HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BÁO CÁO THỰC TẬP CƠ SỞ TUẦN 2
Tìm hiểu sâu về Express.js, Git , Github

Giảng viên hướng dẫn	**: TS. Kim Ngọc Bách**
Họ và tên sinh viên	**: Phạm Trung Kiên**
Mã sinh viên	**: B22DCVT263**
Lớp	**: E22CQCN02-B**

Hà Nội – 2025

I.Express.js

1. What is Express.js?

Express.js is a fast, unopinionated, and minimalist web framework for Node.js that simplifies building web applications and APIs.

Role of Express.js in Node.js development:

- Provides powerful tools to handle HTTP requests/responses.
- Supports Middleware for easy feature extensions.
- Simplifies Routing.
- Integrates well with databases like MongoDB and MySQL.
- Enables template rendering with engines like Pug and EJS.

Why Choose Express.js?

- Reduces boilerplate code compared to native Node.js HTTP modules.
- Provides built-in utilities for request handling.
- Easy to integrate with frontend frameworks like React, Angular, and Vue.js.
- Large community support and frequent updates.

2. Setting up Express.js

Step 1: Install Node.js

Download and install Node.js from the official website: https://nodejs.org/

Step 2: Initialize a Node.js Project

```
- mkdir my-express-app
- cd my-express-app
- npm init -y
```

Step 3: Install Express.js

```
- npm install express
```

Step 4: Create a Basic Express Server

Create a new file index.js and add the following code:

```
- const express = require('express');
- const app = express();
-
- app.get('/', (req, res) => {
- res.send('Hello, Express.js!');
- });
-
- app.listen(3000, () => {
- console.log('Server is running on port 3000');
- });
```

Run the server:

```
- node index.js
```

Open a browser and visit http://localhost:3000 to see the response.

3. Routing in Express.js

Routing in Express.js helps handle HTTP requests like GET, POST, PUT, and DELETE efficiently.

Example of Defining Routes:

```
- const express = require('express');
- const app = express();
-
- app.get('/', (req, res) => {
- res.send('Welcome to the Homepage!');
- });
-
- app.post('/submit', (req, res) => {
- res.send('Form submitted successfully!');
- });
-
- app.put('/update', (req, res) => {
- res.send('Data updated!');
- });
-
- app.delete('/delete', (req, res) => {
- res.send('Data deleted!');
- });
-
- app.listen(3000, () => {
- console.log('Server is running on port 3000');
- });
```

Route Parameters and Query Strings:

```
- app.get('/user/:id', (req, res) => {
- res.send(`User ID: ${req.params.id}`);
- });
-
- app.get('/search', (req, res) => {
- res.send(`Search Query: ${req.query.q}`);
- });
```

4. Middleware in Express.js
Middleware functions process incoming requests before they reach the final route handler.
Example of Custom Middleware:
- app.use((req, res, next) => {
- console.log(`Request Method: ${req.method}, Request URL: ${req.url}`);
- next();
- });

Built-in Middleware:
● express.json(): Parses JSON request bodies.
● express.urlencoded({ extended: true }): Parses URL-encoded bodies.
Third-party Middleware:
● morgan: Logs HTTP requests.
● cors: Enables Cross-Origin Resource Sharing.
Example:
- npm install morgan cors
- const morgan = require('morgan');
- const cors = require('cors');
-
- app.use(morgan('dev'));
- app.use(cors());

5. Connecting Express.js to MongoDB
Using Mongoose to interact with MongoDB.
Step 1: Install Mongoose
- npm install mongoose

Step 2: Connect to MongoDB
- const mongoose = require('mongoose');
-
- mongoose.connect('mongodb://localhost:27017/mydatabase', {
- useNewUrlParser: true,
- useUnifiedTopology: true
- })
- .then(() => console.log('Connected to MongoDB'))
- .catch(err => console.error('Could not connect to MongoDB', err));

Step 3: Define a Mongoose Model
- const UserSchema = new mongoose.Schema({
- name: String,
- email: String,

- age: Number
- });
- const User = mongoose.model('User', UserSchema);

Step 4: CRUD Operations
- app.post('/users', async (req, res) => {
- const user = new User(req.body);
- await user.save();
- res.send(user);
- });
-
- app.get('/users', async (req, res) => {
- const users = await User.find();
- res.send(users);
- });

II. Git và Github

Git Commands

- **Registering a Git User:**
  + Declaring your `user.name` and `user.email` when using Git Bash is necessary because Git needs this information to record the identity of the person performing actions on the repository. When you make a commit, Git attaches this user information to the commit history.

```
git config --global user.name kien
git config --global user.email trungkienpham@gmail.com
```

  + The `--global` flag applies this configuration to all Git projects on your computer.

Check current user configuration:

```
git config user.name
git config user.email
```

- **`cd` Command – Navigate to a Specific Folder or Repository**

The `cd` (Change Directory) command is used to move between folders in your file system. When working with Git, you need to `cd` into the folder containing your Git repository before running Git commands.

```
cd folder_name
cd /c/Users/Trung/Documents/GitHub/MyProject
cd ..      # Go back to the parent folder
cd ~       # Go to the home directory
```

- The `ls` command shows a list of files and folders in the current directory.
- `mkdir`, `touch` – Create Folders and Files
+ mkdir folder_name   # Create a new folder
+ touch file_name.ext   # Create a new file

– `rm`, `rmdir` – Remove Files and Folders
+ rm file_name.ext   # Delete a file
+ rmdir folder_name   # Delete an empty folder

Initialize a Git Repository

- git init

=>This command initializes a new Git repository in the current folder. It creates a `.git` folder that tracks all changes.

Basic Git Concepts

- Repository (repo):
  - A container for code, projects, etc.
  - Two types: local repo and remote repo
  - The `.git` folder is what tracks changes in the repo.

Adding Changes

- `git add .` – Adds all new and modified files, but not deletions
- `git add --all` – Adds all changes: new files, modifications, deletions

Color meanings in Git Bash:

- Green: New file

- Yellow: Modified file

Commit History

- git log   # Full commit history
- git log --oneline   # Simplified one-line format

Undoing Commits

- `git checkout` – View a different branch or commit
- `git revert` – Create a new commit that undoes changes from a previous commit
- `git reset` – Reset the current branch to a specific state

General Git Workflow

1. Configure your username and email
2. Navigate to the folder you want and initialize a local repo
3. Add files to the staging area
4. Commit changes to the local repo
5. Connect the local repo to a remote repo
6. Push code to the remote repo
7. Create a pull request

Working with Branches

- Check branches:

  + git branch
- Create a new branch:

  + git branch new_branch_name
- Switch between branches:

  + git checkout branch_name
- Or with Git 2.23+:

  + git switch -c new_branch_name # Create and switch to new branch

Git Clone

- git clone <url>

=> This command clones a remote repository to your local machine and sets it up as a local repository automatically.

HEAD and Commit Pointer

- `HEAD` is a special pointer representing the current commit you're working on.
- A detached HEAD means you are not on any branch, but viewing a specific past commit.

Upstream Branch

- In Git, an *upstream branch* is the remote branch your local branch is tracking. It tells Git where to push code to or pull code from.

Push and Pull Commands

- git push <remote_name> <local_branch>:<remote_branch>
- git pull <remote_name> <remote_branch>:<local_branch>

Example:

- git push origin main:main