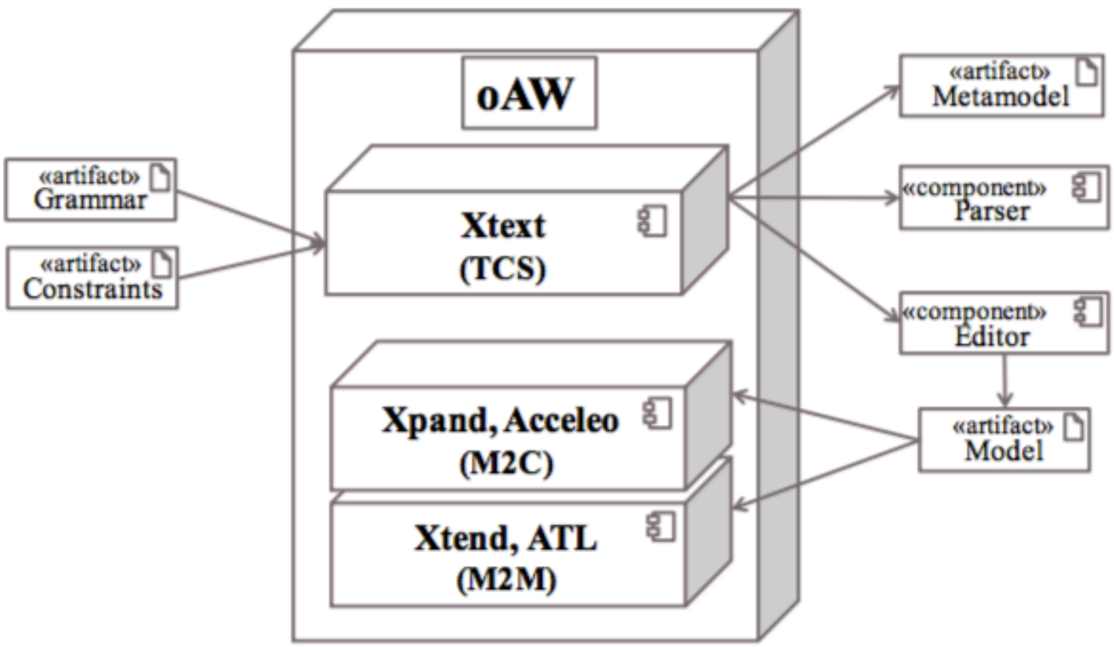
GraphQL Architecture

<u>"from a DSL point of view"</u>

❖ Text-To-Model:

GraphQL using Xtext for developing textual domain specific language, and Xtext creates the **parser**, **metamodel**, and **editor** from grammar definition automatically.



We can find that GraphQL designed as an Application-Layer Protocol and does not require a particular transport. It is a string that is **parsed** and interpreted by a **server**, so in GraphQL, the specification for queries are encoded in the *client* rather than the *server*. These queries are specified at field-level granularity. **The server determines the data returned in its various scripted endpoints**.

On the other hand, GraphQL is strongly-typed. Given a query, tooling **can ensure that the query is both syntactically correct and valid within the GraphQL type system before execution**, i.e. at development time, and the server can make certain guarantees about the shape and nature of the response. This makes it easier to build high quality client tools.

We know that **Parsing** is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar.

So, we can find that the **parser** needed in the server to:

- ensure that the query is correct syntactically within the grammars and valid within GraphQL type
- derives the query string from the start symbol of the grammars
- exposes a tagged template function for parsing GraphQL queries
- exposes lower level API for generating GraphQL AST and traversing it

Also, **Text-To-Model transformation** applied by **Bridging Xtext Grammars to Metamodels** "**Obtaining the Metamodel from the Grammar**" by:

- Production of a metamodel (M2)
  "Mapping a EBNF-based grammar into a MOF-based metamodel"
- Optimization of the metamodel
- Production of a program transformer (M1)
  "Mapping a well-formed program into a metamodel- conformant model"

❖ Model-To-Text:

Code Generation is the process by which a compiler code generator converts a syntactically-correct program into a series of instructions that can be executed by a machine. "**program to generate source code**"

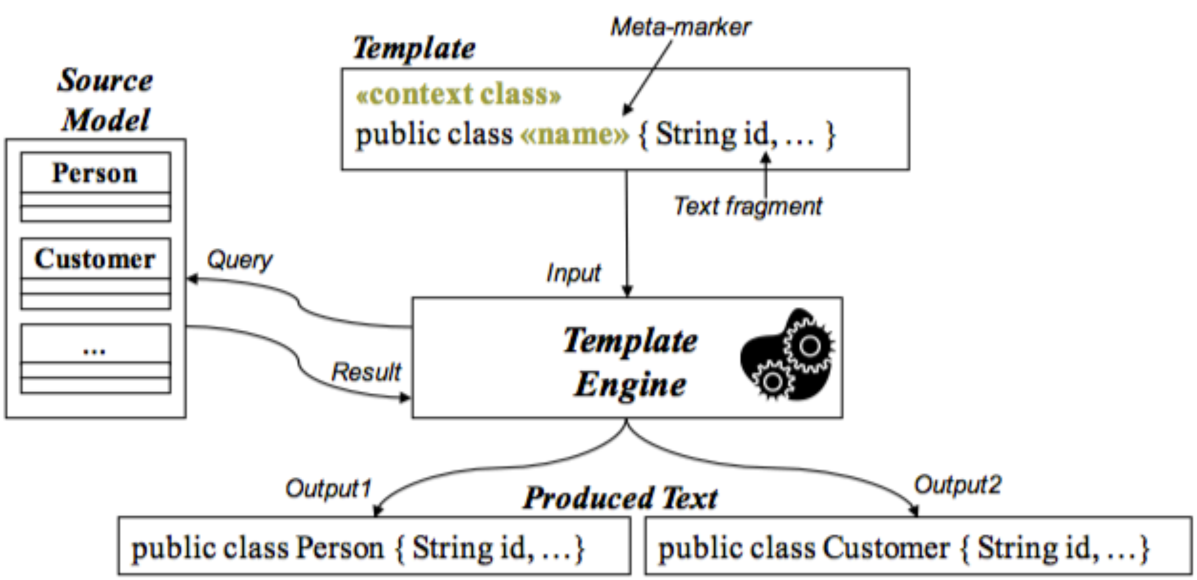
GraphQL used code generation in Model-to-Text transformation, where text is a **program code** using **Xtend** which is a **Template-based** language "a general purpose JVM language with support for code templates", and by **generating Xtext Artefacts**.

Templates are a well-established technique in software engineering and the **components** of the **template-based approach** are:

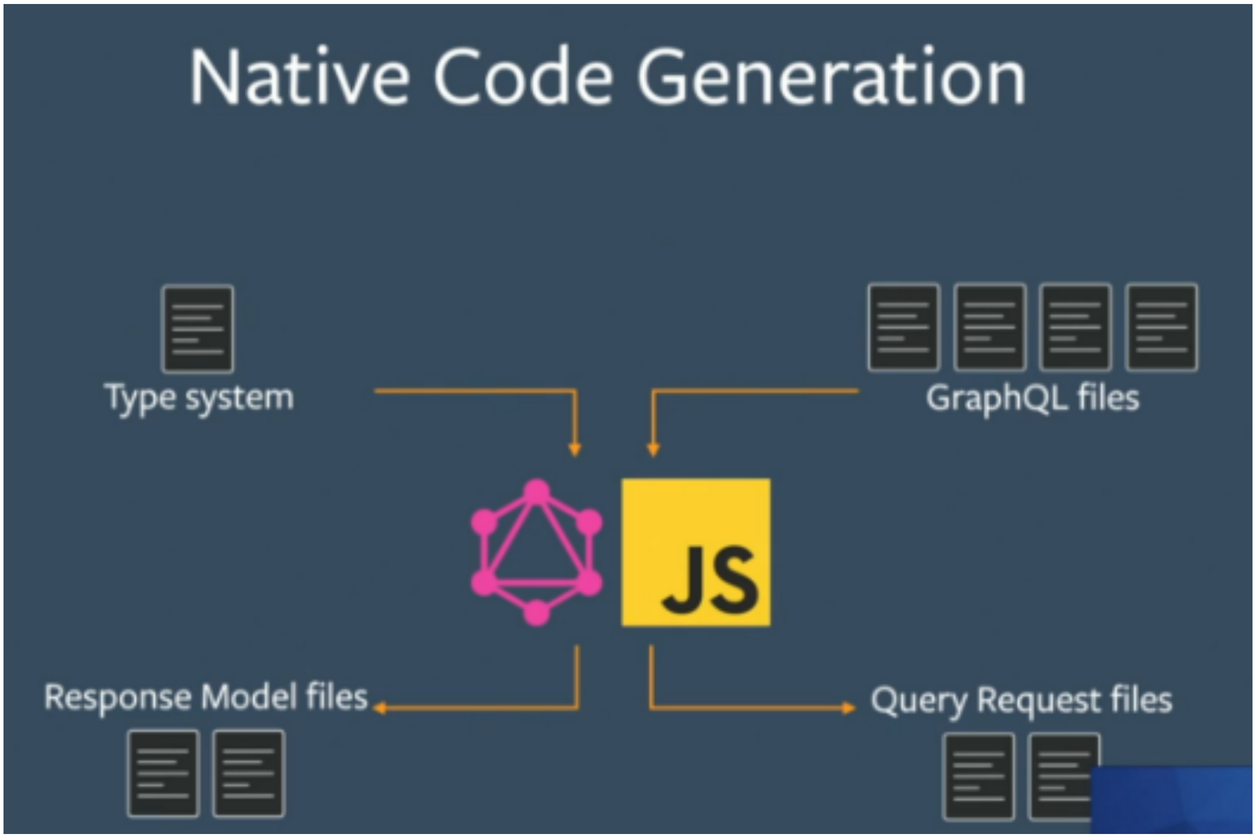
- **Templates**
  "Text fragments and embedded meta-markers "

- **Meta-markers** query an additional data source
  - Have to be interpreted and evaluated in contrast to text fragments
  - Declarative model query: query languages (OCL, XPath, SQL)
  - Imperative model query: programming languages (Java, C#)

- **Template engine**
  "Replaces meta-markers with data at runtime and produces output files"



So, the native **Code Generation** tools **take a description of the type system** and **a set of GraphQL queries** as an **input**, and **output** a **set of response models and query requests**.



Also, **Model-To-Text transformation** applied by:

- Creating a template using a template-base language
- Using GraphQL queries
- Generating Xtext artefacts
- Execute the code generator

The **similarity** between the approach followed in our exercise and the GraphQL approach is that the native code generation tools **take a set of GraphQL queries as an input.**
While the **difference** is it **takes a description of the type system** "the schema-a type system that defines the data you can fetch from a specific GraphQL server" as well in GraphQL approach, but in our exercise, it takes a **custom template**.