```
type Itinerary {

    # Time that the trip departs.
    startTime: Long

    # Time that the trip arrives.
    endTime: Long

    # Duration of the trip on this itinerary, in seconds.
    duration: Long

    # How much time is spent waiting for transit to arrive,
    # in seconds.
    waitingTime: Long

    # How much time is spent walking, in seconds.
    walkTime: Long

    # How far the user has to walk, in meters.
    walkDistance: Float

    # A list of Legs. Each Leg is either a walking (cycling,
    # car) portion of the trip, or a transit trip on a particular
    # vehicle. So a trip where the use walks to the Q train,
    # transfers to the 6, then walks to their destination,
    # has four legs.
    legs: [ Leg ]!

    # Information about the fares for this itinerary
    fares: [ fare ]
}
```

DOCUMENTATION

- Documentation can be automagically generated from the GraphQL schema
- By automating documentation, it is ensured to be up to date
- Due to the nature of GraphQL, documentation will be simpler than it would be for REST. One only needs to know the structure and permissions of the model rather than memorizing endpoints and the data they return

```
import { makeExecutableSchema } from 'graphql-tools';
import resolvers from './resolvers';

const typeDefs = `
type Author {
  id: Int
  firstName: String
  lastName: String
  posts: [Post]
  stars: Int
}

type Post {
  id: Int
  title: String
  text: String
  views: Int
  author: Author
  genre: String
}

type Query {
 author(firstName: String, lastName: String, id: Int):
 authors: [Author]
 posts: [Post]
 post(id: Int): Post
```

BREAKING CHANGES AND SCHEMA VALIDATION

- CI tooling is available to automatically identify breaking changes in a pull request
- Because the tooling is validating static queries against the schema, you can ensure that there will be no client-side runtime errors due to mismatched expectations
- CI integrations allow for the entire team to be notified when a breaking change is merged