Week 1 Recap

Akshaj Kadaveru

You can start coding your websites now!! Project Skeleton is released on Piazza

- **Read the README for instructions on how to use** (don't clone the skeleton!)
- Milestone 2 (Minimal viable product) due Tuesday Jan 18, 2022 (6pm)
- **Hackathon** Friday! Essentially extended office hours from 7pm-1am, we help you with making your site
    - Start asap so that you can get help at office hours on Wednesday / so that you will have better questions to ask during the hackathon

Office Hours tonight

- Monday and Wednesday 7-9pm in 32-082 or on weblab.to/q
- be there!!

Review Resource http://weblab.to/week1

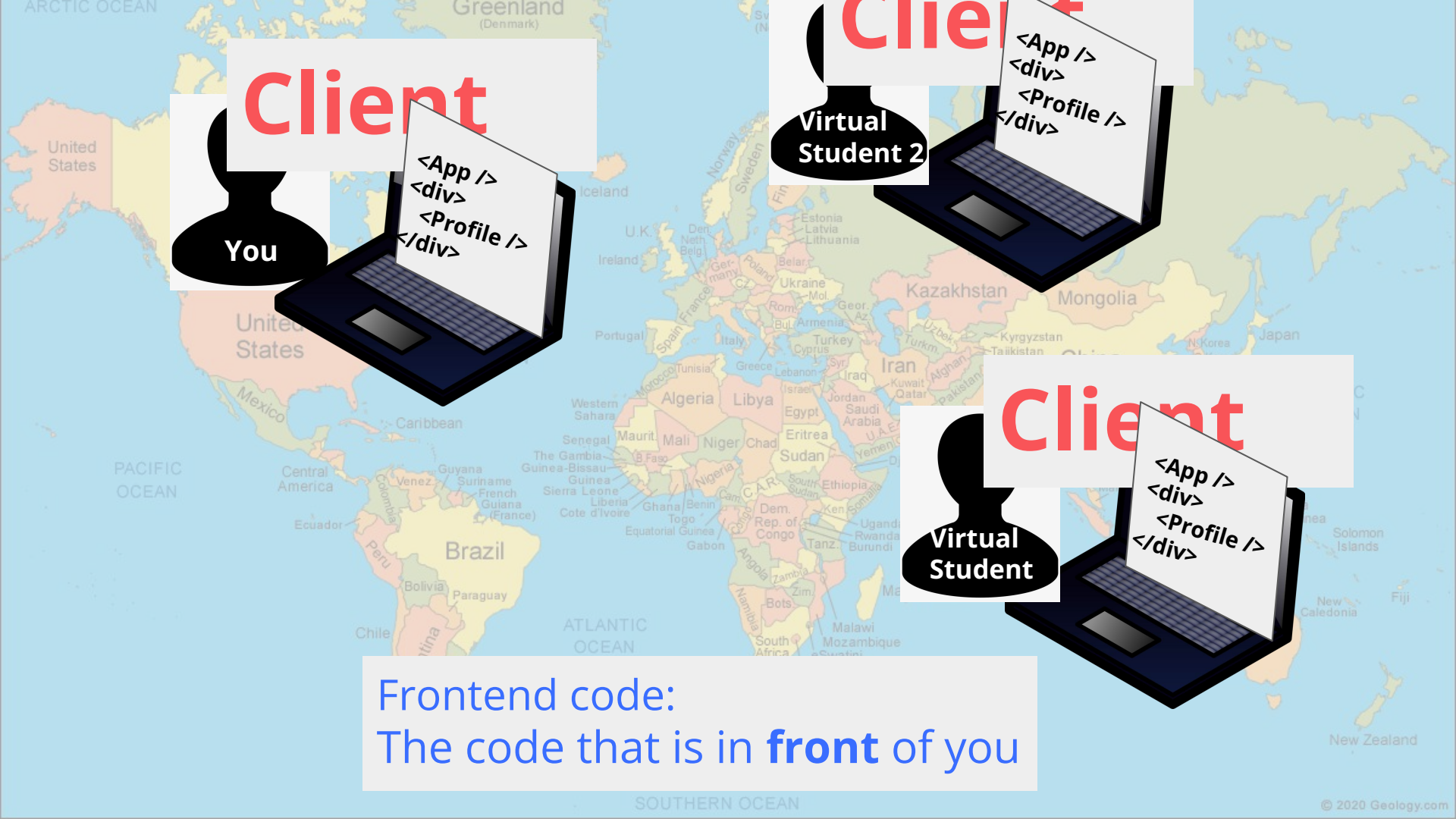
Front End:

- React Study Guide
- Workshop 2 for a simple React app
- Workshop 3 for a more complex app, including how to use ***get*** and ***post*** requests to communicate with the backend
- To-do List React practice

Back End:

- Workshop 5 for Catbook with both a frontend and backend
- Workshop 6 for Catbook with a frontend, backend, and database
- api.js is the important backend file. You don't need to understand the server.js file. Not important.
- The backend is a bunch of descriptions of how to respond to certain get and post requests: Catbook Backend Guide
- Databases are a place to store data, and we will store data on MongoDB Atlas
- A guide to MongoDB/Mongoose

Recap: Frontend (Client) vs Backend (Server)


Client
You
<App />

<Profile />

Client
Virtual Student 2
<App />

<Profile />

Client
Virtual Student
<App />

<Profile />

Frontend code:
The code that is in **front** of you

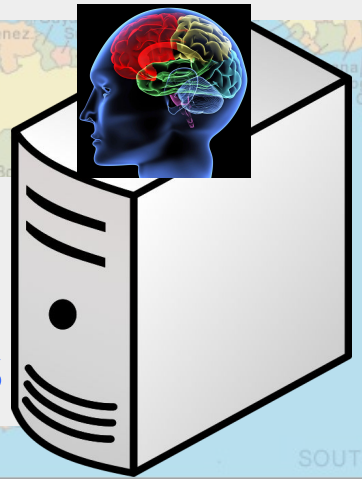
Client

```
get("/api/stories")
```

Client

```
post("/api/story",
  {text: "new story!"})
```

Server!! the central hub



Backend code:
The code that is
behind the scenes

Client

```
get("/api/stories")
```

Front-end (client)

- What the users see
- Every user sees this in their browser
- In front of the user
- Makes your site look pretty / easy to use

Back-end (server)

- Responds to get/post requests from the users
- Runs on the server
- Behind the scenes
- The *core functionality* of your site

Front-end

```
import React from "react";
import NavBar from "./NavBar.js";
import Intro from "./Intro.js";
import Photos from "./Photos.js";

const App = () => {
  return (
    <div>
      <NavBar />
      <div>
        <div>
          <Intro />
          <Photos />
        </div>
        <Post />
      </div>
    </div>
  );
};

export default App;
```

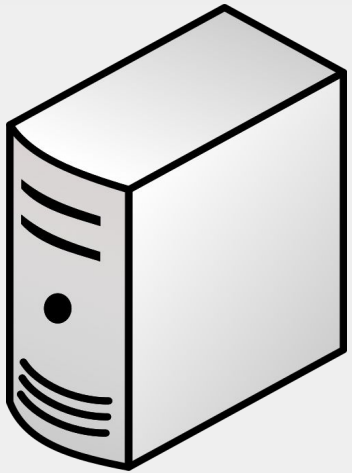
Back-end

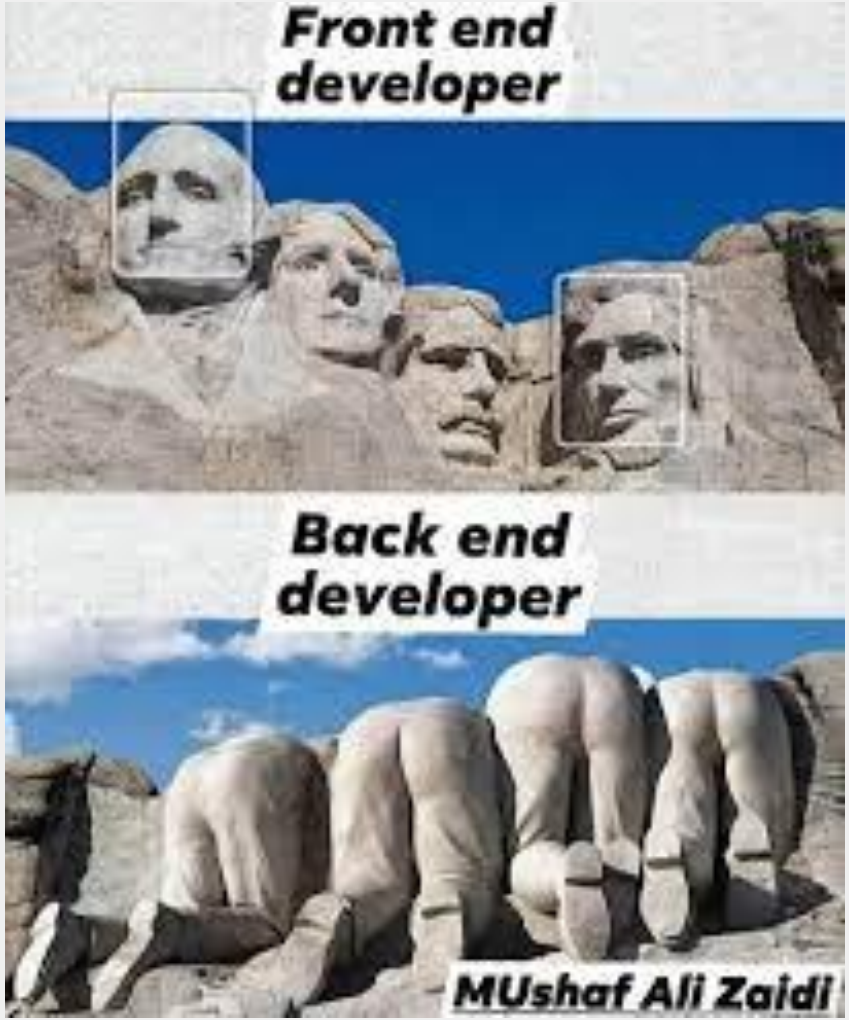
```
router.get("/test", (req, res) => {

});

router.get("/stories", (req, res) => {

});

router.get("/comment", (req, res) => {

});

router.post("/story", (req, res) => {

});

router.post("/comment", (req, res) => {

});
```

Catbook could still exist with no frontend

- We'd have to send our own post requests to the server to add new stories and comments
- We'd have to send our own get requests to the server to read stories and comments
- It **works** but is **bad user experience**

Server

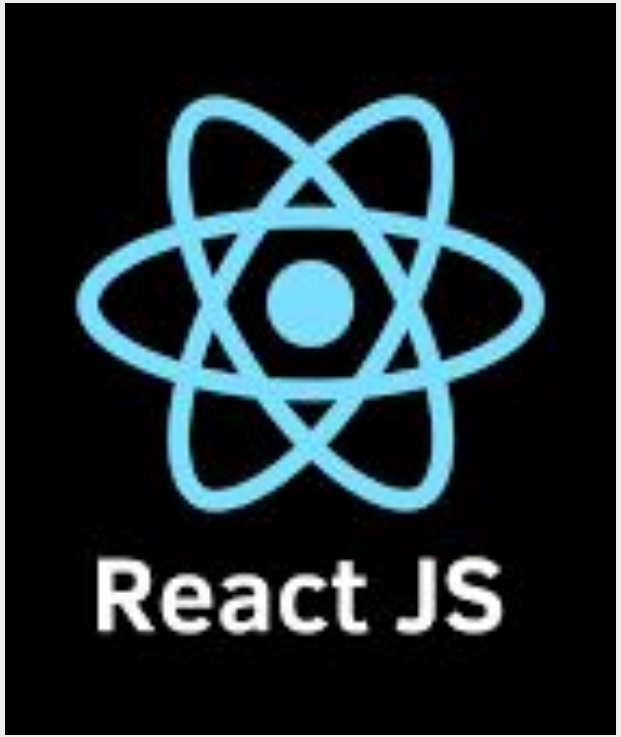



Front end
developer
Back end
developer
MUshaf Ali Zaidi

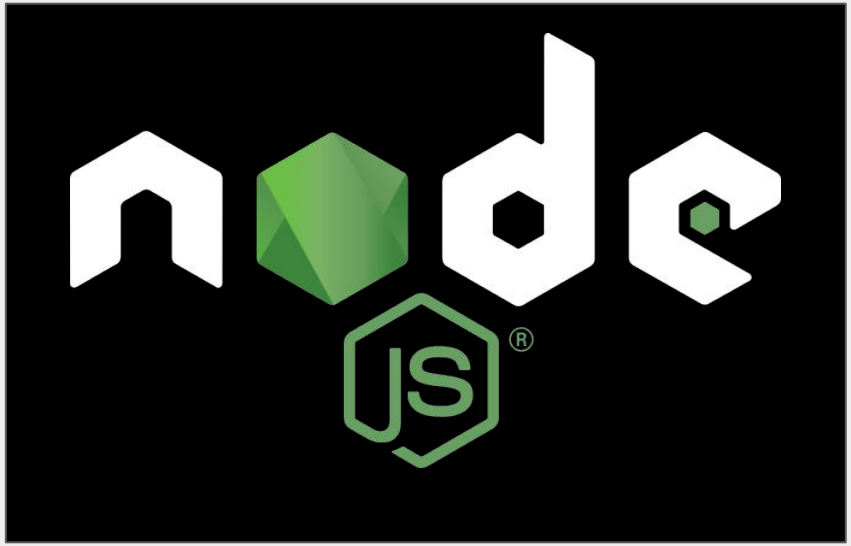
Front-End & Back-End


BACK END DEVELOPER
VIA 9GAG.COM

Front-end



Back-end



Client

Client

get("/api/stories")

find all the stories for me pls!

Server

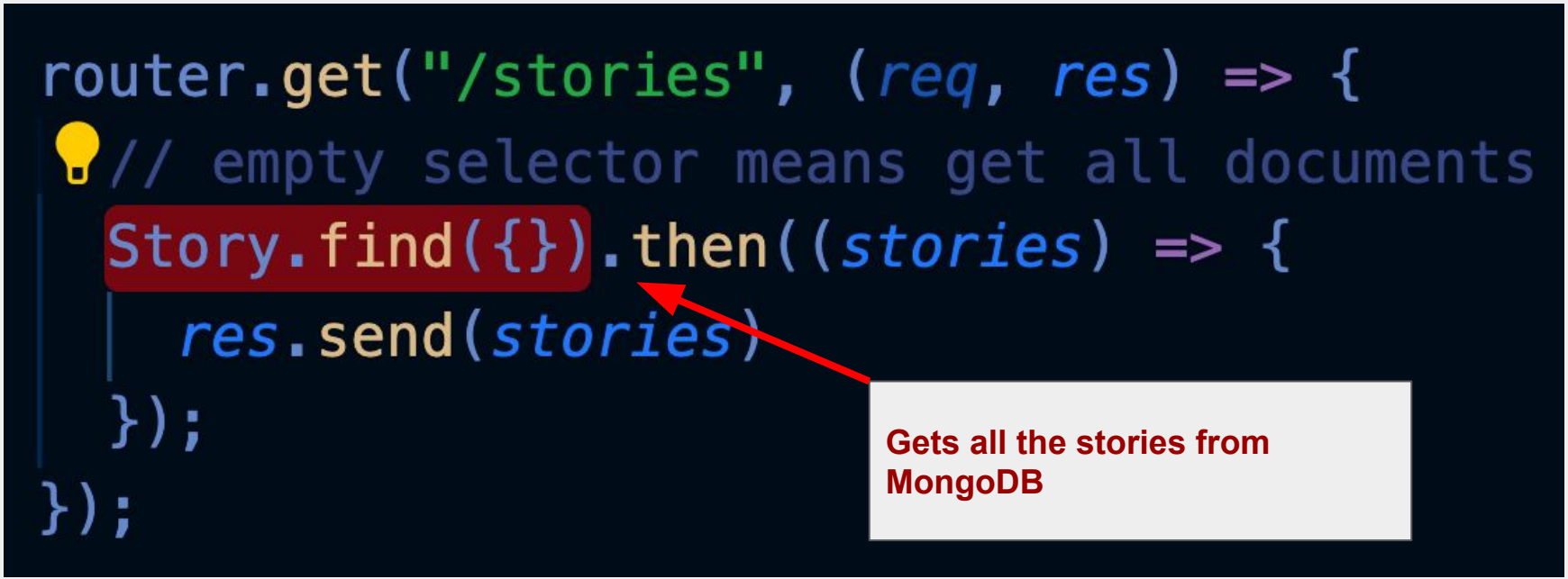
Database

Databases...
Schemas

```
//define a story schema for the database
const StorySchema = new mongoose.Schema({
  creator_name: String,
  content: String,
});
```

```
//define a comment schema for the database
const CommentSchema = new mongoose.Schema({
  creator_name: String,
  parent: String, // links to the _id of a pa
  content: String,
});
```

Databases... Retrieving Data from MongoDB

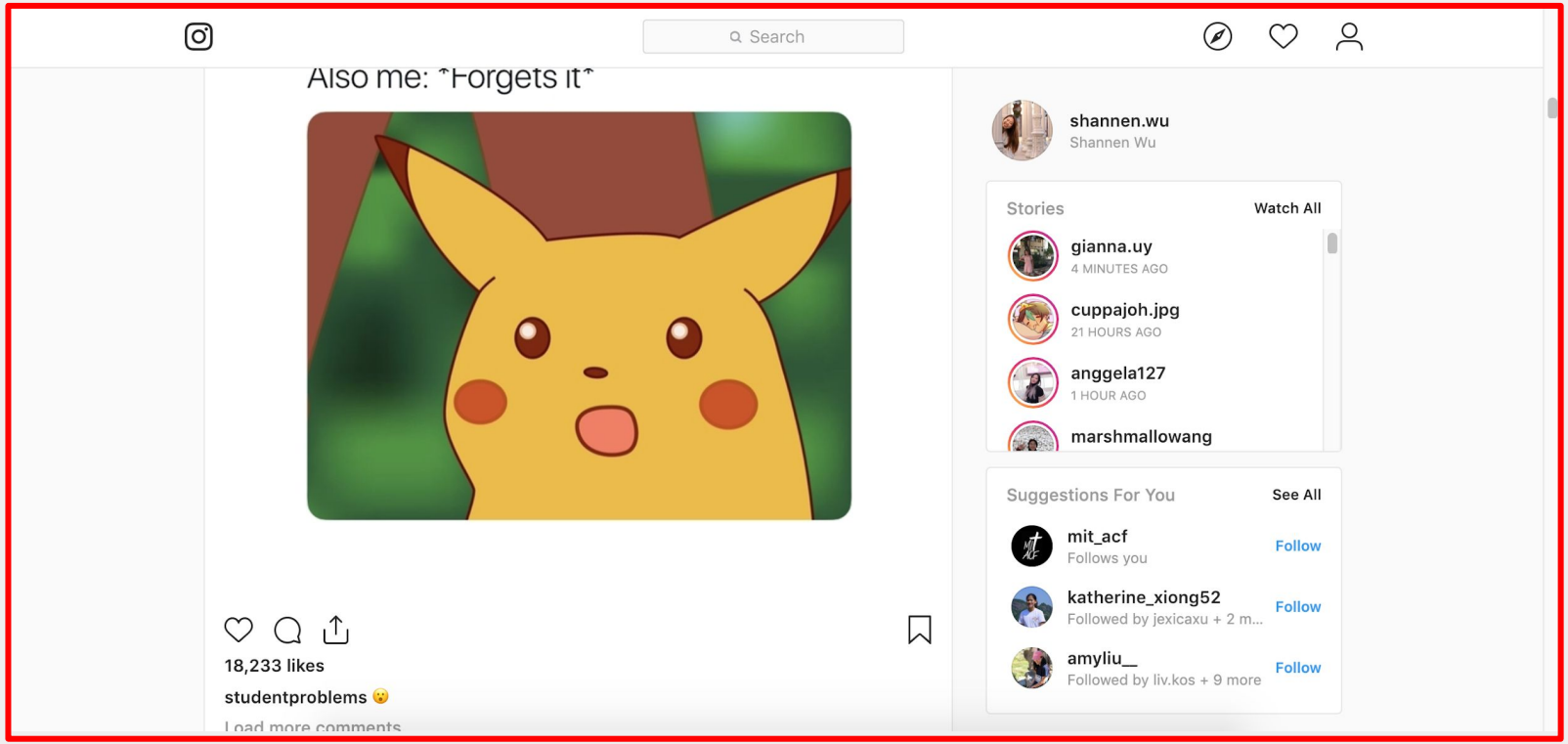


Databases... Adding Data to MongoDB

React

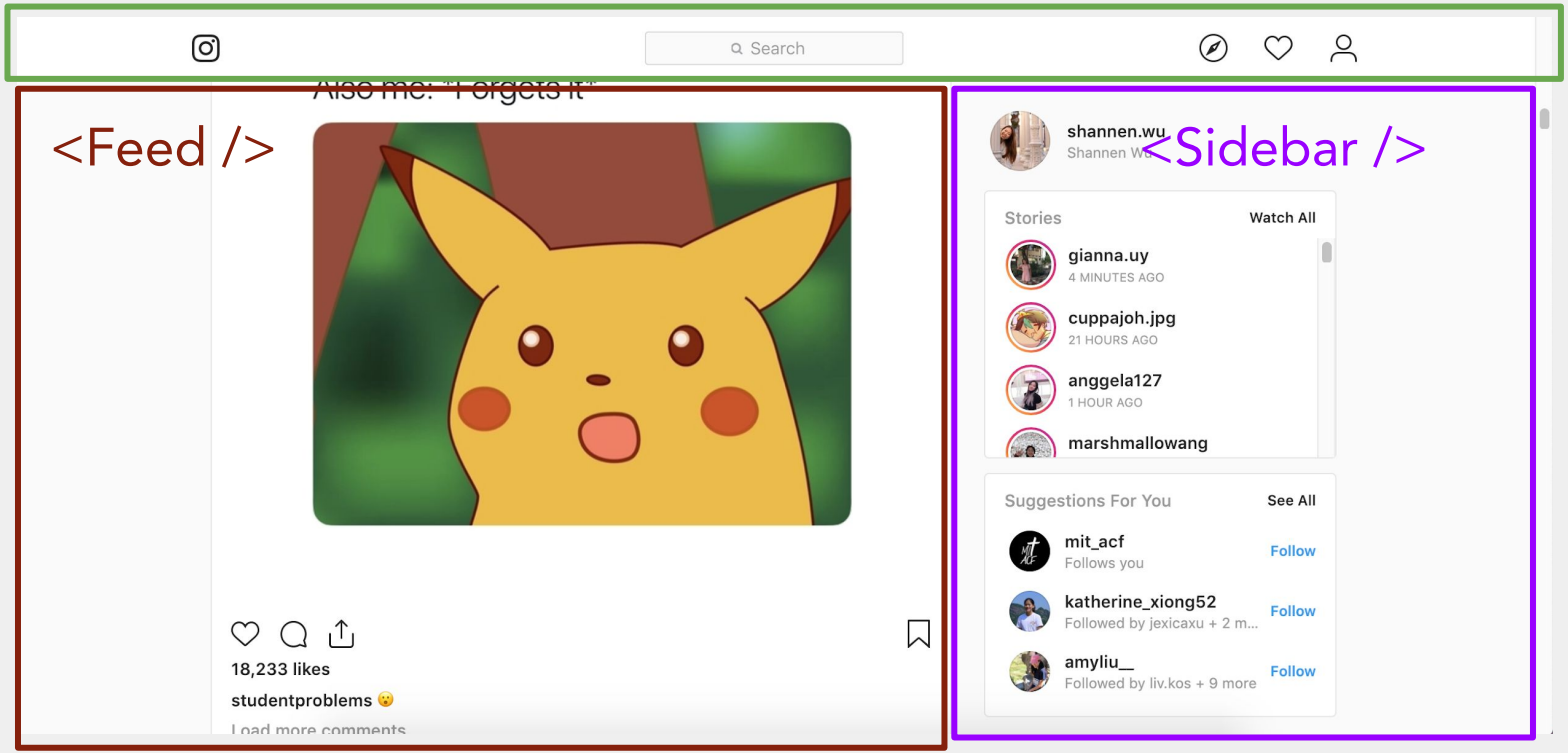
React Apps are "components of components"

The App component (<App />)

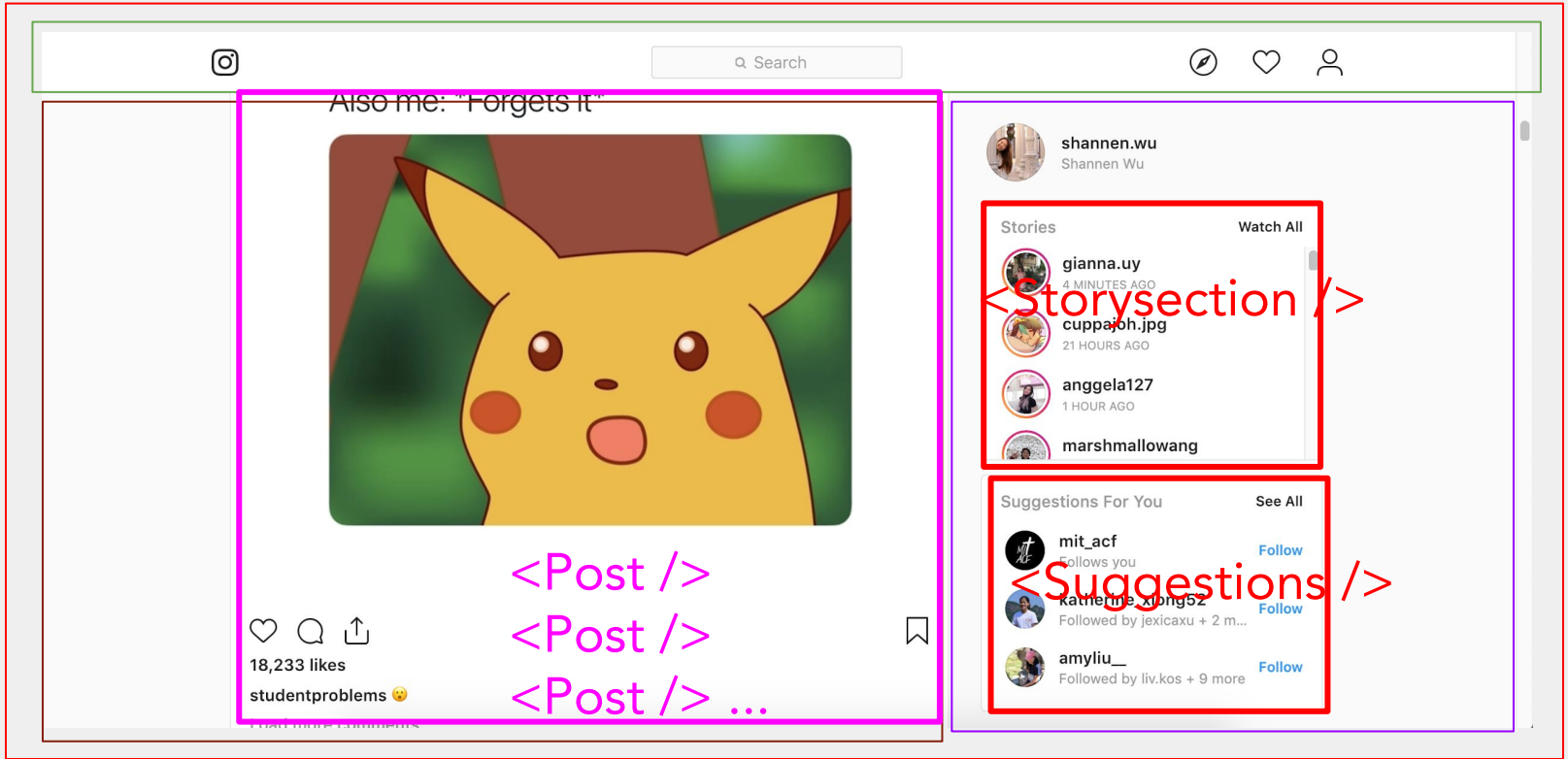


React Apps are "components of components"

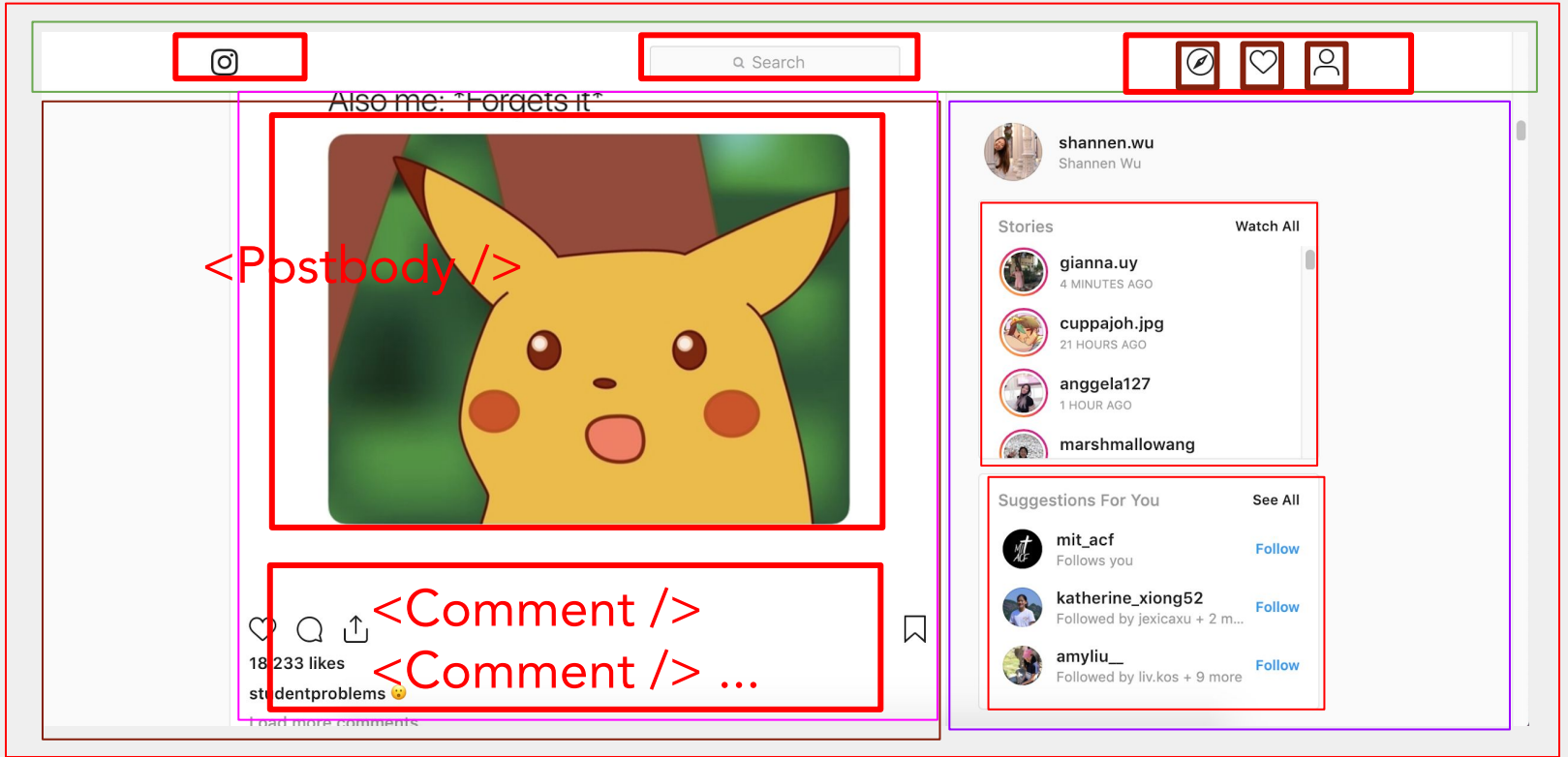
<Navbar />



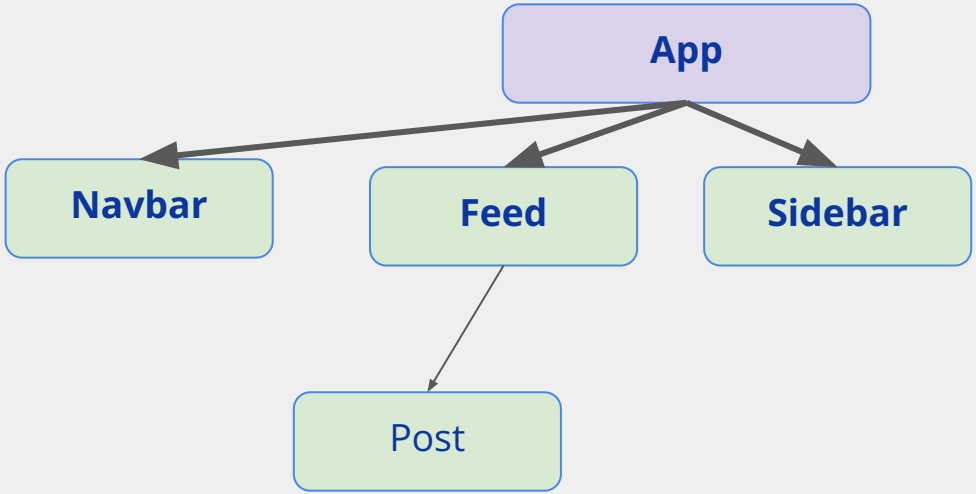
React Apps are "components of components"

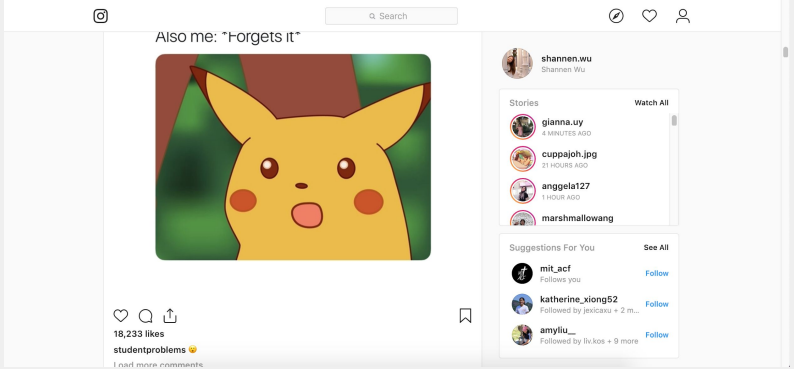


React Apps are "components of components"



Big Picture



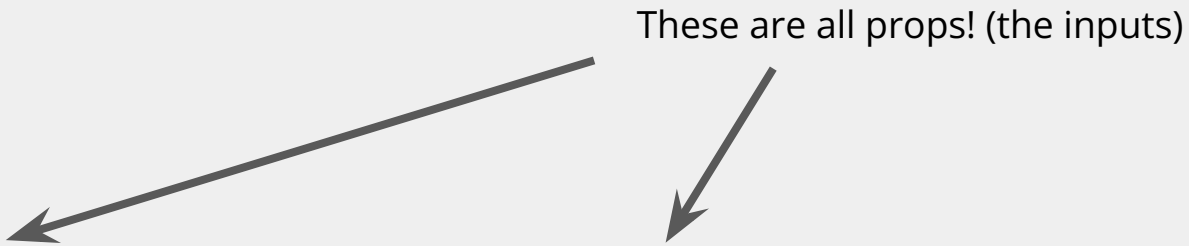


App

```
// App.js

const App = () => {
  return (
    <div>
        <Navbar />
        <div>
            <Feed />
            <Sidebar />
        </div>
    </div>


  )
}
```

Props ('Inputs')

Inputs passed from a parent to a child component




```
<Post name="Akshaj" text="Welcome to web lab!" />
```

here, props = {name: "Akshaj", text: "Welcome to web lab!"}

State

Updatable pieces of information maintained by a component.

```
const [status, setStatus] = useState("busy");
const [isOnline, setIsOnline] = useState(false);
```

What we've learned in React

- Websites are divided into **components**
- components can have inputs, which are **props**
- components can have their own private, updatable information: **state**

React: useEffect and get/post (communicating with server)

Weather app

```
const WeatherApp = () => {
  const [weather, setWeather] = useState("Cloudy");

  return <div>{weather}</div>;
};
```

Why can't we do this

```
const WeatherApp = () => {
  const [weather, setWeather] = useState("Cloudy");

  const actualWeather = get("/api/theweather");
  setWeather(actualWeather);

  return <div>{weather}</div>;
};
```

Why can't we do this

```
const WeatherApp = () => {
  const [weather, setWeather] = useState("Cloudy");

  const actualWeather = get("/api/theweather");
  setWeather(actualWeather);

  return <div>{weather}</div>;
};
```

Asynchronous.. we need to wait for the get request to finish first

Why does this still not work (we get error infinite render??)

```
const WeatherApp = () => {
  const [weather, setWeather] = useState("Cloudy");

  get("/api/theweather").then((actualWeather) => {
    setWeather(actualWeather);
  });

  return <div>{weather}</div>;
};
```

Why does this still not wo e ge infinite render??)

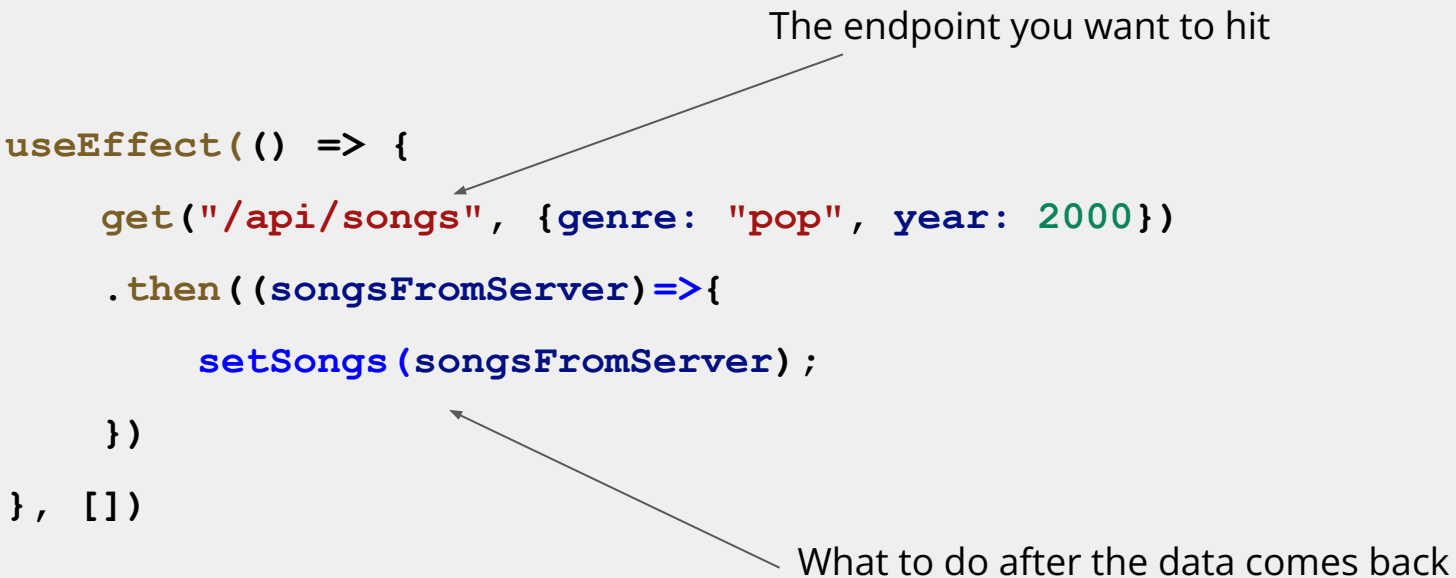
Components re-run their code whenever state or props changes

Solution: useEffect

```
const WeatherApp = () => {
  const [weather, setWeather] = useState("Cloudy");

  useEffect(() => {
    get("/api/theweather").then((actualWeather) => {
      setWeather(actualWeather);
    });
  }, []);

  return <div>{weather}</div>;
};
```

runs only once now!

Talking to the Backend (Server) from the Frontend (Client)

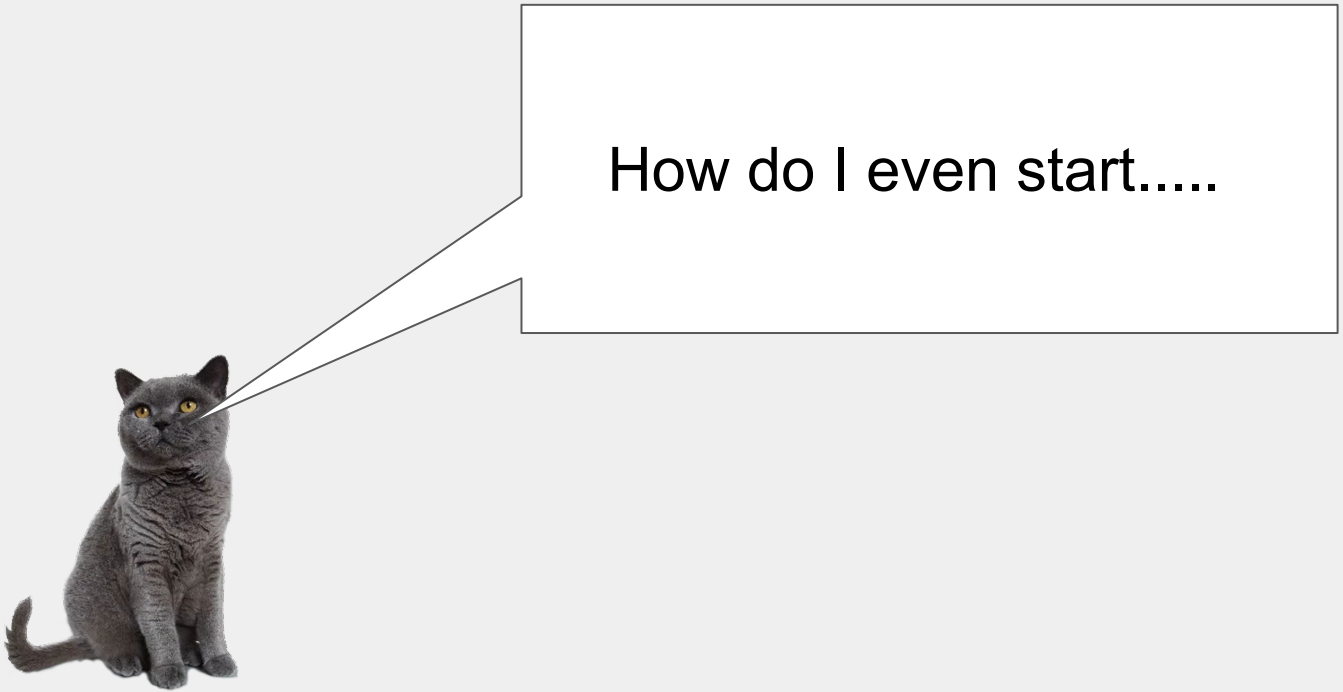


What we've learned in React

- Websites are divided into **components**
- components can have inputs, which are **props**
- components can have their own private, updatable information: **state**
- Components run their code **once at the beginning**, and then **once every time a prop or a state changes**
- To run something only once at the beginning, we use **useEffect**(function, [])
- **get**("/api/something", someData) does a get request to "/api/something", and gets back data (asynchronously)
- **post**("/api/something", someData) does a post request to "/api/something" (asynchronously)
- **Router** is our way of having different 'pages' (with their own URL) on your site

Connecting everything together

Where to start

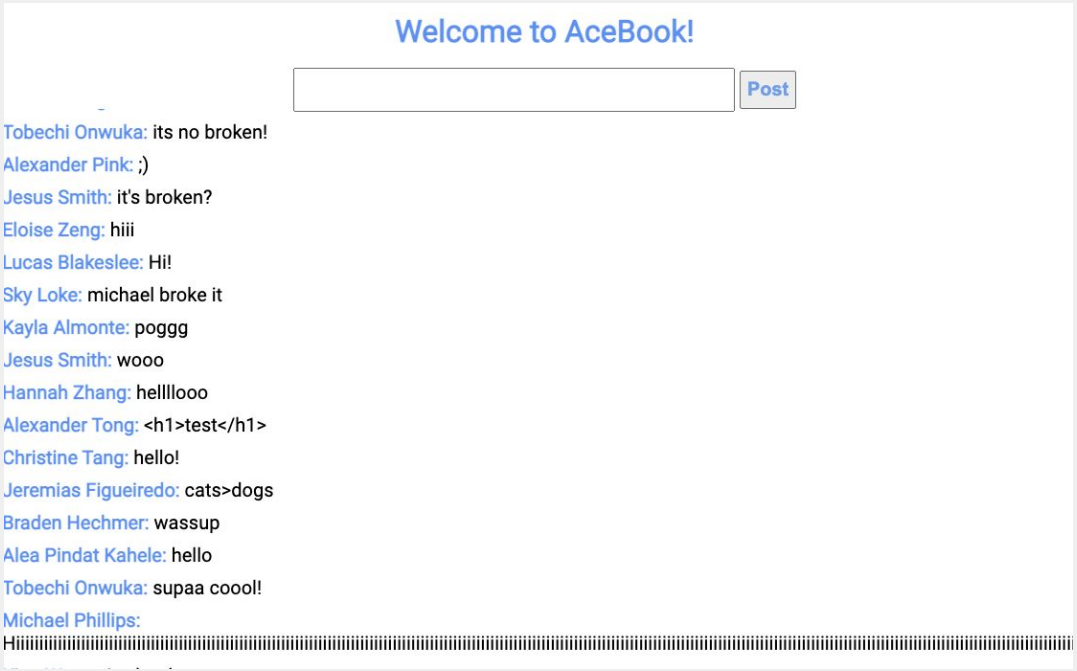


How to start coding up your webapp

1. (Backend) Figure out what you're storing in your database
2. (Backend) Figure out what actions your server will need to be able to do
3. (Frontend) Divide your design into components
4. (Frontend) Figure out the props/state of each component, and where your get/post requests will happen

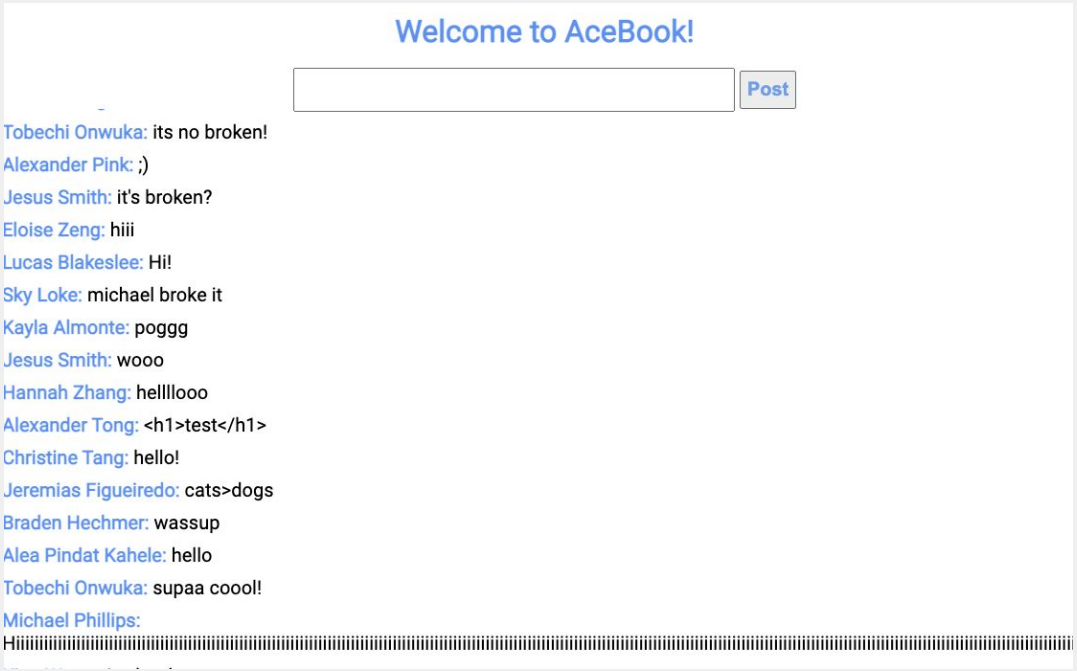
http://weblab.to/recap-app

Step 1: What do we need to store in our database



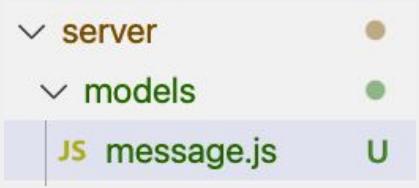
Step 1: What do we need to store in our database

- Messages



Step 1: What do we need to store in our database

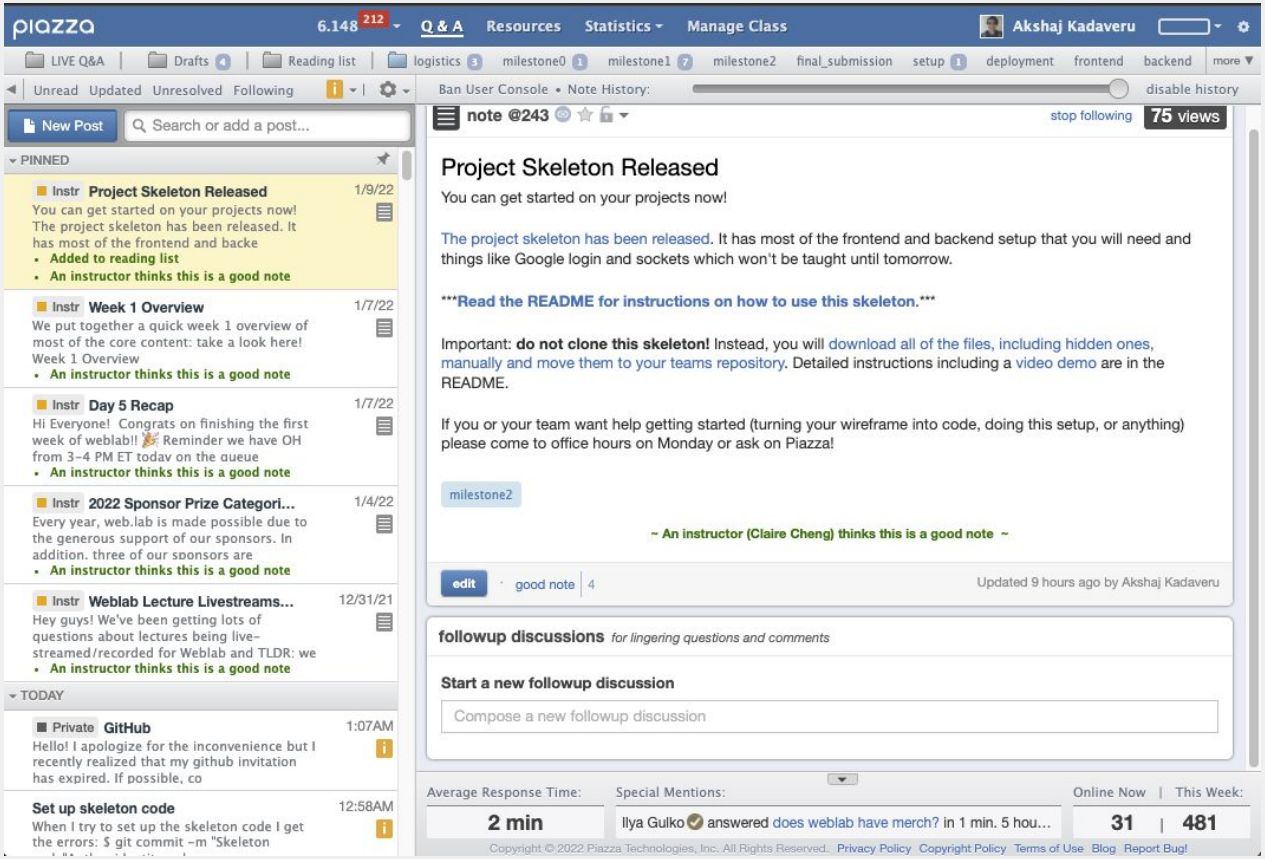
- Add a 'Message' schema



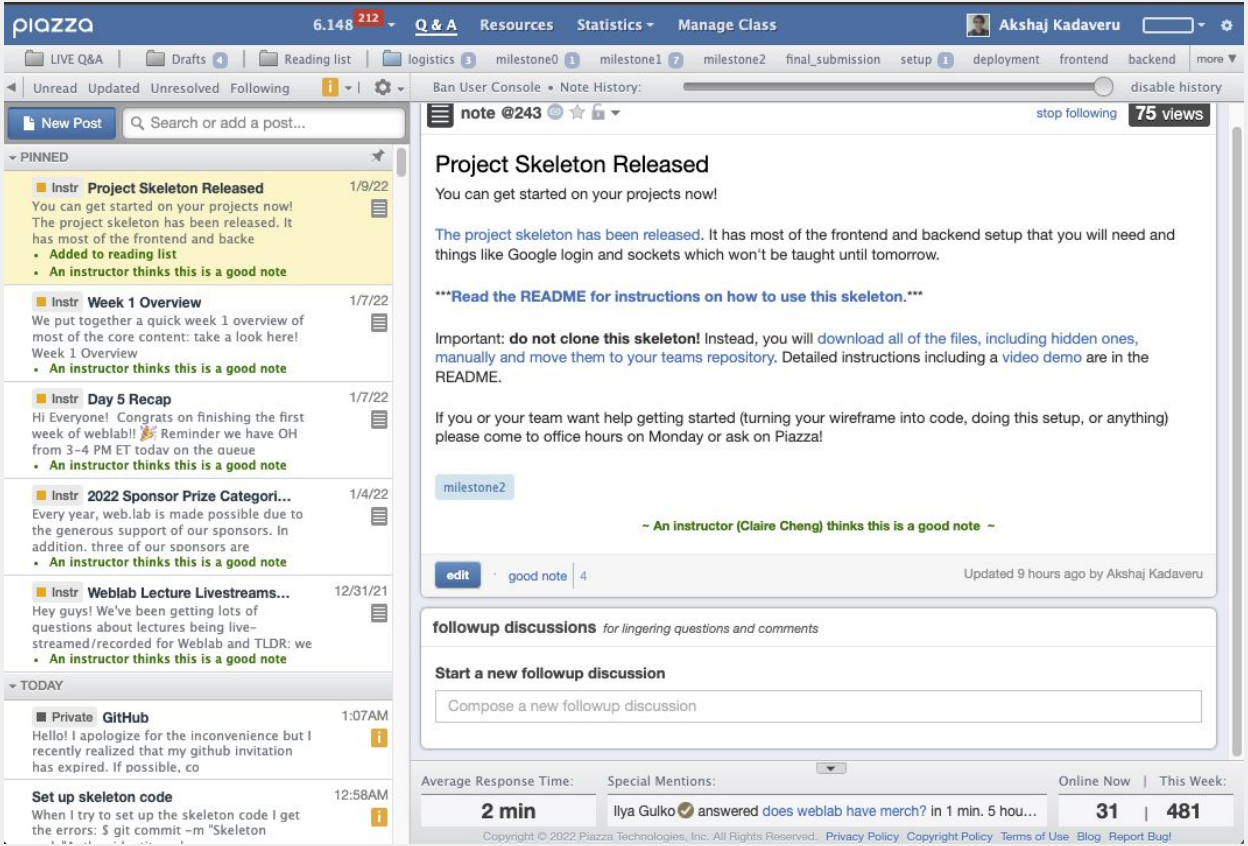
server > models > JS message.js > ...

```
1  const mongoose = require("mongoose");
2
3  const MessageSchema = new mongoose.Schema({
4    name: String,
5    text: String,
6  });
7
8  module.exports = mongoose.model("message", MessageSchema);
```

Step 1 Practice: What Schemas might Piazza store?



Step 1 Practice: What Schemas might Piazza store?



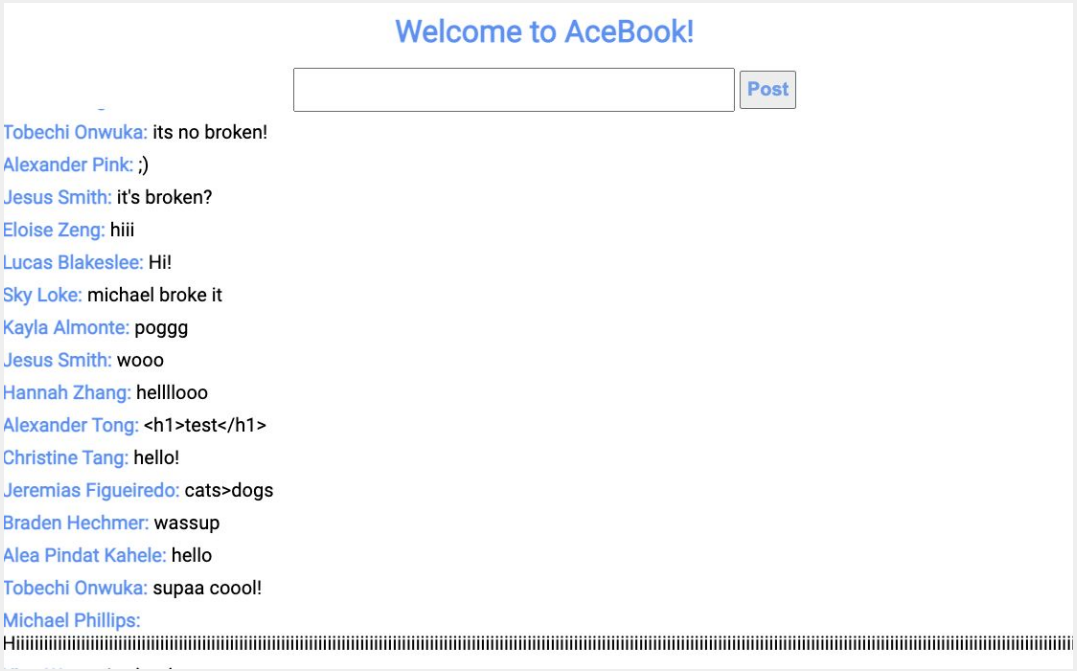
- User
- Class
- Question
- Answer
- School
- Tag
- etc.

Step 1 Practice: What Schemas might Piazza store?

```
const ClassSchema = new mongoose.Schema({
  title: String,
  instructors: [String],
  description: String,
  schoolId: String,
});
```

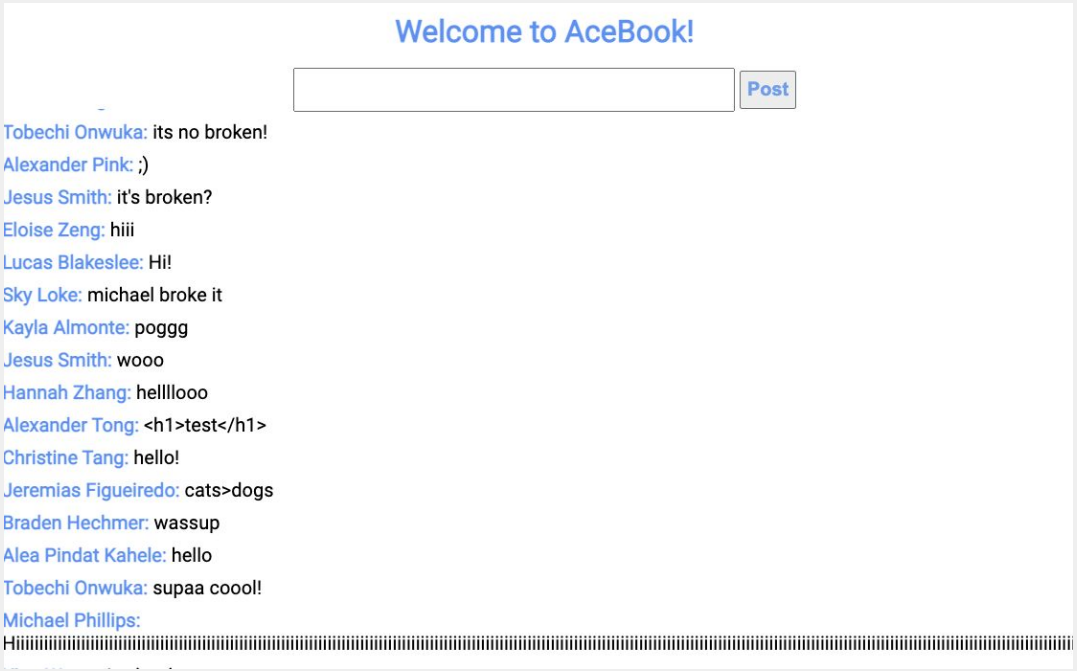
```
const QuestionSchema = new mongoose.Schema({
  questionText: String,
  userId: String,
  classId: String,
  tags: [String],
  time: Date,
});
```

Step 2: What server endpoints will we need?



Step 2: What server endpoints will we need?

- Get all the messages
- Post a new message



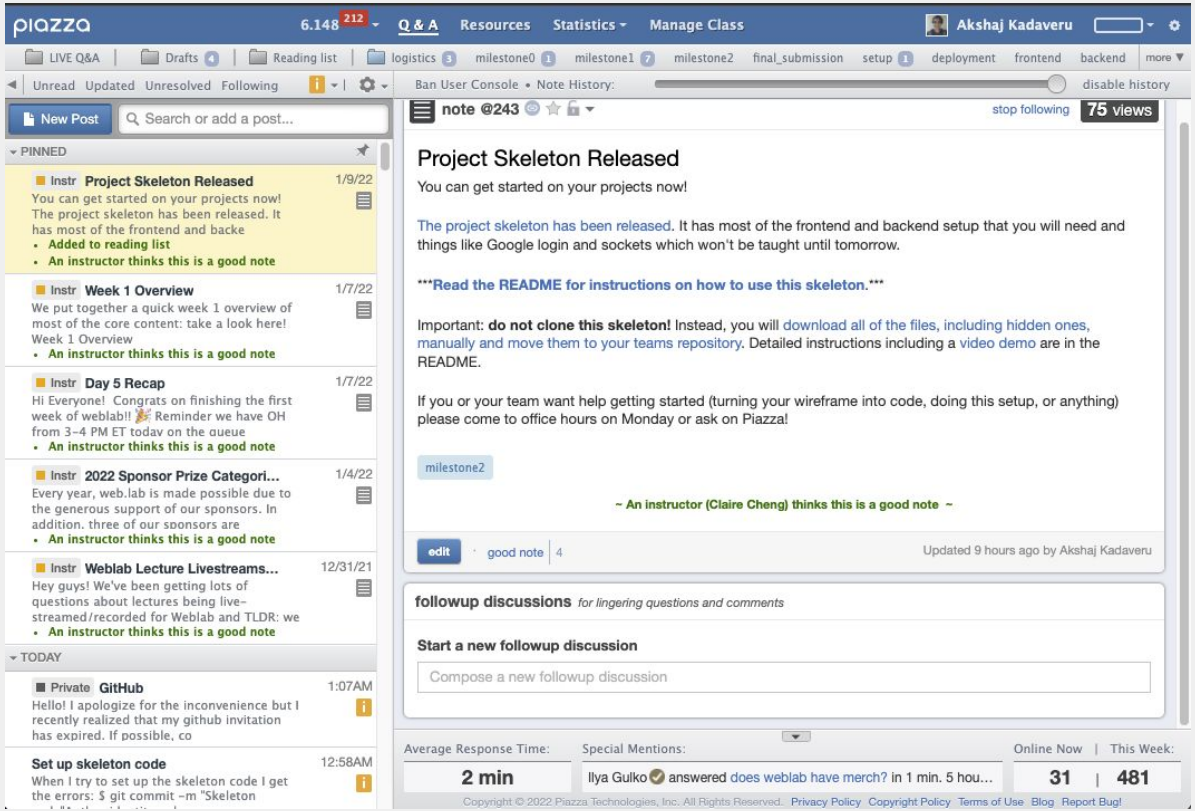
Step 2: What server endpoints will we need?

```
// Get all of the messages
router.get("/allmessages", (req, res) => {


});

// Send a message to everyone
// Needs the message text as an input
router.post("/newmessage", (req, res) => {

});
```

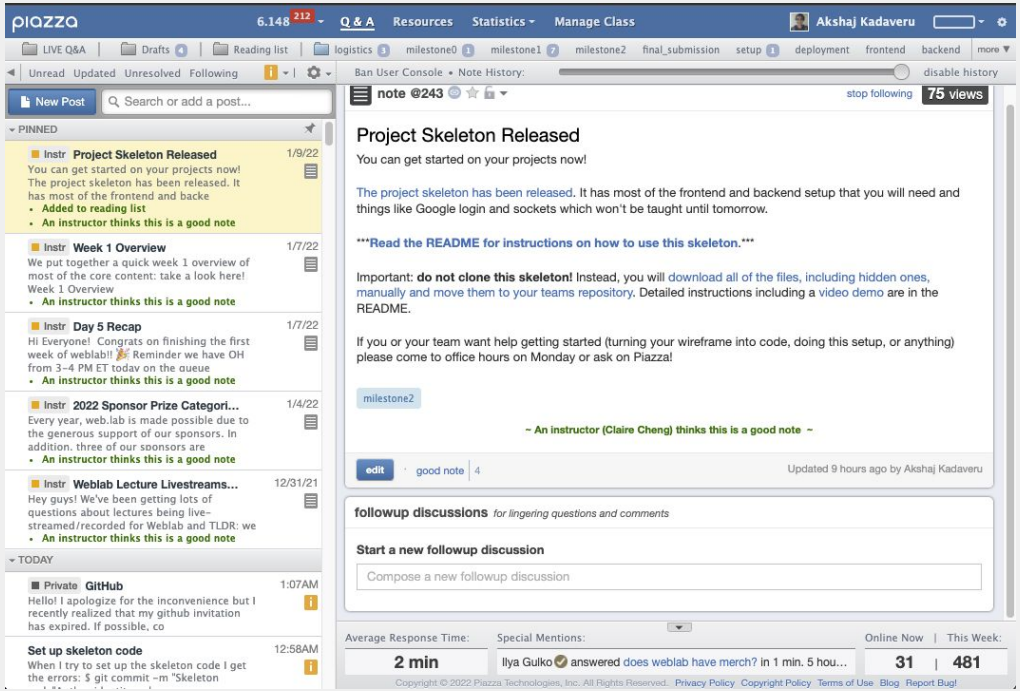
Step 2: What server endpoints will we need?

```
router.get("/allmessages", (req, res) => {
  Message.find({}).then((messagesFromDB) => {
    res.send(messagesFromDB);
  });
});

router.post("/newmessage", (req, res) => {
  const newMessage = new Message({
    name: "Anonymous",
    text: req.body.text,
  });
  newMessage.save();
});
```

Step 2 Practice: What server endpoints will we need?



Step 2 Practice: What server endpoints will we need?



- Add new question
- Edit question
- Delete question
- Add answer
- edit answer
- add user to class
- add new class
- add tag to question
- remove tag from question
- get all the questions for a class
- get all the answers for a question

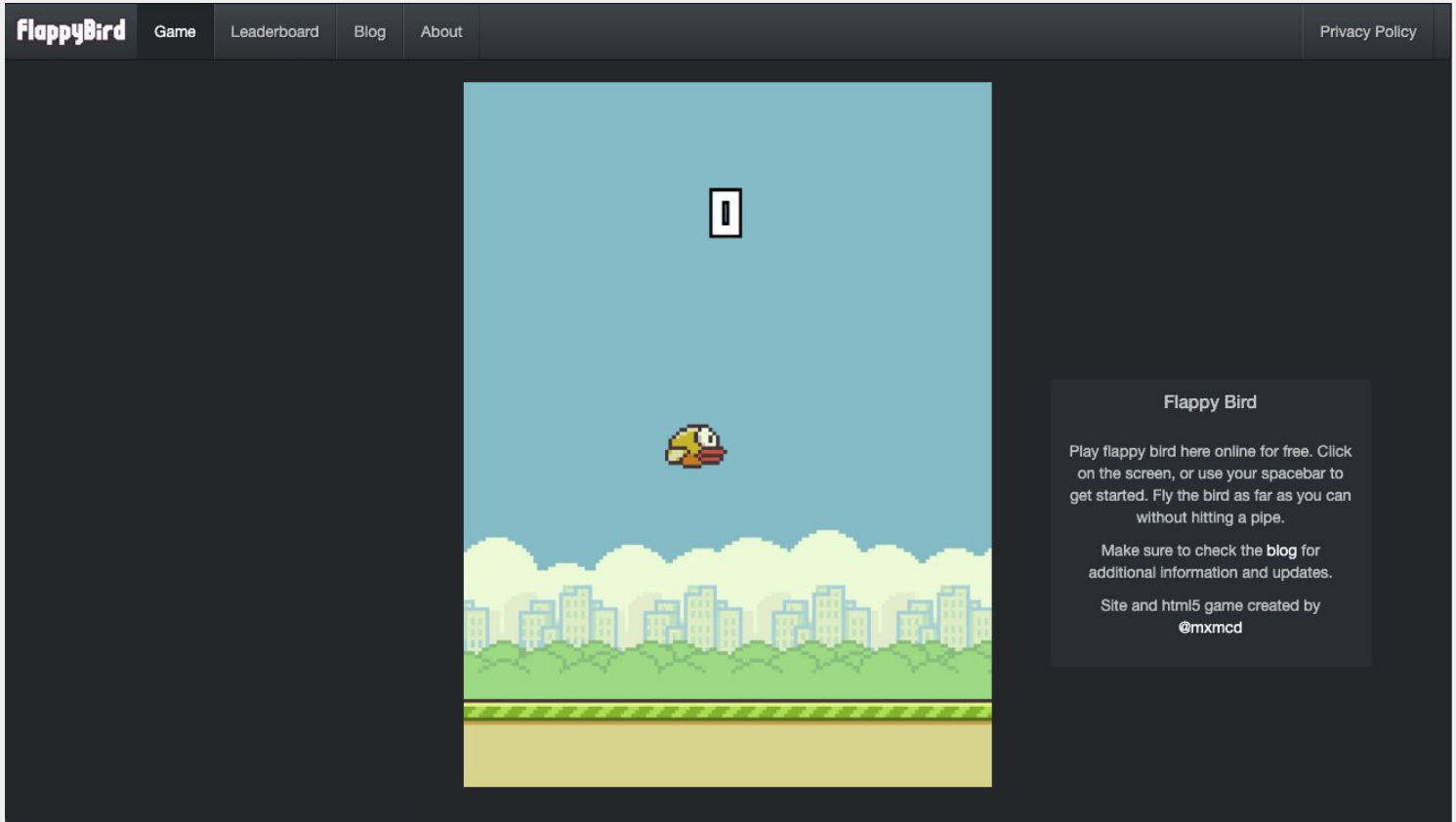
Step 2 Practice: What server endpoints will we need?

```
router.post("/addnewquestion", (req, res) => {

});

router.post("/addnewusertoclass", (req, res) => {

})

router.get("/allanswersforquestion", (req, res) => {

})
```

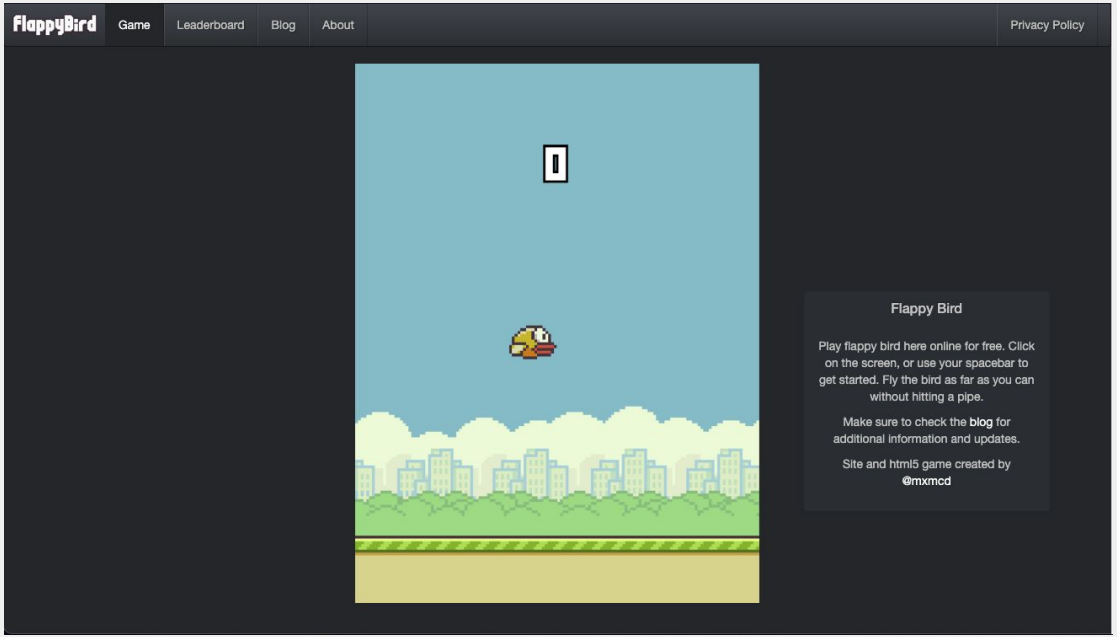
Step 2 Practice: What server endpoints will we need?

```
router.post("/addnewquestion", (req, res) => {
    const newQuestion = new Question({
      questionText: req.body.text,
      classId: req.body.classId
    })
    newQuestion.save();
});

router.get("/allanswersforquestion", (req, res) => {
  const allAnswers = Answer.find({questionId: req.query.questionId}).then((answers) => {
    res.send(answers)
  })
})
```

Step 2 Practice: What server endpoints will we need?



Step 2 Practice: What server endpoints will we need?

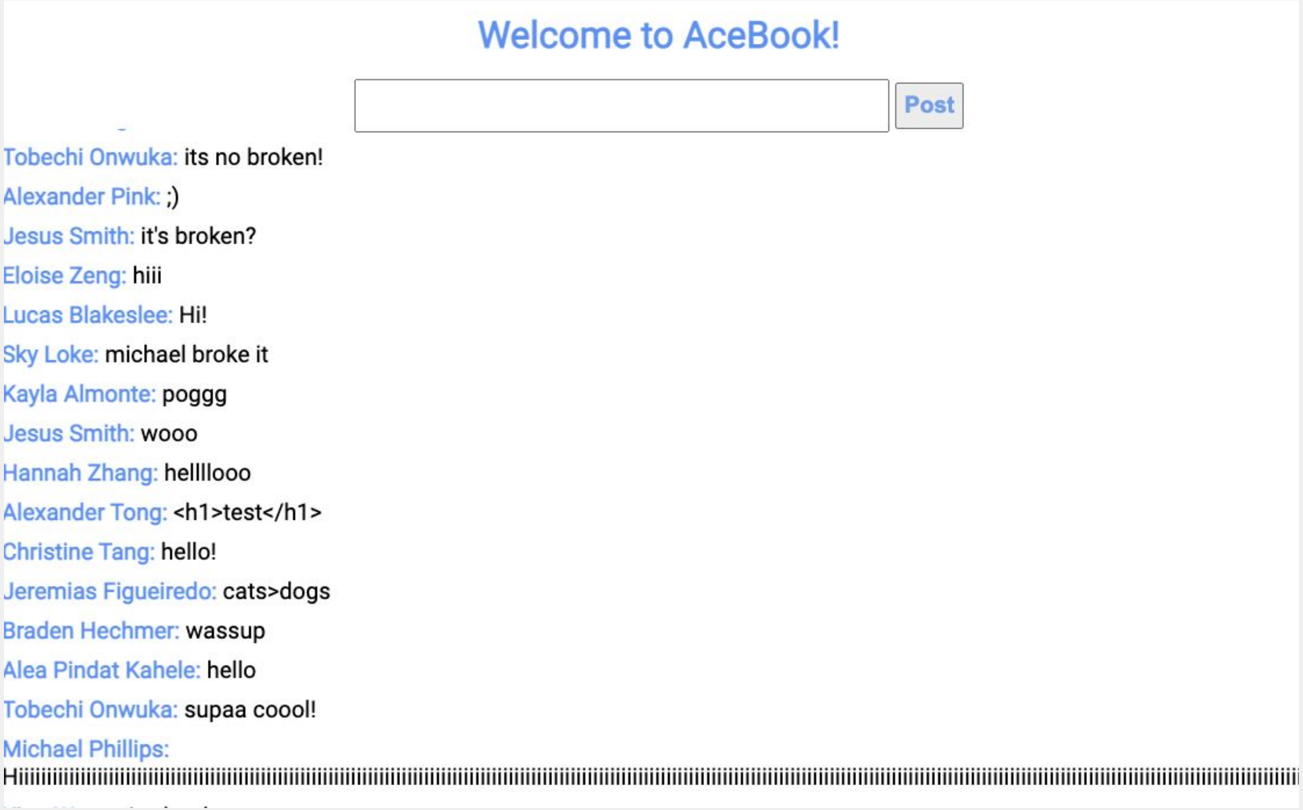


- Add to leaderboard
- Get top ten scores in the last hour
- Get top 40 scores in the last 24 hours

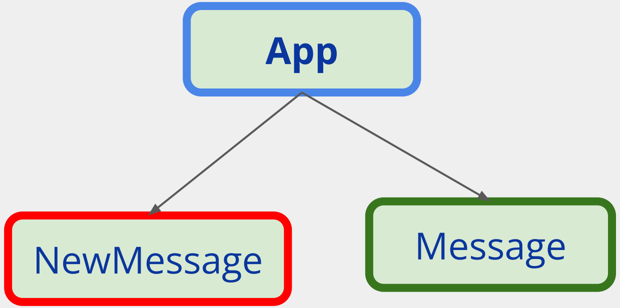
Step 2 Practice: What server endpoints will we need?

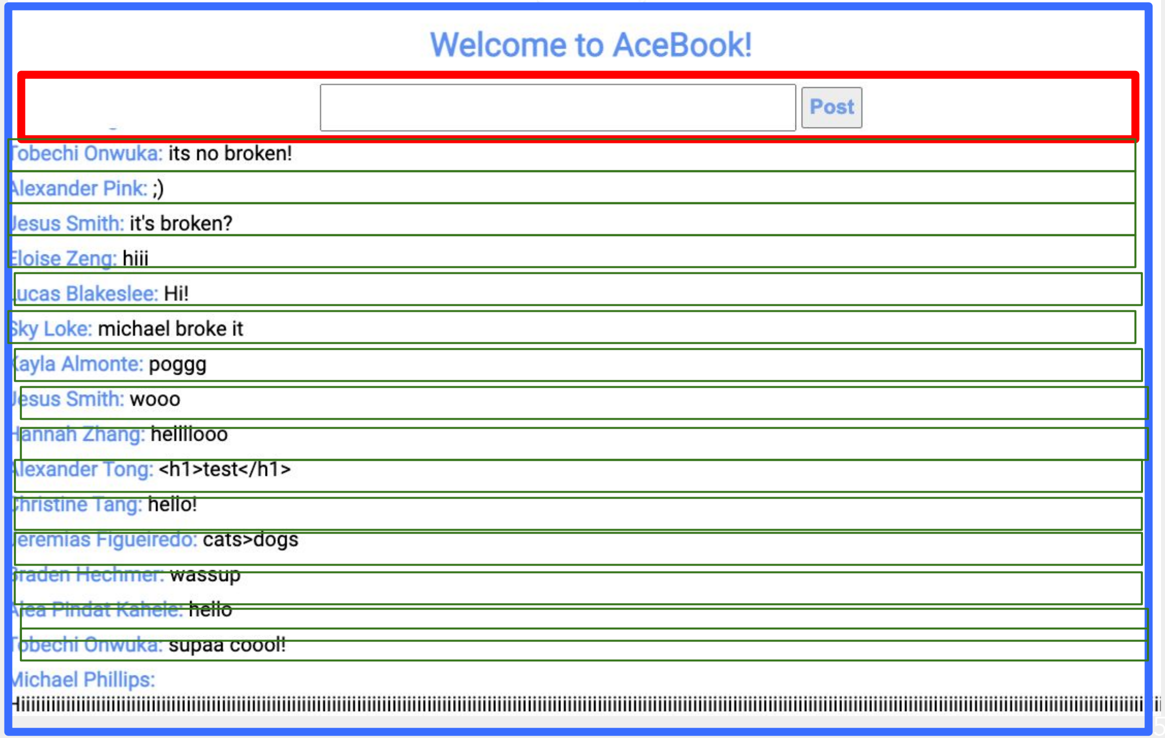
```
router.post("/addscoretoleaderboard", (req, res) => {
    const newLeaderboardEntry = new LeaderboardEntry({
      timestamp: Date.now(),
      score: req.body.score,
      name: req.body.name
    })
    newLeaderboardEntry.save();
});

router.get("/toptenscoresinlasthour", (req, res) => {
  LeaderboardEntry.find({}).then((entries) => {
    // filter entries for top 10 scores in last hour
    return filteredEntries;
  })
})
```

Step 3: What are the components?

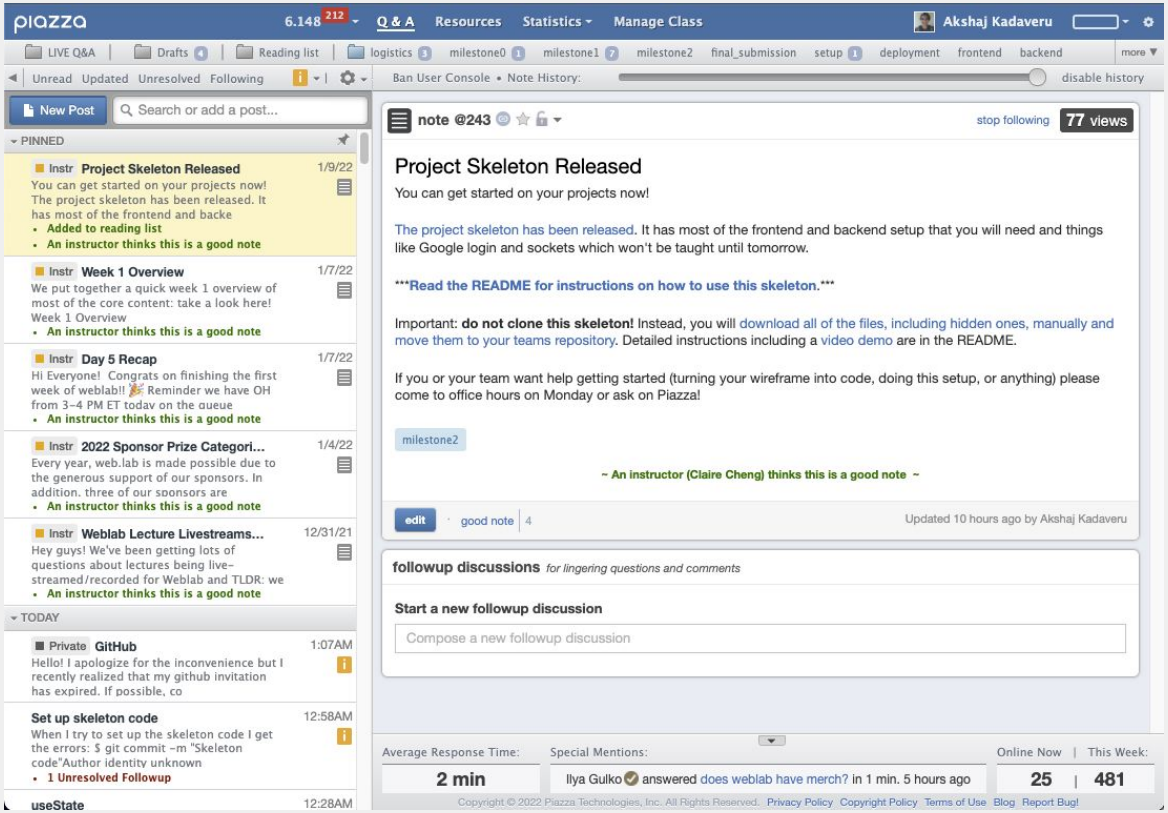


Step 3: What are the components?

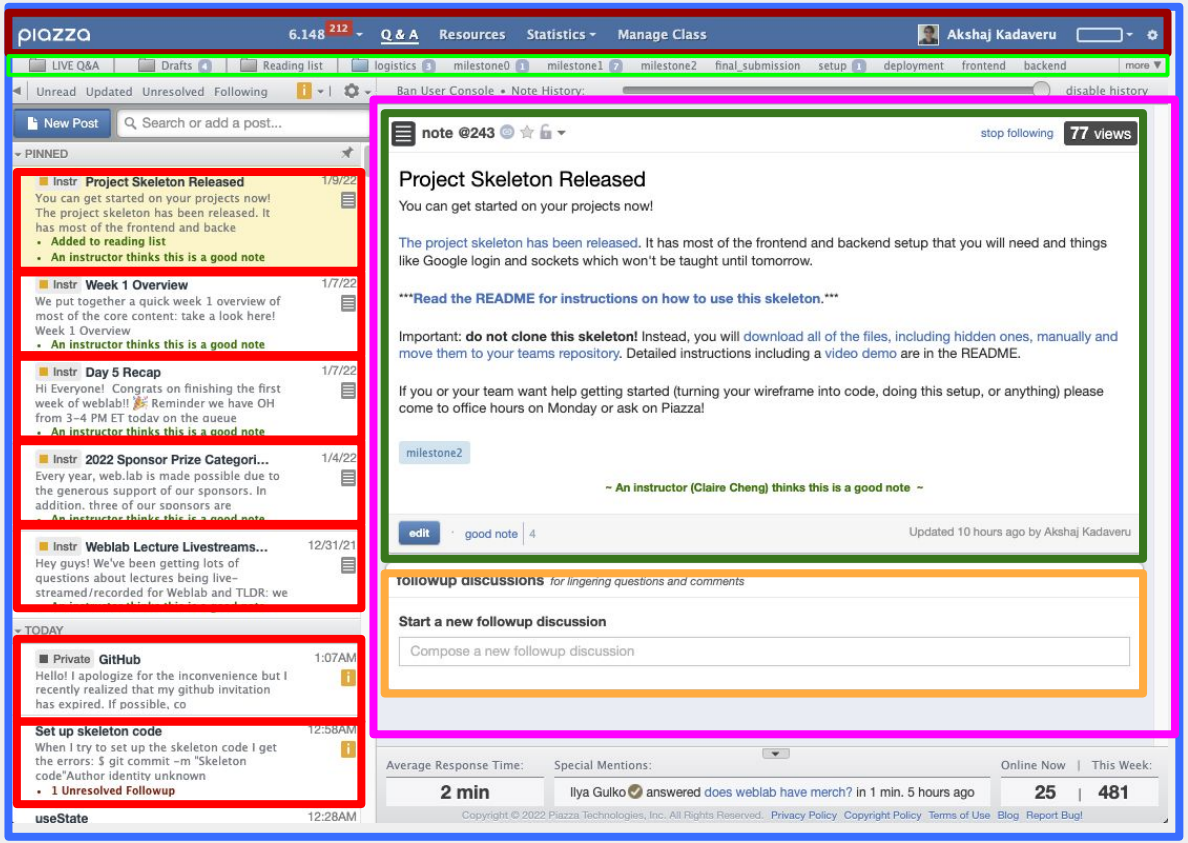




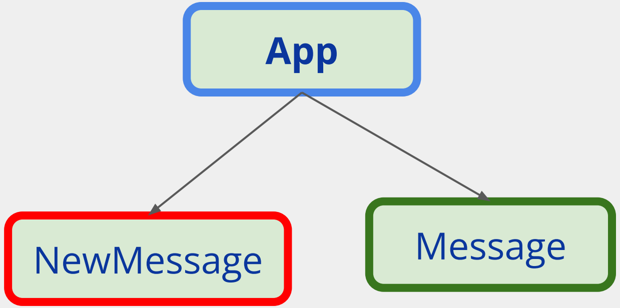
Step 3 Practice: What are the components?

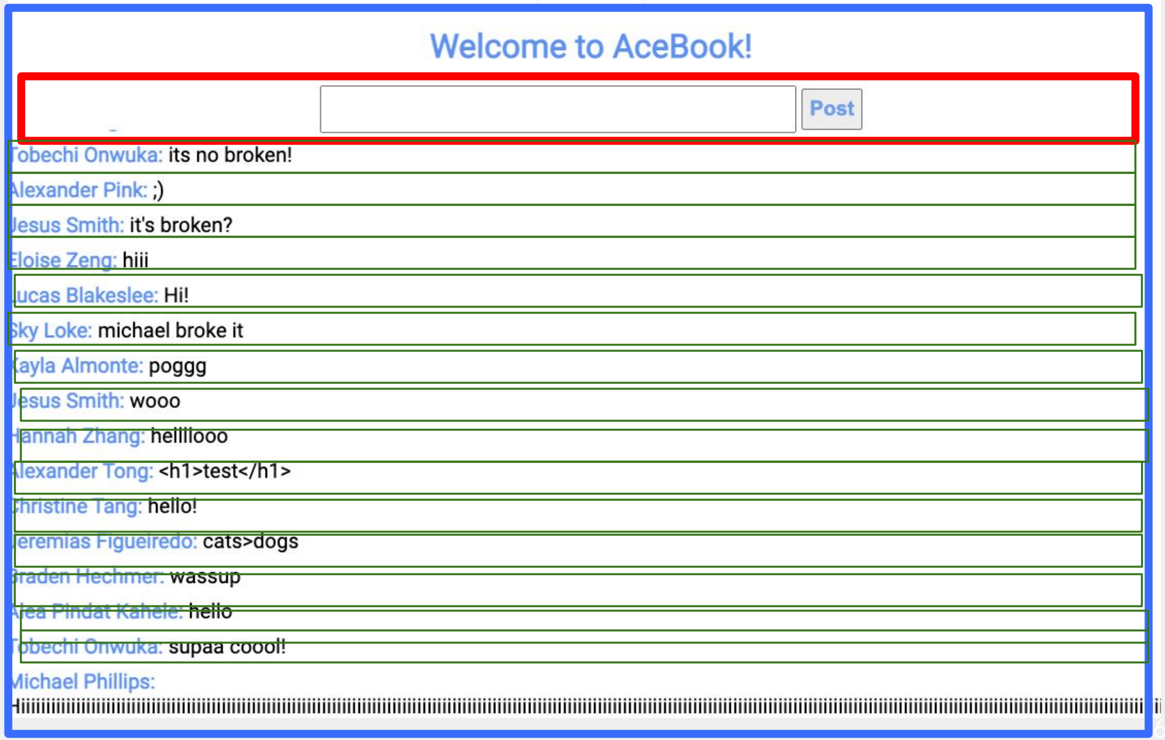


Step 3 Practice: What are the components?

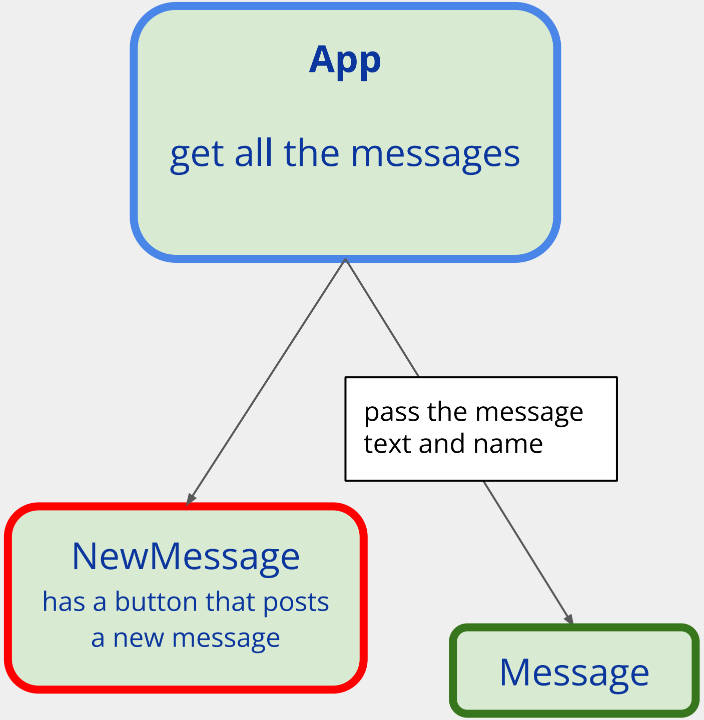


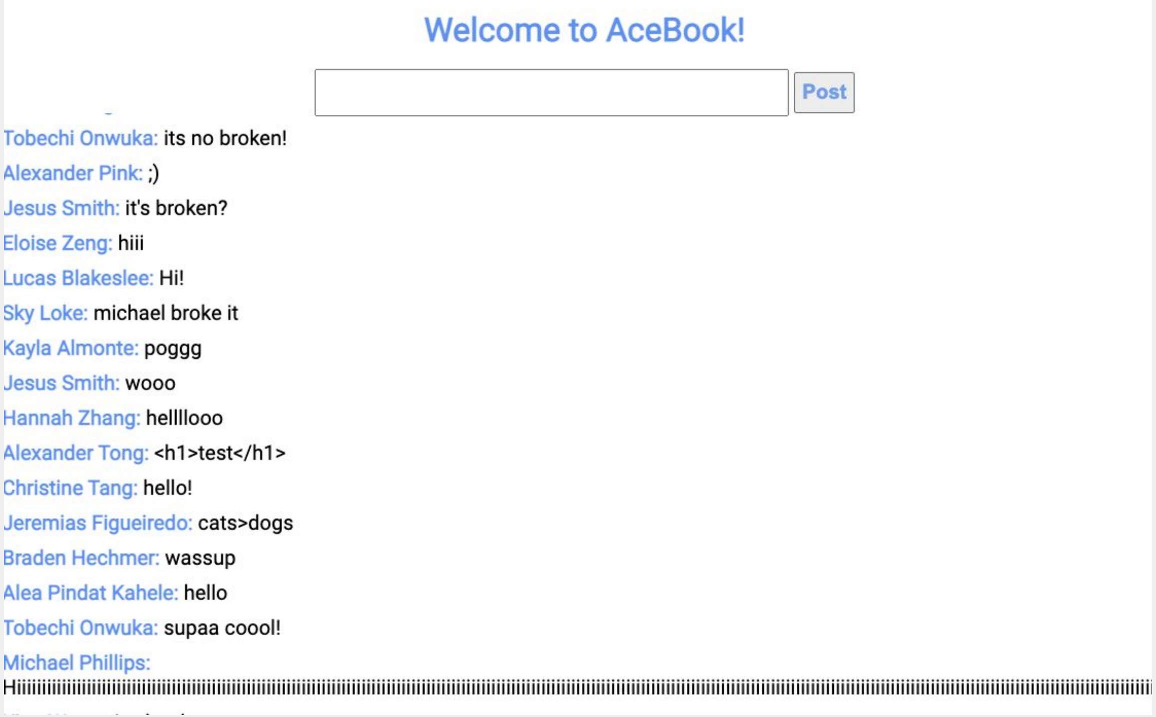
Step 4 : Where do we use our get/post requests? What are the state/props for our components?





Step 4: Where do we use our get/post requests? What are the state/props for our components?





App.js

```
const App = () => {
  const [messages, setMessages] = useState([]);
  useEffect(() => {
    get("/api/messages").then((messagesFromServer) => {
      setMessages(messagesFromServer);
    });
  }, []);
  return (
    <>
      <NewMessage />
      {messages.map((message) => (
        <Message text={message.text} name={message.name} />
      ))}
    </>
  );
};
```

Message.js

```
const Message = (props) => {
  return <div>{props.name + ": " + props.text}</div>;
};
```

NewMessage.js

```
const NewMessage = () => {
  const [content, setContent] = useState("");
  return (
    <>
      <input
        type="text"
        value={content}
        onChange={(event) => {
          setContent(event.target.value);
        }}
      />
      <button
        onClick={() => {
          post("/api/newmessage", { text: content });
        }}
      >
        Post
      </button>
    </>
  );
};
```

Summary: This is all the code we wrote

Database Schemas

message.js

```
const MessageSchema = new mongoose.Schema({
  name: String,
  text: String,
});
```

Backend methods

api.js

```
router.get("/allmessages", (req, res) => {
  Message.find({}).then((messagesFromDB) => {
    res.send(messagesFromDB);
  });
});

router.post("/newmessage", (req, res) => {
  const newMessage = new Message({
    name: "Anonymous",
    text: req.body.text,
  });
  newMessage.save();
});
```

React components (Frontend)

App.js

```
const App = () => {
  const [messages, setMessages] = useState([]);
  useEffect(() => {
    get("/api/messages").then((messagesFromServer) => {
      setMessages(messagesFromServer);
    });
  }, []);
  return (
    <>
      <NewMessage />
      {messages.map((message) => (
        <Message text={message.text} name={message.name} />
      ))}
    </>
  );
};
```

React components (Frontend)

Message.js

```
const Message = (props) => {
  return <div>{props.name + ": " + props.text}</div>;
};
```

NewMessage.js

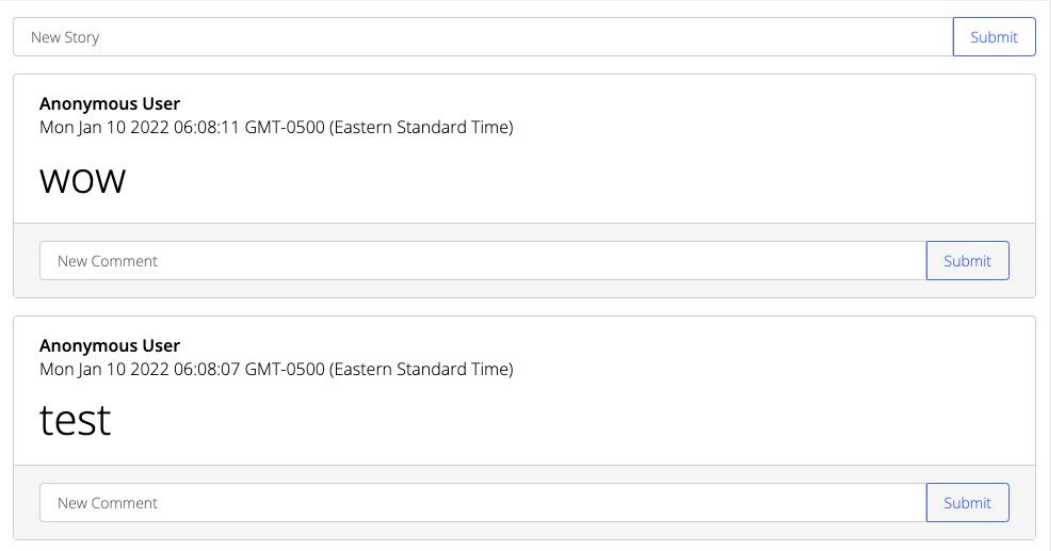
```
const NewMessage = () => {
  const [content, setContent] = useState("");
  return (
    <>
      <input
        type="text"
        value={content}
        onChange={(event) => {
          setContent(event.target.value);
        }}
      />
      <button
        onClick={() => {
          post("/api/newmessage", { text: content });
        }}
      >
        Post
      </button>
    </>
  );
};
```

Mini-workshop: Add the Date to catbook

Hints:

- Add 'date' as a field in the Story model, of type Date
- Pass in the current date Date.now() when adding a new story to the database
- Make sure a story's date is also passed as a prop from Feed.js to Card.js to SingleStory.js, and displayed on the story

- Test if it works by adding new stories! (Old stories won't have dates)




```
git fetch
git reset --hard
git checkout w6-complete
```

Solution

```
git reset --hard
git checkout w6-date
```

```
const StorySchema = new mongoose.Schema({
  creator_name: String,
  content: String,
  date: Date,
});
```

```
router.post("/story", (req, res) => {
  const newStory = new Story({
    creator_name: MY_NAME,
    content: req.body.content,
    date: Date.now(),
  });

  newStory.save().then((story) => res.send(story));
});
```

```
const SingleStory = (props) => {
  return (
    <div className="Card-story">
      <span className="u-bold">{props.creator_name}</s
      <div>{new Date(props.date).toString()}</div>
      <p className="Card-storyContent">{props.content}
    </div>
  );
};
```