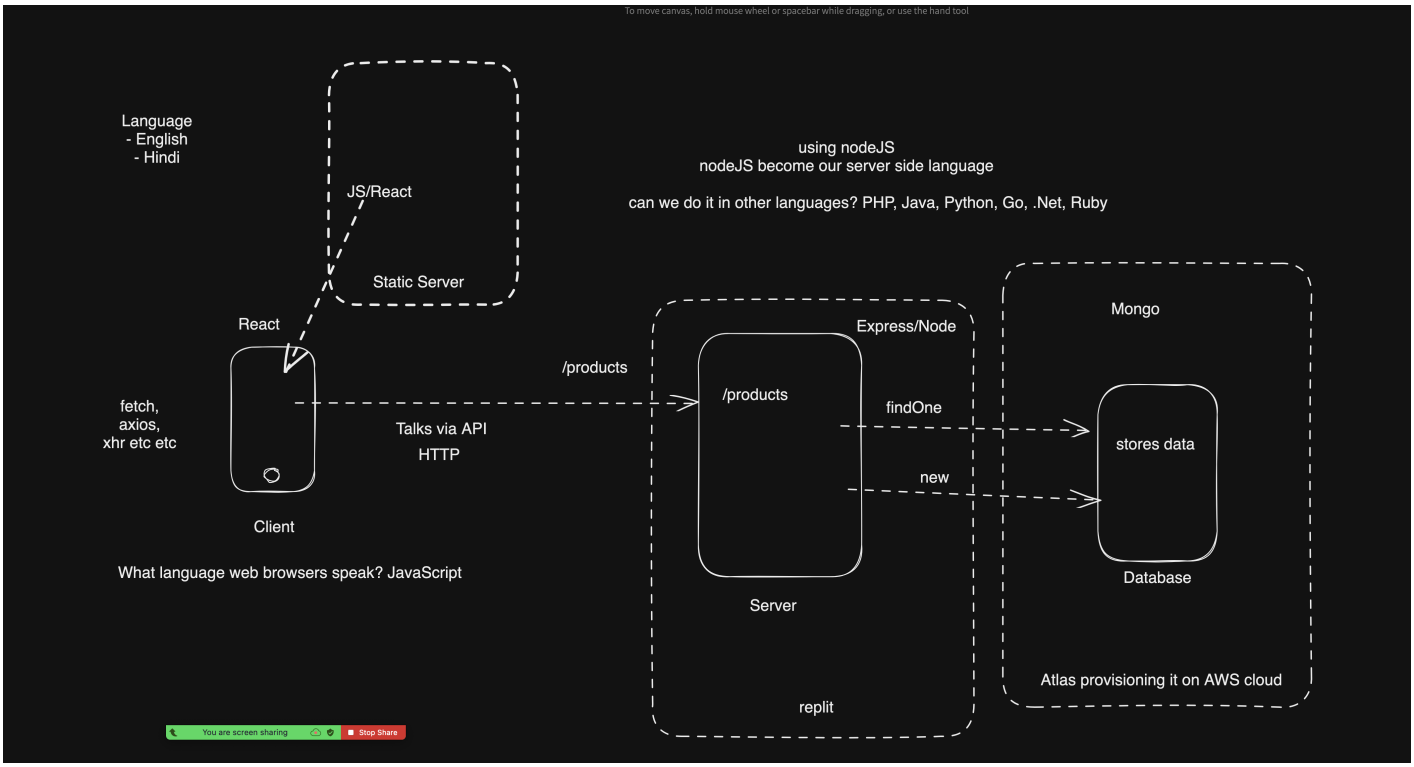
Backend 2.1_CW Exercises

Introduction to MERN stack



Introduction to Express.js

ex01: setting up express.js on replit

challenge

In this exercise, we'll initiate an Express.js project on Replit, getting our environment ready for web development.

1. Open Replit: Log in to your Replit account and create a new project.

2. Install Express: In the terminal, run the command `npm install express` to install the Express.js framework.

3. Create `index.js`: Create a new file named `index.js` in your project's root directory.

4. Require Express: In `index.js`, require the Express module and create an instance of the Express application. Add this to your index.js file:

```
const express = require('express')
const app = express()
```

COPY

ex02: adding a basic GET route

In this exercise, we'll add a simple GET route to our Express.js application and test it using a web browser.

challenge

Add a GET Route on `/`: Create a GET route that responds with a simple message when the root URL `/` is accessed.

Syntax:

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

COPY

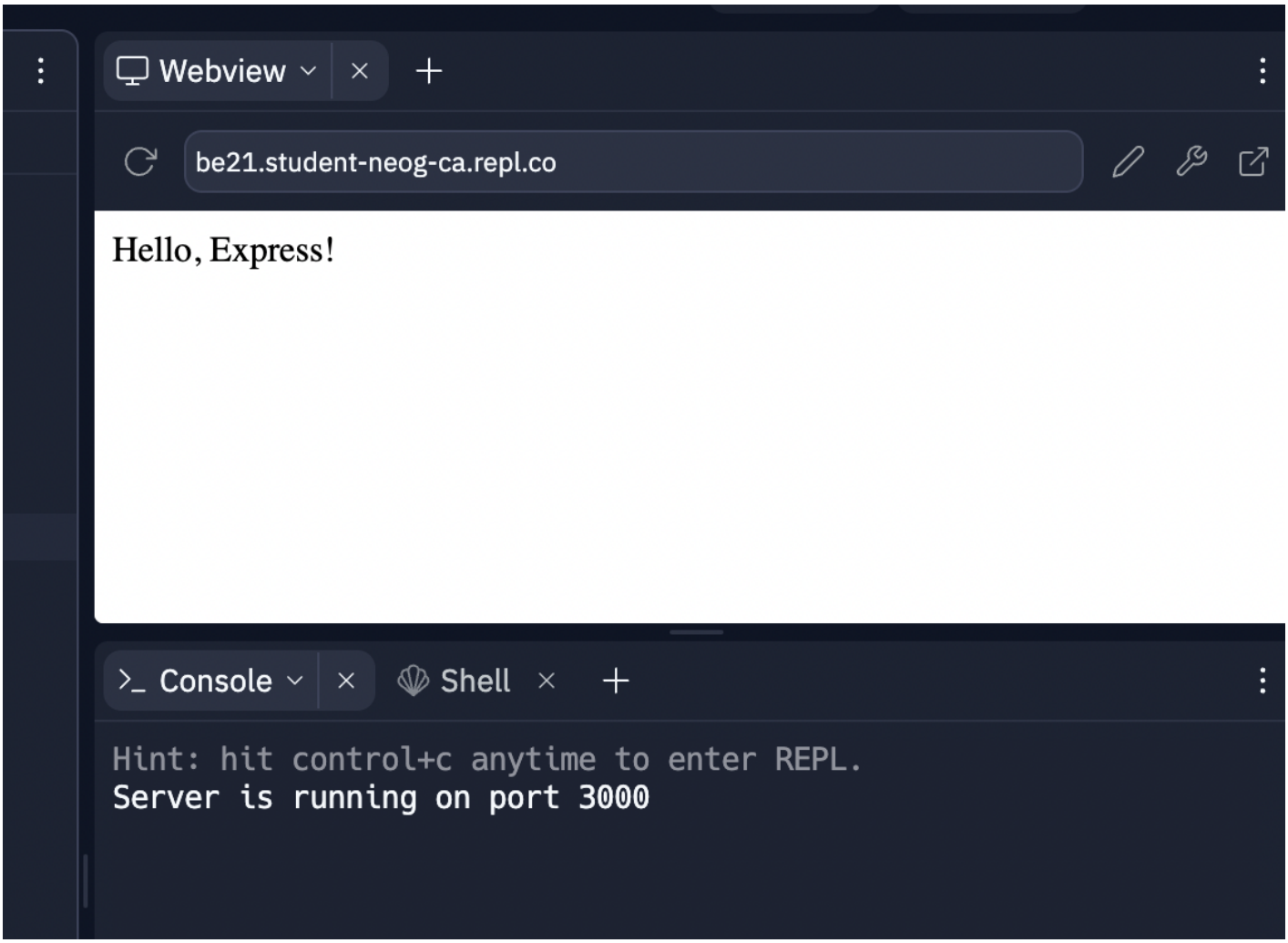
Listen on a Port: Add code to listen on a port (e.g., 3000) and display a message indicating that the server is running.

```
const PORT = process.env.PORT || 3000
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`)
})
```

COPY

testing

Let’s test in Browser: Run your Replit project and access the URL provided in the output console using a web browser. You should see the message "Hello, Express!" displayed.



ex03: creating additional routes

challenge

Add more routes to our Express.js application for the "About" and "Contact" pages.

Create `/about` Route: Add a GET route for the `/about` URL that responds with a message about your application.

Syntax:

```
app.get('/about', (req, res) => {
  res.send('This is the About page.')
})
```

COPY

solution

```
app.get('/about', (req, res) => {
  res.send('This is the About page.')
})
```

COPY

challenge

Create `/contact` Route: Add a GET route for the `/contact` URL that responds with contact information.

solution

```
app.get('/contact', (req, res) => {
  res.send('Contact us at contact@example.com')
})
```

COPY

ex04: demystifying client-server interaction

understanding

Understanding how a client's request is processed by a server is essential in web development. Let's break down the process step by step:

1. Client Sends a Request: A client (usually a web browser) sends a request to a server by specifying a URL in the browser's address bar.
2. Server Decodes the Request: Server receives the request, decodes the URL to identify the desired resource (route), and determines the HTTP verb used.
3. HTTP Verbs: The HTTP verbs (GET, POST, PUT, DELETE, etc.) indicate the type of action the client wants to perform on the resource.
4. GET Requests: A GET request is used to retrieve data from the server. When a client makes a GET request, it's asking the server to provide information.

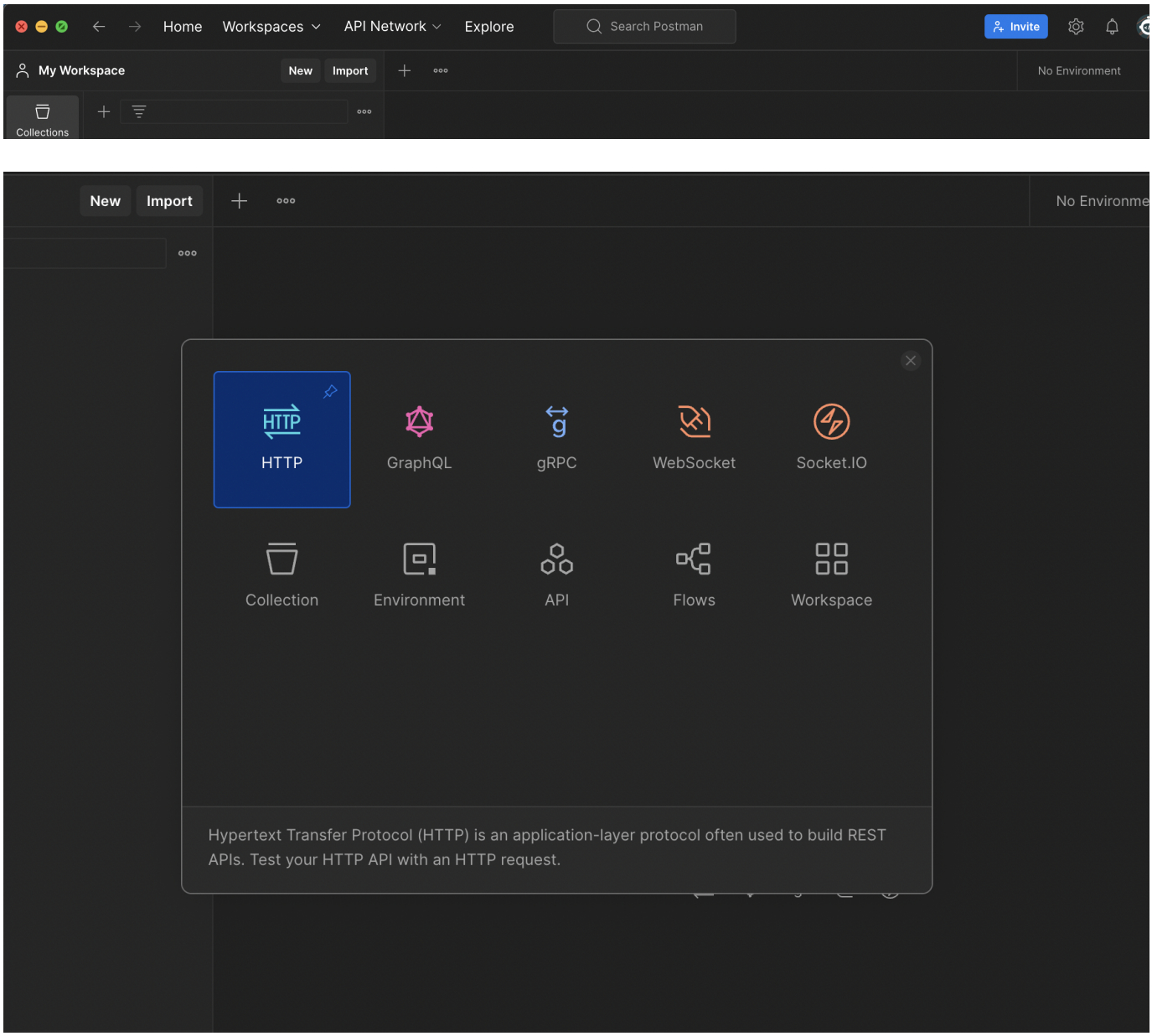
ex05: introducing postman for API testing

Postman is a powerful tool that allows us to test and interact with APIs. Let's see how we can use it to make a GET request to our Express.js routes:

challenge

1. Install Postman: Visit the Postman website and install the Postman app.

2. Create a New Request: Create a new request in Postman by clicking the "New" button and selecting "HTTP"

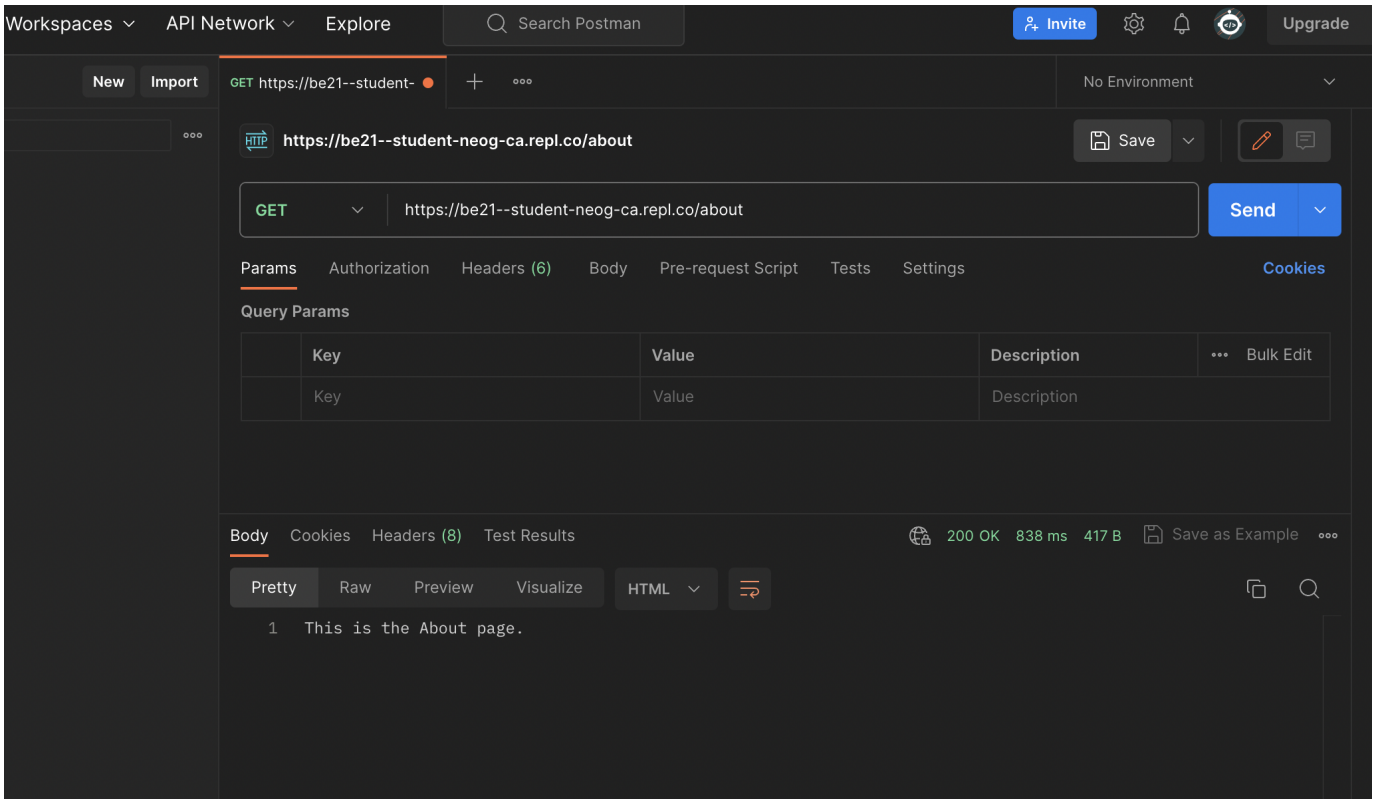


3. Enter Request Details: In the request's URL field, enter the URL of the Express.js app running on Replit, followed by the route you want to test

   (e.g., `https://your-replit-app-url.glitch.me/about`).

4. Select GET: In Postman, select the GET method from the dropdown menu next to the URL field.

5. Send Request: Click the "Send" button to make the GET request. The response will appear in the lower part of the Postman window.



ex06: identifying routes from URLs

Being able to identify routes from URLs is crucial in building and testing web applications. Let's practice identifying routes from different URLs:

challenge

Given the following URLs, identify the routes apart from the domain names:

1. `https://your-replit-app-url.glitch.me/`
2. `https://your-replit-app-url.glitch.me/about`
3. `https://your-replit-app-url.glitch.me/contact`
4. `https://your-replit-app-url.glitch.me/api/users`
5. `https://your-replit-app-url.glitch.me/products/123`

solution

1. /
2. /about
3. /contact
4. /api/users

5. /products/123

ex07: unveiling the MERN stack dynamics

Various components of a MERN stack work together to create a seamless web application experience:

understanding

1. Express as the Backend: Express serves as the backend component of the MERN stack. It handles data-related tasks, communicates with the database (MongoDB), and responds to API requests.
2. MongoDB for Data Storage: MongoDB is the database of choice for storing application data. It's where data is persisted and retrieved by the Express server.
3. React on the Frontend: React is a powerful frontend library that manages the user interface and renders components dynamically. It handles the client-side rendering of views and allows for responsive and interactive user experiences.
4. CDN Hosting for React: React-based applications are hosted on Content Delivery Networks (CDNs). This means that the static assets (HTML, CSS, JavaScript) of the React application are distributed globally for fast and efficient delivery to users.
5. Separation of Concerns: MERN stack follows the principle of separation of concerns. Express handles data and API requests, while React focuses on the user interface. This separation allows for better code organization and maintenance.
6. Communication Between Express and React: While React handles the frontend views, it doesn't directly communicate with Express for static assets. However, React does make API requests to the Express backend to fetch or send data. Express responds to these requests with JSON data.

final solution

https://replit.com/@tanaypratap/BE21CW