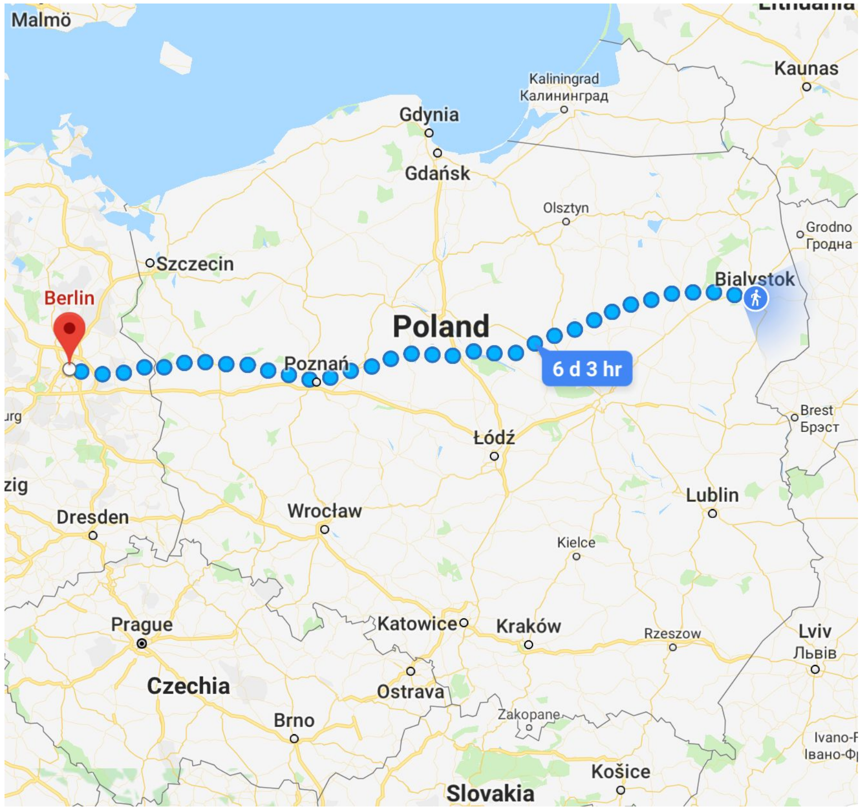
Functional APIs with GraphQL & Elixir

CODE BEAM LITE BERLIN 2018

About me

- Hubert Łępicki
- @hubertlepicki
- https://www.amberbit.com
- Białystok, Poland



elixir

The history behind it (educated guess)

- Frontend: “We need list of posts with thumbnails and short text”
- Back end: “Ok”
- Frontend: “We need to make thumbnail fetching optional and need author info”
- Back end: “Ok”
- Frontend: “We need optional list of comments with each post”
- Back end: “You are ruining my API but okay”
- Frontend: “We need….”

AIN'T NOBODY GOT
TIME FOR THAT
memecenter.com
MemeCenter

Born in 2012

Cambridge Analytica

Made public in 2015

What is this GraphQL thing?

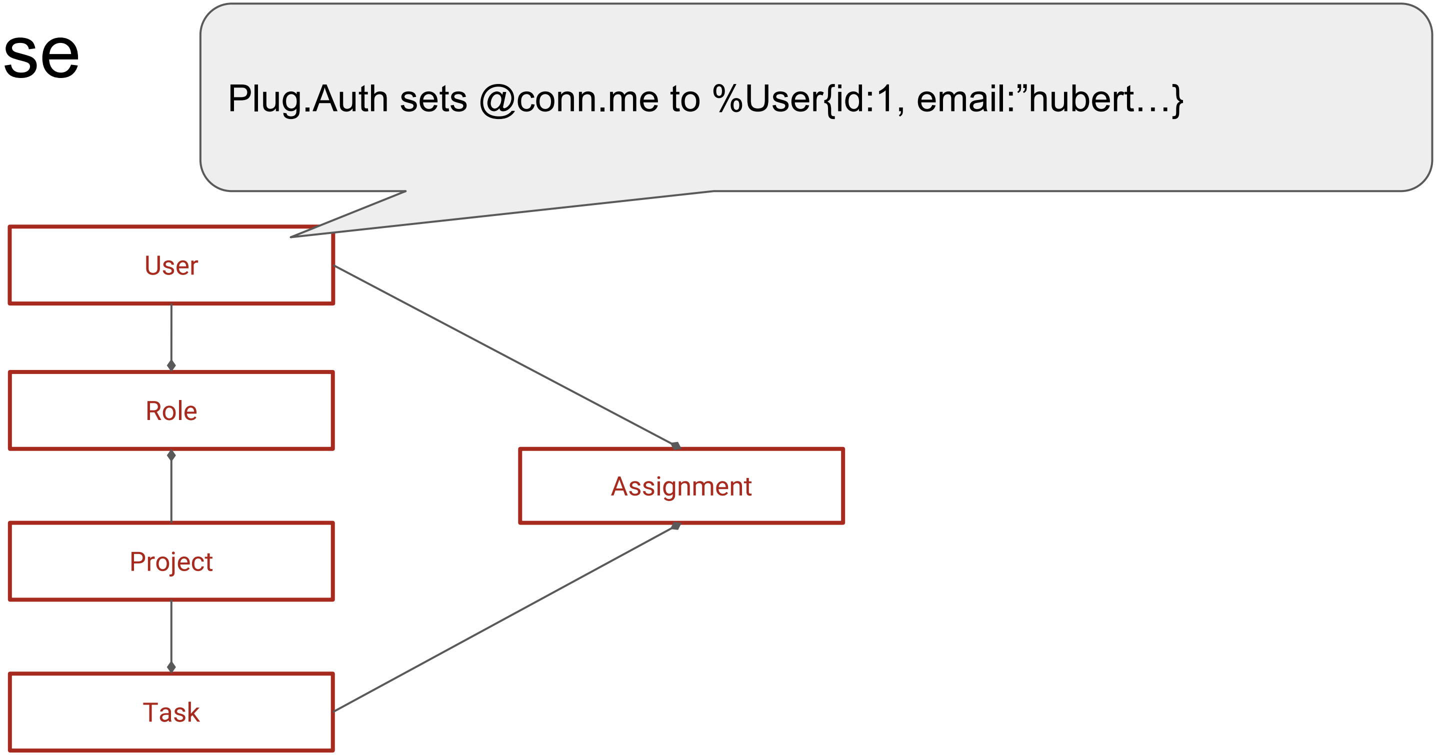
- Graph Query Language
- Specification https://facebook.github.io/graphql
- Describes how you query the data you want to retrieve
- Describes how you modify the data
- Describes how you get notified on data changes
- Transport-independent
- Usually used via HTTP API
- Can be used over WebSocket
- Can be used over custom transports
- Can be used within application internally

GraphQL in Elixir

- Absinthe GraphQL Toolkit https://absinthe-graphql.org/
- One of most complete GraphQL server-side specification implementations
- Modular “toolkit” architecture, consisting of many small repositories (absinthe, absinthe_plug etc.)
- Actively worked on & maintained
- Actively used in production
- Good match (esp. subscriptions)
- Sorts out some architectural design problems for your apps for you

Time for some examples!

Our database



How does it look like?

```
doc = “””query {
  me {
    id,
    email
  }
}”””
```

How does it look like?

```
doc = “””query {
  me {
    email,
    projects {
      id,
      name
    }
  }
}”””
```

How does it look like?

```
doc = “””query {
  me {
    email,
    projects {
      id,
      tasks {
        id,
        name,
        completed
      }
    }
  }
}”””
```

How does it look like?

```
doc = “””query {
  me {
    email,
    projects {
      id,
      tasks(matching: “deploy”) {
        id,
        name
      }
    }
  }
}”””
```

How does it look like?

```
doc = “””query {
  me {
   tasks(completed: false) {
     id,
     name
    }
  }
}”””
```

How does it look like?

```
Absinthe.run(doc, MyApp.Schema, context: %{})

=> {:ok, %{data: … }}

=> {:error, errors}
```

How does it look like?

```
%{data: %{
  "me" => %{
    "email" => "hubert.lepicki@amberbit.com",
    "projects" => [
      %{"id" => 1,
        "tasks" => [
          %{"id" => "1", "name" => "Deploy to staging"},
          %{"id" => "2", "name" => "Deploy to production"}

        ]}]}}}
```

Computed fields

```
query {
  me {
    projects {
      id,
      name,
      completed_percents
    }
  }
}
```

Computed fields

```
%{data: %{
  "me" => %{
    "projects" => [
      %{"id" => 1,
        "name" => “Conquering the World”,
        “completed_percents” => 99
      }]}}}
```

Let’s get our hands dirty!

```
# mix.exs
...
defp deps do
  [ ...
    {:absinthe_phoenix, “~> 1.4”} ]
end
```

```
➜  my_app git:(master) ✗ mix deps.get
Resolving Hex dependencies...
Dependency resolution completed:
  absinthe 1.4.13
  absinthe_phoenix 1.4.3
...
```

Let’s get our hands dirty!

```
# lib/my_app_web/endpoint.ex
defmodule MyAppWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :my_app
  use Absinthe.Phoenix.Endpoint
  ...

# lib/my_app/application.ex
  ...
  supervisor(MyAppWeb.Endpoint, []),
  supervisor(Absinthe.Subscription, [MyAppWeb.Endpoint])
  ...
```

Let’s get our hands dirty!

```
# lib/my_app_web/channels/user_socket.ex
defmodule MyAppWeb.UserSocket do
  use Phoenix.Socket
  use Absinthe.Phoenix.Socket, schema: MyAppWeb.Schema
  ...
```

Let’s get our hands dirty!

```
# lib/my_app_web/router.ex
  ...
  forward("/api/graphiql", Absinthe.Plug.GraphiQL,
          schema: MyApp.Schema, interface: :advanced)

  scope “/api” do
    pipe_through(:api)
    forward(“/”, Absinthe.Plug, schema: MyApp.Schema)
  end
  ...
```

Describe your API with Schema

```elixir
# lib/my_app/schema.ex

defmodule MyApp.Schema do
  use Absinthe.Schema

  # list objects
  ...
  # list queries & mutations
  ...
end
```

Sad news for you

- GraphQL is Object-Oriented
- ...or not really :)

Objects

- Compound types, consisting of one or more fields
- Used for nodes in graph
- RootQueryType
- Me (or maybe User?)
- Project
- Task

Scalars

- Boolean
- Float
- ID
- Int
- String

- Absinthe-specific: :datetime, :naive_datetime, :date, :time, :decimal

Scalars

```
scalar :my_date do
  parse fn input ->
    case Date.from_iso8601(input.value) do
      {:ok, date} -> {:ok, date} _ -> :error
    end
  end

  serialize fn date -> Date.to_iso8601(date) end
end
```

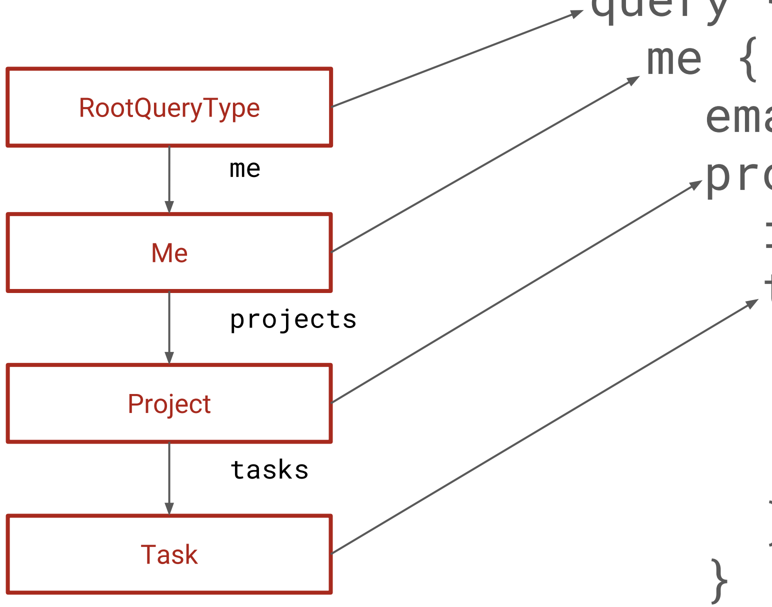
* example from *Craft GraphQL APIs in Elixir with Absinthe*

Types in GraphQL

- Objects
- Scalars
- (and more… Unions, Interfaces, Enumerations...)

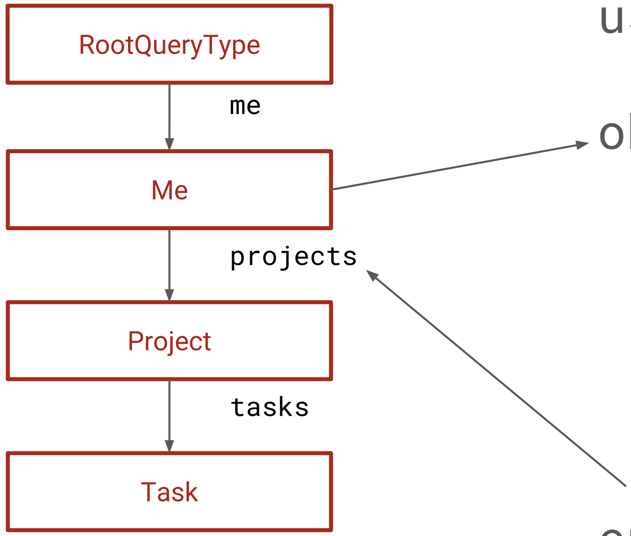
...but where is the graph?

The Graph & The Query



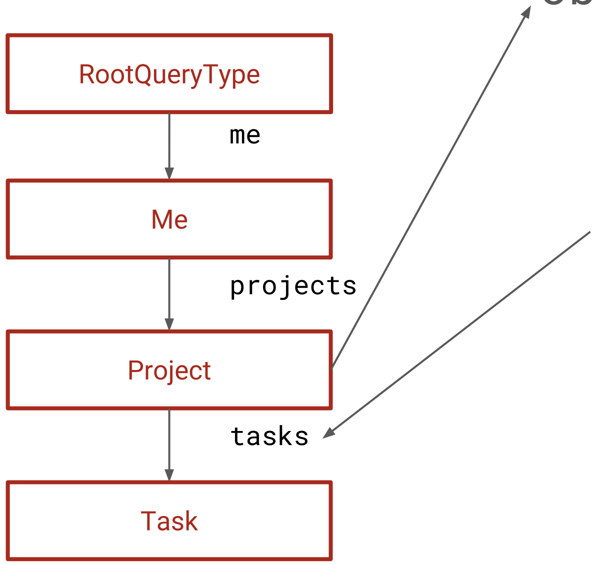
```
query {
  me {
    email,
    projects {
      id,
      tasks(matching: “deploy”) {
        id,
        name,
        completed
      }
    }
  }
}
```

The Graph & The Schema



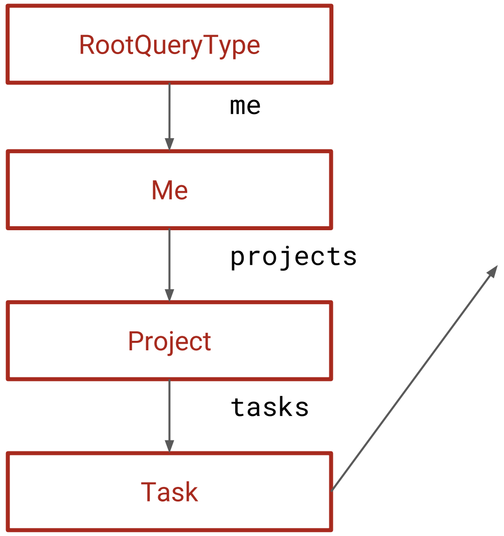
```
defmodule MyApp.Schema
  use Absinthe.Schema

  object :me do
    field :id, non_null(:string)
    field :name, non_null(:string)
    field :email, non_null(:string)
    field :avatar_url, :string

    field :projects, list_of(:project)
  end
  ...
end
```

The Graph & The Schema



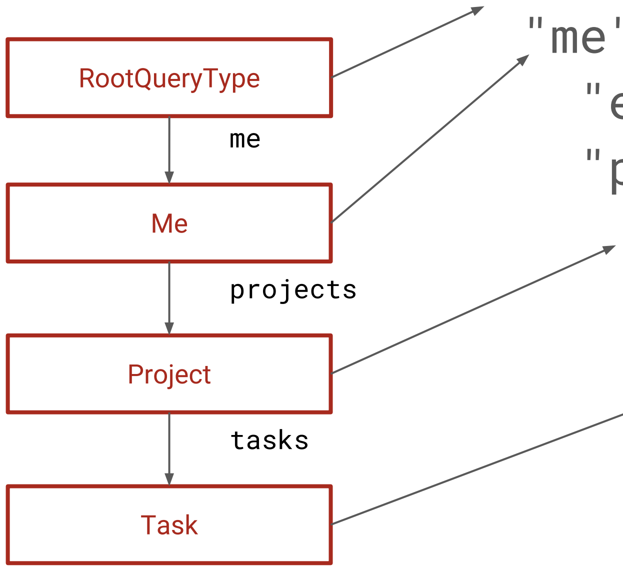
```
object :project do
  field :id, non_null(:id)
  field :name, non_null(:string)

  field :tasks, list_of(:task) do
    @desc “Searches tasks by string”
    arg :matching, :string
  end

  @desc “Computed on the fly!”
  field :completed_percents, :integer
end
```

The Graph & The Schema



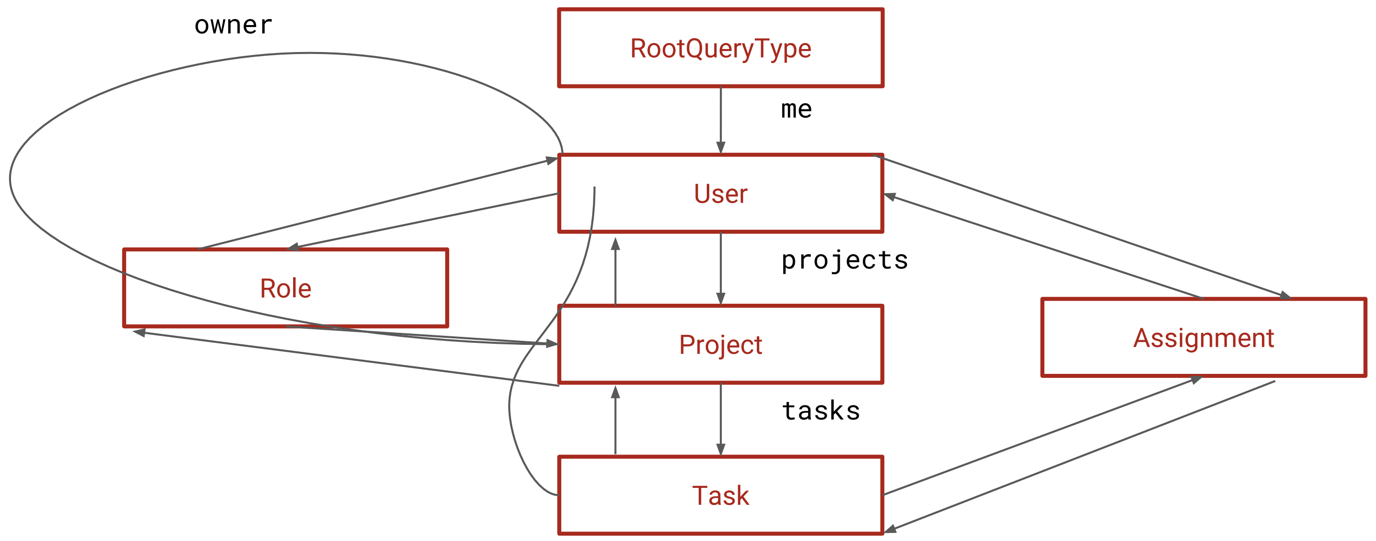
```
object :task do
  field :id, non_null(:id)
  field :name, :string
  field :completed, non_null(:boolean)
end
```

The Graph & The Response

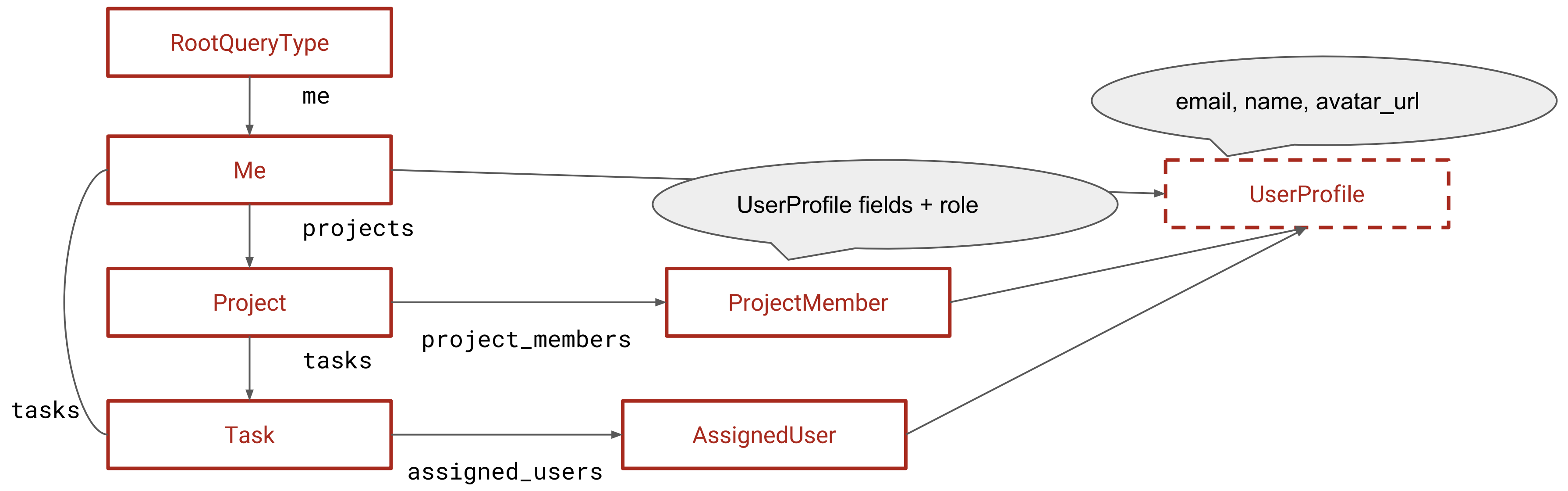


```
%{data: %{
  "me" => %{
    "email" => "hubert@example.com",
    "projects" => [
      %{"id" => 1,
        "tasks" => [
          %{"name" => "Deploy to staging"},
          %{"name" => "Deploy to prod"}

        ]}]}}}
```

Schema design



Schema design



lib/my_app/schema.ex

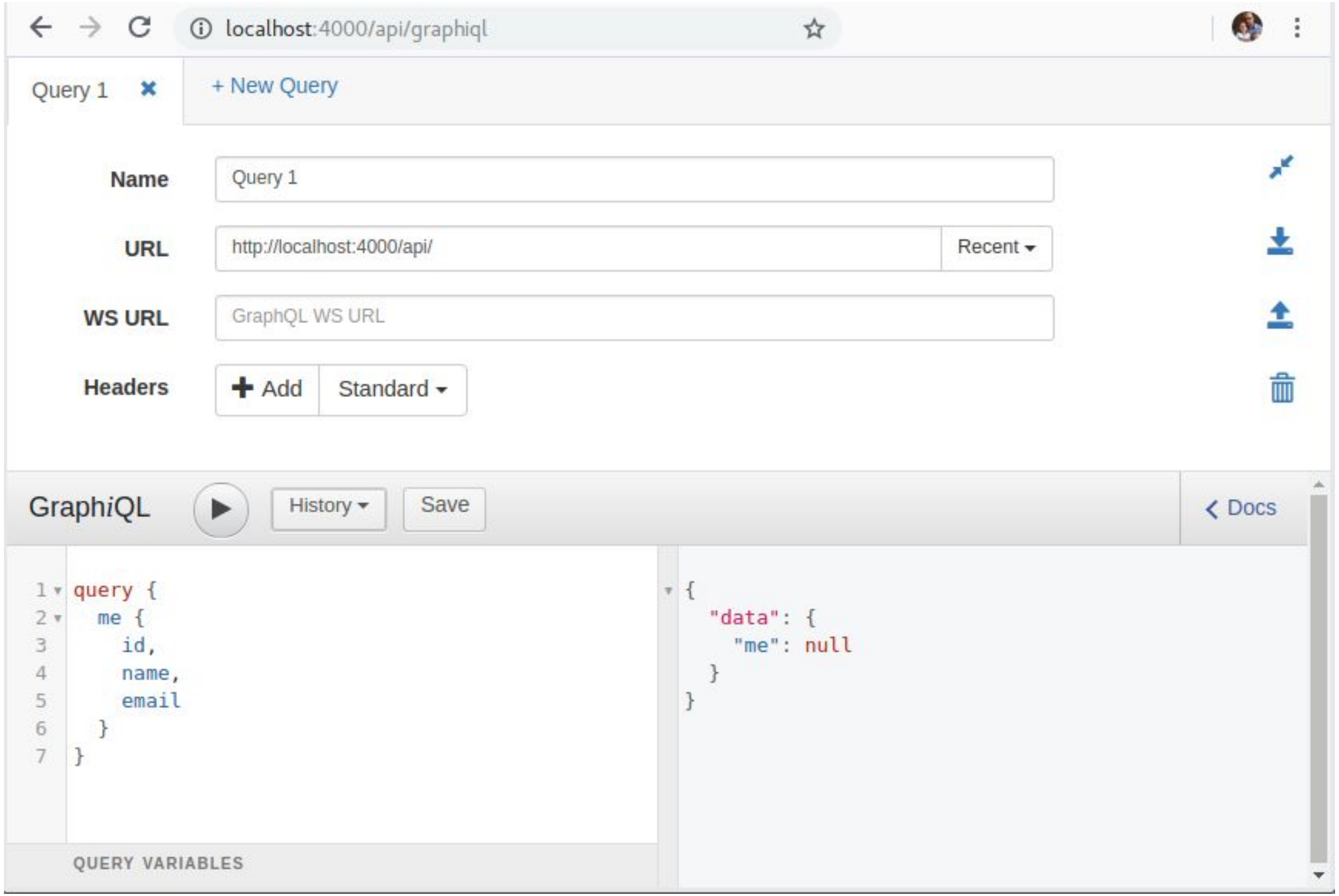
```
defmodule MyApp.Schema do
  use Absinthe.Schema

  object :me do
     field :id, non_null(:string)
     field :name, non_null(:string)
     field :email, non_null(:string)
     field :avatar_url, :string
     field :projects, list_of(:project)
     field :tasks, list_of(:tasks)
  end
  ...
```

lib/my_app/schema.ex

```
...
object :project do
  field :id, non_null(:id)
  field :name, non_null(:string)
  field :tasks, list_of(:task) do
    arg :matching, :string
  end
  field :completed_percents, non_null(:integer)
end
...
```

lib/my_app/schema.ex

```
  ...
  object :task do
    field :id, non_null(:id)
    field :name, :string
    field :completed, non_null(:boolean)
  end

  query do
    field :me, :me
  end
end
```



Name: Query 1

URL: http://localhost:4000/api/

WS URL: GraphQL WS URL

```
query {
  me {
    id,
    name,
    email
  }
}
```

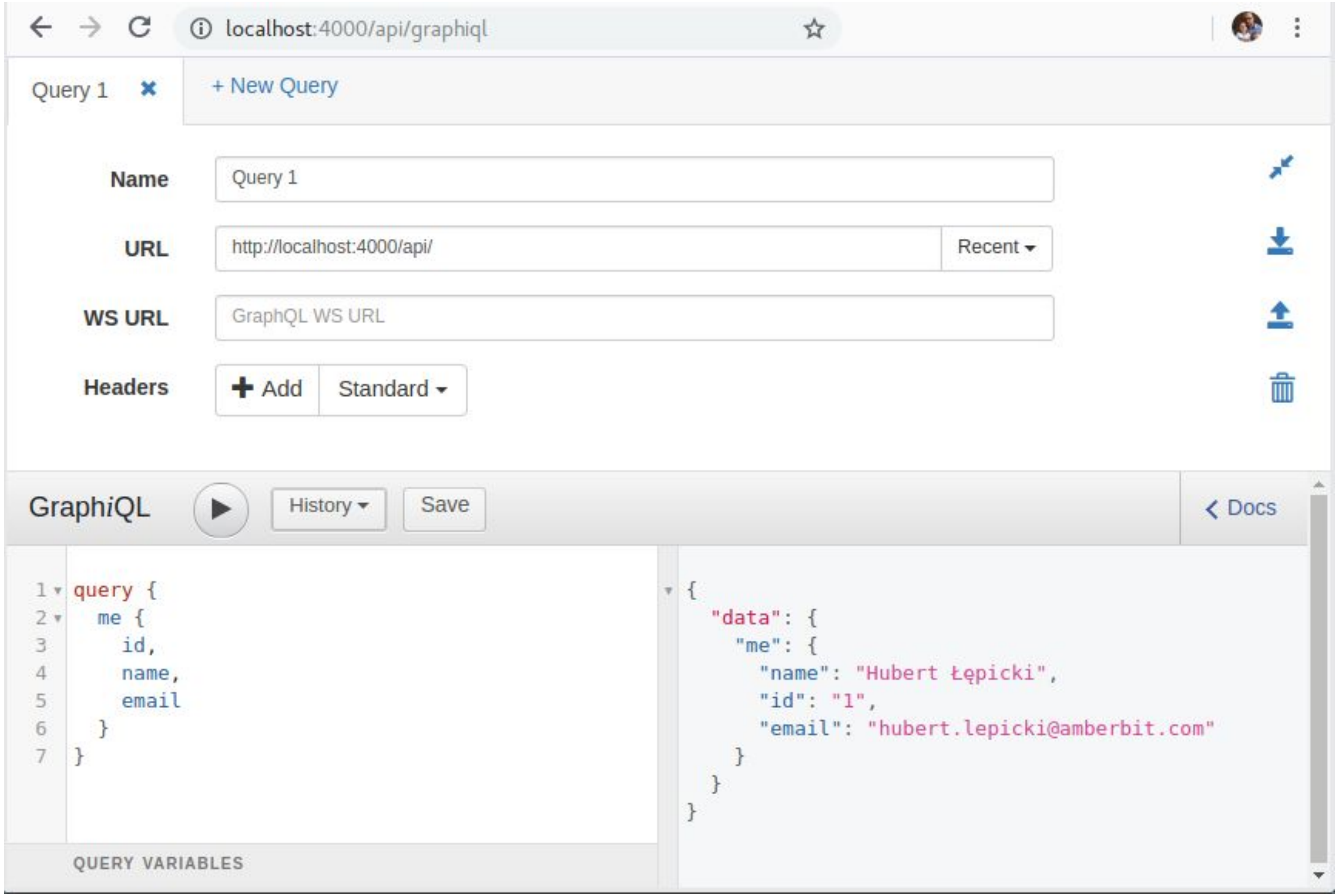
```
{
  "data": {
    "me": null
  }
}
```

lib/my_app/schema.ex

```
  ...
  query do
    field :me, :me do
      resolve fn _parent, _args, _resolution ->
        {:ok, %{id: 1, name: “Hubert Łępicki”,
                email: “hubert.lepicki@amberbit.com”,
                avatar_url: “http://example.com/hub.png”}}
      end
    end
  end
end
```

lib/my_app/schema.ex

```
  ...
  query do
    field :me, :me do
      resolve fn _parent, _args, _resolution ->
        {:ok, Repo.get(User, resolution.context.user_id)}
      end
    end
  end
end
```



Name: Query 1

URL: http://localhost:4000/api/

WS URL: GraphQL WS URL

Headers

```
query {
  me {
    id,
    name,
    email
  }
}
```

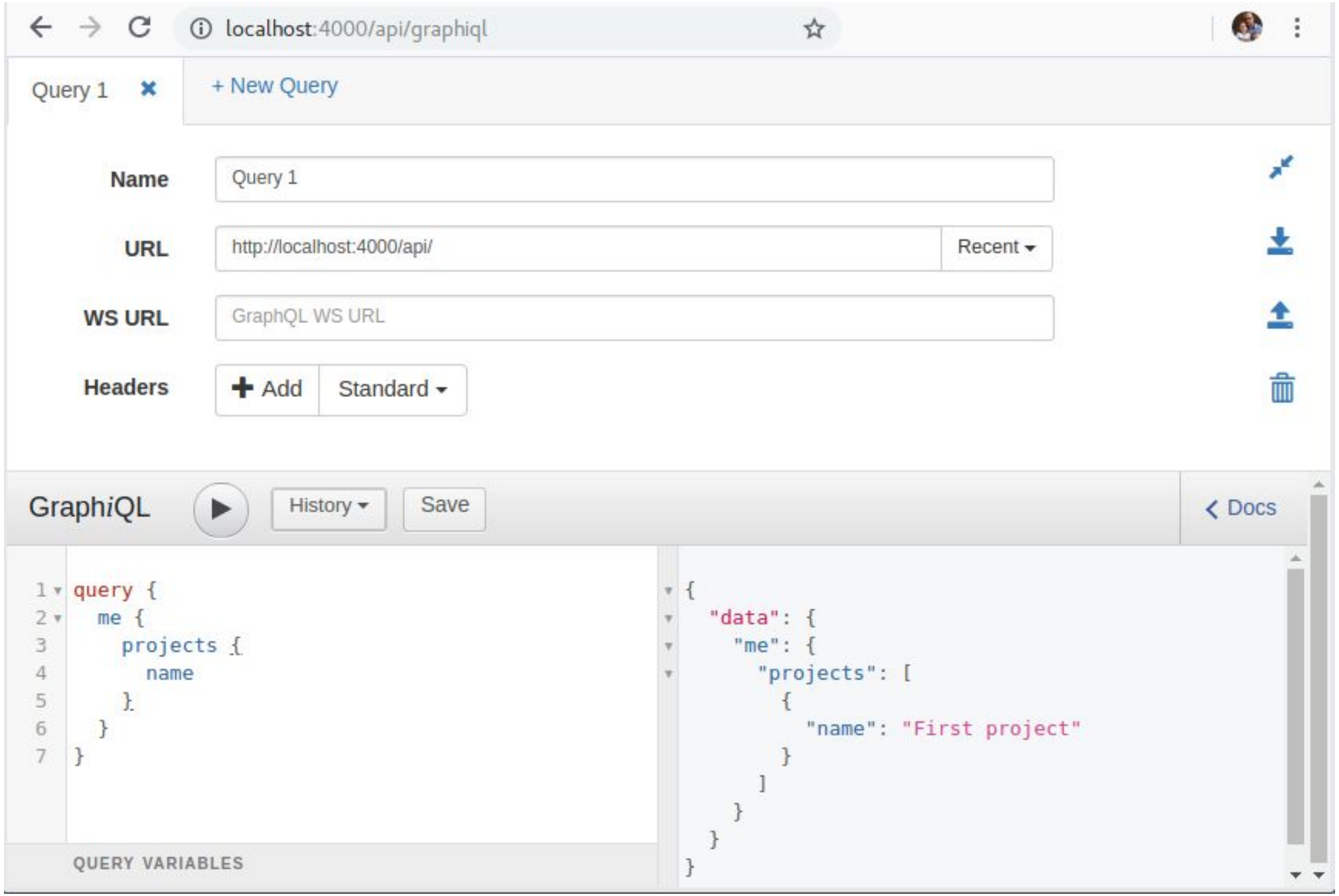
```
{
  "data": {
    "me": {
      "name": "Hubert Łępicki",
      "id": "1",
      "email": "hubert.lepicki@amberbit.com"
    }
  }
}
```

lib/my_app/schema.ex

```
...
  field :me, :me do
    resolve fn _parent, _args, _resolution ->
      {:ok, %{id: 1, name: “Hubert Łępicki”,
              email: “hubert.lepicki@amberbit.com”,
              avatar_url: “http://example.com/hub.png”,
              projects: [%{
                id: 1, name: “First project”,
                tasks: [%{id: 1, name: “First task”}]

              }]}}
...
```

lib/my_app/schema.ex

```
...
object :me do
  ...
  field :projects, list_of(:project) do
    resolve fn _parent, _args, _resolution ->
      {:ok, [
        %{id: 1, name: "First project"}
      ]}
    end
  end
end
...
```



Name: Query 1

URL: http://localhost:4000/api/

WS URL: GraphQL WS URL

Headers

```
query {
  me {
    projects {
      name
    }
  }
}
```

```
{
  "data": {
    "me": {
      "projects": [
        {
          "name": "First project"
        }
      ]
    }
  }
}
```

lib/my_app/schema.ex

```
...
mutation do
  field :create_project, type: :project do
    arg :name, non_null(:string)

    resolve &Resolvers.Projects.create/3
  end
end
...
```

Phoenix integration

```
@graphql """
  query Index @action(mode: INTERNAL) {
    me @put {
      projects
    }
}
"""
def index(conn, result) do
    render(conn, "index.html", projects: result.data.projects)
end
```

Problem #1: N+1 queries

- can be reduced with smart schema design
- cannot be avoided
- can use `batch` with custom Project.by_ids function
- can use Dataloader with Project, Task etc. as sources
- ^^^ generate SQL IN(...) queries. One query per level.
- can also preload data yourself in top-level resolvers
- look ahead into `resolution.path` to see what’s been requested
- use Ecto join + preload to load up data in single query

Problem #2: We’re building DOS endpoint

- Denial of Service
- easy to craft queries that will attempt to load a lot of data
- if you have loops in your schema, you are vulnerable

```
query {
 me {
   projects{
     name,
     users {
       email,
       projects {
         name,
         users {
           ...
```

Problem #2: DOS prevention

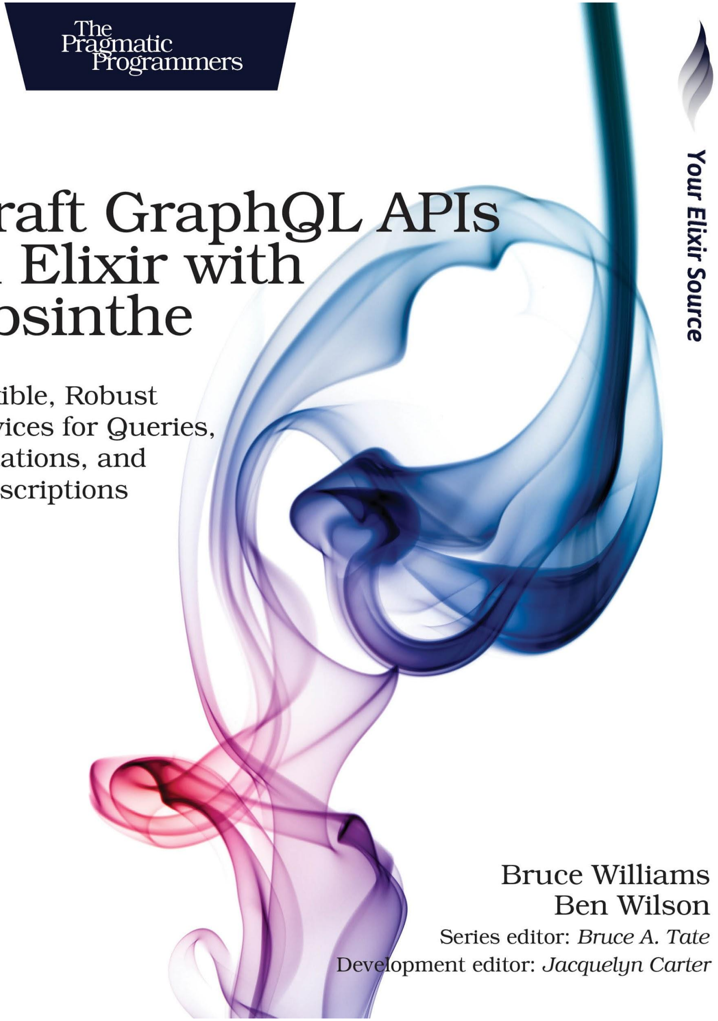
- absinthe has built-in query complexity analysis phase
- give each field / edge complexity
- sums up complexity of overall query
- disallow queries with complexity > MAX_COMPLEXITY
- timeouts & memory limits on resolver processes

Problem #3: Caching

- all queries go to POST /api
- HTTP caching is easier with GET requests
- client-side caching is easy (Apollo!) - need to provide & ask for IDs
- server-side caching blow HTTP layer (in-app)
- use Automatic Persisted Queries (APQ), sent via GET

Problem #4: Hostile developer environments

- JavaScript is bad but could be worse
- Apollo is actually super awesome
- Absinthe is equally super awesome
- Not everyone is so lucky
- Poor/incomplete/outdated implementations are common
- Good Elixir GraphQL *client*?
- Non-dynamic languages often require code generation (sigh)
- Mobile app developers usually hate GraphQL (because of above)
- Can use your GraphQL queries to build REST API if required (sigh)



Questions?

Thanks!