COMP1531

📚 Full-Stack - HTTP Servers

Lecture 4.2

Author(s): Hayden Smith

In This Lecture

- **Why?** 🤔
  - Web servers are fundamental part of web-based full-stack software

- **What?** 📰
  - Networks
  - Express Server
  - APIs
  - Crud

🌐 Networks

- **Network**: A group of interconnected computers that can communicate
- **Internet**: A global infrastructure for networking computers around the entire world together
- **World Wide Web**: A system of documents and resources linked together, accessible via URLs

Network

Internet

Web
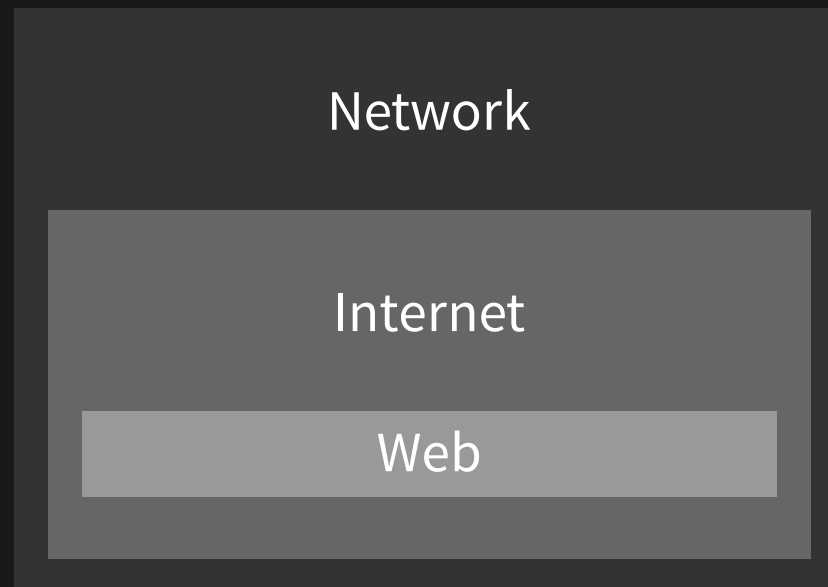
🌐 Networks

If you want to learn more about networking, go and study COMP3331.

🌐 Network Protocols

- Communication over networks must have a certain "structure" so everyone can understand.
- Humans do this all the time - waving, handshakes, clapping. Standard operation procedure that structures how we share info.
- Different "structures" (protocols) are used for different types of communication.
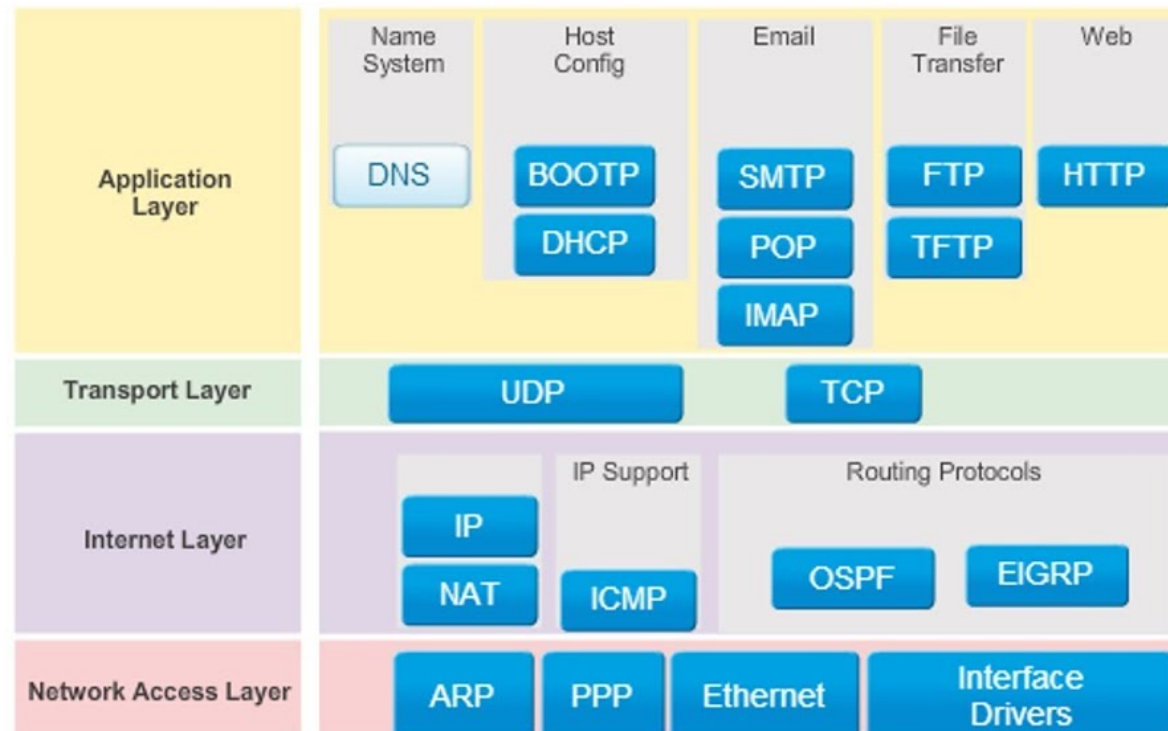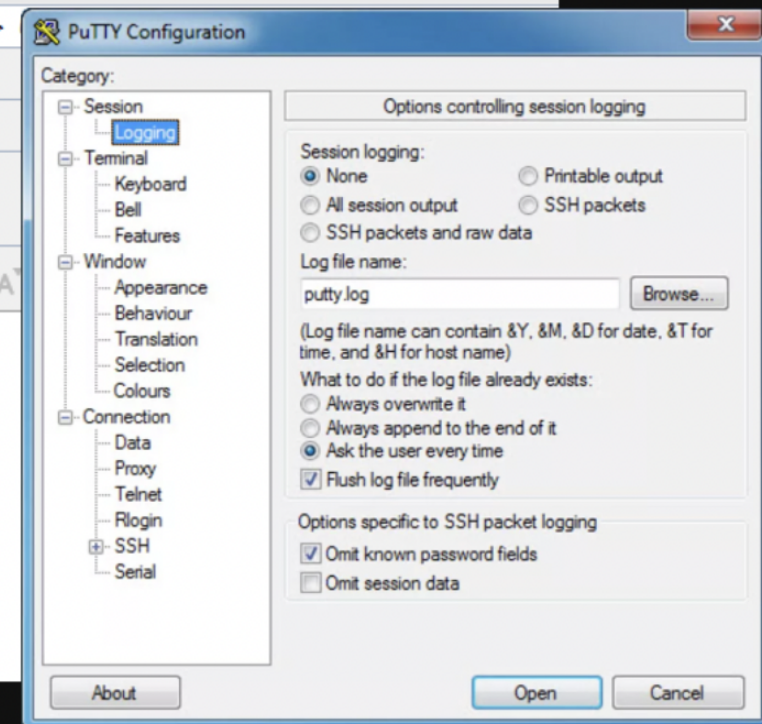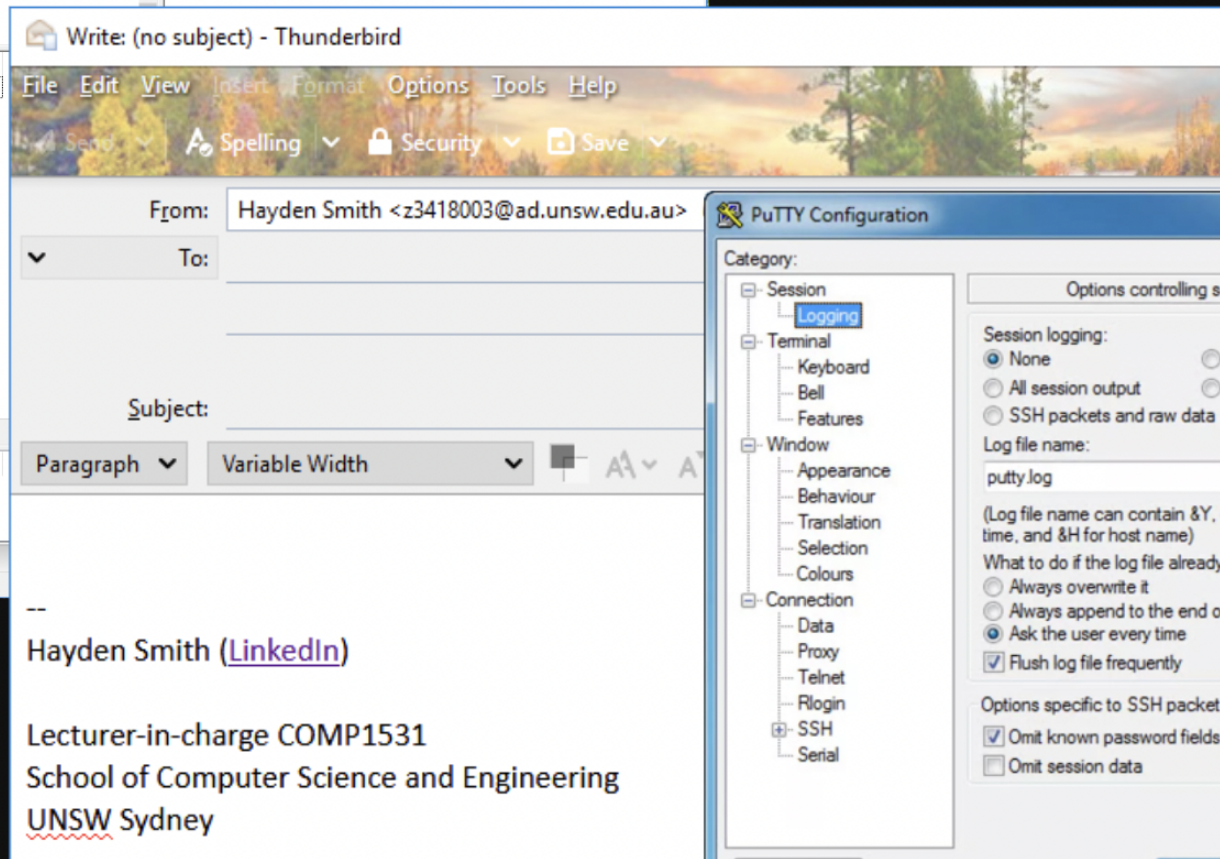
🌐 Network Protocols

source

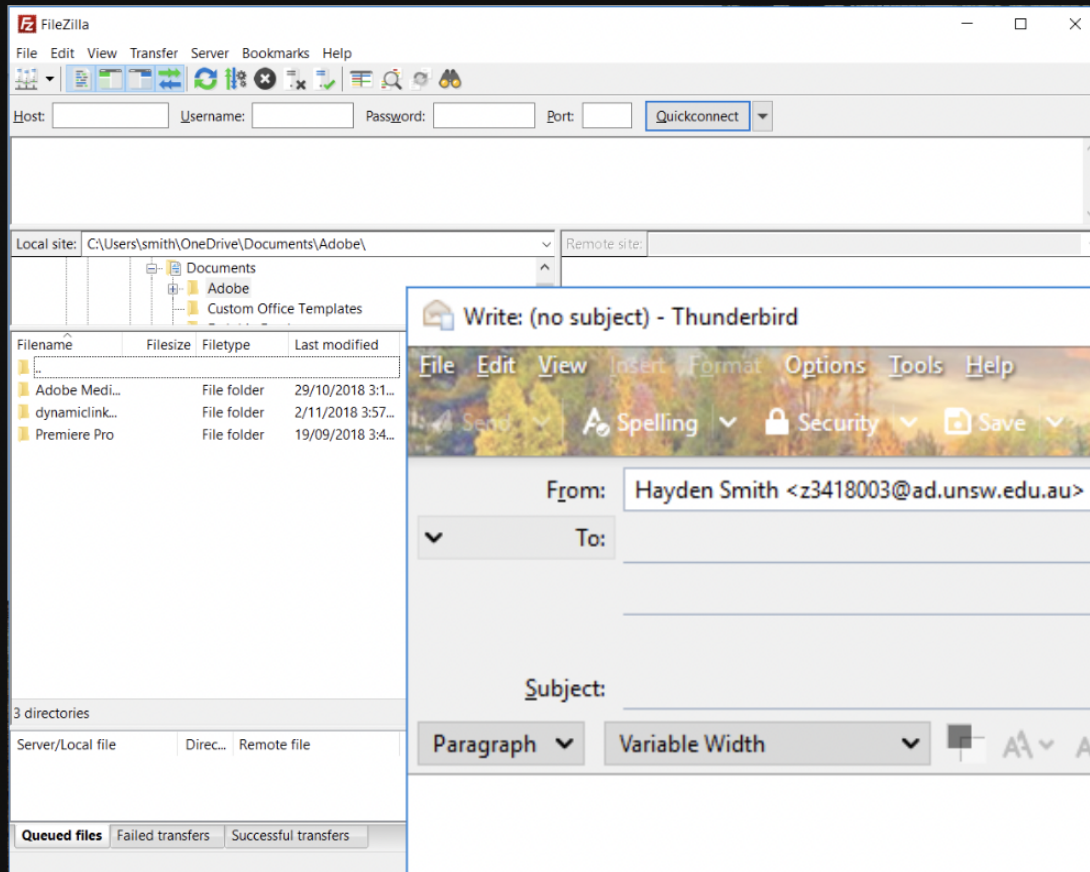HTTP is an example of one of the protocols. It is the protocol of the web. The primary protocol you use to access URLs in your web browser.

Network Protocols

Examples

🕸 HTTP

HTTP: Hypertext Transfer Protocol

I.E. Protocol for sending and receiving HTML documents (nowadays much more)

Web Browsers (Client)   Web Servers

→

←

Web browsers are applications to request and receive HTTP.

🍕 NodeJS Express Server

A very popular npm library exists to allow you to run your own HTTP server with NodeJS.

It's called Express Server.

🍕 NodeJS Express Server

```typescript
import express from 'express';

const app = express();
const port = 3000;

app.use(express.text());

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
```

4.2_express_basic.ts

🍕 NodeJS Express Server

```
1  import express from 'express';
2
3  const app = express();
4  const port = 3000;
5
6  app.use(express.text());
7
8  app.get('/', (req, res) => {
9    res.send('Hello World!');
10 });
11
12 app.listen(port, () => {
13   console.log(`Listening on port ${port}`);
14 });
```

4.2_express_basic.ts

This is us importing the express library

🍕 NodeJS Express Server

```ts
1  import express from 'express';
2
3  const app = express();
4  const port = 3000;
5
6  app.use(express.text());
7
8  app.get('/', (req, res) => {
9    res.send('Hello World!');
10 });
11
12 app.listen(port, () => {
13   console.log(`Listening on port ${port}`);
14 });
```

4.2_express_basic.ts

This creates an instance of a server, and we define the network port to run on.

A port is essentially one of the roads in and out of a computer's network. There are often 65,000-ish.

🍕 NodeJS Express Server

```typescript
 1  import express from 'express';
 2
 3  const app = express();
 4  const port = 3000;
 5
 6  app.use(express.text());
 7
 8  app.get('/', (req, res) => {
 9    res.send('Hello World!');
10  });
11
12  app.listen(port, () => {
13    console.log(`Listening on port ${port}`);
14  });
```

4.2_express_basic.ts

This line is a quirk of `express` that is required in order for the data of many requests to be interpreted.

🍕 NodeJS Express Server

```typescript
1  import express from 'express';
2
3  const app = express();
4  const port = 3000;
5
6  app.use(express.text());
7
8  app.get('/', (req, res) => {
9    res.send('Hello World!');
10 });
11
12 app.listen(port, () => {
13   console.log(`Listening on port ${port}`);
14 });
```

4.2_express_basic.ts

This says that "when URL / is accessed, call this function". The function sends some text to the person accessing that URL.

If we want our server to do more, we need to write lots more of these.

🍕 NodeJS Express Server

```typescript
1  import express from 'express';
2
3  const app = express();
4  const port = 3000;
5
6  app.use(express.text());
7
8  app.get('/', (req, res) => {
9    res.send('Hello World!');
10 });
11
12 app.listen(port, () => {
13   console.log(`Listening on port ${port}`);
14 });
```

4.2_express_basic.ts

This line actually starts the server (on a particular port). It essentially runs an infinite loop so the program runs forever constantly waiting for new people to "access" it via a certain URL.

🍕 NodeJS Express Server

```typescript
 1  import express from 'express';
 2
 3  const app = express();
 4  const port = 3000;
 5
 6  app.use(express.text());
 7
 8  app.get('/', (req, res) => {
 9    res.send('Hello World!');
10  });
11
12  app.listen(port, () => {
13    console.log(`Listening on port ${port}`);
14  });
```

4.2_express_basic.ts

This line actually starts the server (on a particular port). It essentially runs an infinite loop so the program runs forever constantly waiting for new people to "access" it via a certain URL.

Let's take a step back to learn about what servers are used for.
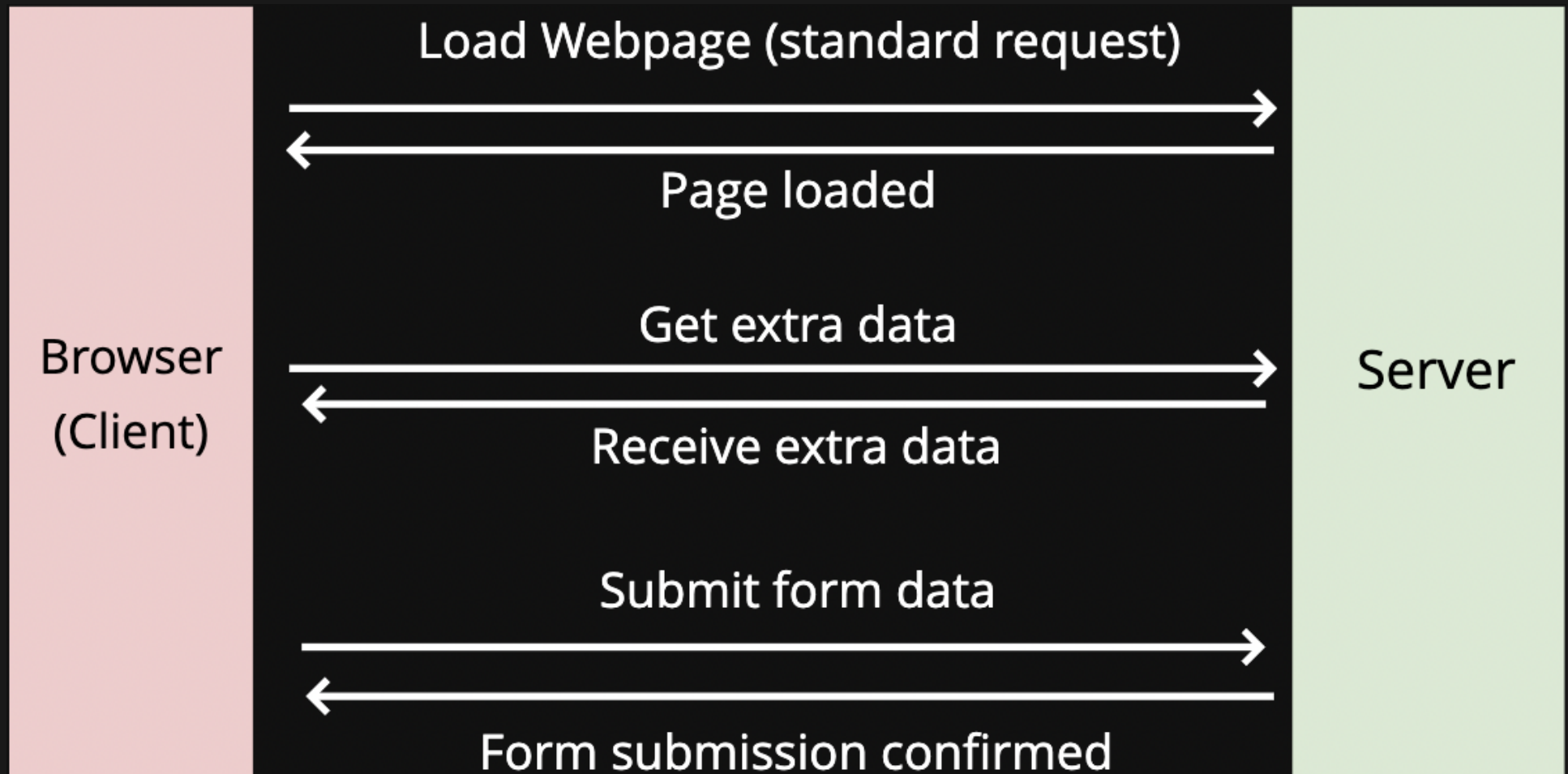
🪟 API

An API (Application Programming Interface) refers to an interface exposed by a particular piece of software.

The most common usage of "API" is for Web APIs, which refer to a "contract" that a particular service provides. The interface of the service acts as a black box and indicates that for particular endpoints, and given particular input, the client can expect to receive particular output.

🪟 Web API

Browser (Client)

Server

Load Webpage (standard request)

→

←

Page loaded

Get extra data

→

←

Receive extra data

Submit form data

→

←

Form submission confirmed

🪟 RESTful API

A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data. These 4 methods describe the "nature" of different API requests.

GET, PUT, POST, DELETE are HTTP Methods. They refer to CRUD operations.

Method	Operation
POST	**C**reate
GET	**R**ead
PUT	**U**pdate
DELETE	**D**elete

🪟 RESTful API

Different CRUD properties require different approaches for input. All output are the same.

```typescript
 1 import express from 'express';
 2
 3 const app = express();
 4 const port = 3001;
 5
 6 app.use(express.text());
 7
 8 app.get('/apple', (req, res) => {
 9   const name = req.query.name;
10   res.send(JSON.stringify({
11     msg: `Hi ${name}, thanks for sending apple!`,
12   }));
13 });
14
15 // same for .put and .delete
16 app.post('/orange', (req, res) => {
17   const body = JSON.parse(req.body);
18   const name = body.name;
19   res.send(JSON.stringify({
20     msg: `Hi ${name}, thanks for sending orange!`,
21   }));
22 });
23
24 app.listen(port, () => {
25   console.log(`Listening on port ${port}`);
26 });
```

4.2_crud.ts

🪟 RESTful API

Different CRUD properties require different approaches for input.

- For inputs:
  - GET|DELETE: via `req.query` (capture URL)
  - PUT|POST: via `req.body` (capture body)
- For outputs:
  - All outputs should be packaged up as JSON

🪟 RESTful API

If we are embrace the use of JSON everywhere, we can make use of other library feature to clean up the code.

```ts
 1  import express from 'express';
 2
 3  const app = express();
 4  const port = 3001;
 5
 6  app.use(express.json());
 7
 8  app.get('/apple', (req, res) => {
 9    const name = req.query.name;
10    res.json({
11      msg: `Hi ${name}, thanks for sending apple!`,
12    });
13  });
14
15  // same for .put and .delete
16  app.post('/orange', (req, res) => {
17    const name = req.body.name;
18    res.json({
19      msg: `Hi ${name}, thanks for sending orange!`,
20    });
21  });
22
23  app.listen(port, () => {
24    console.log(`Listening on port ${port}`);
25  });
```

4.2_crud_json.ts

🪟 RESTful API

If we are embrace the use of JSON everywhere, we can make use of other library feature to clean up the code.

```ts
 1 import express from 'express';
 2
 3 const app = express();
 4 const port = 3001;
 5
 6 app.use(express.json());
 7
 8 app.get('/apple', (req, res) => {
 9   const name = req.query.name;
10   res.json({
11     msg: `Hi ${name}, thanks for sending apple!`,
12   });
13 });
14
15 // same for .put and .delete
16 app.post('/orange', (req, res) => {
17   const name = req.body.name;
18   res.json({
19     msg: `Hi ${name}, thanks for sending orange!`,
20   });
21 });
22
23 app.listen(port, () => {
24   console.log(`Listening on port ${port}`);
25 });
```

4.2_crud_json.ts

🪟 RESTful API

Task

Create a web server that uses CRUD to allow you to create, update, read, and delete a point via HTTP requests

Use a global variable to manage the state.

🔊 Talking To A Web Server

How can we talk to a web server as a client?

1. API client
2. Web Browser
3. An NPM library: `sync-requests`

🔊 API Client (ARC, Postman)

How to download/install ARC:

- Open google chrome
- Google "ARC client"
- Install the addon and open it
- Follow the demo in the lectures

You may need to use a tool like this in the final exam.

📢 API Client (ARC, Postman)

🔊 **Web Browser**

📢 Requests Library

We can use an npm package `sync-requests` to allow us to programatically send RESTful API requests. `npm install sync-requests`.

We can send them to our previous server.

```
1  import request from 'sync-request';
2
3  const res = request(
4    'GET',
5    'http://localhost:3001/apple?name=Hayden'
6  );
7  console.log(JSON.parse(String(res.getBody())));
```

4.2_requests.ts

🔊 Requests Library

We can use an npm package `sync-requests` to allow us to programatically send RESTful API requests. `npm install sync-requests`.

We can send them to our previous server.

```
1  import request from 'sync-request';
2
3  const res = request(
4    'GET',
5    'http://localhost:3001/apple?name=Hayden'
6  );
7  console.log(JSON.parse(String(res.getBody())));
```

4.2_requests.ts

Let's look at the sync-request library to see if we can remove `name=Hayden` from URL.

🍅 Working With jest

```
 1  import request from 'sync-request';
 2
 3  describe('Test Apple', () => {
 4    test('If it returns a name string successfully', () => {
 5      const res = request(
 6        'GET',
 7        'http://localhost:3001/apple',
 8        {
 9          qs: {
10            name: 'Hayden',
11          },
12        }
13      );
14      const bodyObj = JSON.parse(String(res.getBody()));
15      expect(bodyObj.msg).toBe('Hi Hayden, thanks for sending apple!');
16    });
17  });
18  describe('Test Orange', () => {
19    test('If it returns a name string successfully', () => {
20      const res = request(
21        'POST',
22        'http://localhost:3001/orange',
23        {
24          body: JSON.stringify({ name: 'Hayden' }),
25          headers: {
26            'Content-type': 'application/json',
27          },
28        }
29      );
30      const bodyObj = JSON.parse(String(res.getBody()));
31      expect(bodyObj.msg).toBe('Hi Hayden, thanks for sending orange!');
32    });
```

```
33 });
```

4.2_requests.test.ts

🤗 How To Wrap Into Project

In general, iteration 2 requires that you implement an HTTP server. However! Many of the routes that exist in iteration 2 are just wrappers of your iteration 1 functions.

Therefore it should be easy to "wrap" your iteration 1 functions with an HTTP server. I.E. Most of the "server" stuff you'll do is just routing, gathering bodies, and returning responses, while treating your iteration 1 functions like blackboxes.

🎁 Optional! Making Life Easier.

Did you know we can make node auto restart if new files are compiled?

If we:

- Run `npm install --save-dev nodemon`
- Replace `node` with `nodemon` in `package.tson`

Then run `npm run start` in a separate terminal.

🎁 Optional! Making Life Easier.

Did you know we can make `tsc` auto run if source files are changed?

If we:

- Add `--watch` flag to `tsc` command

Then run `npm run tsc` in a separate terminal.

👂 Feedback

Or go to the form here.