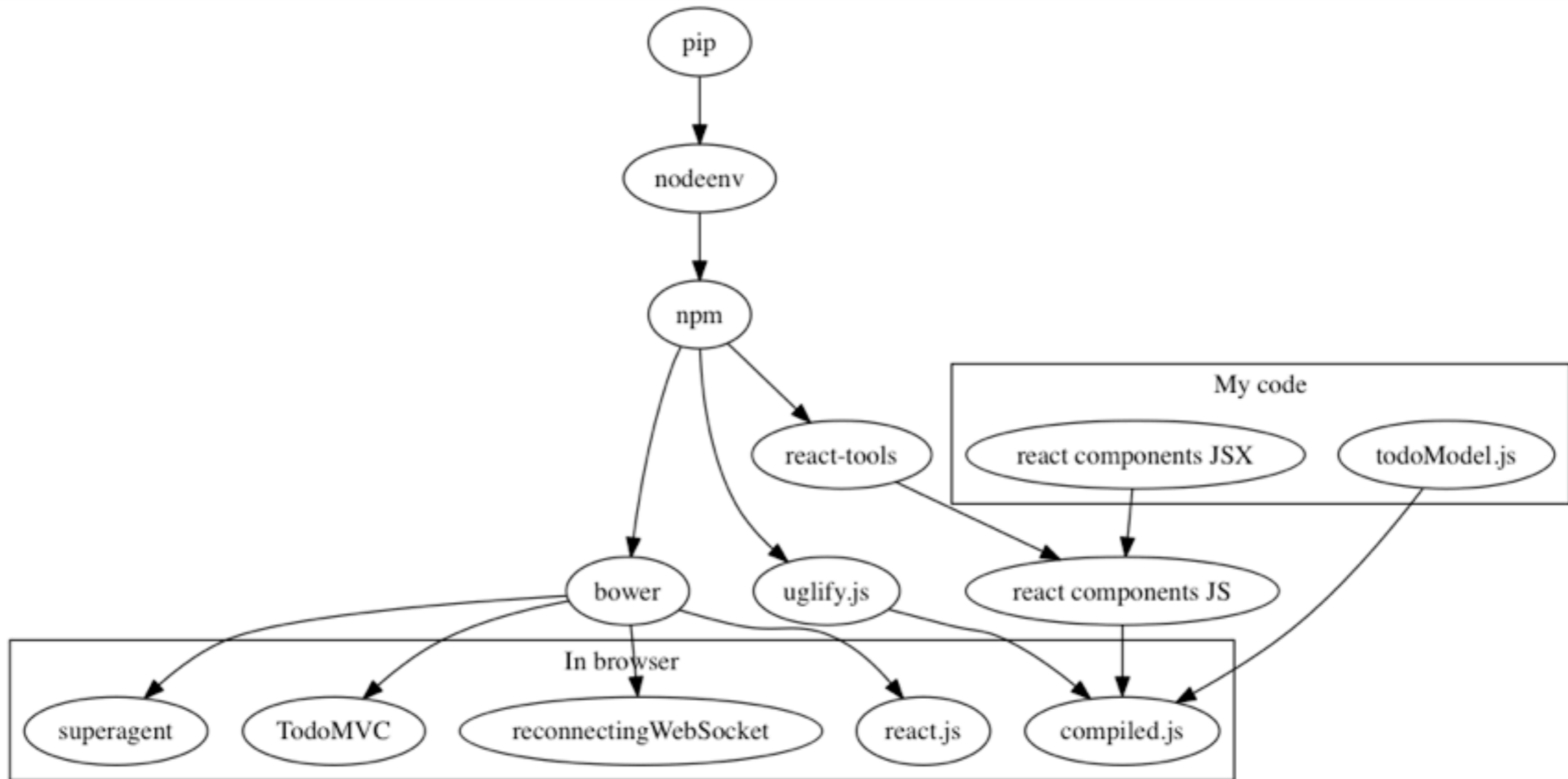
React.js



“React implements one-way reactive data flow which reduces boilerplate and is easier to reason about than traditional [two way] data binding.”

Switching the React TodoMVC app to a WebSocket backend required changing only one file, `todoModel.js`.

Makefile



2 - HOW?

~~... to get real time updates from Postgres.~~

~~... to create an API that cleanly combines REST and
WebSockets.~~

~~... to build a real-time UI on top.~~

UI DEMO

SUMMARY

- Use triggers to NOTIFY Python of changes.
- Use as little framework as possible.
- Use RESTSockets pattern.
- Use React.js.
- Build with make.

QUESTIONS?

<http://btubbs.com/documents/RealTimeWebApps.pdf>