DrupalCon
PORTLAND 2022
25-28 APRIL

Building a GraphQL API - Beyond the basics

Alexander Varwijk

Lead Front-End Engineer @ Open Social

Alexander Varwijk

Lead Front-End Engineer @ Open Social

10 years on Drupal.org (Kingdutch) 🎉

Contributed to (a.o.)

- webonyx/graphql
- drupal/graphql

Twitter: @Kingdutch

Slides available on my website:

https://www.alexandervarwijk.com/talks

open social®

Community Engagement Platform as a Service

Helping organizations across the globe create communities that drive real-world change

holmesglen
Biotechnology Industry Organization
foraus
Pachamama Alliance
milieudefensie
Friends of the Earth Netherlands
CTBTO Preparatory Commission
Global Partnership for Effective Development Co-operation
The Salvation Army
European Commission
Family for every child
Connecting Business Initiative
NWO
iati International Aid Transparency Initiative
Waterschap Rivierenland

Always looking for Drupal developers

https://careers.getopensocial.com

What is(n’t) in this talk?

GraphQL has too many interesting things to cover

In this talk

- Setting up a modular schema
- GraphQL module deep-dive

Not in this talk

- Schema Design Course
- Automated Testing

Disclaimer

Please learn, don’t copy without understanding

I make mistakes, the information provided is based on my learning from the past years. There may be better way to do the things I’m showing you.

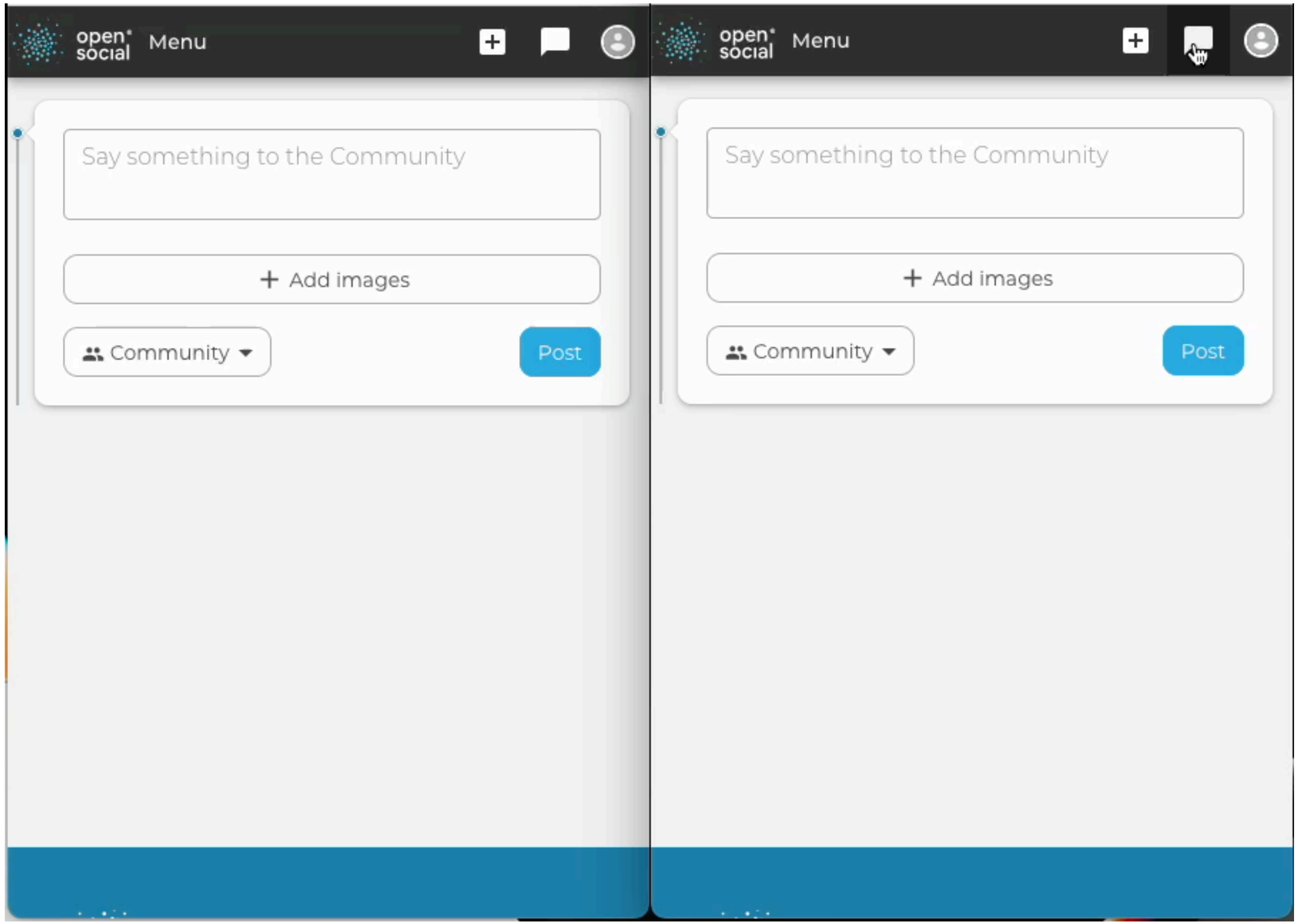
If you see something in this presentation that’s wrong, please teach me after the presentation! Find me here at DrupalCon or on Twitter (@Kingdutch).

Building a Real-Time Chat

Schema Design

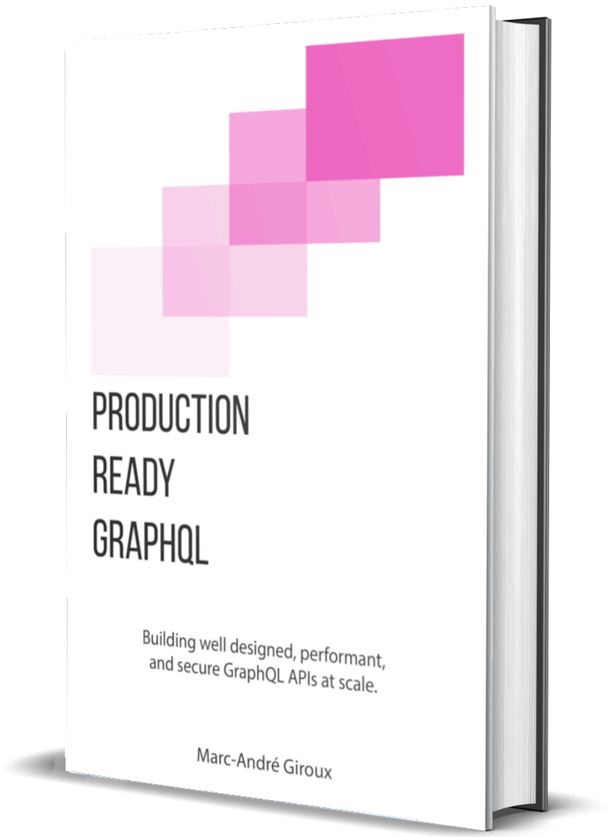
Defining the goal of what we're building

As a chat participant I want to be able to send chat messages to and receive chat messages from other participant(s) without refreshing my page so that I can easily communicate with others


open social
Menu
Say something to the Community
+ Add images
Community
Post
open social
Menu
Say something to the Community
+ Add images
Community
Post

Production Ready GraphQL

Building well designed, performant, and secure GraphQL APIs at scale.



Every developer at Open Social has access to this book

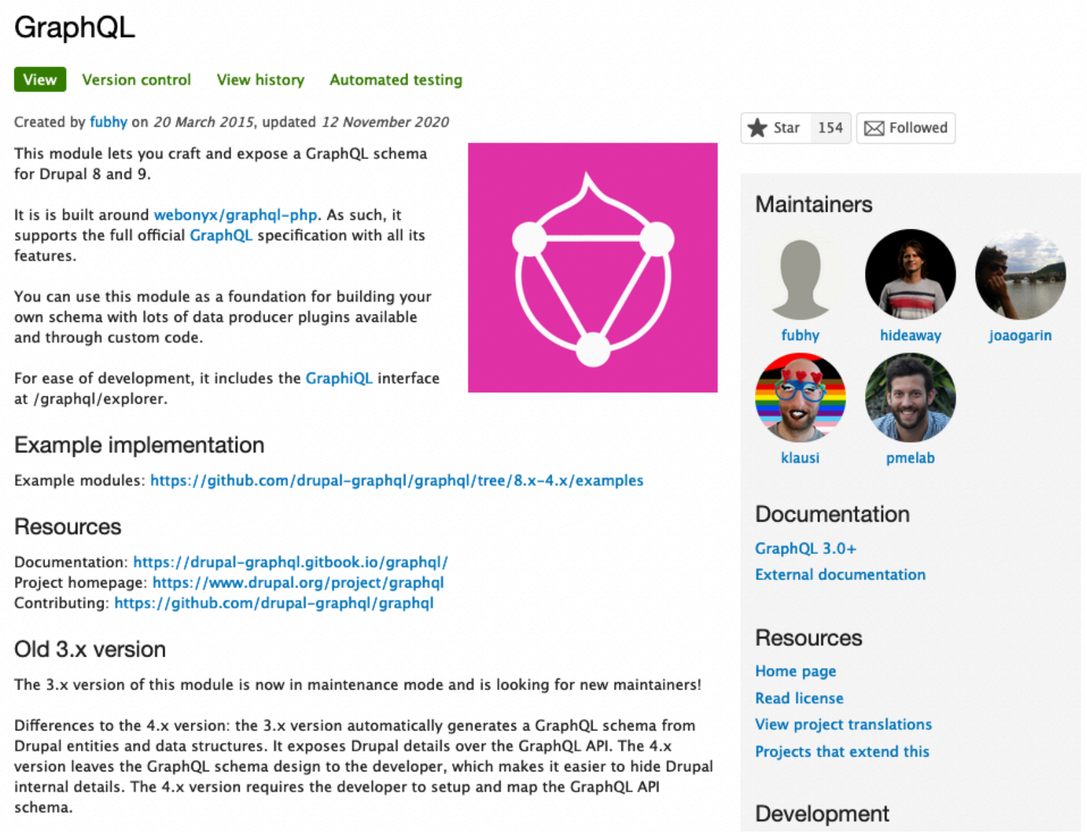
https://book.productionreadygraphql.com/

Written by Marc-Andre Giroux

The GraphQL Module

The GraphQL Module

Because there's always a module for that ;-)



GraphQL

View | Version control | View history | Automated testing

Created by fubhy on *20 March 2015*, updated *12 November 2020*

This module lets you craft and expose a GraphQL schema for Drupal 8 and 9.

It is is built around webonyx/graphql-php. As such, it supports the full official GraphQL specification with all its features.

You can use this module as a foundation for building your own schema with lots of data producer plugins available and through custom code.

For ease of development, it includes the GraphiQL interface at /graphql/explorer.

Example implementation

Example modules: https://github.com/drupal-graphql/graphql/tree/8.x-4.x/examples

Resources

Documentation: https://drupal-graphql.gitbook.io/graphql/
Project homepage: https://www.drupal.org/project/graphql
Contributing: https://github.com/drupal-graphql/graphql

Old 3.x version

The 3.x version of this module is now in maintenance mode and is looking for new maintainers!

Differences to the 4.x version: the 3.x version automatically generates a GraphQL schema from Drupal entities and data structures. It exposes Drupal details over the GraphQL API. The 4.x version leaves the GraphQL schema design to the developer, which makes it easier to hide Drupal internal details. The 4.x version requires the developer to setup and map the GraphQL API schema.

Star 154 | Followed

Maintainers

fubhy | hideaway | joaogarin

klausi | pmelab

Documentation

GraphQL 3.0+
External documentation

Resources

Home page
Read license
View project translations
Projects that extend this

Development

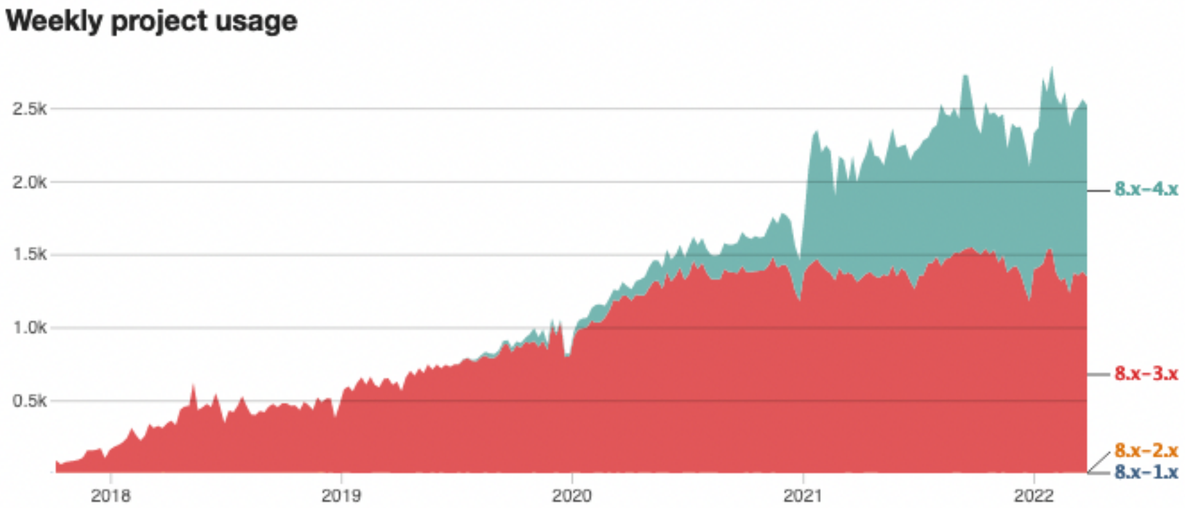
- Uses the webonyx/graphql-php library under the hood
- Contributions and issue tracking happens on GitHub
- 3.x and 4.x are actively used by the community

drupal/graphql

Two major versions

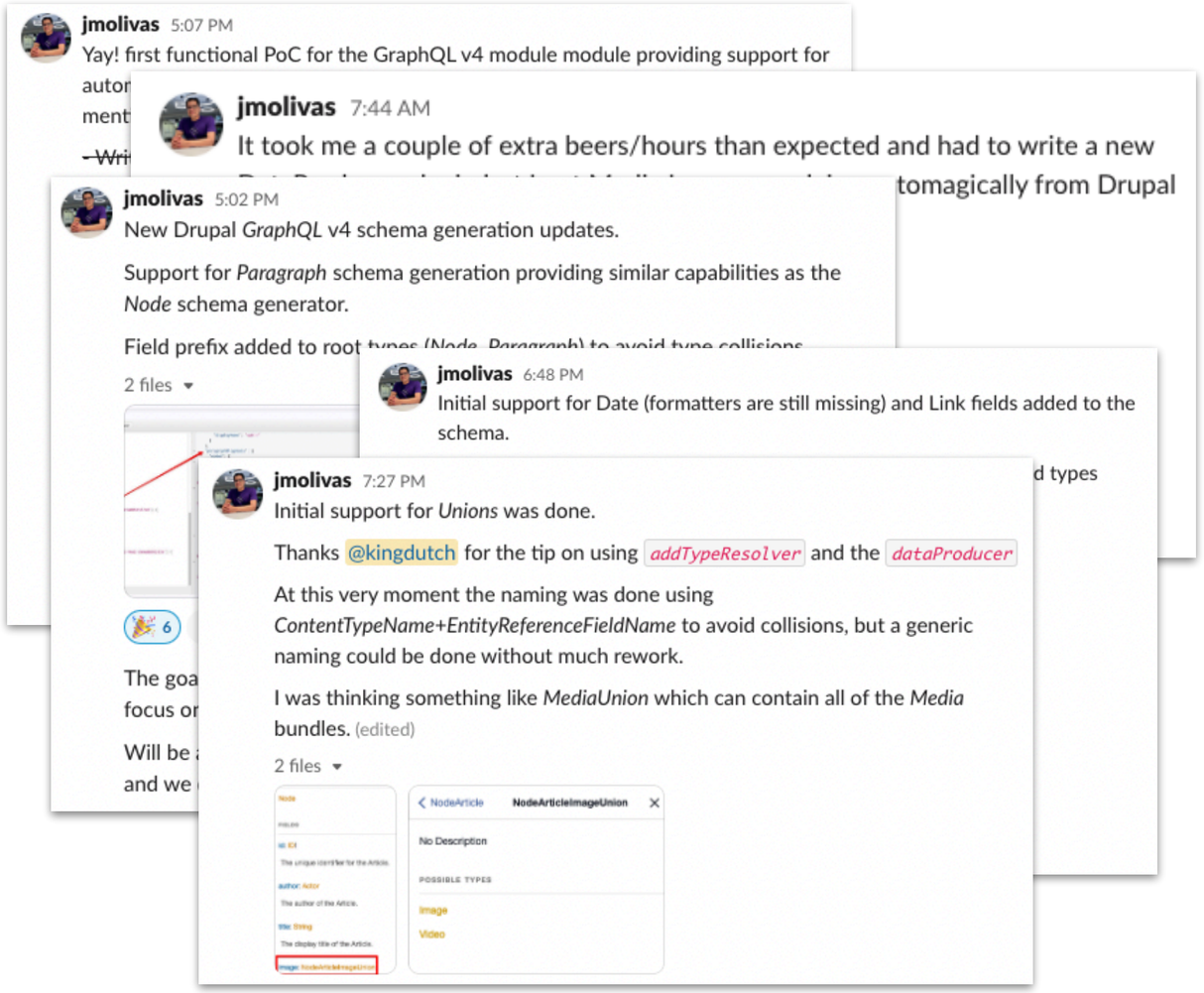
Version 3 features automatic schema generation. This leaks Drupal implementation details.

Version 4 requires writing your own schema. This can result in a cleaner schema but takes more effort.



v3 in v4

Bringing automatic schema generation to the latest version



jmolivas has been hard at work rebuilding automatic schema generation on the new drupal/graphql v4 architecture

Updates in #graphql on the Drupal Slack

[graphql_compose](#) project on Drupal.org

Implementing our API

Defining a base schema

Utilising inheritance and Drupal's modularity

```
/**
 * @Schema(
 *   id = "drupalcon",
 *   name = "A DrupalCon base GraphQL Schema"
 * )
 */
```

Our base schema helps us with some common types

Schema extensions can re-use those types without having to define them

Defining a base schema

1. Our schema must by in the Plugin\GraphQL\Schema namespace to be discovered

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
namespace Drupal\drupalcon\Plugin\GraphQL\Schema;


use Drupal\graphql\Plugin\GraphQL\Schema\SdlSchemaPluginBase;









class DrupalConBaseSchema extends SdlSchemaPluginBase {









}
```

Defining a base schema

1. Our schema must by in the Plugin\GraphQL\Schema namespace to be discovered

2. We use the schema annotation to define our base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
namespace Drupal\drupalcon\Plugin\GraphQL\Schema;



use Drupal\graphql\Plugin\GraphQL\Schema\SdlSchemaPluginBase;

/**
 * @Schema(
 *   id = "drupalcon",
 *   name = "A DrupalCon base GraphQL Schema"
 * )
 */
class DrupalConBaseSchema extends SdlSchemaPluginBase {









}
```

Defining a base schema

1. Our schema must by in the Plugin\GraphQL\Schema namespace to be discovered

2. We use the schema annotation to define our base schema

3. Our only required function is getResolverRegistry

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
namespace Drupal\drupalcon\Plugin\GraphQL\Schema;


use Drupal\graphql\GraphQL\ResolverRegistry;
use Drupal\graphql\Plugin\GraphQL\Schema\SdlSchemaPluginBase;

/**
 * @Schema(
 *   id = "drupalcon",
 *   name = "A DrupalCon base GraphQL Schema"
 * )
 */
class DrupalConBaseSchema extends SdlSchemaPluginBase {

  /**
   * {@inheritdoc}
   */
  public function getResolverRegistry() {
    return new ResolverRegistry();
  }




}
```

Defining a base schema

modules/drupalcon/graphql/drupalcon.graphqls

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

type Query

type Mutation

type Subscription
```

```
type DateTime {
  timestamp: Timestamp!
}

interface Node { id: ID! }

scalar Cursor

interface Connection {
  edges: [Edge!]!
  nodes: [Node!]!
  pageInfo: PageInfo!
}

interface Edge {
  cursor: Cursor!
  node: Node!
}

type PageInfo {
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
  startCursor: Cursor
  endCursor: Cursor
}
```

Only types which can be shared by all the other parts of our schema

Ideally in your version you include comments ;-)

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
namespace Drupal\drupalcon\Plugin\GraphQL\Schema;


use Drupal\graphql\GraphQL\ResolverRegistry;
use Drupal\graphql\Plugin\GraphQL\Schema\SdlSchemaPluginBase;

/**
 * @Schema(
 *   id = "drupalcon",
 *   name = "A DrupalCon base GraphQL Schema"
 * )
 */
class DrupalConBaseSchema extends SdlSchemaPluginBase {

  /**
   * {@inheritdoc}
   */
  public function getResolverRegistry() {
    return new ResolverRegistry();
  }




}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```
public function getResolverRegistry() {
  $registry = new ResolverRegistry();









  return $registry;
}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
/**
 * Contains all the mappings how to resolve a GraphQL request.
 */
class ResolverRegistry implements ResolverRegistryInterface {

/**
 * Add a field resolver for a certain type.
 *
 * @param string $type
 * @param string $field
 * @param \Drupal\graphql\GraphQL\Resolver\ResolverInterface $resolver
 *
 * @return $this
 */
  public function addFieldResolver($type, $field, ResolverInterface $resolver) {
    $this->fieldResolvers[$type][$field] = $resolver;
    return $this;
  }

}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
public function getResolverRegistry() {
  $registry = new ResolverRegistry();
  $builder = new ResolverBuilder();

  $registry->addFieldResolver('DateTime', 'timestamp', $builder->fromParent());

  return $registry;
}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
public function getResolverRegistry() {
  $registry = new ResolverRegistry();
  $builder = new ResolverBuilder();

  $registry->addFieldResolver('DateTime', 'timestamp', $builder->fromParent());

  $registry->addFieldResolver('Connection', 'edges',
    $builder->produce('connection_edges')->map('connection', $builder->fromParent())
  );

  return $registry;
}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/DataProducer/Connection/ConnectionEdges.php

```php
namespace Drupal\drupalcon\Plugin\GraphQL\DataProducer\Connection;

use Drupal\graphql\Plugin\DataProducerPluginCachingInterface;
use Drupal\graphql\Plugin\GraphQL\DataProducer\DataProducerPluginBase;
use Drupal\drupalcon\GraphQL\ConnectionInterface;

/**
 * Produces the edges from a connection object.
 *
 * @DataProducer(
 *   id = "connection_edges",
 *   name = @Translation("Connection edges"),
 *   description = @Translation("Returns the edges of a connection."),
 *   produces = @ContextDefinition("any",
 *     label = @Translation("Edges")
 *   ),
 *   consumes = {
 *     "connection" = @ContextDefinition("any",
 *       label = @Translation("QueryConnection")
 *     )
 *   }
 * )
 */
class ConnectionEdges extends DataProducerPluginBase
  implements DataProducerPluginCachingInterface {}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/DataProducer/Connection/ConnectionEdges.php

```php
class ConnectionEdges extends DataProducerPluginBase
  implements DataProducerPluginCachingInterface {

  /**
   * Resolves the request.
   *
   * @param \Drupal\social_graphql\GraphQL\ConnectionInterface $connection
   *   The connection to return the edges from.
   *
   * @return mixed
   *   The edges for the connection.
   */
  public function resolve(ConnectionInterface $connection) {
    return $connection->edges();
  }

}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```
public function getResolverRegistry() {
  $registry = new ResolverRegistry();
  $builder = new ResolverBuilder();

  $registry->addFieldResolver('DateTime', 'timestamp', $builder->fromParent());

  $registry->addFieldResolver('Connection', 'edges',
    $builder->produce('connection_edges')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'nodes',
    $builder->produce('connection_nodes')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'pageInfo',
    $builder->produce('connection_page_info')
      ->map('connection', $builder->fromParent())
  );









  return $registry;
}
```

Defining a base schema

modules/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```
public function getResolverRegistry() {
  $registry = new ResolverRegistry();
  $builder = new ResolverBuilder();

  $registry->addFieldResolver('DateTime', 'timestamp', $builder->fromParent());

  $registry->addFieldResolver('Connection', 'edges',
    $builder->produce('connection_edges')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'nodes',
    $builder->produce('connection_nodes')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'pageInfo',
    $builder->produce('connection_page_info')
      ->map('connection', $builder->fromParent())
  );

  $registry->addFieldResolver('Edge', 'cursor',
    $builder->produce('edge_cursor')->map('edge', $builder->fromParent())
  );
  $registry->addFieldResolver('Edge', 'node',
    $builder->produce('edge_node')->map('edge', $builder->fromParent())
  );

  return $registry;
}
```

Defining our schema extension

Utilising inheritance and Drupal's modularity

```
/**
 * @SchemaExtension(
 *   id = "drupalcon_chat_schema_extension",
 *   name = "DrupalCon - Chat Schema Extension",
 *   description = "Extend GraphQL schema with chat functionality.",
 *   schema = "drupalcon"
 * )
 */
```

Our schema extension implements a specialised feature in our API

Defining our schema extension

1. Our schema must by in the Plugin\GraphQL\SchemaExtension namespace to be discovered

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
namespace Drupal\drupalcon_chat\Plugin\GraphQL\SchemaExtension;

use Drupal\graphql\Plugin\GraphQL\SchemaExtension\SdlSchemaExtensionPluginBase;

class ChatSchemaExtension extends SdlSchemaExtensionPluginBase {

}
```

Defining our schema extension

1. Our schema must by in the Plugin\GraphQL\SchemaExtension namespace to be discovered

2. We use the SchemaExtension annotation to define our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
namespace Drupal\drupalcon_chat\Plugin\GraphQL\SchemaExtension;



use Drupal\graphql\Plugin\GraphQL\SchemaExtension\SdlSchemaExtensionPluginBase;

/**
 * @SchemaExtension(
 *   id = "drupalcon_chat",
 *   name = "DrupalCon - Chat",
 *   description = "Extend GraphQL schema with chat functionality.",
 *   schema = "drupalcon"
 * )
 */
class ChatSchemaExtension extends SdlSchemaExtensionPluginBase {









}
```

Defining our schema extension

1. Our schema must by in the Plugin\GraphQL\SchemaExtension namespace to be discovered

2. We use the SchemaExtension annotation to define our schema extension

3. Our only required function is registerResolvers

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
namespace Drupal\drupalcon_chat\Plugin\GraphQL\SchemaExtension;

use Drupal\graphql\GraphQL\ResolverBuilder;
use Drupal\graphql\GraphQL\ResolverRegistryInterface;
use Drupal\graphql\Plugin\GraphQL\SchemaExtension\SdlSchemaExtensionPluginBase;

/**
 * @SchemaExtension(
 *   id = "drupalcon_chat",
 *   name = "DrupalCon - Chat",
 *   description = "Extend GraphQL schema with chat functionality.",
 *   schema = "drupalcon"
 * )
 */
class ChatSchemaExtension extends SdlSchemaExtensionPluginBase {

  /**
   * {@inheritdoc}
   */
  public function registerResolvers(ResolverRegistryInterface $registry) : void {
    $builder = new ResolverBuilder();

    /* Chat Resolvers */
  }

}
```

Defining our schema extension

modules/drupalcon_chat/graphql/drupalcon_chat.base.graphqls

Defines new types for the chat functionality

Ideally in your version you include comments as shown during the design phase

```
type ChatMessage implements Node {
  id: ID!
  sent: DateTime!
  sender: Actor
  content: ChatMessageContent!
}

union ChatMessageContent =
  | MediaChatMessageContent
  | UserEventChatMessageContent
  | DeletedChatMessageContent

type MediaChatMessageContent {
  text: String
}

type UserEventChatMessageContent {
  eventType: ChatUserEventType!
  subject: User!
}

enum ChatUserEventType {
  CONVERSATION_CREATED
  JOIN
  PART
}

type DeletedChatMessageContent {
  _: Boolean
}

type ChatMessageConnection
implements Connection {
  pageInfo: PageInfo!
  edges: [ChatMessageEdge!]!
  nodes: [ChatMessage!]!
}

type ChatMessageEdge implements Edge {
  cursor: Cursor!
  node: ChatMessage!
}

input ViewerSendUserChatMessageInput {
  conversation: ID!
  contents: MediaChatMessageContentInput!
}

input MediaChatMessageContentInput {
  text: String!
}

type ViewerSendUserChatMessagePayload {
  errors: [Violation!]
  message: ChatMessage
}
```

Defining our schema extension

modules/drupalcon_chat/graphql/drupalcon_chat.extension.graphqls

```
extend type Query {
  chatMessages(
    first: Int, after: Cursor,
    last: Int, before: Cursor,
    reverse: Boolean = false
  ) : ChatMessageConnection!
  chatMessage(id: ID!) : ChatMessage
}

extend type Mutation {
  viewerSendUserChatMessage(
    input: ViewerSendUserChatMessageInput!
  ) : ViewerSendUserChatMessagePayload
}

extend type Subscription {
  chatMessageReceived : ChatMessage!
}
```

Defines type extensions for the chat functionality

Ideally in your version you include comments :)

Defining our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
namespace Drupal\drupalcon_chat\Plugin\GraphQL\SchemaExtension;

use Drupal\graphql\GraphQL\ResolverBuilder;
use Drupal\graphql\GraphQL\ResolverRegistryInterface;
use Drupal\graphql\Plugin\GraphQL\SchemaExtension\SdlSchemaExtensionPluginBase;

/**
 * @SchemaExtension(
 *   id = "drupalcon_chat",
 *   name = "DrupalCon - Chat",
 *   description = "Extend GraphQL schema with chat functionality.",
 *   schema = "drupalcon"
 * )
 */
class ChatSchemaExtension extends SdlSchemaExtensionPluginBase {

  /**
   * {@inheritdoc}
   */
  public function registerResolvers(ResolverRegistryInterface $registry) : void {
    $builder = new ResolverBuilder();

    /* Chat Resolvers */
  }

}
```

Defining our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();









}
```

Defining our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();









  $registry->addFieldResolver('Query', 'chatMessage',
    $builder->produce('entity_load_by_uuid')
      ->map('type', $builder->fromValue('chat_message'))
      ->map('uuid', $builder->fromArgument('id'))
  );









}
```

Defining our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();









  $registry->addFieldResolver('Query', 'chatMessages',
    $builder->produce('messages')
      ->map('after', $builder->fromArgument('after'))
      ->map('before', $builder->fromArgument('before'))
      ->map('first', $builder->fromArgument('first'))
      ->map('last', $builder->fromArgument('last'))
      ->map('reverse', $builder->fromArgument('reverse'))
  );









}
```

The Messages DataProducer

Implementation on https://github.com/goalgorilla/open_social/ in the Drupal\social_graphql\GraphQL namespace

DataProducer's with pagination return a EntityConnection instance.

EntityConnection contains the pagination logic and takes a ConnectionQueryHelperInterface as only argument.

The implementation of the ConnectionQueryHelperInterface contains:

- The creation of an EntityQuery
- Cursor to object hydration
- Id field configuration
- Sort field configuration
- Creation of the loader promise and object to cursor transformation

See for example Drupal\social_topic\Plugin\GraphQL\QueryHelper\TopicQueryHelper

Defining our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();









  $registry->addFieldResolver('DeletedChatMessageContent', '_',
    $builder->fromValue(NULL)
  );









}
```

Defining our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();









  $registry->addTypeResolver(
    'ChatMessageContent',
    [ChatTypeResolver::class, 'resolveMessageContentType']
  );









}
```

Defining our schema extension

```php
namespace Drupal\drupalcon_chat;

/**
 * Type resolver functions for GraphQL types in social_chat.
 */
class ChatTypeResolver {
  /**
   * Find the concrete type for a MessageContent interface implementation.
   *
   * @param \Drupal\drupalcon_chat\ChatMessageContentInterface $message_content
   *   The message content that should be mapped to a concrete type.
   *
   * @return string
   *   The concrete GraphQL type name.
   */
  public static function resolveMessageContentType(ChatMessageContentInterface $message_content) : string {
    switch ($message_content->getType()) {
      case 'deleted':
        return 'DeletedChatMessageContent';
      case 'media':
        return 'MediaChatMessageContent';
      case 'user_event':
        return 'UserEventChatMessageContent';
      default:
        throw new \RuntimeException("Can not map type '{$message_content->getType()}' to GraphQL Schema");
    }
  }
}
```

Defining our schema extension

modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();



  $this->registerMutationResolver($registry, $builder, 'viewerSendUserChatMessage');

}
```

Defining our schema extension

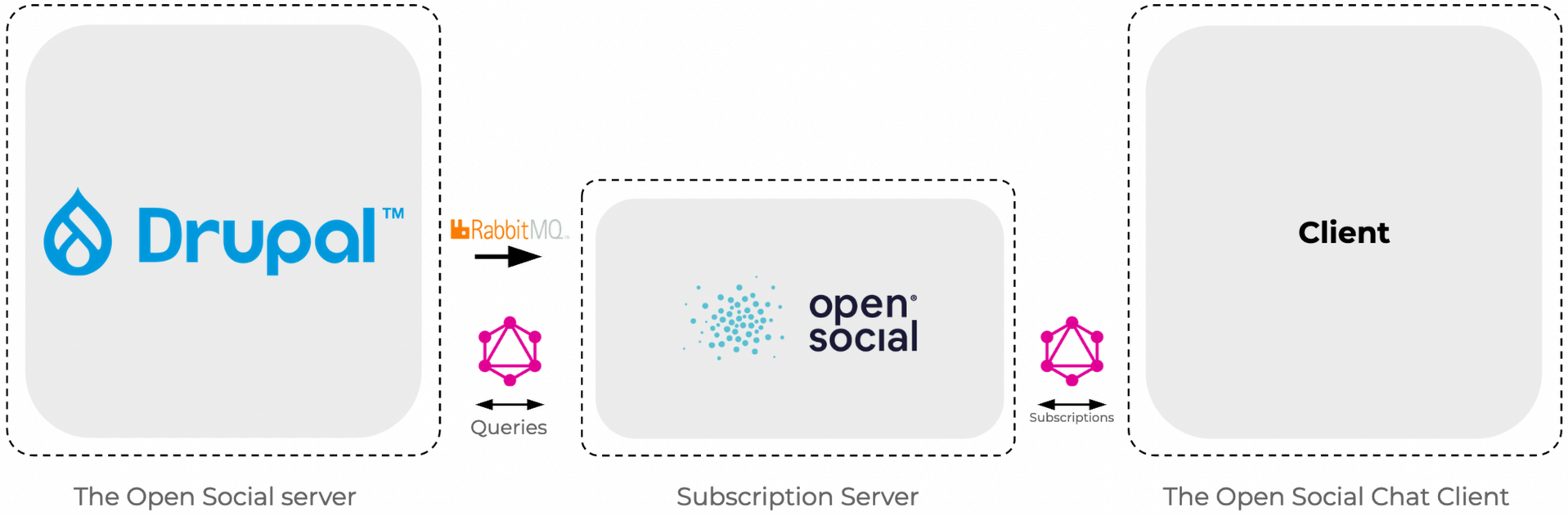
modules/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();



  $this->registerMutationResolver($registry, $builder, 'viewerSendUserChatMessage');









  // Equivalent to the following under the hood.
  $field_name = camelCaseTOsnake_case($fieldName);
  $registry->addFieldResolver('Mutation', $fieldName,
    $builder->compose(
      $builder->produce($field_name . "_input")
        ->map('input', $builder->fromArgument('input')),
      $builder->produce($field_name)
        ->map('input', $builder->fromParent())
    )
  );



}
```

Adding a Subscription Server

Subscription Handling

Real-Time Drupal!



https://www.alexandervarwijk.com/talks/2021-12-10-serving-graphql-subscriptions-using-php-and-drupal

Subscription Handling

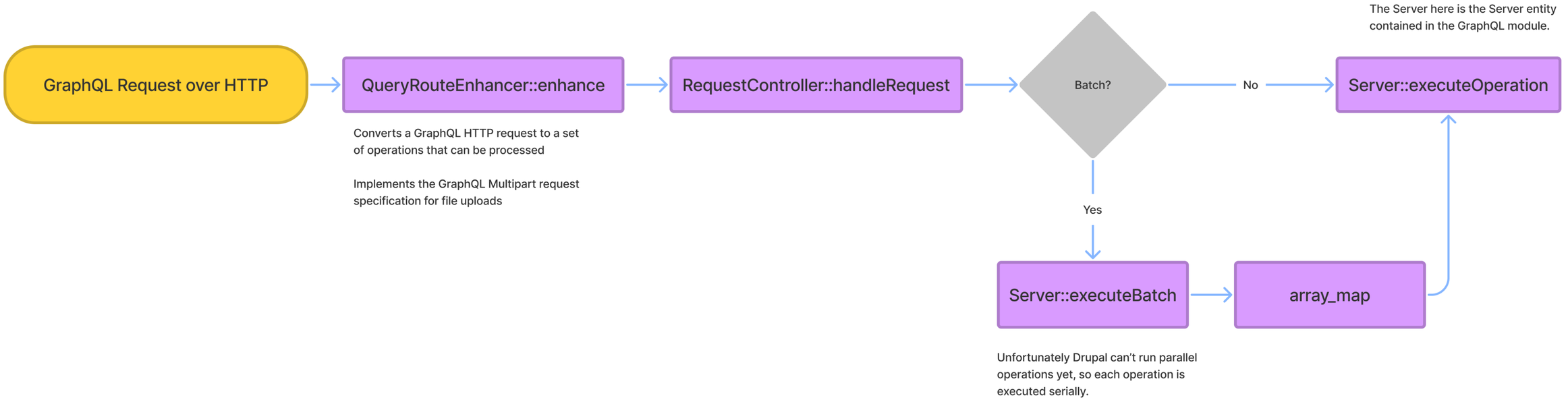
```
type Query {
  chatMessage(id: ID!) : ChatMessage
}

type Subscription {
  chatMessageReceived : ChatMessage!
}
```

Subscription Handling

```php
$reference_query = Parser::parse(<<<GRAPHQL
  query ChatMessage(\$id: ID!) {
    chatMessage(id: \$id) { id }
  }
GRAPHQL);

$query = Printer::doPrint(
  $this->transformSubscriptionForConcreteType(
    $subscription->document(),
    $reference_query,
    $subscription->operationName()
  )
);

// Use the GraphQL client created for the subscription to ask Open
// Social for the fields needed.
/** @var \OpenSocial\RealTime\GraphQL\GraphQLClient $graphql */
$graphql = $subscription->client()->GraphQL;

/** @var \React\Promise\ExtendedPromiseInterface $promise */
$promise = $graphql->execute($query, $subscription->operationName(), ['id' => $data['message']]);
// Pass the returned data on to the subscription.
$promise->done(
    fn (ExecutionResult $result) => $subscription->client()->send(
      json_encode((new NextMessage($subscription_id, $result))->jsonSerialize())
    )
  );
```

Inside the GraphQL Module

The journey of a GraphQL Request



The journey of a GraphQL Request

```
public function executeOperation(OperationParams $operation) {









}
```

The journey of a GraphQL Request

1. Store previous executor

```
public function executeOperation(OperationParams $operation) {
1 $previous = Executor::getImplementationFactory();









}
```

The journey of a GraphQL Request

1. Store previous executor
2. Use our own executor (that's what interfaces with Drupal)

```
public function executeOperation(OperationParams $operation) {
1 $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
2   'create',
  ]);









}
```

The journey of a GraphQL Request

1. Store previous executor
2. Use our own executor (that's what interfaces with Drupal)
3. Get GraphQL configuration

```
public function executeOperation(OperationParams $operation) {
1 $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
2   'create',
  ]);

  try {
3   $config = $this->configuration();









}
```

The journey of a GraphQL Request

1. Store previous executor
2. Use our own executor (that's what interfaces with Drupal)
3. Get GraphQL configuration
4. Use GraphQL Library Helper to execute operation

```
public function executeOperation(OperationParams $operation) {
1 $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
2   'create',
  ]);

  try {
3   $config = $this->configuration();
4   $result = (new Helper())->executeOperation($config, $operation);









  return $result;
}
```

The journey of a GraphQL Request

1. Store previous executor
2. Use our own executor (that's what interfaces with Drupal)
3. Get GraphQL configuration
4. Use GraphQL Library Helper to execute operation
5. Ensure Drupal always has cache information for the result

```
public function executeOperation(OperationParams $operation) {
1 $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
2   'create',
  ]);

  try {
3   $config = $this->configuration();
4   $result = (new Helper())->executeOperation($config, $operation);

    // In case execution fails before the execution stage, we have to wrap the
    // result object here.
    if (!($result instanceof CacheableExecutionResult)) {
      $result = new CacheableExecutionResult($result->data, $result->errors, $result->extensions);
5     $result->mergeCacheMaxAge(0);
    }
  }



  return $result;
}
```

The journey of a GraphQL Request

1. Store previous executor
2. Use our own executor (that's what interfaces with Drupal)
3. Get GraphQL configuration
4. Use GraphQL Library Helper to execute operation
5. Ensure Drupal always has cache information for the result
6. Restore previous executor

```
public function executeOperation(OperationParams $operation) {
1 $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
2   'create',
  ]);

  try {
3   $config = $this->configuration();
4   $result = (new Helper())->executeOperation($config, $operation);

    // In case execution fails before the execution stage, we have to wrap the
    // result object here.
    if (!($result instanceof CacheableExecutionResult)) {
      $result = new CacheableExecutionResult($result->data, $result->errors, $result->extensions);
5     $result->mergeCacheMaxAge(0);
    }
  }
  finally {
6   Executor::setImplementationFactory($previous);
  }

  return $result;
}
```

Configuring the GraphQL PHP Library

```
public function configuration() {









}
```

Configuring the GraphQL PHP Library

```php
public function configuration() {
  $params = \Drupal::getContainer()->getParameter('graphql.config');
  /** @var \Drupal\graphql\Plugin\SchemaPluginManager $manager */
  $manager = \Drupal::service('plugin.manager.graphql.schema');
  $schema = $this->get('schema');

  /** @var \Drupal\graphql\Plugin\SchemaPluginInterface $plugin */
  $plugin = $manager->createInstance($schema);
  if ($plugin instanceof ConfigurableInterface && $config = $this->get('schema_configuration')) {
    $plugin->setConfiguration($config[$schema] ?? []);
  }









}
```

Configuring the GraphQL PHP Library

```php
public function configuration() {
  $params = \Drupal::getContainer()->getParameter('graphql.config');
  /** @var \Drupal\graphql\Plugin\SchemaPluginManager $manager */
  $manager = \Drupal::service('plugin.manager.graphql.schema');
  $schema = $this->get('schema');

  /** @var \Drupal\graphql\Plugin\SchemaPluginInterface $plugin */
  $plugin = $manager->createInstance($schema);
  if ($plugin instanceof ConfigurableInterface && $config = $this->get('schema_configuration')) {
    $plugin->setConfiguration($config[$schema] ?? []);
  }

  // Create the server config.
  $registry = $plugin->getResolverRegistry();
  $server = ServerConfig::create();
  $server->setDebugFlag($this->get('debug_flag'));
  $server->setQueryBatching(!!$this->get('batching'));
  $server->setValidationRules($this->getValidationRules());
  $server->setPersistentQueryLoader($this->getPersistedQueryLoader());
  $server->setSchema($plugin->getSchema($registry));
  $server->setPromiseAdapter(new SyncPromiseAdapter());
  $server->setContext($this->getContext($plugin, $params));
  $server->setFieldResolver($this->getFieldResolver($registry));

  return $server;
}
```

The journey of a GraphQL Request

```php
public function executeOperation(OperationParams $operation) {
  $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
    'create',
  ]);

  try {
    $config = $this->configuration();
    $result = (new Helper())->executeOperation($config, $operation);

    // In case execution fails before the execution stage, we have to wrap the
    // result object here.
    if (!($result instanceof CacheableExecutionResult)) {
      $result = new CacheableExecutionResult($result->data, $result->errors, $result->extensions);
      $result->mergeCacheMaxAge(0);
    }
  }
  finally {
    Executor::setImplementationFactory($previous);
  }

  return $result;
}
```

The journey of a GraphQL Request

Server::executeOperation → Helper::executeOperation → GraphQL::promiseToExecute → Executor::promiseToExecute → Executor::doExecute

- Ensures the server has a schema
- Ensures the server supports query batching if this is a batched request
- Validates the parameters for the operation
- Loads persisted query if queryId is provided
- Determines the operation to execute
- Ensures HTTP GET is only used for Query and no other operation types

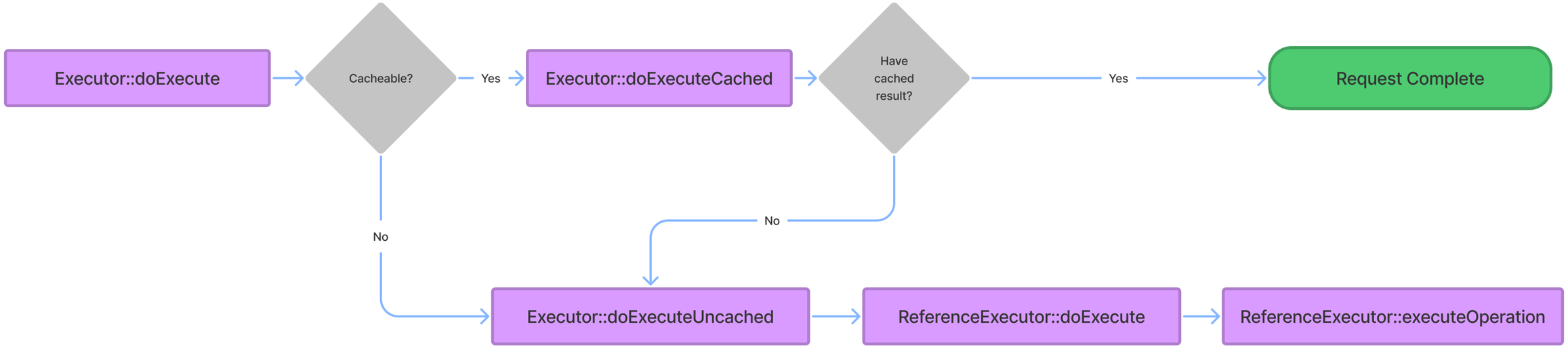
Applies error handling to the result if needed

- Applies our configured operation validation rules. Returns an error if any validation rules are broken.

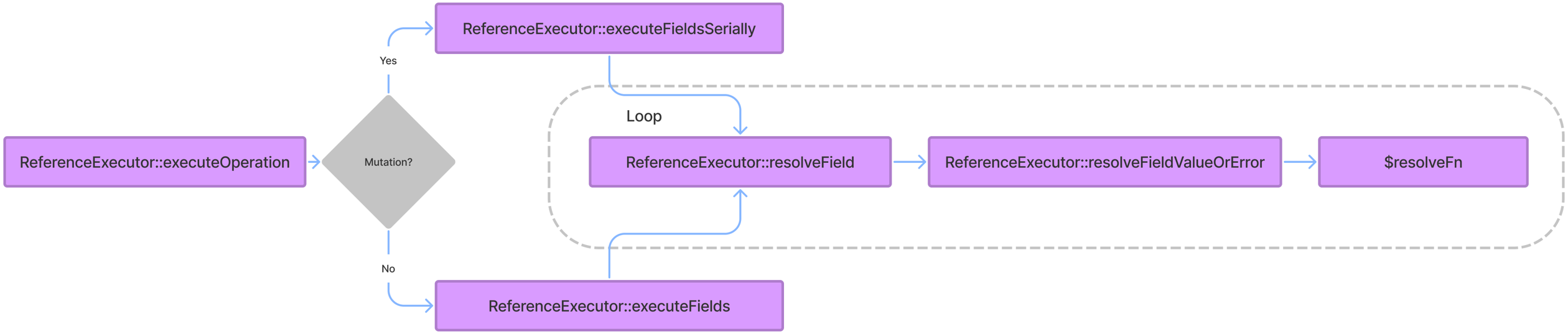
- This happens in the GraphQL library
- Creates an instance of the configured executor

- This is now the Executor of the GraphQL module

The journey of a GraphQL Request



The journey of a GraphQL Request



The journey of a GraphQL Request

```php
/**
 * Returns the default field resolver.
 *
 * @todo Handle this through configuration on the server.
 *
 * Fields that don't explicitly declare a field resolver will use this one
 * as a fallback.
 *
 * @param \Drupal\graphql\GraphQL\ResolverRegistryInterface $registry
 *   The resolver registry.
 *
 * @return null|callable
 *   The default field resolver.
 */
protected function getFieldResolver(ResolverRegistryInterface $registry) {
  return function ($value, $args, ResolveContext $context, ResolveInfo $info) use ($registry) {
    $field = new FieldContext($context, $info);
    $result = $registry->resolveField($value, $args, $context, $info, $field);
    return DeferredUtility::applyFinally($result, function ($result) use ($field, $context) {
      if ($result instanceof CacheableDependencyInterface) {
        $field->addCacheableDependency($result);
      }

      $context->addCacheableDependency($field);
    });
  };
}
```

The journey of a GraphQL Request

```
      function ($value, $args, ResolveContext $context, ResolveInfo $info) use ($registry) {
    $field = new FieldContext($context, $info);
    $result = $registry->resolveField($value, $args, $context, $info, $field);
    return DeferredUtility::applyFinally($result, function ($result) use ($field, $context) {
      if ($result instanceof CacheableDependencyInterface) {
        $field->addCacheableDependency($result);
      }

      $context->addCacheableDependency($field);
    });
  }
```

The journey of a GraphQL Request

$value The value of the parent type for the field we're resolving. For root fields this is rootValue from our ServerConfig

$args The arguments provided to the field in the query

$context The context we set in our ServerConfig. A callable that returns a new ResolveConfig instance to keep track of cache metadata

$info Information that may be useful to the field resolver: the field definition, the field name, the expected return type, the nodes in the query referencing the field, the parent type, the path to this field from the root value, the schema used for execution, all fragments in the query, the root value for query execution, the AST of the operation, and variables passed to the query execution

```
      function ($value, $args, ResolveContext $context, ResolveInfo $info) use ($registry) {
  $field = new FieldContext($context, $info);
  $result = $registry->resolveField($value, $args, $context, $info, $field);
  return DeferredUtility::applyFinally($result, function ($result) use ($field, $context) {
    if ($result instanceof CacheableDependencyInterface) {
      $field->addCacheableDependency($result);
    }

    $context->addCacheableDependency($field);
  });
}
```

Resolving a field with our ResolverRegistry

```
public function resolveField($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
  // First, check if there is a resolver registered for this field.
  if ($resolver = $this->getRuntimeFieldResolver($value, $args, $context, $info)) {
    if (!$resolver instanceof ResolverInterface) {
      throw new \LogicException(sprintf('Field resolver for field %s on type %s is not callable.', $info->fieldName, $info->parentType->name));
    }

    return $resolver->resolve($value, $args, $context, $info, $field);
  }

  return call_user_func($this->defaultFieldResolver, $value, $args, $context, $info, $field);
}




protected function getRuntimeFieldResolver($value, $args, ResolveContext $context, ResolveInfo $info) {
  return $this->getFieldResolverWithInheritance($info->parentType, $info->fieldName);
}
```

Resolving a field with our ResolverRegistry

```php
public function getFieldResolverWithInheritance(Type $type, string $fieldName) : ?ResolverInterface {
  if ($resolver = $this->getFieldResolver($type->name, $fieldName)) {
    return $resolver;
  }

  if (!$type instanceof ImplementingType) {
    return NULL;
  }

  // Go through the interfaces implemented for the type on which this field is
  // resolved and check if they lead to a field resolution.
  foreach ($type->getInterfaces() as $interface) {
    if ($resolver = $this->getFieldResolverWithInheritance($interface, $fieldName)) {
      return $resolver;
    }
  }

  return NULL;
}



public function getFieldResolver($type, $field) {
  return $this->fieldResolvers[$type][$field] ?? NULL;
}
```

Resolving a field with our ResolverRegistry

```
public function resolveField($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
  // First, check if there is a resolver registered for this field.
  if ($resolver = $this->getRuntimeFieldResolver($value, $args, $context, $info)) {
    if (!$resolver instanceof ResolverInterface) {
      throw new \LogicException(sprintf('Field resolver for field %s on type %s is not callable.', $info->fieldName, $info->parentType->name));
    }

    return $resolver->resolve($value, $args, $context, $info, $field);
  }

  return call_user_func($this->defaultFieldResolver, $value, $args, $context, $info, $field);
}




protected function getRuntimeFieldResolver($value, $args, ResolveContext $context, ResolveInfo $info) {
  return $this->getFieldResolverWithInheritance($info->parentType, $info->fieldName);
}
```

Resolving a field with our ResolverRegistry

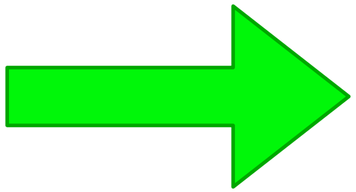
```php
/**
 * Resolves by an argument name lookup.
 */
class Argument implements ResolverInterface {

  protected string $name;

  /**
   * @param string $name
   *   Name of the argument.
   */
  public function __construct($name) {
    $this->name = $name;
  }

  /**
   * {@inheritdoc}
   */
  public function resolve($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
    return $args[$this->name] ?? NULL;
  }

}
```

Resolving a field with our ResolverRegistry

```
public function resolve($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
  $plugin = $this->prepare($value, $args, $context, $info, $field);

  return DeferredUtility::returnFinally($plugin, function (DataProducerPluginInterface $plugin) use
($context, $field) {
    foreach ($plugin->getContexts() as $item) {
      /** @var \Drupal\Core\Plugin\Context\Context $item */
      if ($item->getContextDefinition()->isRequired() && !$item->hasContextValue()) {
        return NULL;
      }
    }

    if ($this->cached && $plugin instanceof DataProducerPluginCachingInterface) {
      if (!!$context->getServer()->get('caching')) {
        return $this->resolveCached($plugin, $context, $field);
      }
    }

    return $this->resolveUncached($plugin, $context, $field);
  });
}
```

Resolving a field with our ResolverRegistry

```
query {
    chatMessages(first: 1) {
        pageInfo {
            hasPreviousPage
            hasNextPage
            startCursor
            endCursor
        }
        edges {
            node {
                id
                sender {
                    id
                    displayName
                }
                sent {
                    timestamp
                }
                content {
                    ... on MediaChatMessageContent {
                        text
                    }
                }
            }
        }
    }
}
```




```
{
  "data": {
    "chatMessages": {
      "pageInfo": {
        "hasPreviousPage": false,
        "hasNextPage": true,
        "startCursor": “TRUNCATED",
        "endCursor": "TRUNCATED"
      },
      "edges": [
        {
          "node": {
            "id": "4bcb1252-e27d-4f2f-abd3-4f4f64cd2584",
            "sender": {
              "id": "c61e077f-f4c1-4b40-a403-869ee9a6e944",
              "displayName": "DrupalCon"
            },
            "sent": {
              "timestamp": "1649750295"
            },
            "content": {
              "text": "Welcome to DrupalCon!"
            }
          }
        }
      ]
    }
  }
}
```

What are we working on?

OAuth2 Integration

Using scopes to authorise fields in your GraphQL API

Simple OAuth (OAuth2) & OpenID Connect

Issues for Simple OAuth (OAuth2) & OpenID Connect

Create a new issue Advanced search E-mail notifications

Search for

Status – Open issues – **Priority** – Any – **Category** – Any –

Version – Any 6.* – **Component** – Any – Search

Displaying 1 – 7 of 7

Title	Status	Priority	Category	Version	Component	Replies	Last updated	Assigned to	Created
Decoupled (static/dynamic) scopes	Needs work	Normal	Feature request	6.0.x-dev	Code	9	4 days 7 hours	bojan_dev	1 month 4 weeks
Enhance/update consumer entity new	Needs review	Normal	Feature request	6.0.x-dev	Code	8 2 new	1 week 4 days		1 month 4 weeks
Create dedicated authorization server service and implement new (scope/consumer) data model new	Active	Normal	Feature request	6.0.x-dev	Code	3 1 new	3 weeks 4 days		1 month 4 weeks
Migrate roles (used as scopes) to the new "Scope" entity new	Active	Normal	Feature request	6.0.x-dev	Code	1 1 new	3 weeks 4 days		3 weeks 4 days
Support generating static scopes via Drupal console or drush new	Active	Normal	Feature request	6.0.x-dev	Code	1 1 new	3 weeks 5 days		3 weeks 5 days
Introspection/debug response should be conform OAuth2 specs new	Active	Normal	Feature request	6.0.x-dev	Code	4 1 new	3 weeks 5 days		1 month 4 weeks
Make registration possible in the authorize flow	Active	Normal	Feature request	6.0.x-dev	Code	2	1 month 2 days		1 month 4 weeks

Coming up: Simple OAuth 6.0

Schema Linting

Sticking to the promise you made to your API clients

Make sure we don’t break backwards compatibility in our API

Ensure this works with Drupal’s modularity

```
~ $ graphql-schema-linter schema/**/*.graphql
schema/comment.graphql
2:1  The enum `CommentStatus` should be sorted alphabetically.  enum-values-sorted-alphabetically
10:1 The object type `Comment` is missing a description.        types-have-descriptions

schema/query.graphql
4:20 The field `Query.users` is deprecated but has no deprecation reason.  deprecations-have-a-reason

schema/user.graphql
2:6 The object type `user` should start with a capital letter.       types-are-capitalized
2:1 The type `user` is defined in the schema but not used anywhere.  defined-types-are-used
3:3 The field `user.email` is missing a description.                 fields-have-descriptions

✖ 6 errors detected
```

Rate Limiting

Prevent malicious users from using all the server resources

Implement leaky-bucket rate limiting to limit number of requests per timespan

Implement query complexity analysis and complexity limiting.

Want more?

Testing
Directives
Schema Design
Custom validation
QueryLoader Plugins

#graphql on drupal.slack.com

twitter.com/Kingdutch

Thank you

Alexander Varwijk