Fast API

1. Basic FastAPI Application and Path Parameters (1 hour)

Start by initializing a new repository on GitHub. Everyone should clone this repo and create a new branch for their work.
1.1. Create a FastAPI application (a new Python script) with the following routes:

- GET /items/{item_id} : This route should return an item given an id. The item id should be a path parameter. If the item is not found, it should return an HTTP 404 Not Found response.
- POST /items : This route should accept a JSON body to create a new item in the system.

1.2. To simulate a real system, create a list of dictionary objects to act as your items data. Each dictionary should represent an item.

1.3. Use path parameters and type hints to handle different types of inputs. Make sure to run your application and test all routes using an API testing tool (e.g. Postman, curl etc.).

1.4. When done, commit your changes to your local branch and push it to the remote repository. Create a pull request and ask your team members for a code review. After the code has been approved, merge it into the main branch.

Note: Make sure to use meaningful commit messages and include useful descriptions in your pull requests. This will help your teammates understand your changes and give more effective feedback.

Fast API

2. Query Parameters (1 hour)

Start this practice by creating a new branch from the updated main branch.

2.1. Extend your application with the following routes:

- GET /items : This route should return a list of items. Implement pagination using query parameters (e.g., page number and items per page).

- GET /items/search : This route should search for an item given a search term as a query parameter.

2.2. Use query parameters to filter results based on the search term and page number.

2.3. Make sure to test your application ensuring all routes work as expected.

2.4. Similar to Practice 1, once you're done, push your changes, create a pull request, get it reviewed, and merge it into the main branch.

Note: Make sure to use meaningful commit messages and include useful descriptions in your pull requests. This will help your teammates understand your changes and give more effective feedback.

Fast API

3. Pydantic Schemas and Data Validation (45 minutes)

Again, start by creating a new branch.

3.1. Define Pydantic models for your item object. This should be a new class in your Python script.

3.2. Update your POST /items route to use the Pydantic model for data validation. This means the route should accept a Pydantic object as a parameter and FastAPI will automatically validate the request body.

3.3. Update your GET /items/{item_id} and GET /items routes to return Pydantic models instead of dictionary objects.

3.4. Test your application ensuring all routes work as expected.

3.5. Follow the same GitHub flow as before to merge your changes into the main branch.

Note: Make sure to use meaningful commit messages and include useful descriptions in your pull requests. This will help your teammates understand your changes and give more effective feedback.

Fast API

4. Basic Error Handling (45 minutes)

Start with a new branch as usual.

4.1. Implement error handling in your application. If a client sends a request for a non-existing item or with invalid data, your application should return an appropriate HTTP response with a helpful error message.

4.2. Test your application ensuring all routes work as expected.

4.3. Merge your changes into the main branch after code review.

Note: Make sure to use meaningful commit messages and include useful descriptions in your pull requests. This will help your teammates understand your changes and give more effective feedback.

Fast API

5. Jinja Templates (1.5 hours)

Finally, create one last branch for this practice.

5.1. Create a new directory for your Jinja templates. Create two templates: one for an items list page and one for an item detail page.

5.2. Update your GET /items and GET /items/{item_id} routes to return rendered templates instead of JSON responses.

5.3. In your item detail page, implement a form to update the item. The form should POST to a new route you'll create in the next step.

5.4. Create a POST /items/{item_id} route to handle form submission. This route should validate the form data using Pydantic and update the item in your data.

5.5. Test your application ensuring all routes work as expected.

5.6. After getting your code reviewed, merge your changes into the main branch.

Note: Make sure to use meaningful commit messages and include useful descriptions in your pull requests. This will help your teammates understand your changes and give more effective feedback.