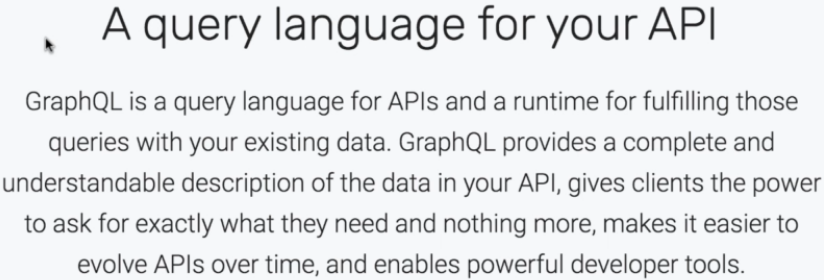
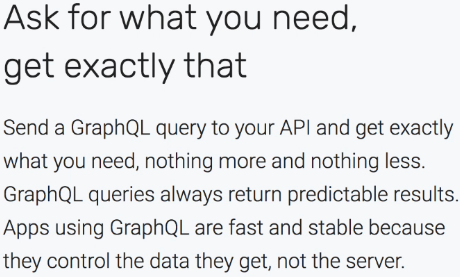
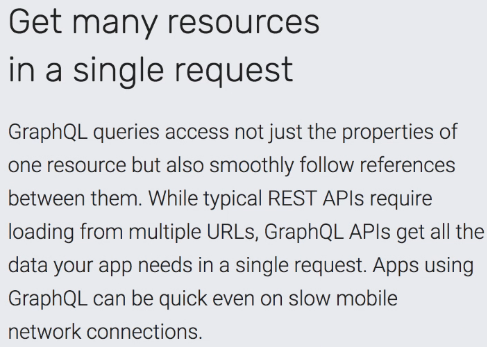
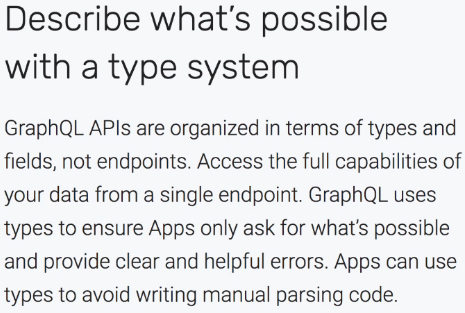
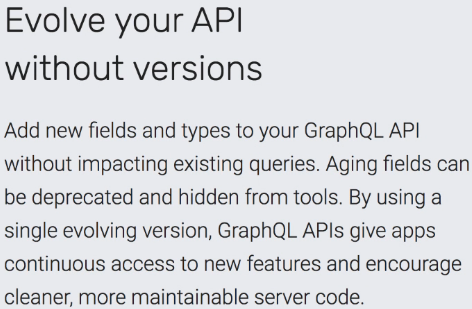
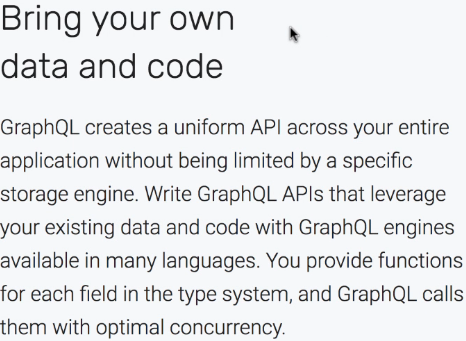
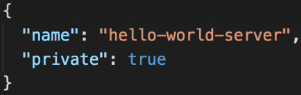
**What is GraphQL**  
\* We’ll take a very practical approach to learning GraphQP, we’ll mostly learn by doing, that is by writing actual code.  
\* Let’s start with a quick overview of what GraphQL is and what are its advantages compared to other approaches to designing APIs.  
<https://graphql.org/>  
\* **A query language for your API**.  
  
\* **We could say that GraphQL is a bit like SQL but for querying Web APIs rather than databases**.  
\* Why would we want GraphQL?  
  
=> This means that the client has full control over which data it wants from the server.  
\* **With RESTful APIs when you request a resource, you always get all the data from that resource**.  
=> **That can result in “overfetching”, fetching too much data, data that’s not actually used**.  
\* **With REST you sometimes have the opposite problem as well - “underfetching” if you want to fetch 2 different resources, you need to make 2 separate calls to the server**.  
  
\* **Get many resources in a single request**.

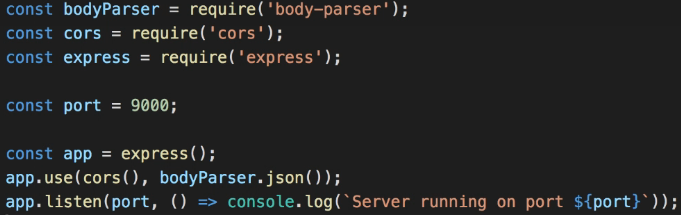
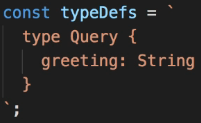
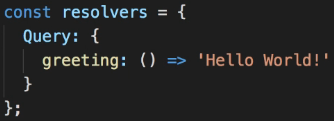
  
\* **Describe what’s possible with a type system**.  
=> You write a schema that fully describes your API.  
\* This is not a unique feature of GraphQL by any means, there are many schema-based approaches out there. Even with REST if you really want to, you can use something like the OpenAPI specification, formerly known as Swagger.  
\* **But I definitely like the schema-first approach**.  
\* **Having a clear contract between the server and clients**.  
\* **Having a type system also makes it easier for people to build powerful developer tools and there are many useful tools for working with GraphQL**.  
\* GraphiQL is a web interface for running GraphQL queries in the browser.  
  
\* Along with the language and the tools, there are also some interesting ideas in the GraphQL community.  
\* One of them is “version-less API”.  
=> **That means you can modify your API over time without breaking backwards compatibility**.  
\* This is more of a best practice rather than a technical feature but GraphQL does offer a way to deprecate fields.  
  
=> **Means that GraphQL is a layer that you can add on top of your existing code**. **There’s no need to re-write your entire application. GraphQL takes care of exposing an API. All your data and business logic can stay the same**. Also, GraphQL is available for many different languages.

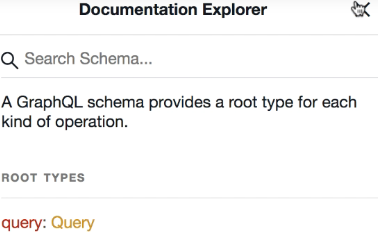
\* **In this course, we’ll use JavaScript for both the server and the client**.  
\* GraphQL is an open specification with multiple implementations for different languages.  
\* GraphQL was first developed at Facebook, initially as an internal project in 2012 and then released as open source in 2015. The reason why Facebook started working on it was that essentially they wanted to speed up their mobile apps. They were too slow because they were making too many calls to the server.  
\* **So that explains why GraphQL puts a lot of emphasis on avoiding overfetching and underfetching data**.  
\* The GitHub’s REST API was considered to be one of the best examples of RESTful APIs, but then in 2016 GitHub adopted GraphQL for v4 of their Developer API. So that was a significant step.  
\* Today GraphQL is used by many many companies and there’s a growing community around it.

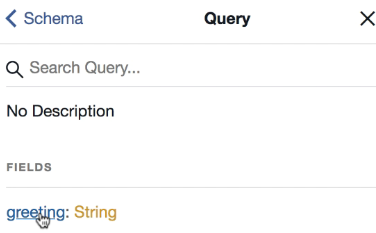
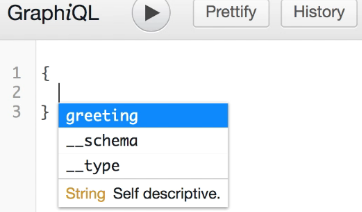
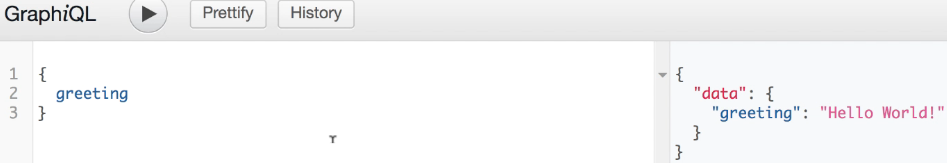
**Pre-Requisites**  
\* To follow the examples in this course you will need:  
=> A computer running Linux, macOS, or Windows.  
=> A web browser, preferably the latest version of Google Chrome.  
=> A recent version of Node.js installed. The latest LTS version is recommended, currently v8.11.  
=> Your favourite code editor. In the videos the instructor uses **Visual** **Studio** **Code**, which is **cross-platform, free and open source**.

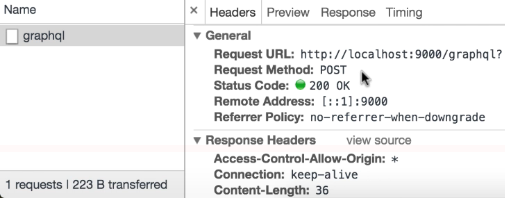
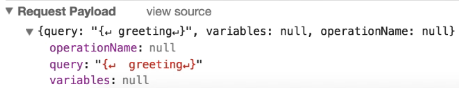
**Important Update**  
\* A new major version of apollo-server-express, one of the packages used in the examples, was published after the course was first recorded. Unfortunately the new version 2.x is not compatible with v1.x used in the videos.  
\* To avoid problems, please make sure to install version 1.x. When following the instructions in Lectures 4 and 8 run:  
**npm install apollo-server-express@1** instead of just apollo-server-express which would fetch the latest.  
\* The differences between the two versions are small and do not affect the main GraphQL concepts explained in the course. You'll find a new video showing how to upgrade to Apollo Server 2.x at the end of the course.

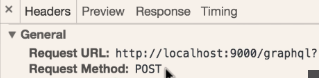
**Minimal GraphQL Server**  
\* Let’s write our first GraphQL server based on Node.js, Express and Apollo.  
\* We’ll create a very simple API thata just returns a single value but it will show you how to put together all the parts required to run a GraphQL server.  
\* Let’s add **package.json**  
 **npm install express body-parser cors**  
\* Create **server.js**  
\* **The body-parser middleware will automatically parse HTTP Requests that contain a JSON body**.  
\* **GraphQL Requests are in JSON format**.  
\* **The cors middleware allows cross-origin requests**.

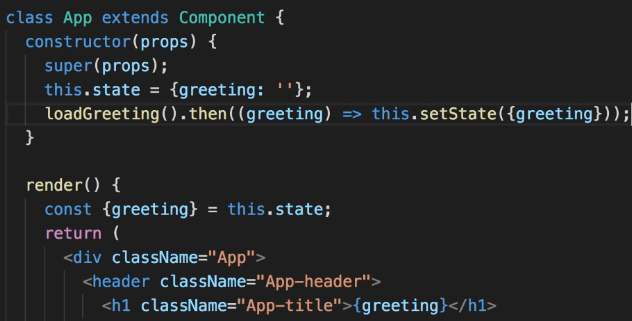
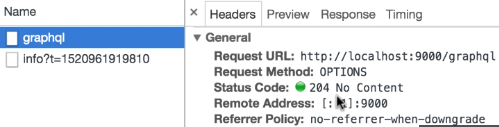
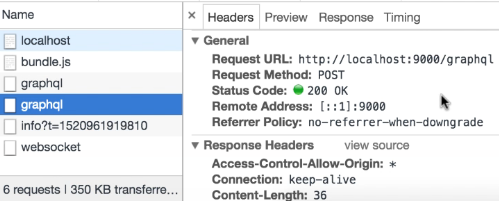
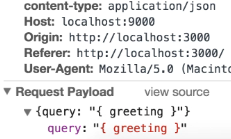
  
**node server.js**  
\* Now let’s add some dependencies for the GraphQL functionality.  
**npm install graphql graphql-tools apollo-server-express**  
**graphql** => **contains the core GraphQL functionality like parsing queries**.  
**graphql-tools** => **provides some utility functions**.  
**apollo-server-express** => **provides the glue for serving GraphQL over HTTP on top of Express**.  
\* **With GraphQL we need to describe the data we want to expose in our API using a Type System**.  
**Type System**  
**GraphQL Schema Definition Language** => special syntax.  
=> In pretty much any schema we need a top-level type called Query that describes which queries can be sent by the client.  
=> A type can have some fields.  
\* If you’re familiar with TypeScript or Flow, you’ll find this syntax very similar.  
  
\* We’re declaring that all a client can do with this API is ask for the “greeting” that is a string value.  
**resolvers** => **to process a request, the structure of this object reflects our type definitions**.  
  
=> **The “greeting” function is a resolver in a sense that it resolves the value of the “greeting” field**.  
**typeDefs** => **we declare what the client can ask for.**  
**resolvers** => **we have the logic that says how the server will respond.**  
**makeExecutableSchema({typeDefs, resolvers})** =>  
  
\* I’m using the shorthand syntax since the names of the object properties are the same as the variable names we pass as values.  
  
\* At this point we’re ready to tell Express to server our GraphQL Schema.  
**graphqlExpress** => **a function, with it we can declare a new route in our app**.  
  
  
\* So to recap: we define what our API looks like in the typeDefs, in resolvers we specify how each request is handled, we put the 2 together into an executable schema and we expose it over HTTP as the ‘/graphql’ route using the graphqlExpress() function.  
\* We could run this server as it is but I want to add something else.

=> The apollo-server module provides another function  
graphiqlExpress => GraphiQL is a web interface that lets us run GraphQL queries in the browser.  
  
  
\* **It says that on this server, the GraphQL API is mapped to the ‘/graphql’ path**.  
\* This is all the code we need for our first GraphQL server.  
**node server.js**   
\* Navigate to /graphiql  
  
\* **The** **Docs**:  


\* **If we click on Query**:  
  
=> **Here we can explore our schema**.  
\* **This can be really useful if you need to call a GraphQL API written by somebody else**.  
\* The syntax for a query is we write {} that represent the root type (i.e. Query) and then we can specify which fields we want.  
\* If you press CTRL + Space, you should see what fields are available.  
  


\* **We can see the server response on the right panel**.  
\* It’s a JSON object that has a “data” property.  
\* **We just ran our first GraphQL query**.  
\* This GraphiQL interface is great for testing queries but I also want to show you what happens behind the scenes in terms of HTTP Requests and Responses.  
\* Let’s open Chrome Dev Tools with the Network Tab.  
  
\* If we look at the Request body, it’s a JSON object that has a “query” property with the value being the query string we entered into the GraphiQL tool exactly like we typed it, new lines and everything.  
  
\* And here is the Response, it’s the same apart from the indentation.  
  
\* So at the Transport Layer, GraphQL Requests are just regular HTTP Requests where we POST some JSON and get some JSON back.  
\* That’s our first GraphQL server.  
\* It’s not a lot of code but we’ve seen a few different GraphQL concepts like type definitions, resolvers, and so on.

**Minimal GraphQL Client**  
\* Let’s see how to call the server from a Client application.  
\* We’ll use React for the Web Frontend because React is very popular.  
**npx create-react-app hello-world-client**  
**npm start**  
\* We basically want our React app to make the GraphQL query that we entered in the GraphiQL tool.  
\* As we’ve seen, executing a GraphQL query effectively means making an HTTP Request.  
=> **POST Request to the GraphQL Endpoint that on our server is /graphql**.  
  
\* **With the Content-Type being JSON**.  
\* **And sending the query in the Request body wrapped in a JSON object**.

\* **We’ll use the FETCH API just because it doesn’t need any additional packages installed, you could of course use axios for example**.  
\* **Now we want to get the Response that is asynchronous**.  
\* **To handle the Promise returned by fetch(), we can use the async/await feature (ES2017)**.  
\* **In the FETCH API, the .json() is also asynchronous, it returns a Promise**.  
  
\* As a temporary step, I’m going to call this function outside of the component so it will be executed as soon as this file is loaded.  
  
\* That’s good, our call returns what we need.  
\* So we can extract it.  
  
\* **Our loadGreeting() is an async function so it returns a Promise**.  
  
  
\* In Network Tab we see:  
  
=> **This is an OPTIONS Request, it’s the pre-flight Request made by the browser to check the Cross-Origin permissions**.  
\* **We can ignore this one**.  
\* **Let’s select All Requests in the Network Tab and we see**:  
  
  
\* That’s it for this example.  
\* We’ve seen that we can send GraphQL Requests to the server by making standard HTTP Requests.  
\* Just passing the GraphQL query in the Request body.  
\* **There’s no need to use a special GraphQL client, you can just use FETCH or some other library for making HTTP Requests**.  
\* **There are a number of GraphQL Client Libraries that add extra features - like caching**.  
\* But for simple requirements, making HTTP Requests directly is a perfectly good choice.  
\* And I wanted to show you this approach first to make sure you understand what GraphQL queries look like over HTTP.

**Resources**  
GraphQL Official Page  
<https://graphql.org/>