The Ultimate To-Do App Project: The Path to becoming the Ultimate Developer

This project is designed you give you an introduction and a chance to explore the latest state of the software development techniques, practices, tools, and frameworks. In this project you are required to build a To-Do List App. Once you have completed all the steps you will have a comprehensive introduction to most of the latest app development technologies. We will use this project in all the App development classes and bootcamps i.e. Web, Hybrid, Android, iOS, Python App Development, Entrepreneurship, IoT, and Magic Leap/Augmented Reality. Happy coding and May the Force be with you.

Note:
Please make a separate repo for each step and continuously push your code to the Github repository.

Part I: Server REST API's

Step 1: Todo RESTful API - A TDD Approach to Building an API using NoSQL Database

Introduction
Testing is an integral part of the software development process, which helps improve the quality of the software. There are many types of testing involved like manual testing, integration testing, functional testing, load testing, unit testing, and other. In this project, we'll write our code following the rules of Test Driven Development (TDD).

What is a unit test?
Martin Fowler defines unit tests as follows:

> Firstly, there is a notion that unit tests are low-level, focusing on a small part of the software system. Secondly, unit tests are usually written these days by the programmers themselves using their regular tools - the only difference being the use of some sort of unit testing framework. Thirdly, unit tests are expected to be significantly faster than other kinds of tests.

In this project step, we'll be building a Todo API using the TDD method with Node.js/Express/Typescript or Python/Flask or Python/Django or Kotlin/Spring Boot or Vapor/Swift and MongoDB as our NoSQL Database. We'll write unit tests for the production code first, and the actual production code later.

Test Driven Development
Typical of all our web apps, we'll use the TDD approach. It's really simple. Here's how we do Test Driven Development:

- Write a test. – The test will help flesh out some functionality in our app
- Then, run the test – The test should fail, since there's no code(yet) to make it pass.
- Write the code – To make the test pass
- Run the test – If it passes, we are confident that the code we've written meets the test requirements
- Refactor code – Remove duplication, prune large objects and make the code more readable. Re-run the tests every time we refactor our code
- Repeat – That's it!

What is REST?
The characteristics of a REST system are defined by six design rules:

- **Client-Server**: There should be a separation between the server that offers a service, and the client that consumes it.
- **Stateless**: Each request from a client must contain all the information required by the server to carry out the request. In other words, the server cannot store information provided by the client in one request and use it in another request.
- **Cacheable**: The server must indicate to the client if requests can be cached or not.
- **Layered System**: Communication between a client and a server should be standardized in such a way that allows intermediaries to respond to requests instead of the end server, without the client having to do anything different.
- **Uniform Interface**: The method of communication between a client and a server must be uniform.
- **Code on demand**: Servers can provide executable code or scripts for clients to execute in their context. This constraint is the only one that is optional.

What is a RESTful web service?
The REST architecture was originally designed to fit the HTTP protocol that the world wide web uses.
Central to the concept of RESTful web services is the notion of resources. Resources are represented by URIs. The clients send requests to these URIs using the methods defined by the HTTP protocol, and possibly as a result of that the state of the affected resource changes.
The HTTP request methods are typically designed to affect a given resource in standard ways:

HTTP Method	Action	Examples
GET	Obtain information about a resource	http://example.com/api/orders (retrieve order list)
GET	Obtain	http://example.com/api/orders/123

	information about a resource	(retrieve order #123)
POST	Create a new resource	http://example.com/api/orders (create a new order, from data provided with the request)
PUT	Update a resource	http://example.com/api/orders/123 (update order #123, from data provided with the request)
DELETE	Delete a resource	http://example.com/api/orders/123 (delete order #123)

The REST design does not require a specific format for the data provided with the requests. In general data is provided in the request body as a JSON blob, or sometimes as arguments in the query string portion of the URL.

Designing a simple todo web service
The task of designing a web service or API that adheres to the REST guidelines then becomes an exercise in identifying the resources that will be exposed and how they will be affected by the different request methods.

Let's say we want to write a To Do List application and we want to design a web service for it. The first thing to do is to decide what is the root URL to access this service. For example, we could expose this service as:

http://[hostname]/todo/api/v1.0/

Here I have decided to include the name of the application and the version of the API in the URL. Including the application name in the URL is useful to provide a namespace that separates this service from others that can be running on the same system. Including the version in the URL can help with making updates in the future, since new and potentially incompatible functions can be added under a new version, without affecting applications that rely on the older functions.
The next step is to select the resources that will be exposed by this service. This is an extremely simple application, we only have tasks, so our only resource will be the tasks in our to do list.

Our tasks resource will use HTTP methods as follows:

HTTP Method	URI	Action
GET	http://[hostname]/todo/api/v1.0/tasks	Retrieve list of tasks
GET	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Retrieve a task
POST	http://[hostname]/todo/api/v1.0/tasks	Create a

		new task
PUT	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Update an existing task
DELETE	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Delete a task

We can define a task as having the following fields:

- **id**: unique identifier for tasks. Numeric type.
- **title**: short task description. String type.
- **description**: long task description. Text type.
- **done**: task completion state. Boolean type.

And with this we are basically done with the design part of our web service. All that is left is to implement it!

References:
https://kotlinlang.org/docs/tutorials/spring-boot-restful.html
https://github.com/vapor/vapor

Development Languages:
You can do the project using TypeScript/Node.js or/and Python/Flask.

Note: Please continuously push your code to a Github repository.

Step 2: Todo RESTful API - A TDD Approach to Building an API using SQL Database

In this project step, we'll be port our Todo API using the TDD method in the step 1 to using PostgreSQL as our SQL Database.

Reference:
https://www.postgresql.org/
https://blog.panoply.io/postgresql-vs.-mysql

Note: Please continuously push your code to a public Github repository.

Step 3: Todo gRPC API Project: A TDD Approach to Building an API using SQL and NoSQL Databases

Introduction

"gRPC is a modern, open source remote procedure call (RPC) framework that can run anywhere. It enables client and server applications to communicate transparently, and makes it easier to build connected systems." Some frequently asked questions are answered here. This step explains a little bit about gRPC,

why we chose it, how we implement it, and some basic tips and lessons we've learnt along the way.

At Panacloud we sometimes chose gRPC over JSON/HTTP RESTful APIs for multiple reasons:

1. JSON encoding/decoding/transmission was a significant portion of latency profiling
2. When managing many microservices, REST was too loosely defined and promoted mistakes
3. HTTP2 by default

The above can be loosely summarised as speed, capabilities and more robust API definition. Below is an outline of each of these and why they are important to us. We believe gRPC will end up replacing many REST based services as the advantages are too many once the initial concepts are grasped.

gRPC is very fast
gRPC is faster than REST. Others have documented this, but in a nutshell, it runs on HTTP2 by default (see a comparison by Brad Fitz) and when using Google's protocol buffers (you don't have to) for encoding, the information comes on an off the wire much faster than JSON. There's also bidirectional streaming, header compression, cancellation propagation and more.
Our backend services are very low-level optimized, to the point where using JSON encoding was actually a noticeable portion of the latency when profiling. Other applications that don't care about speed might not worry about this, but for a distributed and data heavy application this actually makes a noticeable difference.

gRPC is robust
RPC allows a connection to be maintained between two machines for the purpose of executing functions remotely. Where as in REST calls the request and response are totally de-coupled: you can send anything you like and hopefully the other end understands what to do with it. RPC is the opposite, it effectively defines a relationship between two systems and enforces strict rules which govern communication between them. That can be pretty annoying at first, as you need to update both the server and the client to keep things working when you make changes, but after a while that actually becomes invaluable as it prevents mistakes. This is particularly evident when your backend is comprised of many microservices, which is the case for us in Panacloud.

SDK generation is easy
gRPC generates pretty nice basic SDK skeletons. It also supports all the functionality we needed like nesting, so that was a great start.

Backend compatibility
Another key advantage is we wanted to use gRPC internally for our backend service communication, so it was nice to have the same technology across both internal and external APIs.

Other advantages of gRPC
gRPC supports useful additions like standard error responses and meta data. Meta data is particularly useful for us, as it's possible to embed things like data access details, auth, SDK version info and other useful information that cascades through many backend services.
gRPC is designed to support mobile SDKs for Android and iOS. Keeping a communication pipe open between a phone app and backend service is very useful for creating low latency mobile applications.
Tracing is also built in, which is extremely useful. Because requests can be tracked using the same context through multiple services, it's possible to cancel requests on different systems and or trace them to see what is causing delays, etc.

References:
https://grpc.io/
https://github.com/grpc/grpc-web

Now your job in this step is to build a gRPC version of the Todo API that you built in step 1 and 2.

Note: Please continuously push your code to a Github public repository.

Part II: Client Apps

Step 1: Build a Simple in Memory Todo Client App using the TDD Approach

We are going to start off by building very simple in memory Todo list app for the Web (PWA) or Android or iOS or Magic Leap platforms. The data is not required to be persisted either on the client or the server. Every time we restart the app it will contain no entries. You are also required to write tests for your app.

Step 2: Build a persistent Simple client Todo Client App using the TDD Approach

In this step we will build very simple in Todo list app for the Web or Android or iOS or Magic Leap platforms and persist data only on the client side. In case of web app it will use IndexedDB, Room for Android, and Core Data for iOS.

References:
https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

Step 3: Build a Simple Todo Client App using the TDD Approach

In this step we will build very simple in Todo list app for the Web or Android/Kotlin or iOS/Swift or Magic Leap platforms and using the server side REST and gRPC API's developed in Part 1.

Step 4: Build a UI Component based Todo Client App using the TDD Approach

In this step we will build a reusable UI component based Todo list app for the Web or Android or iOS or Magic Leap platforms and using the server side REST and gRPC API's developed in Part 1. You can use the latest versions of React or Angular component technologies for web development. You should use ReactiveX API for asynchronous programming.

Reference:
http://reactivex.io/
https://proandroiddev.com/mvvm-with-kotlin-android-architecture-components-dagger-2-retrofit-and-rxandroid-1a4ebb38c699
https://proandroiddev.com/clean-architecture-on-android-using-feature-modules-mvvm-view-slices-and-kotlin-e9ed18e64d83
https://www.twilio.com/blog/2018/06/build-reusable-ios-components-swift.html

Note: When I say Web App it means Progressive Web App (PWA).

Part III: Realtime App

Build a Simple Realtime Todo Client App using the TDD Approach

In this step we will build very simple in Todo list app for the Web or Android or iOS or Magic Leap platforms and using Google's Firebase Firestore Database. The app should be deployed using Firebase hosting service in case of the web app.

Reference:
https://firebase.google.com/products/firestore/

Part IV: Serverless REST API's

Step 1: Todo Serverless RESTful API - A TDD Approach to Building an API using SQL and NoSQL Database

In this step we will rebuild the RESTful API using the TDD approach using the Serverless technologies. You have the option to build it using either the AWS Lambda Functions or Cloud Functions for Firebase or Google Cloud Functions.

Note:
AWS Lambda supports code written
in **Node.js** (**JavaScript/Typescript**), **Python**, **Java** (**Java 8** compatible), and C# (.**NET** Core) and Go. Your code can include existing libraries, even native ones.
Google Cloud Functions code is written in Node.js
Google Cloud Functions can be written in Node.js and Python, and are executed in language-specific runtimes.

Reference:
https://www.manning.com/books/serverless-architectures-on-aws-second-edition
https://aws.amazon.com/lambda/
https://aws.amazon.com/serverless/developer-tools/#Frameworks
https://serverless.com/
https://firebase.google.com/docs/functions/http-events
https://cloud.google.com/functions/docs/writing/

Step 2: Build CI/CD Pipeling for Todo Serverless RESTful API

In this step we will build a CI/CD pipeline which will allow Serverless API developed in previous Step to be committed to Github and automatically updated to the selected Serverless cloud.

Part V: Containerized Server API's

Step 1: Containerizing the RESTful and gRPC API's

Docker is all about taking applications and running them in containers. The process of taking an application and configuring it to run as a container is called "containerizing". Sometimes we call it "Dockerizing".

In this step you'll go through the process of containerizing the API's developed in Step 1 and 2.

References:
https://www.docker.com/
https://www.amazon.com/Docker-Deep-Dive-Nigel-Poulton/dp/1521822808/ref=sr_1_1

Note: Please continuously push your code to a Github public repository.

Step 2: Deploying the RESTful and gRPC API's Containers to Heroku Cloud

Your next job is to deploy your containerized API's to Heroku cloud and test them.

Use MongoDB Atlas for hosting the NoSQL Database:
https://www.mongodb.com/cloud/atlas

For SQL Database use Heroku Postgres:
https://www.heroku.com/postgres

References:

https://devcenter.heroku.com/articles/container-registry-and-runtime

https://blog.heroku.com/container-registry-and-runtime

https://medium.com/travis-on-docker/how-to-run-dockerized-apps-on-heroku-and-its-pretty-great-76e07e610e22

Step 3: Build CI/CD Pipeline Using GitHub, Docker, CircleCI, and Heroku

In this step you will setup a Continuous Integration and Continuous Deployment (CI/CD) pipeline with CircleCI, Docker, and Heroku. Every time you will make a commit to Github to the code that you developed in step 3B , it should immedately be deployed to Heroku, and should be available to the customers.

References:
https://circleci.com/docs/2.0/
https://dzone.com/articles/cicd-pipeline-using-github-docker-circleci-amp-her
https://gist.github.com/mlabouardy/bb4f8b467f6b3bc6a2ab229e15ff8fb8

Part VI: Fargate Server API's

Step 1: Deploying the RESTful and gRPC API's Containers to Fargate in the AWS Cloud

Your next job is to deploy your containerized API's to Fargate an AWS Cloud Service and run all the tests on them.

References:
https://aws.amazon.com/fargate/
https://dzone.com/articles/deploy-docker-image-to-fargate

Step 2: Build CI/CD Pipeline Using GitHub, Docker, and AWS Fargate

This step you will setup a Continuous Integration and Continuous Deployment (CI/CD) pipeline with Github, Docker, and Heroku. You can use any tool for CI/CD. Every time you will make a commit to Github the code that you developed in step 3C should be immediately deployed to AWS Fargate, and should be available to the customers.

Part VII: Cloud Native App

Step 1: Deploying the RESTful and gRPC API's Containers to Kubernetes in the AWS Cloud or Google Cloud

Your next job is to deploy your containerized API's to Kubernetes in the AWS Cloud or Google Cloud and run all the tests them.

References:
https://cloud.google.com/kubernetes-engine/
https://aws.amazon.com/eks/
https://www.manning.com/books/kubernetes-in-action

Step 2: Building and Deploying Microservices using CI/CD in Kubernetes in the AWS Cloud or Google Cloud

What are microservices?
Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of loosely coupled services, which implement business capabilities. The microservice architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

Microservices are becoming a new trend, thanks to the modularity and granularity they provide on top of advantages like releasing applications in a continuous manner.

Keeping that in mind, we will make a simple application that can give an example of how microservices are built and how they interact. In this step, you will be building a small application using the Microservice Architecture (MSA). The application will be a super simple To-Do management list.

it is leveraging the MSA where the whole application is divided into a set of services that specialize in doing a specific task using a simple set of protocols. All the communication between different services occur over the network.

So, let's talk about the architecture of the application. Our application will consist of two services namely the User service and the To-Do service:

- User Service: The user service provides a RESTful endpoint to list the users in our application and also allows to query the user lists based on their usernames.
- To-Do Service: The ToDo service provides a RESTful endpoint to list all the lists as well as providing the list of projects filtered on the basis of usernames.

Now, for building our Application, we will be making use of Node.js/Express/Typescript or Python/Flask or Python/Django or Kotlin/Spring

Boot or Vapor/Swift. To demonstrate that different microservices can use different technologies the requirement is that you use different technology stacks to develop the two microservices mentioned above. For example, you can use the Python and SQL database combination to develop one service, and TypeSceipt and NoSQL to build the other microservice.

You are also required to build a CI/CD pipeline and use the TDD approach to building these microservices. These microservices should be continuously deployed to a Kubernetes cloud.

Step 3: Build CI/CD Pipeline Using GitHub, Docker, and Kubernetes

This step you will setup a Continuous Integration and Continuous Deployment (CI/CD) pipeline with Github, Docker, and Kubernetes. You can use any tool for CI/CD. Every time you will make a commit to Github the code that you developed in step 3D should be immediately deployed to Kubernetes, and should be available to the customers.

Step 4: Building and Deploying Microservices using Kubernetes, Istio and Jenkins

Istio is a joint effort launched just over a year ago by Google, IBM and Lyft to create an open technology framework to connect, secure, manage and monitor networks of cloud microservices. Each of the three participants contributed existing technologies they had developed separately. In this step your job is to deploy the microservices developed in Step 3 to Istio and Kubernetes.

Reference:
https://diginomica.com/2018/08/03/google-istio-bigger-kubernetes-serverless/
https://cloud.google.com/blog/products/gcp/istio-reaches-1-0-ready-for-prod
https://cloudplatform.googleblog.com/2018/07/cloud-services-platform-bringing-the-best-of-the-cloud-to-you.html
https://istio.io/docs/concepts/what-is-istio/

Step 5: Building and Deploying Microservices using Kubernetes, Istio, and Jenkins in a Private Cloud

As you know Kubernetes is an open-source specification as well as a reference implementation for container orchestration. Applications can be built, tested, and deployed via continuous integration and delivery pipelines using an integration of Kubernetes and Jenkins pipelines. The open-source nature of the full stack allows deployment to public as well as private clouds.

In the step you are required to build your own private cloud on your own server machines using Kubernetes and Jenkins. You will deploy the microservies you developed in step 4E to this private cloud with Istio.

Part VIII: Professional Todo App

Step 1: Design a Professional Todo App

You are required to design a professional Todo list app for the Web/Mobile Web (Responsive) or Android or iPhone or Magic Leap. It should have most of the functionality of the Microsoft To-Do App. Special attention should be given to the UI/UX and ease of use. Good design is a requirement. You are required to develop your design using Adobe XD. Adobe XD is a user experience design software application. It supports vector design and wireframing, and creating simple interactive click-through prototypes

Reference:
https://todo.microsoft.com/
https://www.adobe.com/products/xd.html
https://www.cnet.com/news/adobe-xd-experience-design-tool-now-available-free-creative-cloud/

Step 2: Build a Professional Todo App using Microservices and Reusable UI Components

You are required to build the professional Todo list app for the Web/Mobile Web (Responsive) or Android or iPhone or Magic Leap that you designed in Step 1. It should use Microservices architecture and Kubernetes on the server side and any reusable UI component technology on the client side. You can use the latest versions of React or Angular component technologies for web development. You should use ReactiveX API for asynchronous programming.

Reference:
http://reactivex.io/
https://proandroiddev.com/mvvm-with-kotlin-android-architecture-components-dagger-2-retrofit-and-rxandroid-1a4ebb38c699
https://proandroiddev.com/clean-architecture-on-android-using-feature-modules-mvvm-view-slices-and-kotlin-e9ed18e64d83
https://www.twilio.com/blog/2018/06/build-reusable-ios-components-swift.html

Step 3: Deploy Professional Todo App to the Public Cloud

In this step you are required to deploy the app built in step 2 to a public Kubernetes cloud. A CI/CD pipeline must also be established.

Step 4: Deploy Professional Todo App to the Public Cloud using Kubernetes and Istio

In this step you are required to deploy the app built in step 2 to a public Kubernetes/Istio cloud. A CI/CD pipeline must also be established.

Step 5: Deploy Professional Todo App to the Private Cloud

In this step you are required to deploy the app built in step 2 to a private Kubernetes, Istio and Jenkins cloud. You will build this private cloud using your own machines on your own premises.

//****** Given on Web, Sep 12 to Bootcampers

Panacloud $10K Bootcamp and Developer Training: The Path to Create the Ultimate Developers

To All Panacloud Sharia Faisal Developers and $10K Bootcampers Karachi and Faisalabad Teams:
My job is to train you be able to earn $10,000 per month ASAP, therefore I am going to push you to the extreme, no mercy on anyone. Either you will become a world class developer or you will run away and dropout.

Today everyone will appear in the Linux exams after only two days of preparation.

After your Linux exam your next tasks will be to do some coding. All coding will be done in teams.

Each team leaders will create a public GitHub repo and complete Parts I, II, and III of the following Ultimate Todo List Project before next Friday 6:00 pm (You have about 10 days):
https://www.dropbox.com/s/dg1ugsur6bugybe/The%20Ultimate%20To%20Do%20List%20Project.docx?dl=0

Each team will give me the URL of Github repo, which I will publish on Facebook. I and others will continuously monitor the project progress and what contribution each team member has made in the repo. We will monitor the number of pushes as well as quality of code pushes. All code should be properly documented.

We will continually calculate a Cryptowit Developer Skill Index (Wit-Skill-Index), which will show to the world how good a developer you are.

The server code should be written in TypeScript/Node.js and/or Python, depending on the developers in the team. If you have both Node.js and Python developers in the team you are required to do both.

The client code can be written in TypeScript/JavaScript or Swift. Addition points of doing it in React or Angular. If you have both Web and iOS developers in the team you are required to do both.