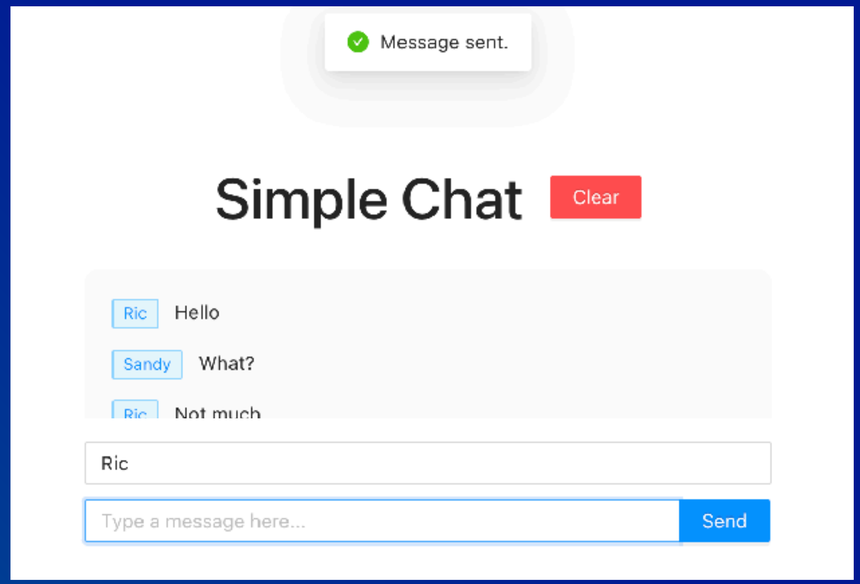
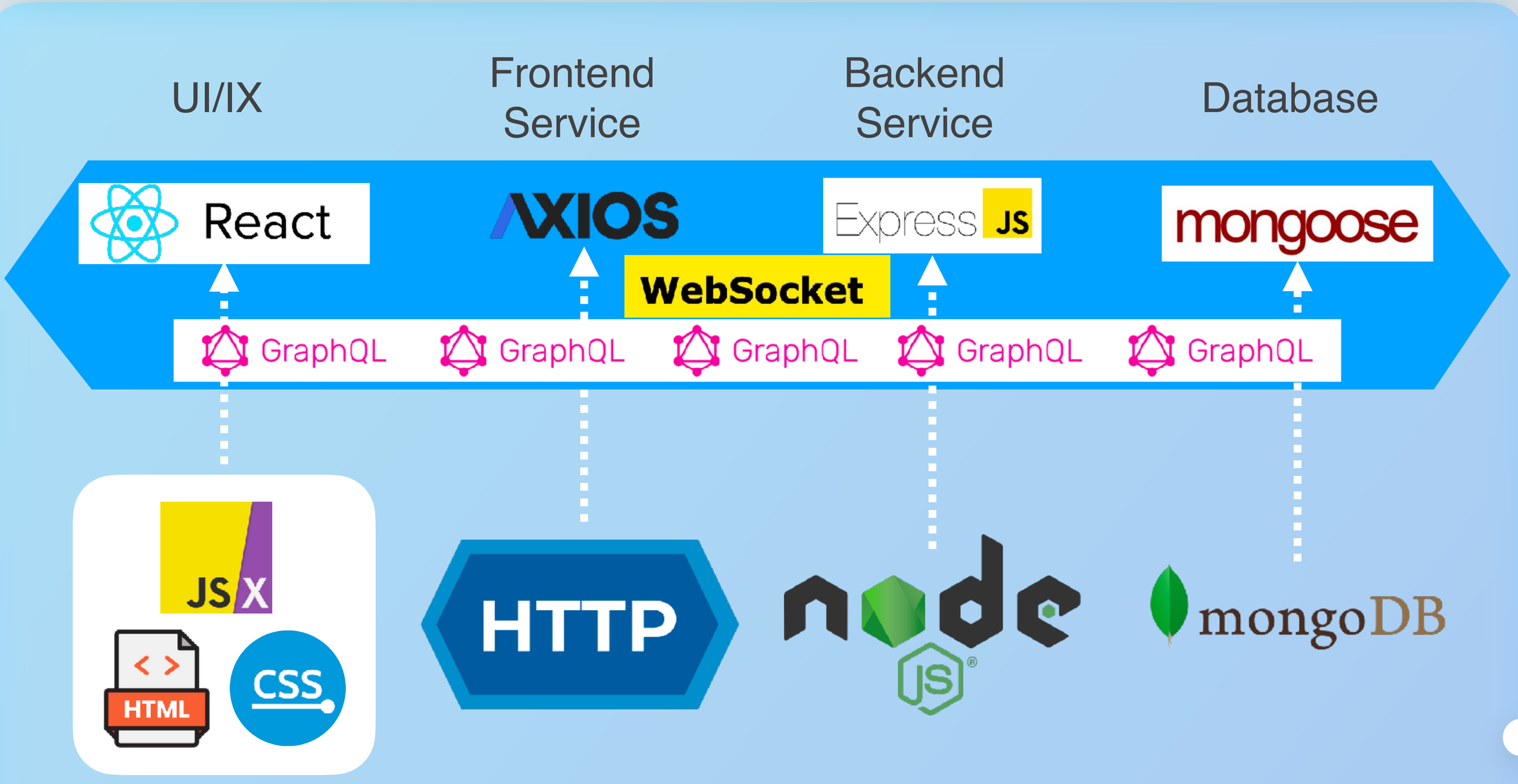
08. Full-Stack Web Applications



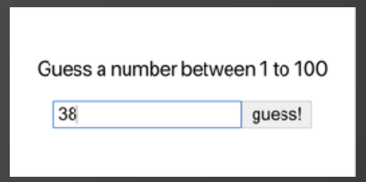
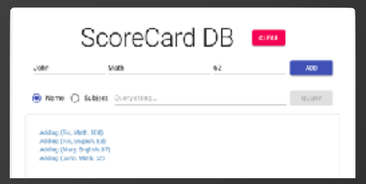
Ric Huang / NTUEE

(EE 3035) Web Programming

Recap: A Typical Web Service Infrastructure



What we have learned...

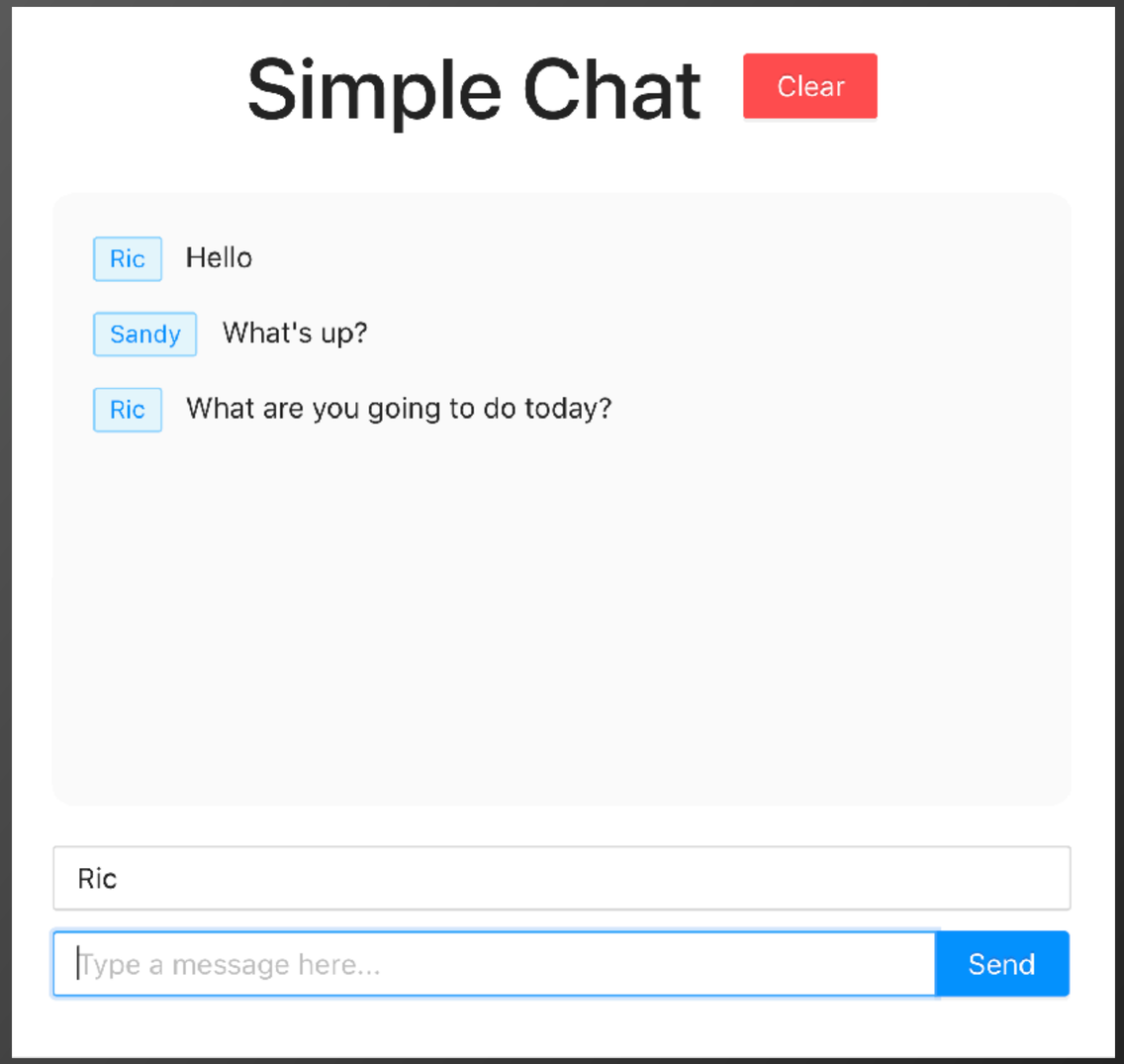
	Frontend	Middleware	Backend	DB
Lecture Note #6 HW#5				
Number-Guessing Game	React Functional Components, Hooks	HTTP, Routing, async/await	Node.js Memory (non-DB)	
Lecture Note #7 HW#6				
ScoreCard App	MaterialUI, Styled Components, React Context, user-defined hooks	HTTP, Routing, async/await	Node.js RESTful API, curl	MongoDB, SQL

來看看大家常見的全端 projects

- 資訊平台/工具
- 論壇/評價網站
- 媒合平台/社群平台
- 線上互動遊戲
- 生產力管理工具

有沒有發現，有一些必要的功能，以目前所學是做不到的呢？

Think: 一個簡單的 chat app (聊天室)...



- 在 "遊戲"、"對話" 類的 web application, 會需要有 "即時性的互動"、"主動通知" 的功能
- 目前學到的後端技術，可以做到嗎？還差什麼？

Server 主動通知 Client 端

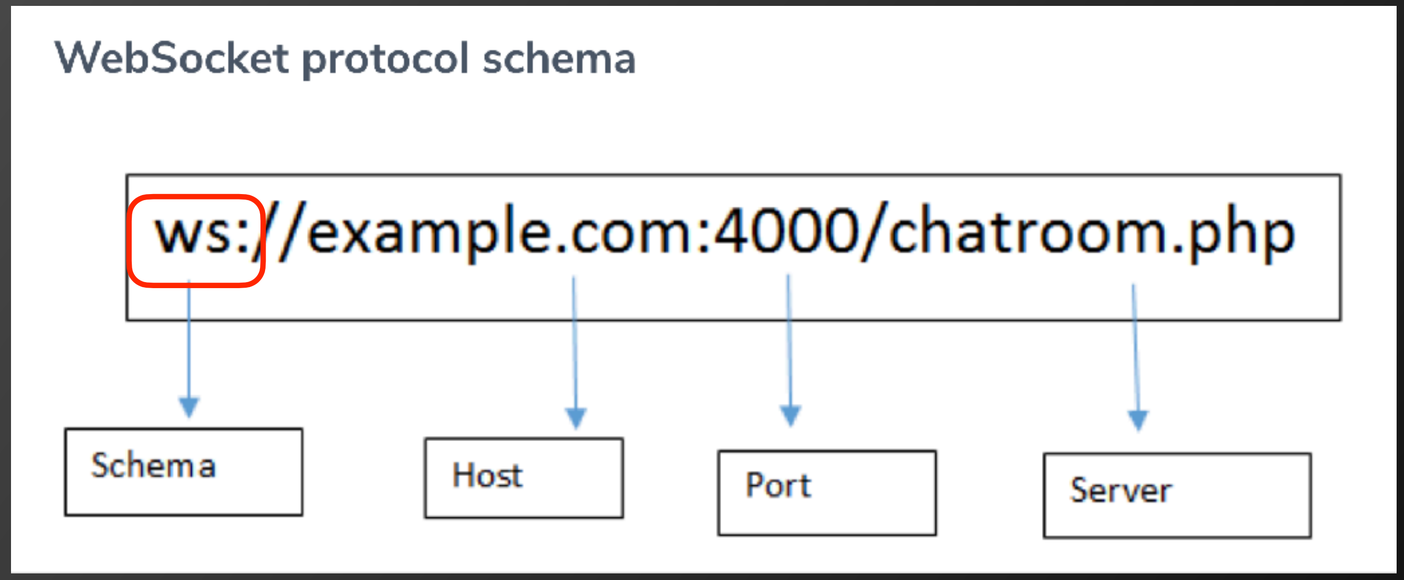
- 之前的例子都是用 HTTP 或是相對應的 middleware 來處理 client / server 之間的溝通
- HTTP 的 server 是很懶惰的，只有當 client 端傳送 CRUD 的請求，他才會對此 cleint 作出對應的動作
- 想像我們在遊戲或是社群的場景，'A' client 對 server 做出 CRUD, 然後 'B' client 應該要即時知道這個變化，但如果 server 不主動通知 'B' 的話，除非 'B' 定時去詢問 server, 否則他就不可能知道 => 這十分沒有效率

WebSocket

- WebSocket is the communication protocol which provides bidirectional communication (duplex) between the Client and the Server over a TCP connection.
- WebSocket remains open all the time so they allow the real-time data transfer. When client triggers the request to the server, server does not close the connection on receiving the response. It rather persists and waits for client or server itself to terminate the request.

HTTP vs. WebSocket

HTTP

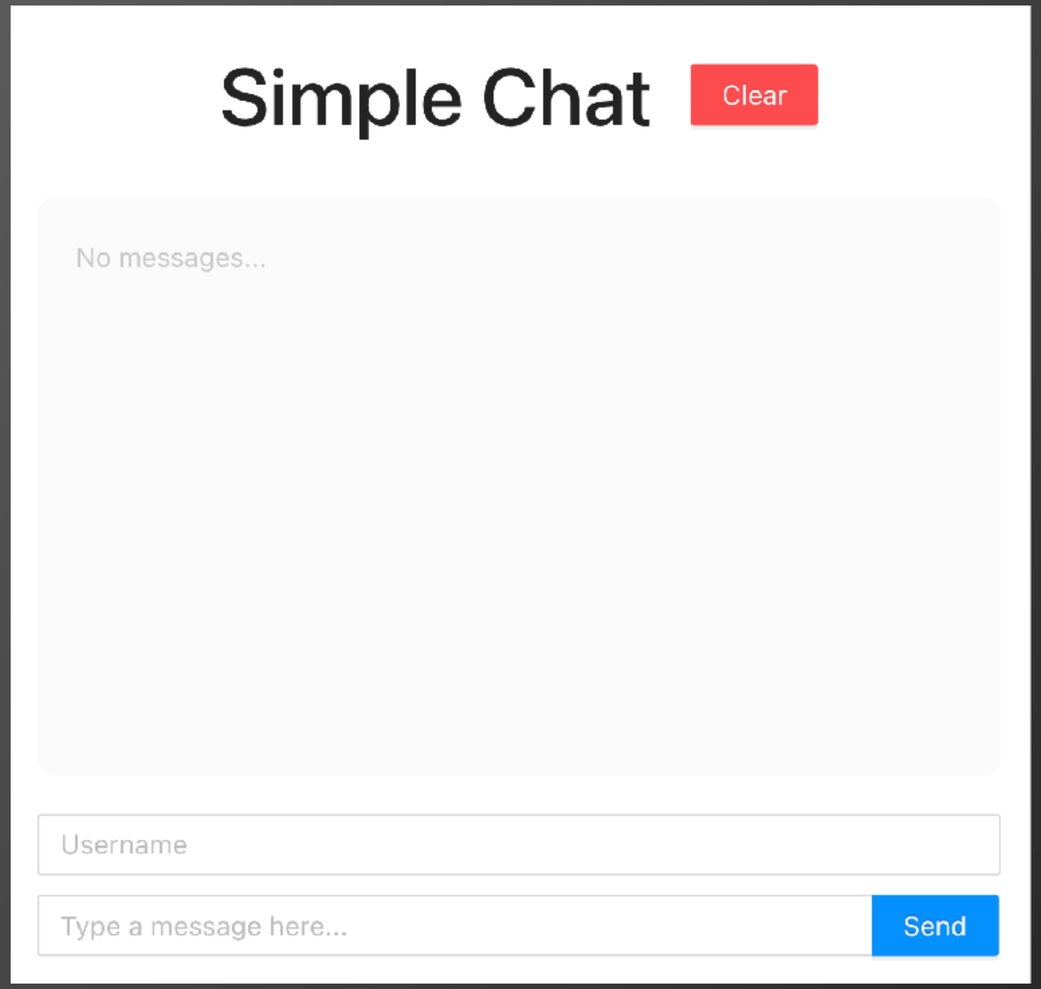


WebSocket

Real-time Duplex Communication (FYI)

- WebSocket offers a full real-time duplex communication.
- FYI, there are many so-called duplex communication techniques between server and client, such as: (ref)
  - Polling
  - Long Polling
  - Streaming
  - Postback and AJAX
  - HTML5

Preparing for the Simple Chat App



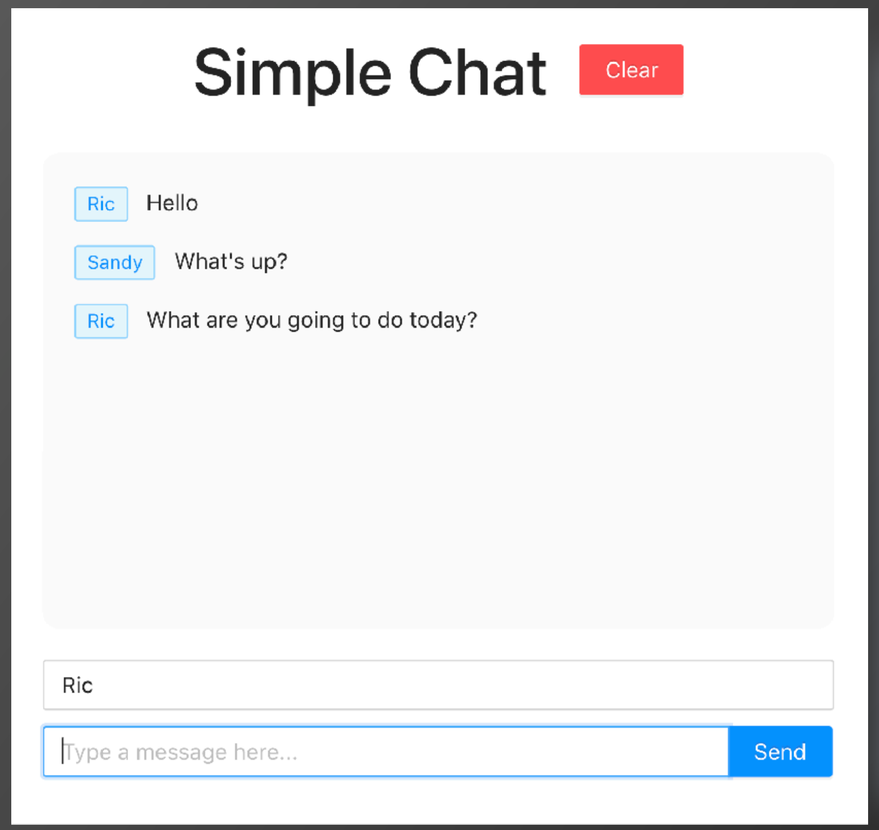
- 我們接下來將會用 "Simple Chat" 這個 app 來學習如何串接 React + WebSocket + Express + Mongoose
- 如同先前的範例，先創建一個 simpleChat (or hw7) project, 然後用 create-react-app 建立 frontend, 再到 NTU Cool 下載 chat-starter-src.tgz, 在 frontend 目錄底下解壓縮 (應該會取代 src 目錄)
- 安裝 "yarn add antd"
- yarn start 後應該會看到 simple chat 的靜態網頁 (如左圖)

你可能會看到一些 webpack/antd 的 warnings => 可以先無視他們

Simple Chat App - Frontend

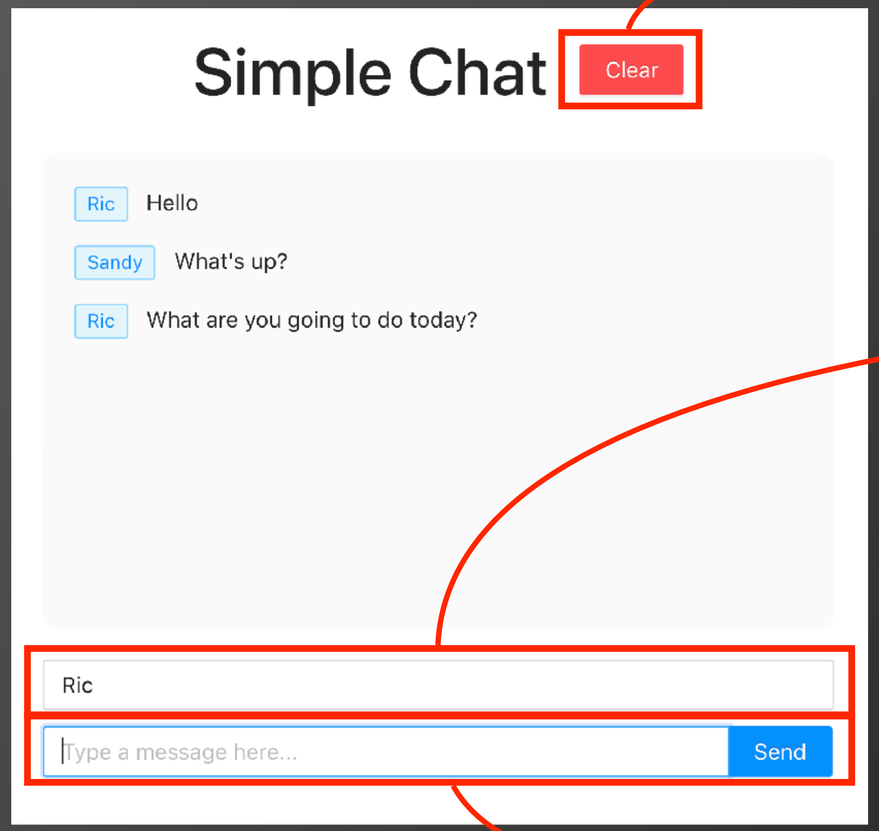
- 前端畫面 (App.js)：使用 Ant Design 的 UI library (ref)

根據畫面，猜猜看要用 antd 的什麼 components?




```
import { Button, Input } from 'antd'
...
  <div className="App">
    <div className="App-title">
      <h1>Simple Chat</h1>
      <Button> Clear </Button>
    </div>
    <div className="App-messages" />
    <Input />
    <Input.Search />
  </div>
```

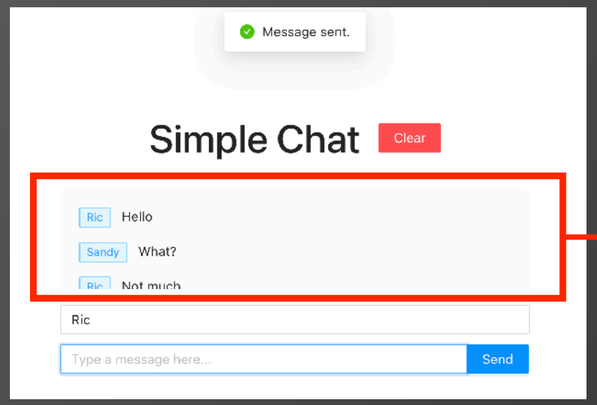
Frontend • antd (ref)




```
<Button type="primary" danger >
  Clear
</Button>
```

```
<Input
 placeholder="Username"
 style={{ marginBottom: 10 }}
></Input>
```

```
<Input.Search
 enterButton="Send"
 placeholder="Type a message here..."
></Input.Search>
```



接下來
就是要定義 chat message body

```
<div className="App-messages" />
```

已經有一個 DOM node 了... 所以？
改成 class component？

定義一個 Hook => useChat()

【Hooks 是什麼？】

Hooks 是用來串連 React class 的 states

把 states 包裝成 functional components

裡頭的 local variables

並且定義一些 methods

讓 functional components

來操作這些 state variables

【Hooks 是什麼？】

React APP

```
const { status, messages, sendMessage }
      = useChat()
const [username, setUsername]
      = useState('')
const [body, setBody]
      = useState('')
useEffect(() => {
  ...
}, [...]);
return (
  <div className="App">
  ...
    </div>
)
```

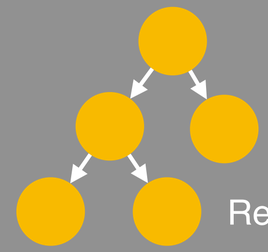
User-Defined Hooks

```
const [status, setStatus]
      = useState({})
const [messages, setMessages]
      = useState([])
const sendMessage = () => ...
...
return {
  status, messages,
  sendMessage
}
```

React Hooks

- useState()
- useEffect()
- useContext()
- useRef()

React Engine



React V-DOM

- Mounting
- Updating
- Unmounting

Component Life-Cycle

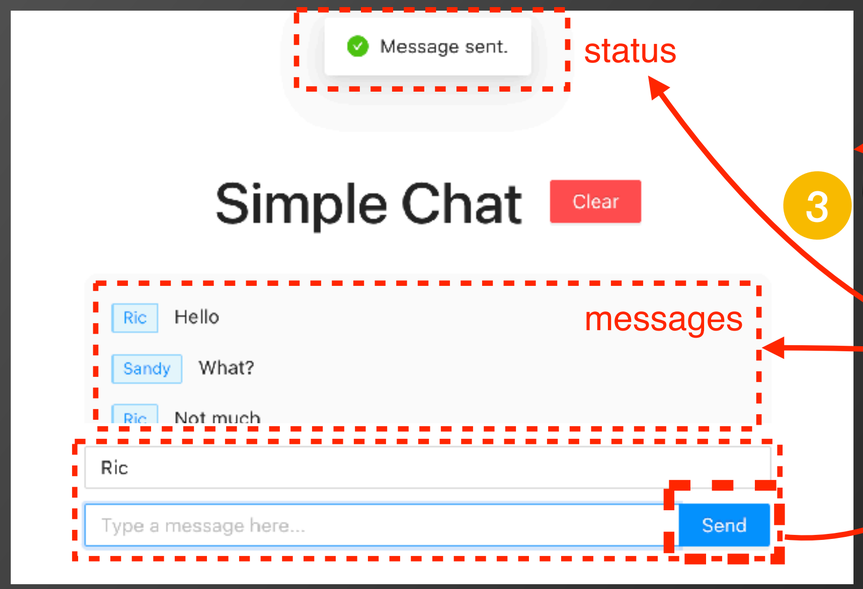
【為什麼要用 "user-defined hooks"?】

理論上你可以把所有的 states 以及操作
states 的 functions 都定義在 APP 的
containers/components 裏頭，
但這樣就像是把所有的 variables,
functions 等寫在同一個檔案裏頭一樣，
這樣不但 code 難讀，也很難 maintain

Simple Chat: Implementation Steps

0. 先用 Ant Design 實現一個靜態畫面

1. 利用 User-Defined Hook 來實現前端功能



App.js

1 Import

3 Fake response (update)

2 Call "sendMessage ({name, body})"

4 Fake backend

console.log({name, body})

User-Defined Hooks

```
const [status, setStatus]
      = useState({})
const [messages, setMessages]
      = useState([])
const sendMessage = () => ...
...
return {
  status, messages,
  sendMessage
}
```

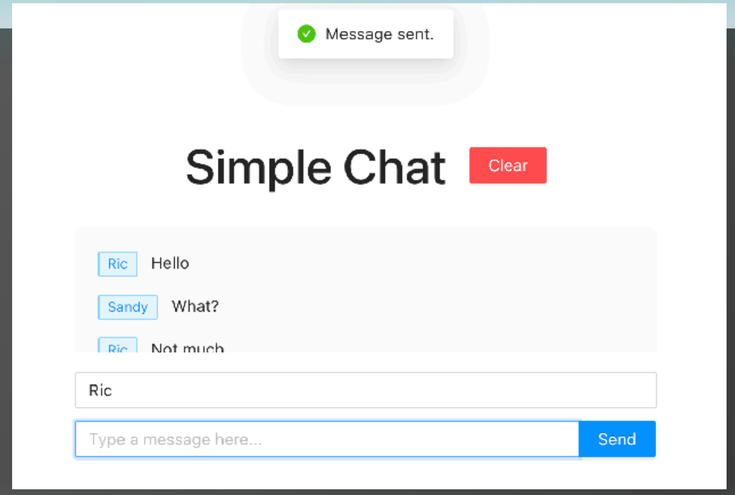
useChat.js

useChat Hook 如何定義？（useChat.js）

- 用 useState 定義 "messages", "status" 兩個 states
- 定義 "sendMessage()" method
- return { status, messages, sendMethod }

```
import { useState } from "react";
const useChat = () => {
  const [messages, setMessages] = useState([]);
  const [status, setStatus] = useState({});
  const sendMessage = (payload) => {
    ... // update messages and status
    console.log(payload);
  }
  return {
    status, messages, sendMessage
 };
};
export default useChat;
```

Frontend Design • Use "useChat" Hook in App.js

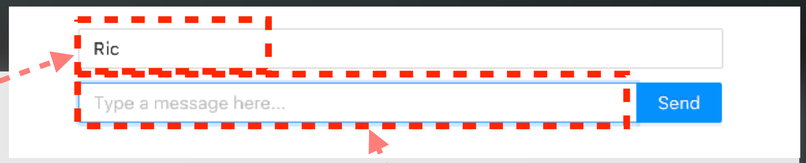



```
import { useState } from 'react'
import { Button, Input, Tag } from 'antd'
import useChat from './useChat'
function App() {
  const { status, messages, sendMessage } = useChat()
  const [username, setUsername] = useState('')
  const [body, setBody] = useState('')
```

Import from useChat.js

Local state variables from message input

Frontend Design ◦ Input (name) and Input.Search (msg)



Save and store the username

```
<Input
  placeholder="Username"
  value={username}
  onChange={(e) => setUsername(e.target.value)}
  style={{ marginBottom: 10 }}
></Input>
<Input.Search
  value={body}
  onChange={(e) => setBody(e.target.value)}
  enterButton="Send"
  placeholder="Type a message here..."
  onSearch={(msg) => {
    sendMessage({ name: username, body: msg })
    setBody('')
  }}
></Input.Search>
```

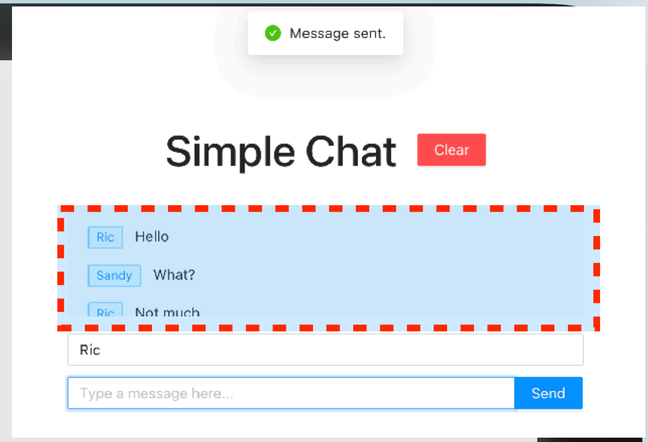
Save and store the textBody

When “Send”, call sendMessage()

[ref]

Frontend Design • Update the messages display

```
import { useState } from 'react'
import { Button, Input, Tag } from 'antd'
import useChat from './useChat'
function App() {
  const { status, messages, sendMessage } = useChat()
  const [username, setUsername] = useState('')
  const [body, setBody] = useState('')  // textBody
 return (
    <div className="App">
      <div className="App-title"> ... </div>
      <div className="App-messages">
        {messages.length === 0 ? (
          <p style={{ color: '#ccc' }}> No messages... </p>
        ) : (
          messages.map(({ name, body }, i) => (
            <p className="App-message" key={i}>
              <Tag color="blue">{name}</Tag> {body}
            </p>
          ))
        )}
      </div>
```



Initial or when cleared

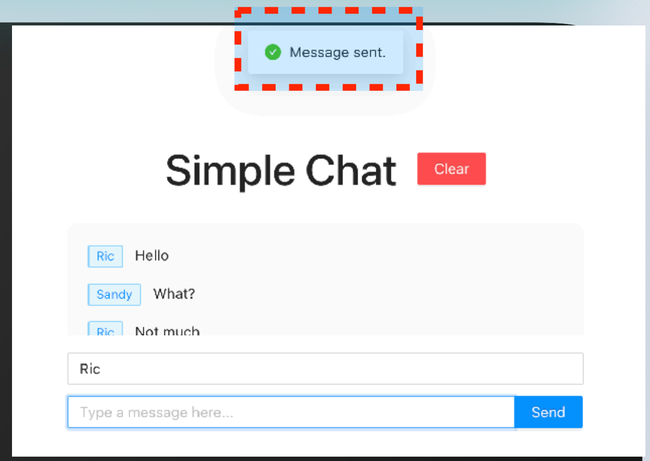
Print each message: { name, textBody }

Frontend Design ◦ Display the "status"

- Pop up when "status" state is updated
- useState() or useEffect()?

```
import { Button, Input, message, Tag } from 'antd'
// in function App()
  const displayStatus = (s) => {
    if (s.msg) {
      const { type, msg } = s;
      const content = {
        content: msg, duration: 0.5 }
      switch (type) {
        case 'success':
          message.success(content)
          break
        case 'error':
        default:
          message.error(content)
          break
  }}}
  useEffect(() => {
    displayStatus(status)}, [status])
```

App.js




```
// in useChat()
const sendMessage = (msg) => {
  setMessages(...);
  setStatus({
    type: "success",
    msg: "Message sent." });
  console.log(msg);
}
```

useChat.js

yarn start again.

Do you see:
(1) messages printed
(2) status displayed
(3) { username, body }
printed on console?

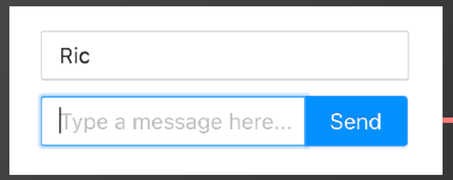
接下來，
我們將把前端的訊息
透過 Web Socket
傳到後端 & DB
然後再從後端傳回給前端

Simple Chat: Implementation Steps

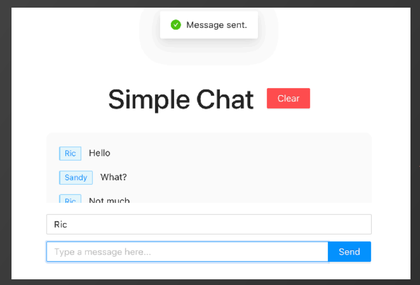
0. 先用 Ant Design 實現一個靜態畫面

1. 利用 User-Defined Hook 來實現前端功能

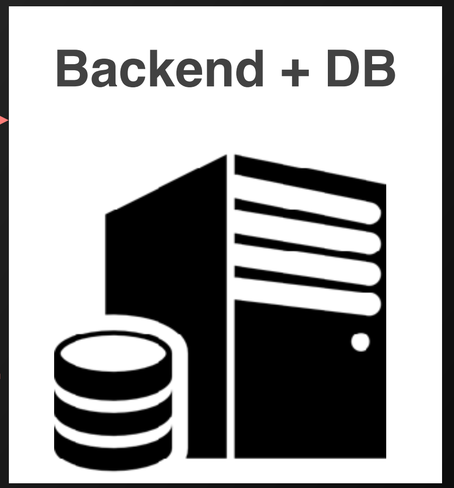
2. 利用 Web Socket 來完成前後端的溝通



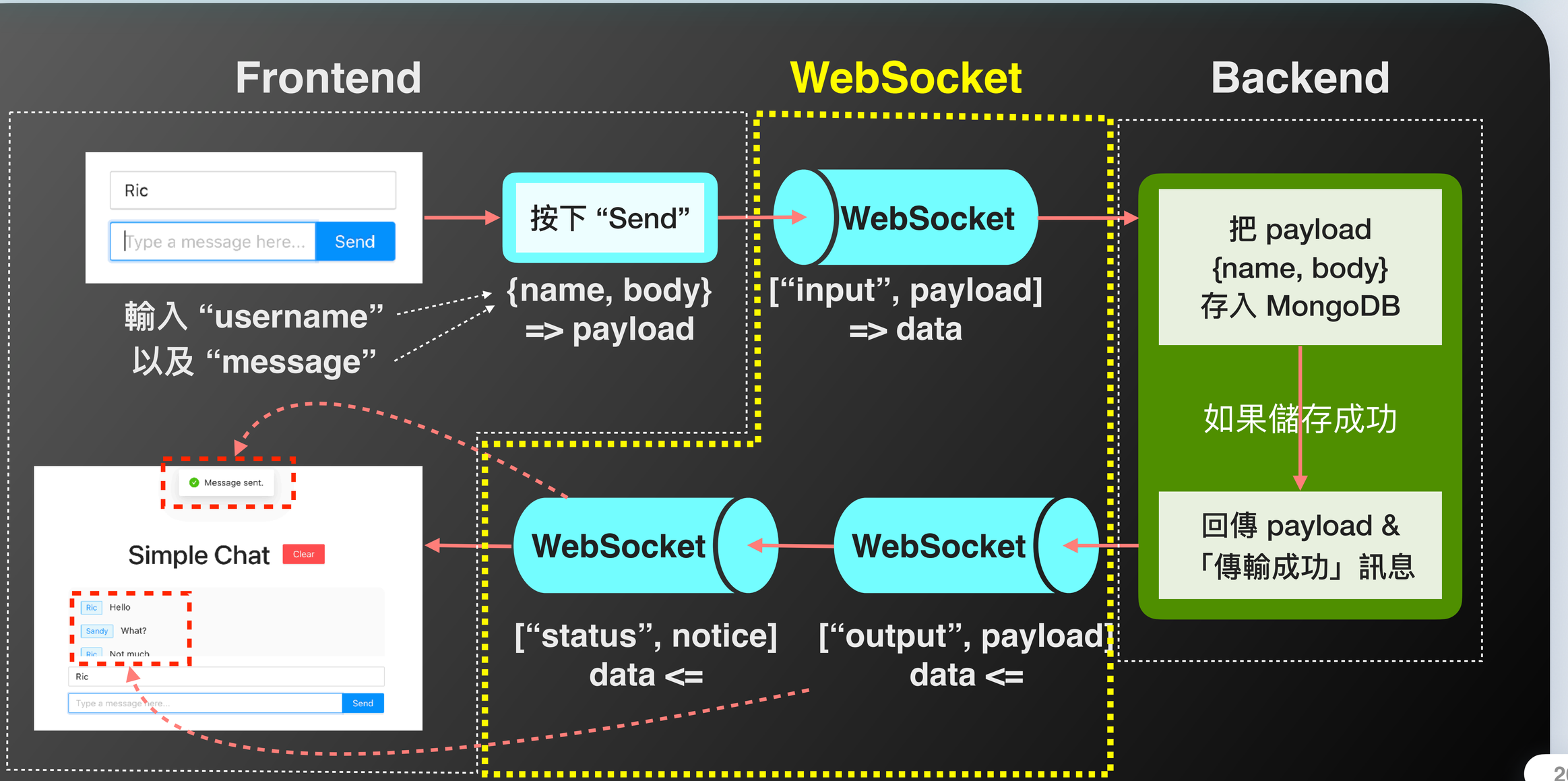
[{“input”, {name, message}}]



[{“output”, messages},
{"status", notice}]



Duplex message communication process



建立 Client WebSocket Service (ref)

- Client side — `new WebSocket(url)`

```
const client = new WebSocket('ws://localhost:4000')
```

In which file?

- Note: On the client side, you can't require or import 'ws'. If you do, you will receive the following error:

```
Error: ws does not work in the browser. Browser clients must use the native WebSocket object
```

建立 Client WebSocket Service (ref)

- 在 useChat() hook 裏頭加入 Web Socket client, 並定義一個 function "sendData()" 用來跟後端溝通
  => 用 client.send() 來傳送 data

```
const client = new WebSocket
                    ('ws://localhost:4000')
const sendData = async (data) => {
  await client.send(
    JSON.stringify(data));
};
```

Why "JSON.stringify()"?

Sending Internet Data as "Byte String"

- Note: data sent thru internet is treated as "byte string”
- JSON.stringify(obj) // converting object to string

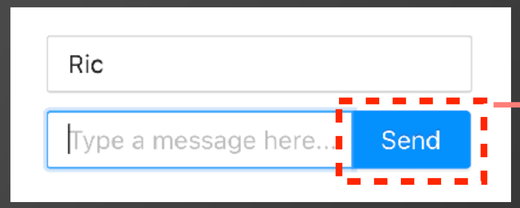
```
JSON.stringify({});                  // '{}'
JSON.stringify(true);                // 'true'
JSON.stringify('foo');               // '"foo"'
JSON.stringify([1, 'false', false]); // '[1,"false",false]'
JSON.stringify({ x: 5 });            // '{"x":5}'
```

- JSON.parse(str[, reviver]) // converting string to object

```
JSON.parse('{}');              // {}
JSON.parse('true');            // true
JSON.parse('"foo"');           // "foo"
JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
JSON.parse('{"x":5}');         // { x: 5 }
```

ws_client.sendData()

- 在使用 "byte string" 來傳遞 internet data 的時候，除了主要的訊息 (i.e. payload) 之外，通常還會帶著一些如 status, type 等附帶資訊
  => 語法：ws_client.sendData(type, payload)
- 因此，從 <Input.Search> 的按鈕，到 ws.client.sendData(), 會經過底下的包裝：



onSearch

{name, msg}
as **payload**

```
const sendMessage = (payload) => {
  sendData(["input", payload]);
};
```

```
const sendData = async (data) => {
  await client.send(
    JSON.stringify(data));
};
```

建立 Server-side WebSocket Service (1/3) (ref)

- Server side —
  `new WebSocket.Server({server: serverObj})`

  e.g. HTTP Server
- Prepare backend directory and packages

```
> mkdir backend
> cd backend; yarn init -y
> yarn add express nodemon mongoose ws
> yarn add -D @babel/cli @babel/core @babel/node @babel/preset-env
@babel/plugin-proposal-class-properties @babel/plugin-proposal-
object-rest-spread @babel/plugin-transform-arrow-functions dotenv-
defaults
```

- Copy ".babelrc" and modify "package.json" as in HW#6

建立 Server-side WebSocket Service (2/3) (ref)

- 建立 DB 連線 (ref: HW#6)
- "mongo.js"

```
import mongoose from 'mongoose';
import dotenv from 'dotenv-defaults';

export default {
  connect: () => {
    dotenv.config();
    if (!process.env.MONGO_URL) {
      console.error("Missing MONGO_URL!!!");
      process.exit(1);
    }
    mongoose
    .connect(process.env.MONGO_URL, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    })
    .then((res) => console.log("mongo db connection created"));
    mongoose.connection.on('error',
      console.error.bind(console, 'connection error:'));
  }
};
```

建立 Server-side WebSocket Service (3/3) (ref)

- "server.js"

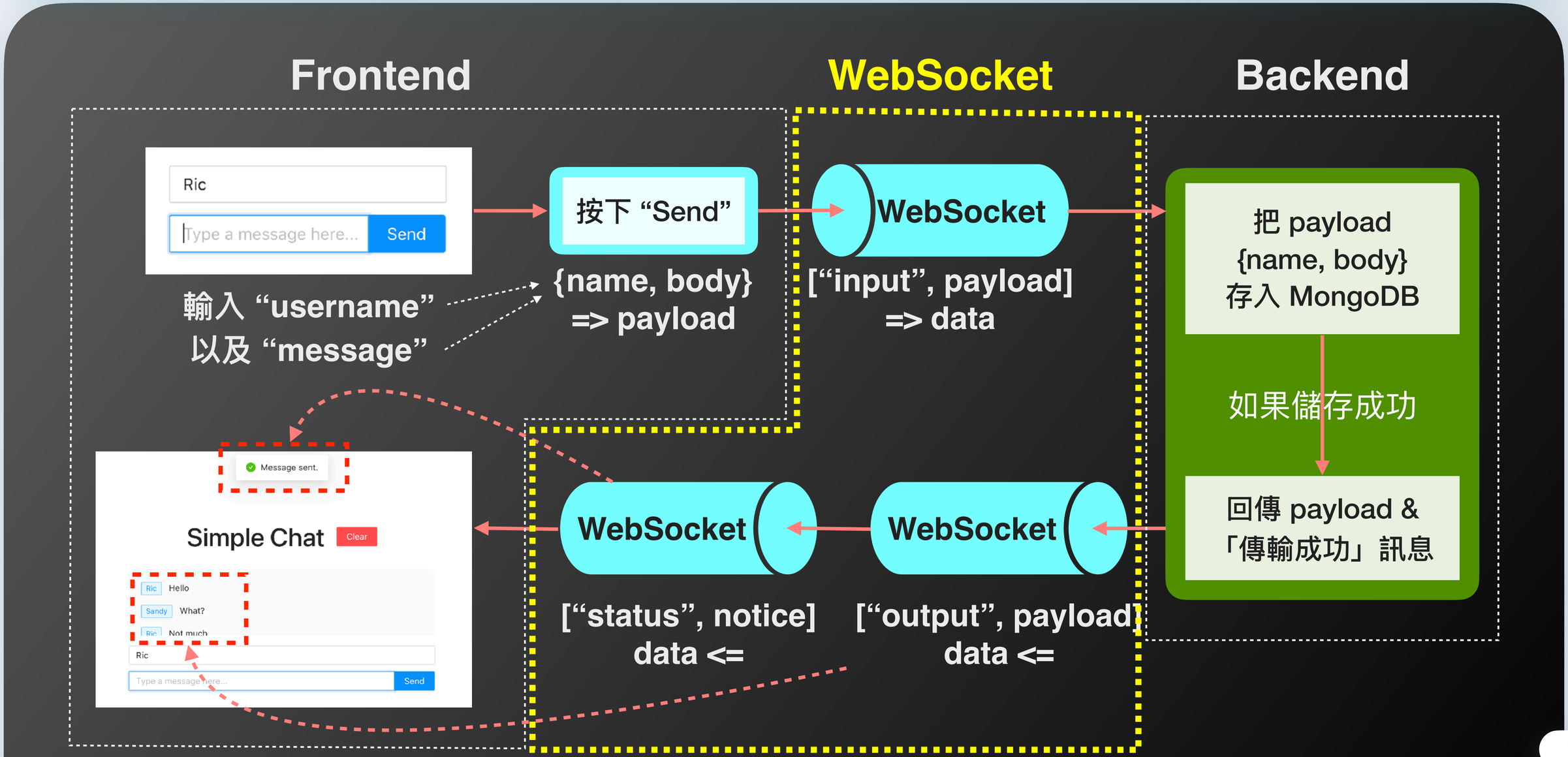
```
import http, express, dotenv-defaults, mongoose, WebSocket... etc.
import mongo from './mongo'

mongo.connect();

const app = express()
const server = http.createServer(app)
const wss = new WebSocket.Server({ server })
const db = mongoose.connection

db.once('open', () => {
  console.log("MongoDB connected!");
  wss.on('connection', (ws) => {
    // Define WebSocket connection logic
    ...
  });
});

const PORT = process.env.PORT || 4000;
server.listen(PORT, () => { ... });
```

Connecting DB

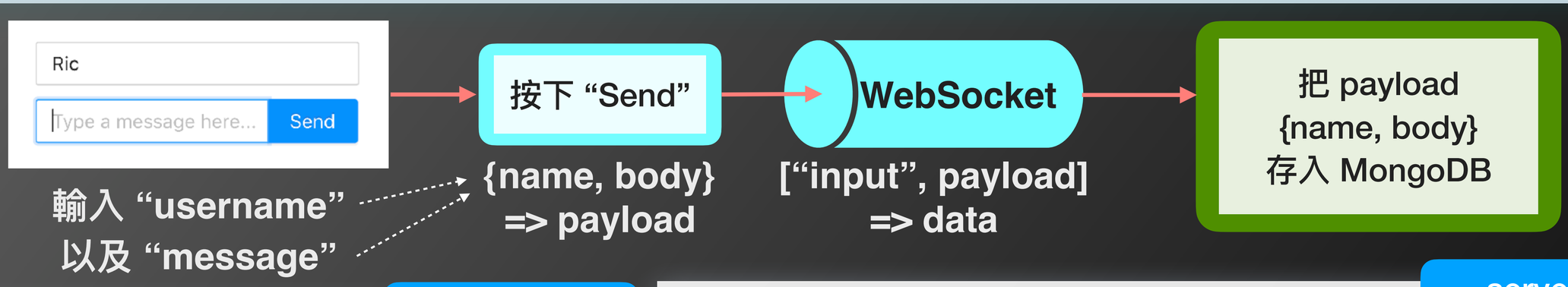
Client-side WebSocket

接下來 Backend 的 code 幾乎都在 slides 裡頭，請大家搞懂之後，自行把他們拼湊起來！

Recap: Duplex message communication process



WebSocket Server - (1) Parsing byteString & payload



Client

```
<Input.Search
  onSearch={(msg) => {
    sendMessage({
      name: username, body: msg })
></Input.Search>
```

```
const sendMessage = (payload) => {
  sendData(["input", payload]);
};
```

```
const sendData = async (data) => {
    await client.send(
          JSON.stringify(data));
  };
```

server.js

```
import wsConnect from './wsConnect'
db.once('open', () => {
  wss.on('connection', (ws) => {
    wsConnect.onMessage(ws); });
});
```

wsConnect.js

```
export default {
  onMessage: (ws) => (
    async (byteString) => {
      const { data } = byteString
      const [task, payload] = JSON.parse(data)
      switch (task) {
        case 'input': {
          const { name, body } = payload
          // Save payload to DB
          // Respond to client
        ...
```

WebSocket Server。(2) Define Message MongoDB Schema

- 如前所述，MongoDB 的儲存格式就像是 JSON format 一樣，因此，在 JS 裡頭就用 object 的格式來定義即可

models/message.js

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema
// Creating a schema, sort of like working with an ORM
const MessageSchema = new Schema({
  name: {
    type: String,
    required: [true, 'Name field is required.']
  },
  body: {
    type: String,
    required: [true, 'Body field is required.']
  }
})
// Creating a table within database with the defined schema
const Message = mongoose.model('message', MessageSchema)
// Exporting table for querying and mutating
export default Message;
```

WebSocket Server - (3) Save payload to DB

wsConnect.js

```
export default {
  onMessage: (ws) => (
    async (byteString) => {
      const { data } = byteString
      const [task, payload] = JSON.parse(data)
      switch (task) {
        case 'input': {
          const { name, body } = payload
          // Save payload to DB
          const message
          = new Message({ name, body })
          try { await message.save();
          } catch (e) { throw new Error
            ("Message DB save error: " + e); }
          // Respond to client
      ...
```

Save to DB

Backend

把 payload
{name, body}
存入 MongoDB

如果儲存成功

回傳 payload &
「傳輸成功」訊息

WebSocket

WebSocket

[“status”, notice]
data <=

[“output”, payload]
data <=

WebSocket Server - (4) Respond to Client

wsConnect.js

```
export default {
  onMessage: (ws) => (
    async (byteString) => {
      ...
      switch (task) {
      ... // Save payload to DB
          // Respond to client
          sendData(['output', [payload]], ws)
          sendStatus({
            type: 'success',
            msg: 'Message sent.'
          }, ws)
          break
        }
        default: break
}})}
```

Send message to client

Send Notification message

Backend

把 payload {name, body} 存入 MongoDB

如果儲存成功

回傳 payload & 「傳輸成功」訊息

WebSocket

WebSocket

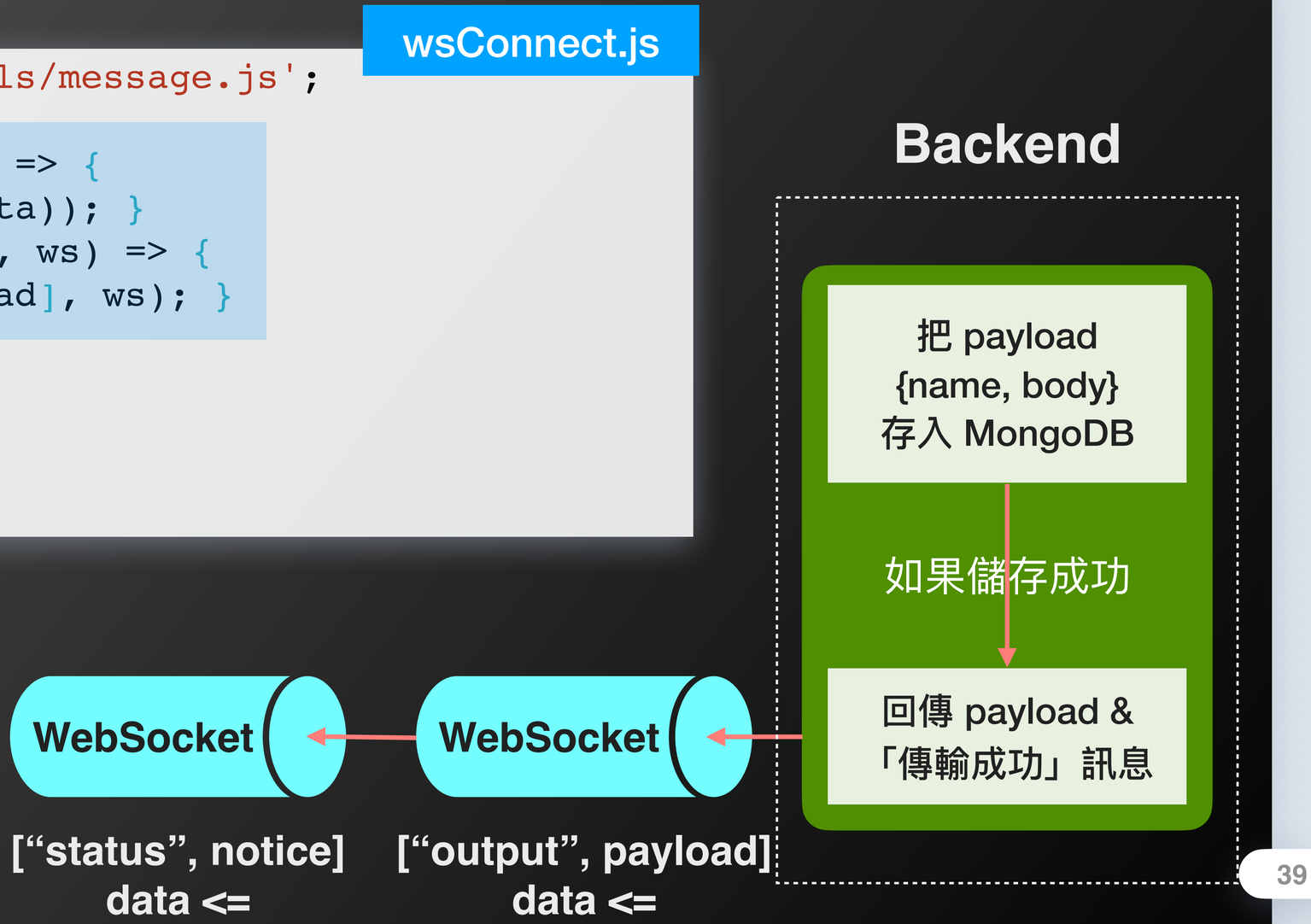
[“status”, notice] data <=

[“output”, payload] data <=

WebSocket Server - (5) Add Helper Functions

wsConnect.js

```
import Message from './models/message.js';

const sendData = (data, ws) => {
  ws.send(JSON.stringify(data)); }
const sendStatus = (payload, ws) => {
  sendData(["status", payload], ws); }

export default {
  onMessage: (ws) => (
      ...
)}
```



Restart server and client
Open "MongoDB dashboard" to see
if the data is properly saved.
Does it work now?

Need to define how client
receives data

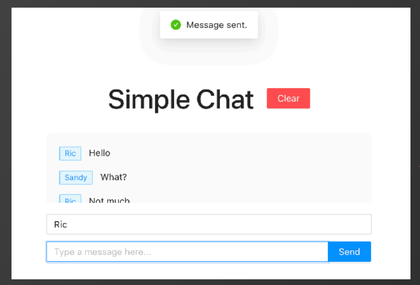
Simple Chat: Implementation Steps

0. 先用 Ant Design 實現一個靜態畫面

1. 利用 User-Defined Hook 來實現前端功能

2. 利用 Web Socket 來完成前後端的溝通

3. 前端處理收到的資料並更新畫面



[{“output”, messages},
{"status", notice}]

Frontend ◦ (1) Receiving Data

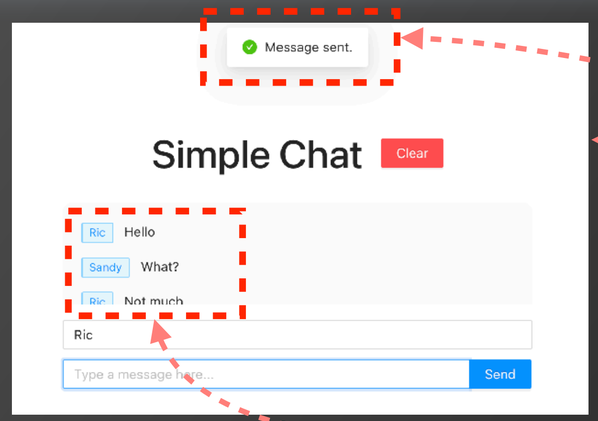
useChat.js

```
const useChat = () => {
  ... // define messages, status
  client.onmessage = (byteString) => {
    const { data } = byteString;
    const [task, payload] = JSON.parse(data);
    switch (task) {
      case "output": {
        setMessages(() =>
        [...messages, ...payload]); break; }
      default: break;
    }
  }
  ... // define sendMessage(), sendData()
}
```

Received from server

App.js

```
import useChat from './useChat'
function App() {
  ...
  return (
    ...
    <div className="App-messages">
      {messages.length === 0 ? (...) : (
        messages.map(({name,body},i) => (
          <p key={i}>
            <Tag color="blue">
            {name}</Tag> {body}
          </p>))}
    </div>
```



WebSocket

WebSocket

回傳 payload &
「傳輸成功」訊息

["status", notice]
data <=

["output", payload]
data <=

Frontend ◦ (2) Displaying Status

useChat.js

```
const useChat = () => {
  ...
  client.onmessage = (byteString) => {
  ...
    switch (task) {
      case "output": { ... }
      case "status": {
        setStatus(payload); break; }
      default: break;
    }
  }
  ...
}
```

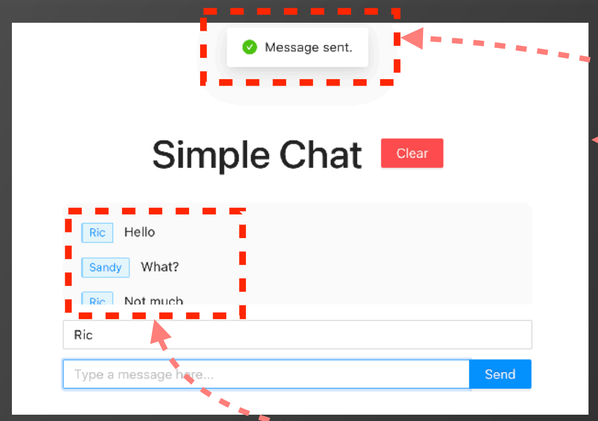
Received from server

App.js

```
function App() {
  const displayStatus = (payload) => {
    if (payload.msg) {
      const { type, msg } = payload
      const content = {
        content: msg, duration: 0.5 }
      switch (type) {
        case 'success':
          message.success(content)
          break
        case 'error':
        default:
          message.error(content)
          break
    }}}
  useEffect(() => {
    displayStatus(status)}, [status])
```

antd component

每次 re-render 後



WebSocket

WebSocket

回傳 payload &
「傳輸成功」訊息

["status", notice]
data <=

["output", payload]
data <=

Have you successfully seen the chat?

To improve:

1. 檢查 username, textBody 是否為空值
2. Username 輸入之後自動跳到 textBody input
3. 一開始就顯示 DB 之前的對話記錄
4. Clear message 還沒做

1. 檢查 username, textBody 是否為空值

- 在 <Input.Search onSearch> 的時候檢查 —

```
<Input.Search
  ...
  onSearch={(msg) => {
    if (!msg || !username) {
      displayStatus({
        type: 'error',
        msg: 'Please enter a username and a message body.'
      })
      return
    }

    sendMessage({ name: username, body: msg })
    setBody('')
  }}
></Input.Search>
```

2. Username 輸入之後自動跳到 textBody input

- How? By "useRef()"

1. 在 App.js 先宣告一個 useRef() hook for textBody

```
const bodyRef = useRef(null)
```

2. 把這個 bodyRef 指定給 <Input.Search ref>

```
<Input.Search
    ref={bodyRef}
```

3. 當 <Input> 輸入 "Enter" 時，把 focus 移到 bodyRef

```
<Input
  onKeyDown={(e) => {
    if (e.key === 'Enter') {
      bodyRef.current.focus()
  }}}>
```

3. 一開始就顯示 DB 之前的對話記錄

- 在 backend/src/wsConnect.js 加個 initData()

```
export default {
  initData: (ws) => {
    Message.find().sort({ created_at: -1 }).limit(100)
      .exec((err, res) => {
        if (err) throw err;
        // initialize app with existing messages
        sendData(["init", res], ws);
      });
  },
  onMessage: (ws) => ( ... )
}
```

記得要在 server.js
wss.on(“connection”, …)
裡頭一開始就呼叫

- 在 useChat 裡頭加個 "init" 的 task

```
const useChat = () => {
  client.onmessage = (byteString) => {
    switch (task) {
      case "init": {
        setMessages(payload);
        break;
}}}}
```

4. Clear message

- 在 useChat 裡頭加個 "clearMessages()" 的 method

```
const useChat = () => {
  const clearMessages = () => {
    sendData(["clear"]);
  };
  return { status, messages, sendMessage, clearMessages };
}
```

- 在 App.js 裡頭 handle <Button onClick>

```
function App() {
  const {status, messages, sendMessage, clearMessages}= useChat()
  return (...
        <Button type="primary" danger onClick={clearMessages}>
          Clear
        </Button>
    ...)
}
```

4. Clear message (continued)

- 在 wsConnect 的 onMessage 裡頭加個 case "clear"

```
case 'clear': {
  Message.deleteMany({}, () => {
    sendData(['cleared'], ws)
    sendStatus
    ({ type: 'info', msg: 'Message cache cleared.'}, ws)
  })
  break
}
```

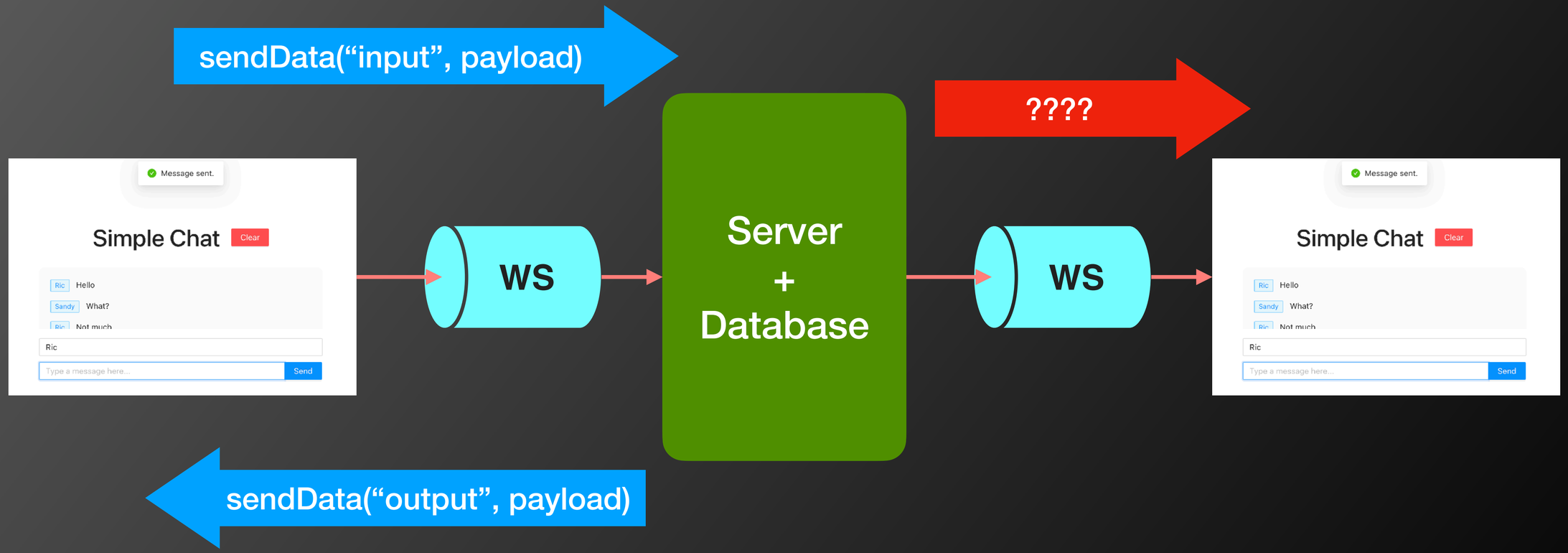
- 在 useChat 裡頭的 client.onmessage 加個 "cleared" 的 task

```
client.onmessage = (byteString) => {
  switch (task) {
    case "cleared": {
      setMessages([]);
      break;
  }}
}
```

可以對話了嗎？

還有什麼問題？

Uh，可以對話嗎？還缺什麼？



要利用 Broadcast 做到對話的互動

Backend - Broadcast Message

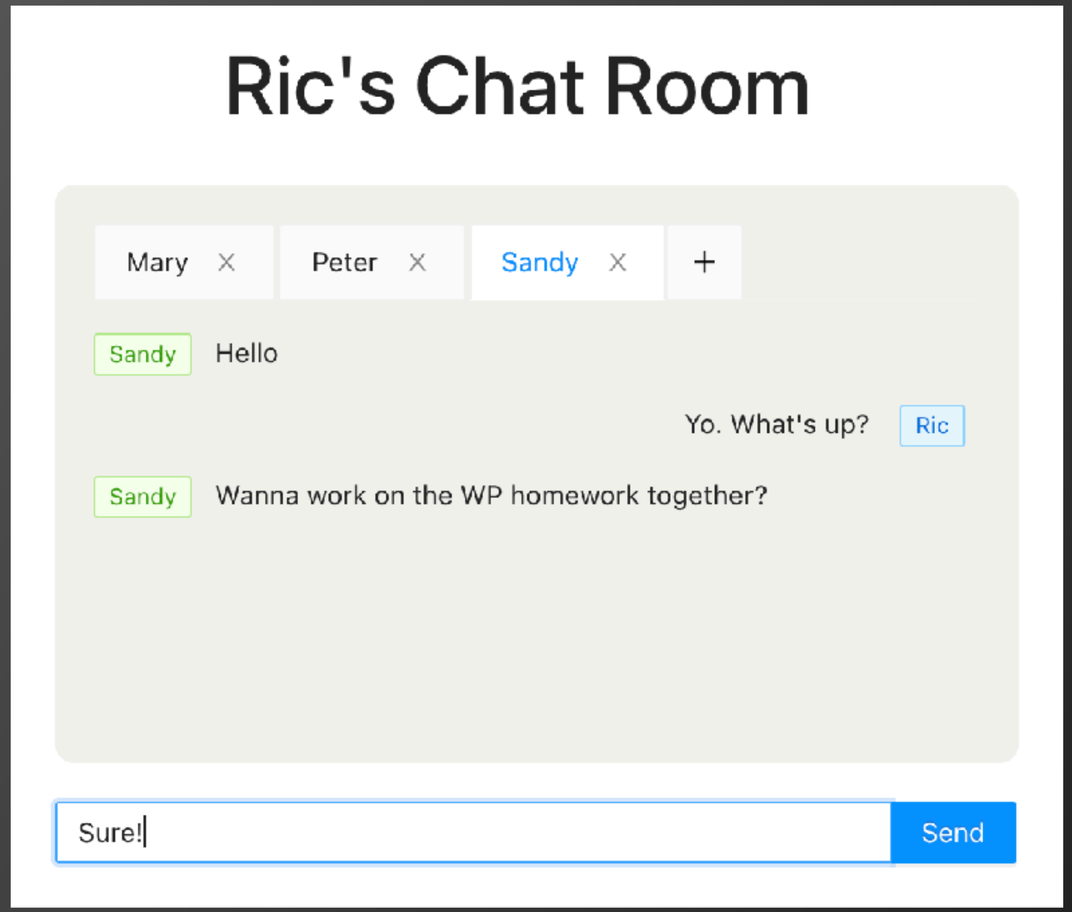
- 用在當 server 上到某人的對話之後，要把它「廣播」(broadcast) 給所有連到 server 的 clients
- 包含 clear 動作也需要 broadcast
- 在 wsConnect.js 中加入 broadcastMessage() function, 並將所有 sendData() 以及 sendStatus() 用 broadcastMessage() 取代

```
const broadcastMessage = (wss, data, status) => {
  wss.clients.forEach((client) => {
    sendData(data, client);
    sendStatus(status, client);
  });
};
```

當然，上述簡單的 chat app,
還有許多還可以進化的地方，包含：

1. 建立登入的機制，區分不同使用者的資料
2. 從聊天室進化成可以指定人聊天的 chat app
3. 使用第三方的 API 服務 (e.g. Google, FB)
4. 更有效率的 DB 存取/搜尋服務 (e.g. GraphQL)

Next Week: HW#7, Lecture Note#9



We are going to build a Chatroom app, in which you can chat with your friends on the same server.

感謝聆聽！

Ric Huang / NTUEE

(EE 3035) Web Programming