Fullstack with

GRAPHQL, PRISMA, REACT AND APOLLO BOOST

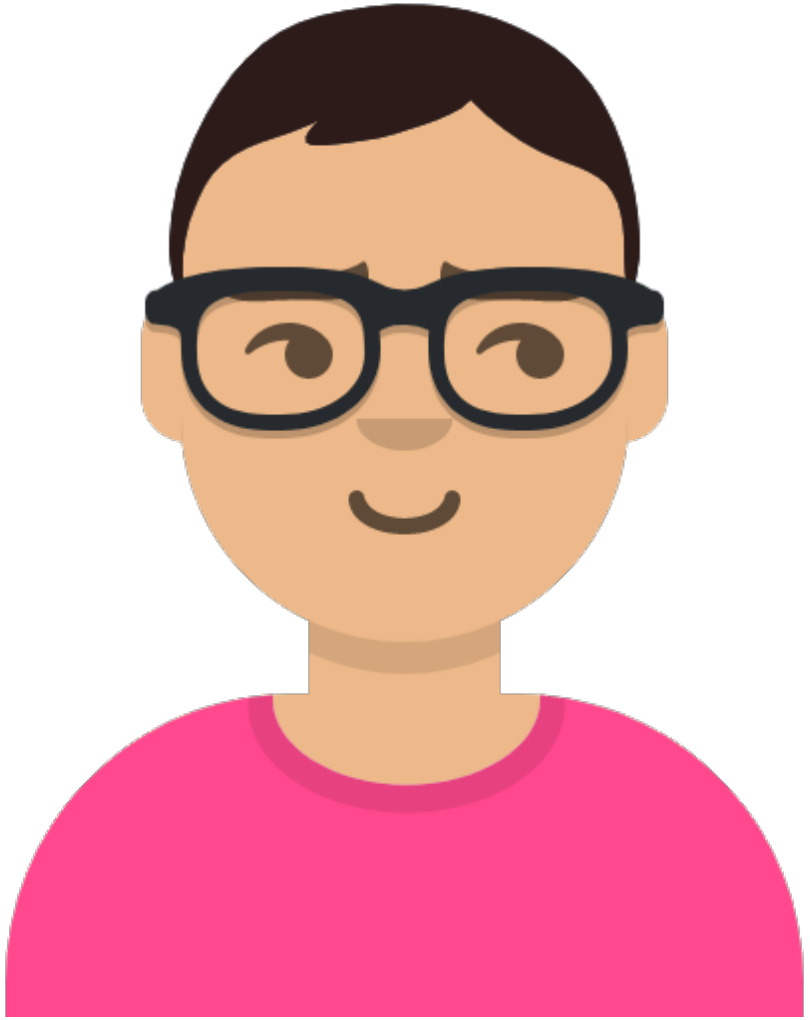
Somprasong Damyos

Technical Manager @Open Source Technology Co., Ltd.

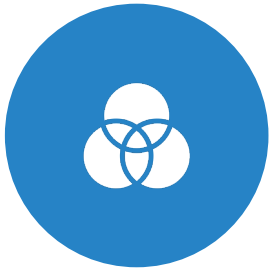
@somprasongd

@somprasongd

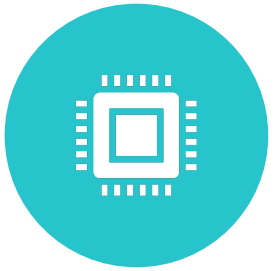
@somprasongd



Agenda



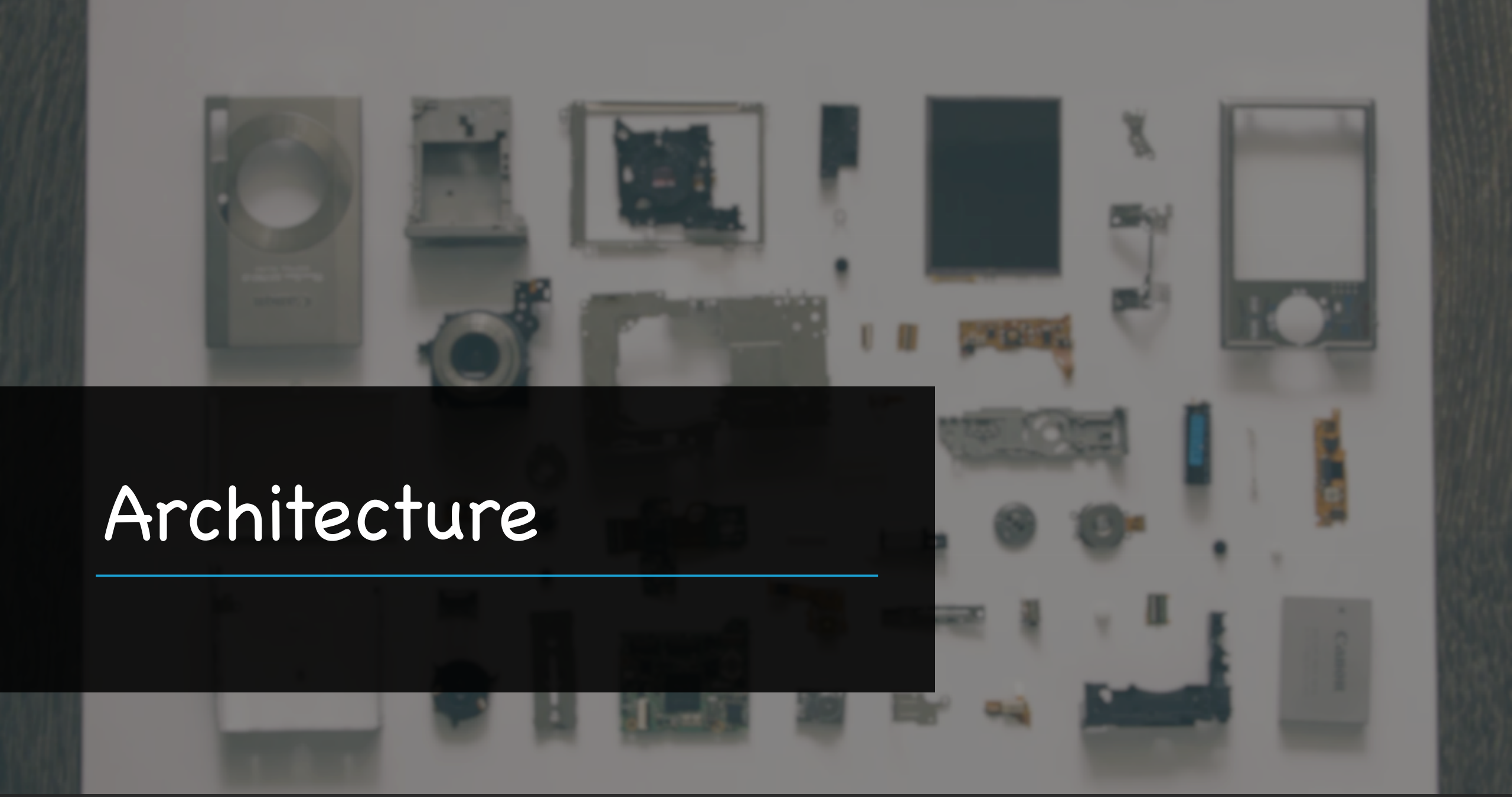
ARCHITECTURE



BUILD GRAPHQL SERVER

DATA ACCESS LAYER WITH PRISMA

CREATE FRONTEND APP WITH REACT AND APOLLO BOOST



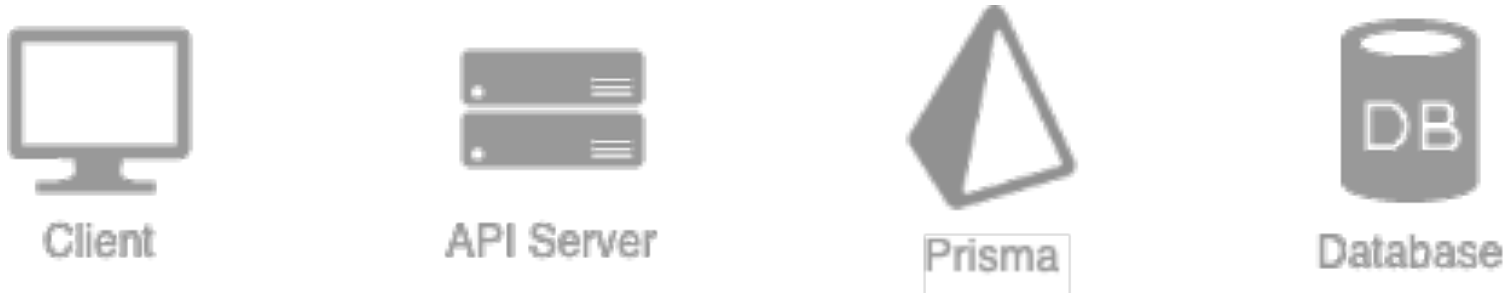
Architecture

Typical 3-tier Architecture

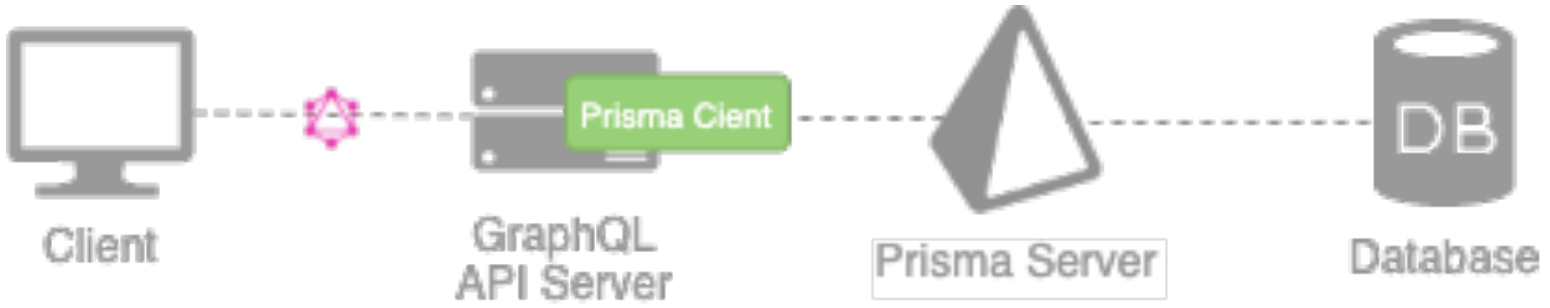
Typical 3-tier Architecture

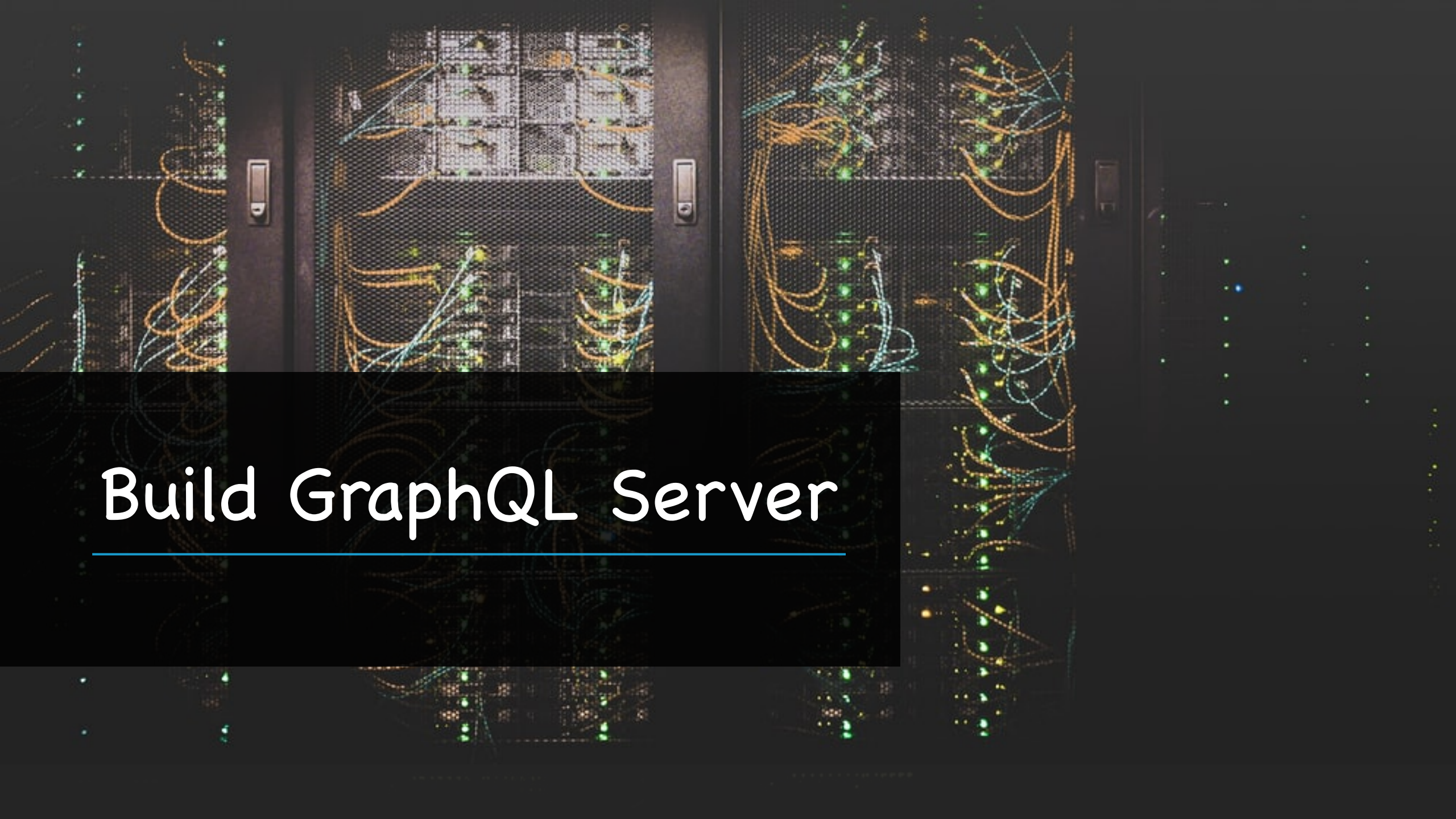
The Data Access Layer

Prisma is the Data Access Layer



Architecture





Build GraphQL Server

What is GraphQL?

New API standard developed by Facebook

Specification for type system & query language

Core primitives: Query, Mutation & Subscription

Rest vs GraphQL

Single vs Multiple Endpoint

Server vs Client decides how data return

Schemaless vs Schemaful

How GraphQL Works?

Describe your data

```
type Task {
  id: ID! @id
  title: String!
  completed: Boolean!
}
```

Schema

Ask for what you want

```
query {
  tasks {
    text
  }
}
```

Query/Mutation

Get predictable results

```
{
  "data": {
    "tasks": [
      {
        "text": "Learn GraphQL"
      },
      {
        "text": "Learn Prisma"
      }
    ]
  }
}
```

Server Response

Understand GraphQL Servers

1. Definition: The GraphQL Schema

2. Implementation: Resolver Functions

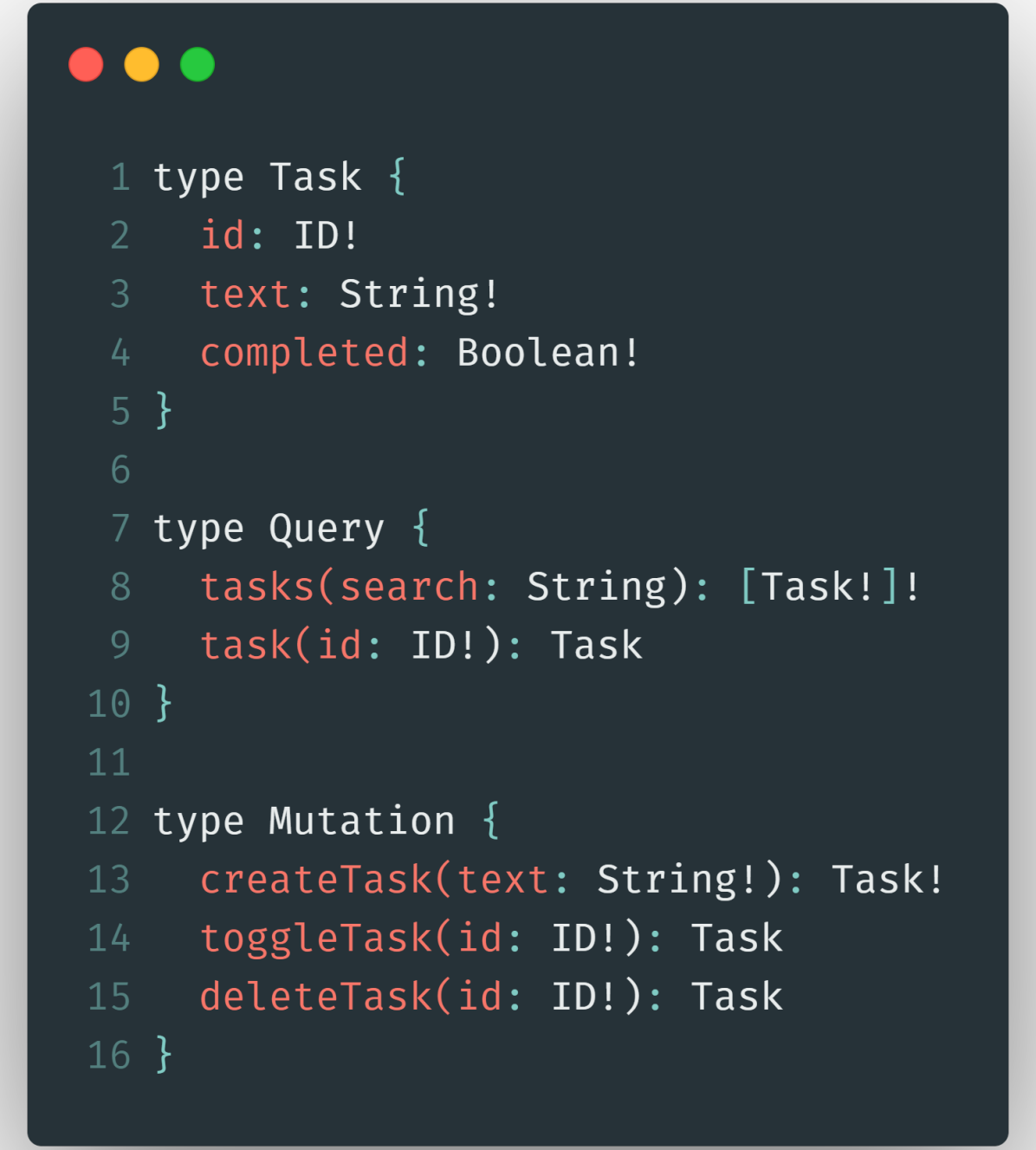
3. Setup: Framework, Network (HTTP), Middlewares

1. The GraphQL Schema

Strongly typed & written in GraphQL Schema Definition Language (SDL)

Defines API capabilities (contract for client-server communication)

Used for: Auto-generated docs, codegen, automated tests, ...



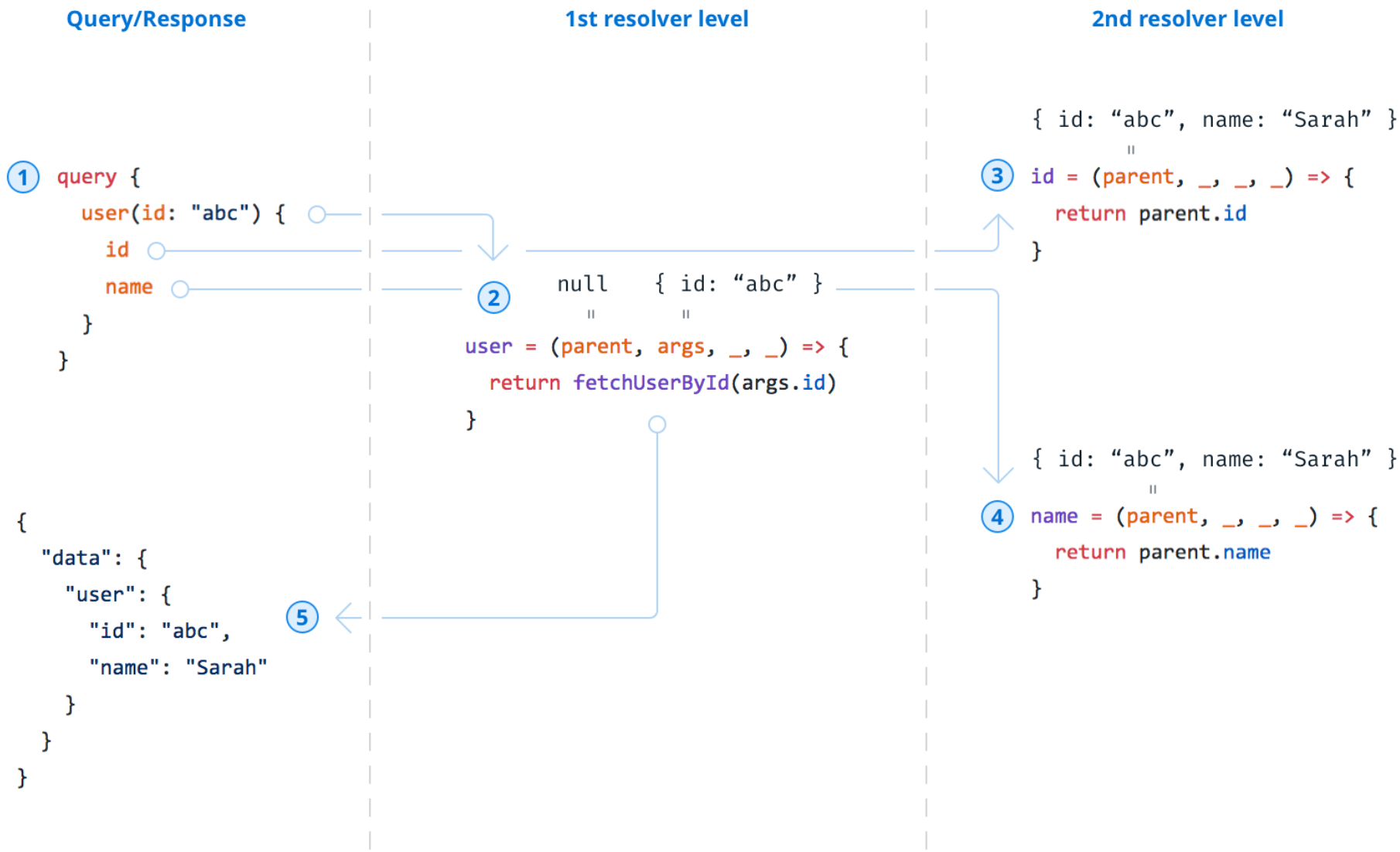
```
type Task {
  id: ID!
  text: String!
  completed: Boolean!
}

type Query {
  tasks(search: String): [Task!]!
  task(id: ID!): Task
}

type Mutation {
  createTask(text: String!): Task!
  toggleTask(id: ID!): Task
  deleteTask(id: ID!): Task
}
```

2. Resolver Function

Concrete implementation of the API

One resolver function per field in SDL schema

Query execution: Invoke resolvers for all fields in query


Query/Response
1st resolver level
2nd resolver level
{ id: “abc”, name: “Sarah” }
query {
user(id: "abc") {
id
name
}
}
3
id = (parent, _, _, _) => {
return parent.id
}
null
{ id: “abc” }
2
user = (parent, args, _, _) => {
return fetchUserById(args.id)
}
{ id: “abc”, name: “Sarah” }
4
name = (parent, _, _, _) => {
return parent.name
}
{
"data": {
"user": {
"id": "abc",
"name": "Sarah"
}
}
}
5

```
const resolver = {
  Query: {
    tasks: (root, args, context, info) => context.db.getTasks(args.search),
    task: (root, args, context, info) => context.db.getTask(args.id)
  },
  Mutation: {
    createTask: (root, args, context, info) => context.db.createTask(args.text),
    toggleTask:(root, args, context, info) => context.db.toggleStatus(args.id),
    deleteTask: (root, args, context, info) => context.db.deleteTask(args.id)
  }
};
```

3. Setup

"GraphQL engine" to orchestrate resolver invocations

Network layer based on "graphql-yoga" (network configuration: port, endpoints, CORS ...)

Middleware (analytics, logging, crash reporting ...)

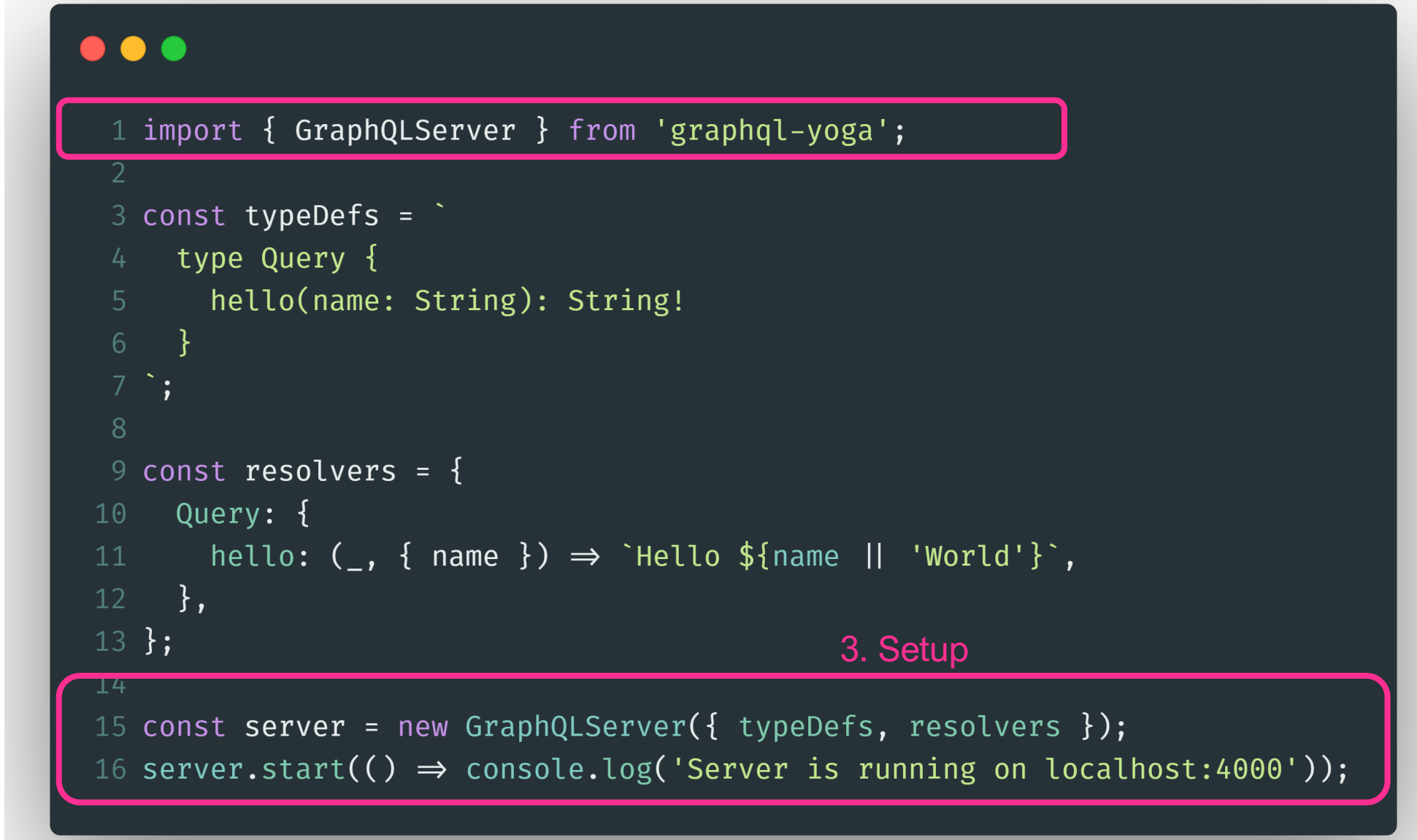

import { GraphQLServer } from 'graphql-yoga';
const typeDefs = `
type Query {
hello(name: String): String!
}
`;
const resolvers = {
Query: {
hello: (_, { name }) => `Hello ${name || 'World'}`,
},
};
const server = new GraphQLServer({ typeDefs, resolvers });
server.start(() => console.log('Server is running on localhost:4000'));

```
1  import { GraphQLServer } from 'graphql-yoga';
2
3  const typeDefs = `
4    type Query {
5      hello(name: String): String!
6    }
7  `;
8
9  const resolvers = {
10   Query: {
11     hello: (_, { name }) => `Hello ${name || 'World'}`,
12   },
13 };
14
15 const server = new GraphQLServer({ typeDefs, resolvers });
16 server.start(() => console.log('Server is running on localhost:4000'));
```

1. Schema

```
1  import { GraphQLServer } from 'graphql-yoga';
2
3  const typeDefs = `
4    type Query {
5      hello(name: String): String!
6    }
7  `;
8
9  const resolvers = {
10   Query: {
11     hello: (_, { name }) => `Hello ${name || 'World'}`,
12   },
13 };
14
15 const server = new GraphQLServer({ typeDefs, resolvers });
16 server.start(() => console.log('Server is running on localhost:4000'));
```

2. Resolver


import { GraphQLServer } from 'graphql-yoga';
const typeDefs = `
type Query {
hello(name: String): String!
}
`;
const resolvers = {
Query: {
hello: (_, { name }) => `Hello ${name || 'World'}`,
},
};
3. Setup
const server = new GraphQLServer({ typeDefs, resolvers });
server.start(() => console.log('Server is running on localhost:4000'));

Build GraphQL Server

```
mkdir todo-app

cd todo-app

git clone https://github.com/somprasongd/boilerplate-node-babel.git gql-server

cd gql-server

npm install graphql-yoga
```

Build GraphQL Server

```
type Task {
  id: ID!
  text: String!
  completed: Boolean!
}

type Query {
  tasks(search: String): [Task!]!
  task(id: ID!): Task
}

type Mutation {
  createTask(text: String!): Task!
  toggleTask(id: ID!): Task
  deleteTask(id: ID!): Task
}
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
query {
  tasks {
    id
    text
    completed
  }
}
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
query {
  tasks(search: "graphql") {
    id
    text
    completed
  }
}
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
mutation {
  createTask(text: "Learn Prisma") {
    id
    text
  }
}
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
mutation createTask{
  createTask(text: "Learn Prisma") {
    id
    text
  }
}

query getAllTask{
  tasks(search: "") {
    id
    text
    completed
  }
}
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
1  mutation createTask {
2    createTask(text: "Learn React") {
3      ...taskFields
4    }
5  }
6
7  query getAllTask {
8    tasks(search: "") {
9      ...taskFields
10   }
11 }
12
13 fragment taskFields on Task {
14   id
15   text
16   completed
17 }
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
query getAllTask {
  allTasks: tasks(search: "") {
     ... taskFields
  }
}

fragment taskFields on Task {
  taskId: id
  text
  completed
}
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
query getAllTask($search: String) {
  allTasks: tasks(search: $search) {
     ... taskFields
  }
}
```

Query & Mutation

Query

Arguments

Mutation

Operation Name

Fragment

Alias

Variable

Directive

```
query getAllTask($search: String, $withStatus: Boolean!) {
  tasks(search: $search) {
    id
    text
    completed @include(if: $withStatus)
  }
}
```

Data Access Layer with Prisma

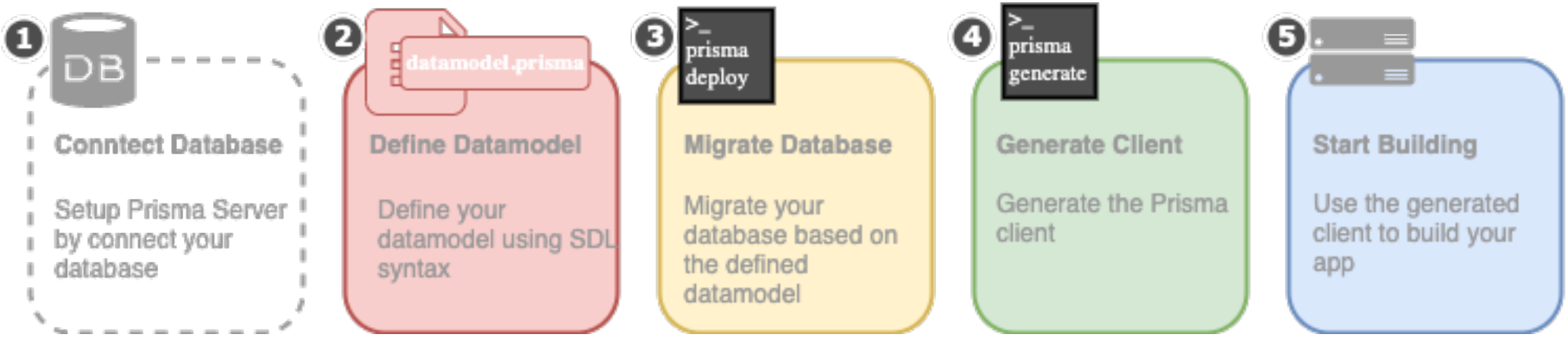
What is Prisma?

DB-agnostic data access layer (think ORM)

Prisma server & Prisma client

Declarative data modelling & migrations

How Prisma Works



3-Main Concept

1. Datamodel

2. Prisma Server

3. Prisma Client

1. Datamodel

Defines the models of your application and is foundation for the Prisma client API.

The datamodel is written in .prisma-files and uses GraphQL SDL syntax.

Using the datamodel for database migrations (optional)

Example Datamodel

```
type Task {
  id: ID! @id
  text: String!
  completed: Boolean! @default(value: false)
  createdAt: DateTime! @createdAt
  updatedAt: DateTime! @updatedAt
}
```

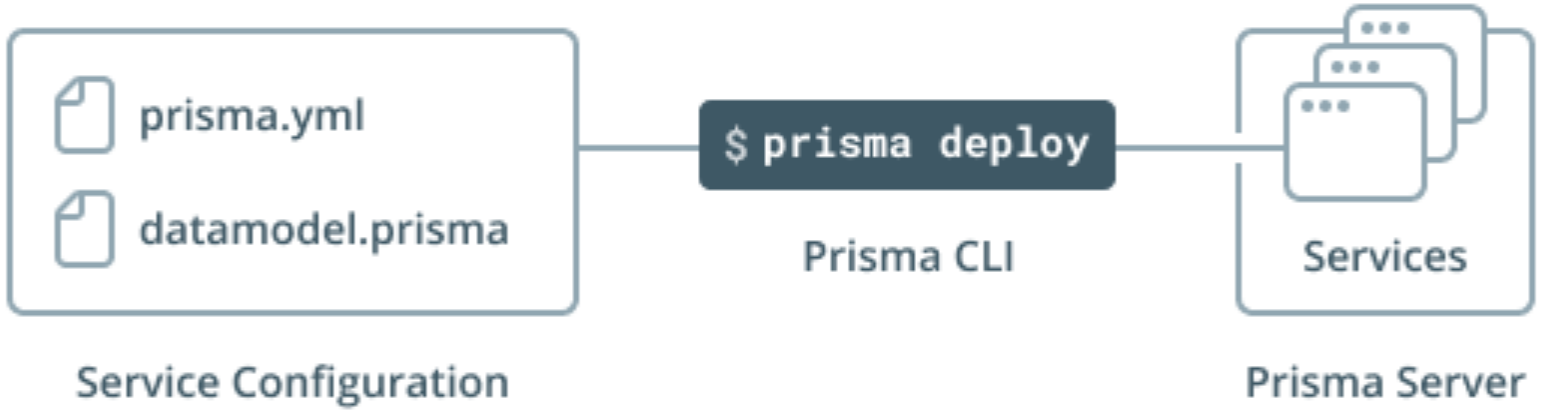
2. Prisma Server

The Prisma server is a standalone infrastructure component that is connected to your database

Prisma servers can be run with Docker

Prisma Services

2. Prisma Server



Minimal prisma.yml

```
endpoint: http://localhost:4466

datamodel: datamodel.prisma
```

3. Prisma Client

The Prisma client is an auto-generated library that replaces a traditional ORM in your API

It connects to a Prisma server which sits on top your database

Use `prisma generate` to create client

3. Prisma Client

Update prisma.yml

```
endpoint: http://localhost:4466
datamodel: datamodel.prisma
generate:
  - generator: javascript-client
    output: ./generated/prisma-client/
hooks:
  post-deploy:
    - prisma generate
```

Play with Prisma Client

```
cd prisma

npm init -y

npm install prisma-client-lib

touch playground.js

code .
```

Create

```
// npm i prisma-client-lib
const {prisma} = require('./generated/prisma-client');

// A `main` function so that we can use async/await
async function main() {
  // Create a new task
  const newTask = await prisma.createTask({ text: 'Learn React' });
  console.log(`Created new task: ${newTask.text} (ID: ${newTask.id})`);
}

main().catch(e => console.error(e));
```

Read (All)

```
1 // npm i prisma-client-lib
2 const {prisma} = require('./generated/prisma-client');
3
4 // A `main` function so that we can use async/await
5 async function main() {
6   // Read all tasks from the database and print them to the console
7   const allTasks = await prisma.tasks();
8   console.log('All Tasks:', allTasks);
9 }
10
11 main().catch(e => console.error(e));
```

Read (All with Fliter)

```
1 // npm i prisma-client-lib
2 const {prisma} = require('./generated/prisma-client');
3
4 // A `main` function so that we can use async/await
5 async function main() {
6   // Filter task list
7   const filteredTask = await prisma.tasks({
8     where: {
9       text: 'React',
10    },
11  });
12  console.log('Filtered: ', filteredTask);
13 }
14
15 main().catch(e => console.error(e));
```

Read (Single)

```
1 // npm i prisma-client-lib
2 const {prisma} = require('./generated/prisma-client');
3
4 // A `main` function so that we can use async/await
5 async function main() {
6   const taskId = 'ck36yql76001e08149494j1gj';
7   // Fetch single task
8   const task = await prisma.task({ id: taskId });
9   console.log('Task:', task);
10 }
11
12 main().catch(e ⇒ console.error(e));
```

Update

```
// npm i prisma-client-lib
const {prisma} = require('./generated/prisma-client');

// A `main` function so that we can use async/await
async function main() {
  const taskId = 'ck36yql76001e08149494j1gj';
  // Update task to completed
  const updatedUser = await prisma.updateTask({
    where: { id: taskId },
    data: { completed: true },
  });
  console.log('Updated:', updatedUser);
}

main().catch(e => console.error(e));
```

Delete

```
// npm i prisma-client-lib
const {prisma} = require('./generated/prisma-client');

// A `main` function so that we can use async/await
async function main() {
  const taskId = 'ck36yql76001e08149494j1gj';
  // Delete task
  const deletedTask = await prisma.deleteTask({ id: taskId });
  console.log('Deleted:', deletedTask);
}

main().catch(e => console.error(e));
```

Add Prisma to GraphQL Server

```
# change output path to gql-server/src in prisma.yml

prisma generate

cd ../gql-server

npm install prisma-client-lib

code .
```

Add database layer with prisma

```
1 import { prisma } from './generated/prisma-client';
2
3 // ...
4
5 const server = new GraphQLServer({
6   typeDefs,
7   resolvers,
8   context: {
9     prisma,
10  },
11 });
```

Complete API operations against the database

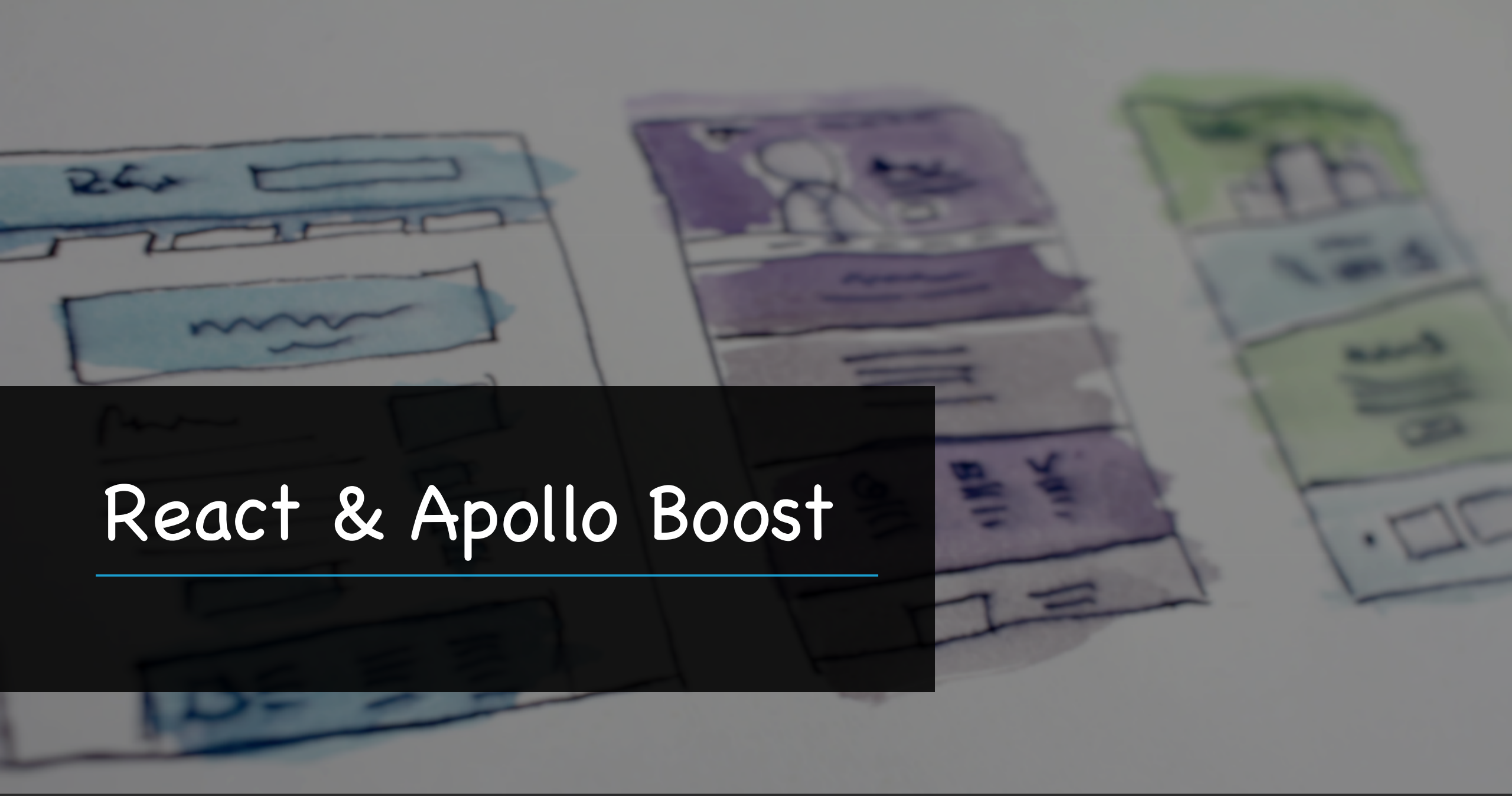
```
// resolver/Query.js
const Query = {
  tasks: (parent, { search }, { prisma }, info) ⇒
    prisma.tasks({
      where: {
        text_contains: search,
      },
    }),
  task: (parent, { id }, { prisma }, info) ⇒ prisma.task({ id }),
};

export { Query }
```

Complete API operations against the database

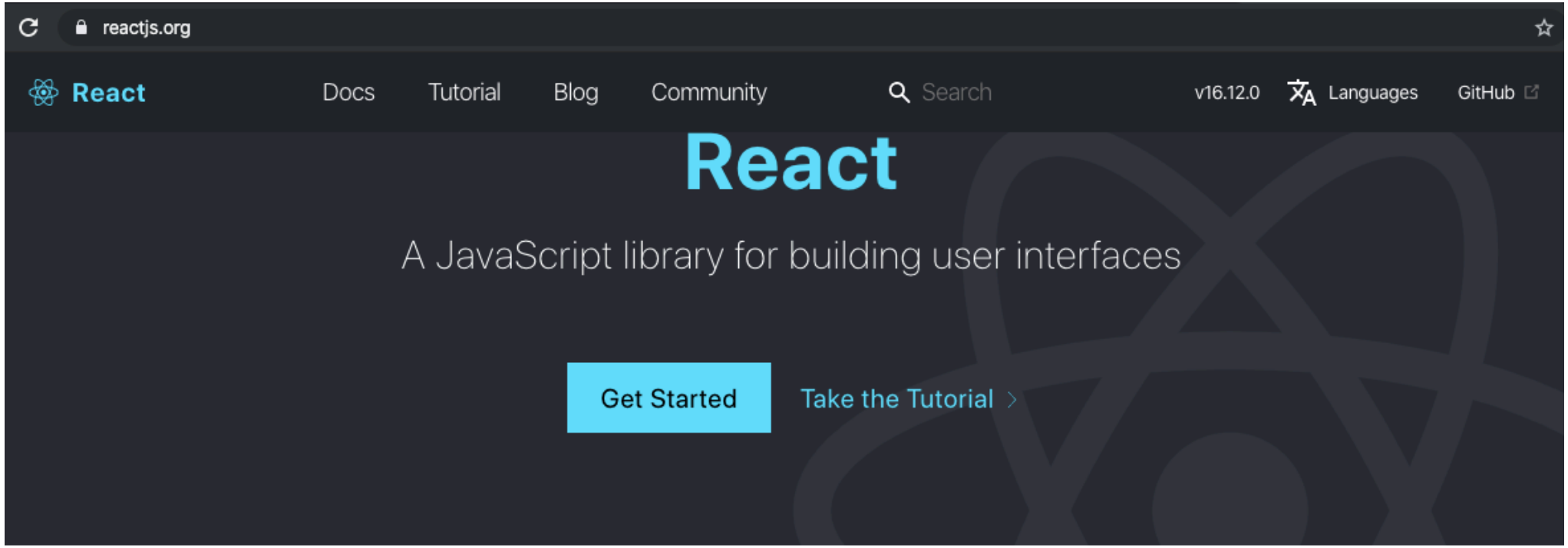
```
// resolver/Mutation.js
const Mutation = {
  createTask: (root, { text }, { prisma }, info) ⇒ prisma.createTask({ text }),
  toggleTask: async (root, { id }, { prisma }, info) ⇒ {
    const taskToUpdate = await prisma.task({ id });
    return prisma.updateTask({
      where: { id },
      data: {
        completed: !taskToUpdate.completed,
      },
    });
  },
  deleteTask: (root, { id }, { prisma }, info) ⇒ prisma.deleteTask({ id }),
};

export { Mutation };
```

Complete API operations against the database

```
// resolvers/index.js
import { Query } from './Query';
import { Mutation } from './Mutation';

const resolvers = {
  Query,
  Mutation,
};

export { resolvers };
```



React & Apollo Boost



Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

Learn Once, Write Anywhere

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using React Native.

Create React App

```
npx create-react-app client

cd client

npm start
```

Workshop: React Basic

JSX

Use Semantic UI

Component & Props

State & Events Handler

Form

Life Cycle

HTTP Request

https://github.com/somprasongd/todo-react-app

Apollo Client

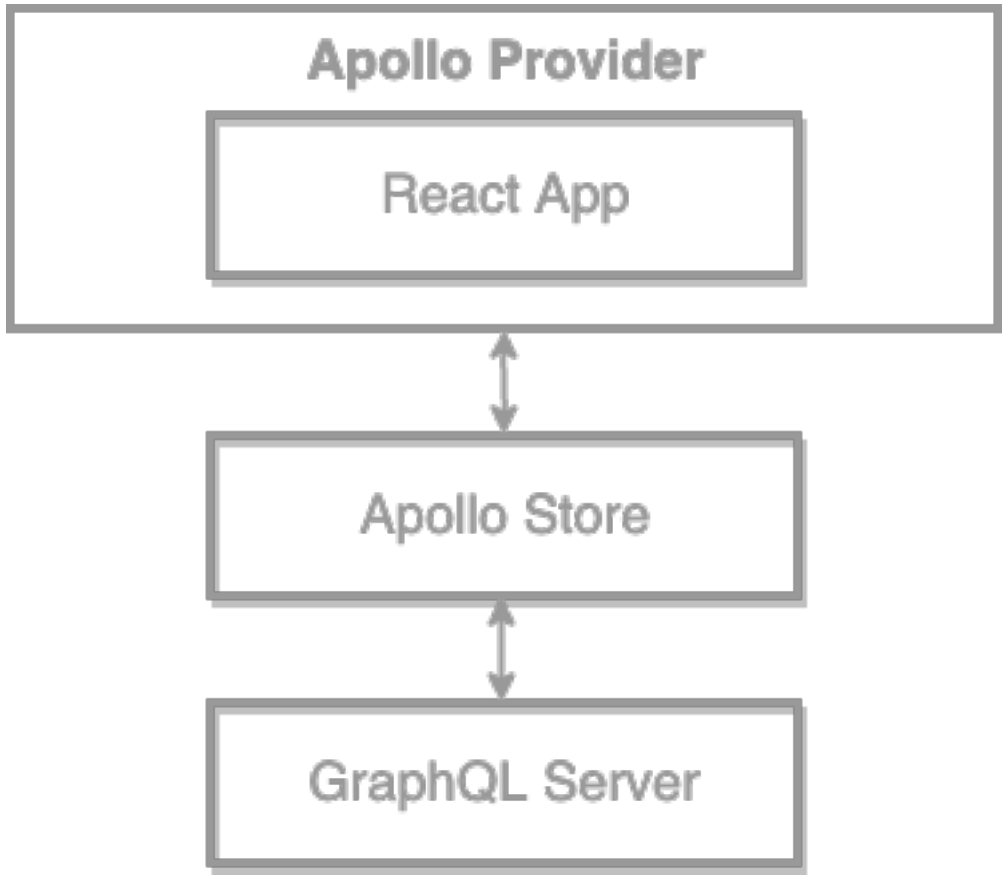
Apollo Client is a complete state management library for JavaScript apps

Simply write a GraphQL query

Apollo Client will take care of requesting and caching your data, as well as updating your UI

https://www.apollographql.com/docs/react/

How Apollo Client Works



Setup Apollo Client

```
npm install apollo-boost @apollo/react-hooks graphql


# apollo-boost: Package containing everything you need to set up Apollo Client

# @apollo/react-hooks: React hooks based view layer integration

# graphql: Also parses your GraphQL queries
```

Create Apollo Client

```
1 import ApolloClient from 'apollo-boost';
2
3 const client = new ApolloClient({
4   uri: 'http://localhost:4000',
5 });
```

Make your first query

```
1  import { gql } from 'apollo-boost';
2  // or you can use `import gql from 'graphql-tag';` instead
3
4  //  ...
5
6  client
7    .query({
8      query: gql`
9        query fetchAllTask {
10         tasks {
11           id
12           text
13           completed
14         }
15       }
16     `
17   })
18   .then(result => console.log(result));
```

Connect your client to React

```
1 import ApolloClient from 'apollo-boost';
2 import { ApolloProvider } from '@apollo/react-hooks';
3
4 const client = new ApolloClient({
5   uri: 'http://localhost:4000',
6 });
7
8
9 ReactDOM.render(
10   <ApolloProvider client={client}>
11     <App />
12   </ApolloProvider>
13   , document.getElementById('root'));
```

Workshop

Query & Mutation

https://gist.github.com/somprasongd/b380459844057 30a020560c09f943c04