Test-Automation Infrastructure

Our project utilizes the following test automation tools:

- **Mocha & Chai**: For unit and integration testing on the back-end (Node.js/Express).
- **Jest**: For front-end testing of JavaScript logic and React components.
- **Postman/Newman**: For API validation and integration testing.
- **MySQL Test Containers**: To facilitate database integration testing in an isolated environment.

Justification for Choosing These Tools

- **Mocha & Chai**:
    - Widely used for Node.js applications.
    - Supports asynchronous testing.
    - Provides flexible and readable reporting.
    - Requires additional setup for assertions.
- **Jest**:
    - Optimized for JavaScript and front-end testing.
    - Includes built-in mocking capabilities.
    - Fast execution and easy configuration.
    - Can be slower for larger test suites.
- **Postman/Newman**:
    - Facilitates API testing with minimal setup.
    - Provides automation for integration tests.
    - Useful for validating endpoints before deployment.
    - Requires GUI-dependent setup, which can be complex.
- **MySQL Test Containers**:
    - Enables database testing without external dependencies.
    - Ensures data integrity during test execution.
    - Requires Docker for setup and execution.

Adding a New Test

1. **Back-end Test (Mocha & Chai):**
    - Navigate to server/tests directory.
    - Create a new test file, e.g.,transactions.test.js.
    - Use Mocha syntax for writing tests:

```
const chai = require('chai');
const chaiHttp = require('chai-http');
const app = require('../server');
```

```
chai.use(chaiHttp);
describe('Transactions API Tests', () => {
  it('should retrieve all transactions', (done) => {
    chai.request(app)
      .get('/api/transactions')
      .end((err, res) => {
        chai.expect(res).to.have.status(200);
        done();
      });
  });
});
```

2. Front-end Test (Jest):

- Navigate to client/src/__tests__.
- Create a new test file, BudgetTool.test.js.
- Write Jest-based test:

```
import { render, screen } from '@testing-library/react';
import BudgetTool from '../components/BudgetTool';

test('renders budget tool component', () => {
  render(<BudgetTool />);
  expect(screen.getByText(/Set Your Budget/)).toBeInTheDocument();
});
```

CI Service and Repository Linkage

We have chosen **GitHub Actions** as our CI service.

Justification for Choosing GitHub Actions

- Fully integrated with GitHub, making configuration straightforward.
- Free for open-source projects and provides cost-effective CI/CD.
- Supports extensive custom workflows with YAML configuration.
- Runs CI/CD pipelines directly within GitHub, reducing external dependencies.
- Can be slower compared to dedicated CI tools like CircleCI.
- Trade-off: May have slower performance for large projects and limited parallelism on the free tier.

CI Service Comparison

CI Service	Pros	Cons
GitHub Actions	- Deep integration with GitHub - Free for open-source projects - Flexible YAML workflow definitions	- May have slower performance for large projects - Free tier has limited parallelism and minutes
Travis CI	- Long-standing reputation and proven integration with GitHub - Simple configuration with a straightforward .travis.yml file	- Build queues can be slow during peak times - Limited free build minutes for private repositories
CircleCI	- Excellent parallelism and customizable workflows - Good performance with advanced caching options - Detailed build insights and reporting	- Steeper learning curve for configuration - Pricing may be less competitive for larger teams

CI Service Configuration

1. **Creating the CI Workflow**
    - Add a .github/workflows/ci.yml file to the repository:

```
name: CI Pipeline
on:
  push:
    branches:
      - main
  pull_request:
    branches:
```

```
    - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test
```

2. Repository Linkage

- GitHub Actions automatically integrates with our repository.
- We ensure that every push or pull request triggers the CI pipeline.

Tests Executed in CI Builds

- **Unit Tests**: Run using Mocha/Chai and Jest.
- **Integration Tests**: API endpoint validation with Postman/Newman.
- **Database Tests**: Run MySQL test containers to validate queries.
- **End-to-End Tests**: (Future Implementation)

CI Triggers

- **On Push**: Runs the full test suite whenever code is pushed to the main branch.
- **On Pull Request**: Runs the test suite before merging PRs to ensure stability.
- **Scheduled Runs**: (Future Implementation) Nightly builds to detect regressions.

Development Actions Triggering CI:

- Commits, merges, and pull requests all trigger CI builds.
- Future enhancements may include additional triggers for updates to critical configuration files or dependencies.