DART

Bullseye: Your First Dart App

Google IO Codelab - 2012

Bullseye

Your first Dart app

Dart Codelab for Google IO 2012

Instructors

Seth Ladd
Developer Advocate
@sethladd
+Seth Ladd

Jaime Wren
Software Engineer, Dart Editor
+Jaime Wren

TAs

- Kathy Walrath, Technical Writer
- Devon Carew, Software Engineer
- Dan Grove, Eng Manager
- Jacob Richman, Software Engineer
- Vijay Menon, Software Engineer

Introduction

This codelab will help you build and run a simple chat application using Dart. Along the way, you will learn:

- The fundamentals of the Dart programming language
- How to use the Dart HTML libraries
- Bi-directional communication with WebSockets
- The basics of Dart Editor
- How to find answers from the online resources

Prerequisites

This codelab assumes you are familiar with fundamental web programming. You should know the basics of HTML, CSS, and JavaScript. You should also have experience with a programming language such as Java, C#, or JavaScript.

This codelab assumes you have watched the following video tutorials:

- Google I/O 101: Introduction to Dart with Seth Ladd[1] on YouTube
- Google I/O 101: Dart Editor with Devon Carew[2] on YouTube

Installs

This codelab requires the Dart Editor. You can find builds of the editor for Mac, Windows, and Linux at http://dartlang.org/editor/.

[1] http://youtu.be/vT1KmTQ-1Os
[2] http://youtu.be/9PHMKzgrmxE

Additional materials

This codelab provides easy to follow, step-by-step instructions. However, you'll find it quite useful to have the following online resources loaded up and ready to access.

- http://api.dartlang.org
  - The Dart API docs list the functions, classes, and methods for all the libraries.
- http://www.dartlang.org/docs/language-tour/
  - The Dart Language Tour is your high-level guide through the Dart language.
- http://www.dartlang.org/docs/library-tour/
  - The Dart Library Tour covers most of the bundled libraries of the Dart platform.
- http://synonym.dartlang.org/
  - Know JavaScript? This is your resource! Translate common idioms, patterns, and snippets from JavaScript to Dart.

Step 1: Set up your environment

Dart offers better productivity through powerful and helpful tools. At the center of the toolchain is the Dart Editor, a lightweight text editor that understands how to analyze, run, and debug Dart apps. The editor works with the Dart SDK and Dartium (a build of Chromium with the Dart VM) to give you an integrated experience.

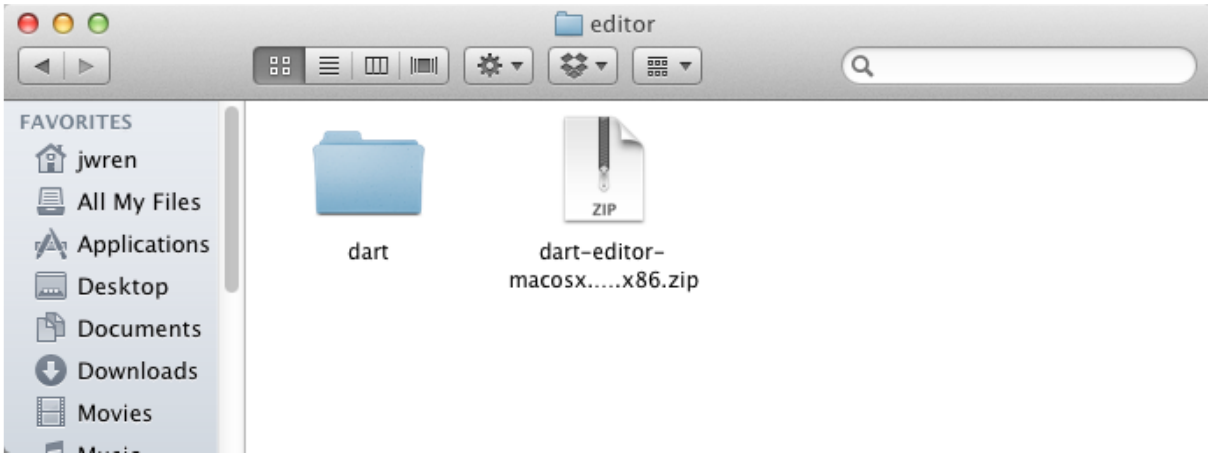
Objectives

1. Install Dart Editor
2. Send feedback to the editor team
3. Run the sample clock Dart app
4. Learn about Dartium

Walkthrough

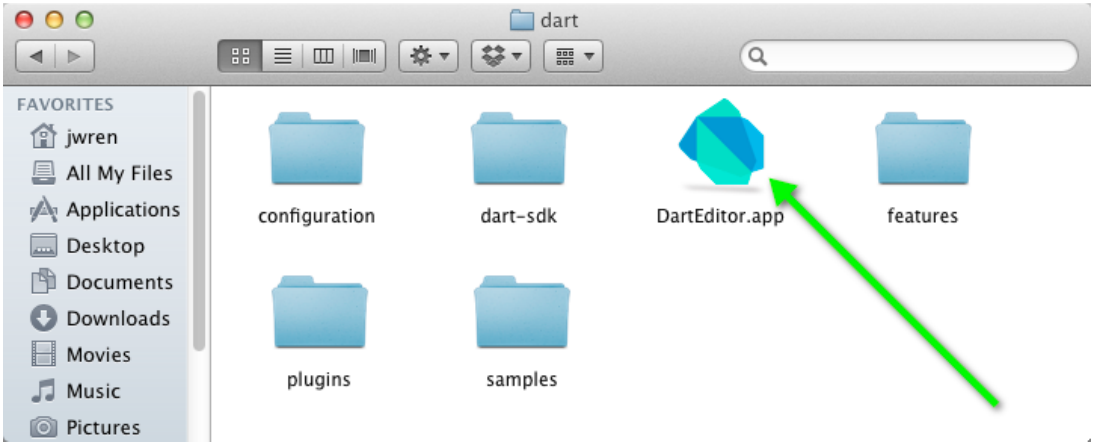
Install the Dart Editor

To get your environment set up, plug in the provided USB. Open the USB drive and find the `editor/` directory inside. Copy over the correct Dart Editor version for your OS/bit combination directory to your machine, and unzip it.

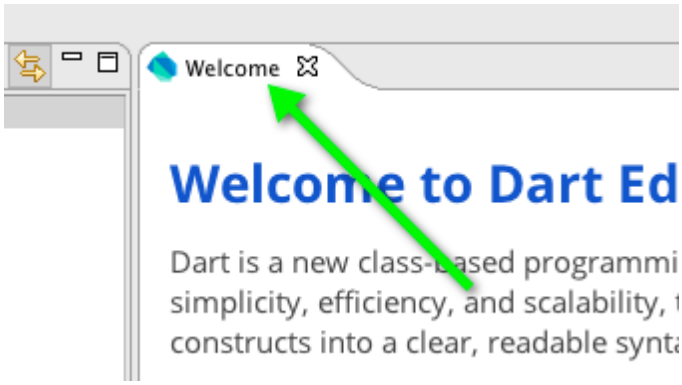


Open up the newly unzipped `dart/` directory, and double click the executable:

- `DartEditor.app` (Mac)
- `DartEditor` (Linux)
- `DartEditor.exe` (Windows)



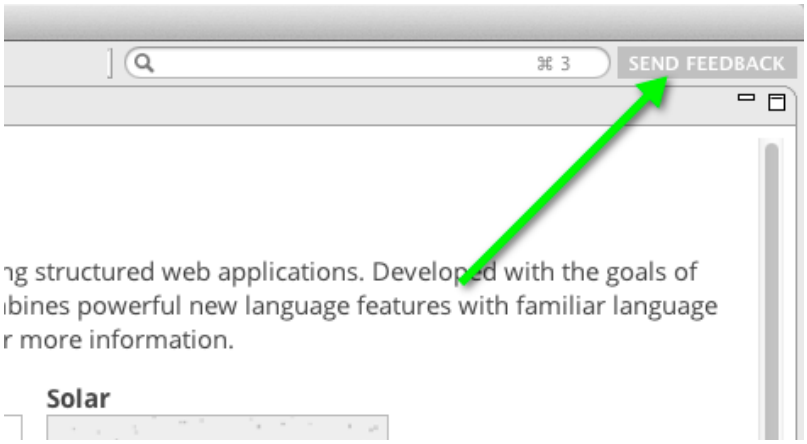
When Dart Editor first opens, the Welcome window appears.



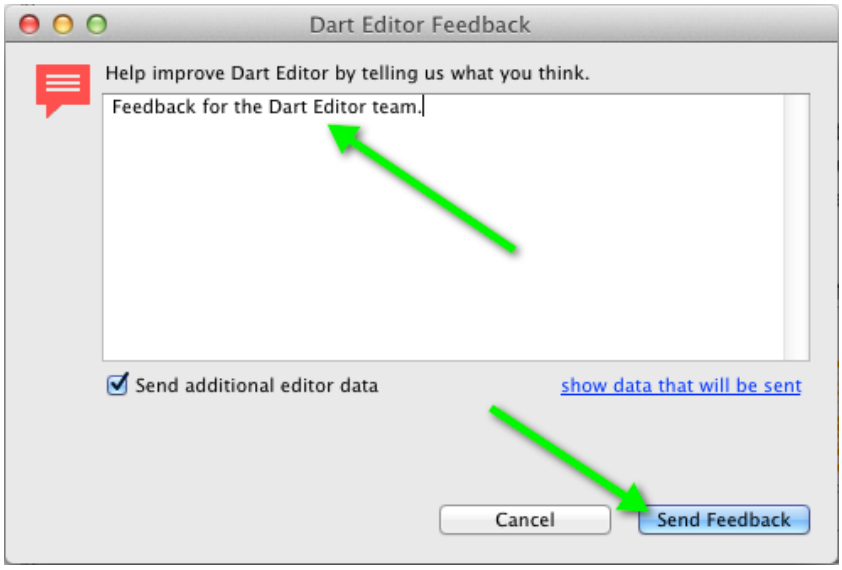
Use the Send Feedback button

The Dart Editor team really appreciates feedback. The easiest way to let them know your thoughts is to use the Send Feedback button in the upper right of the editor's toolbar.

Locate the Send Feedback button and click on it.



The Dart Editor Feedback dialog allows you to share bugs and requests directly with the editor team, as well as the larger Dart team. Feel free to send us any and all comments, especially during this codelab. We'll turn your suggestions into bug reports and feature requests[3] as appropriate.



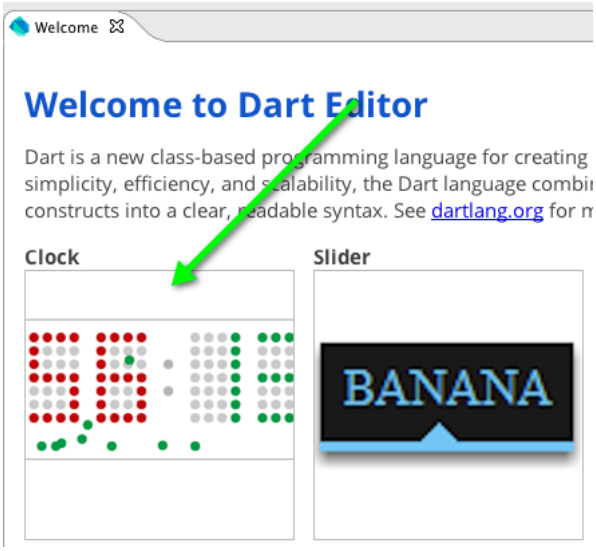
Run the Clock sample

(If the feedback window is still open, close it now.)

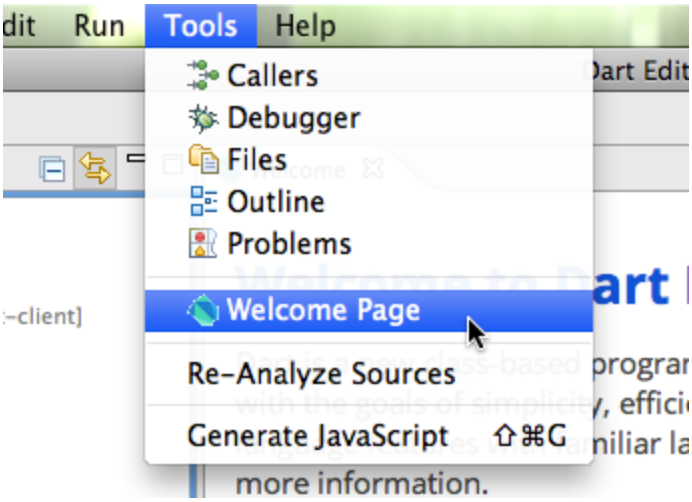
[3] http://dartbug.com is your direct link to our issue tracker.

Time to run a Dart app!

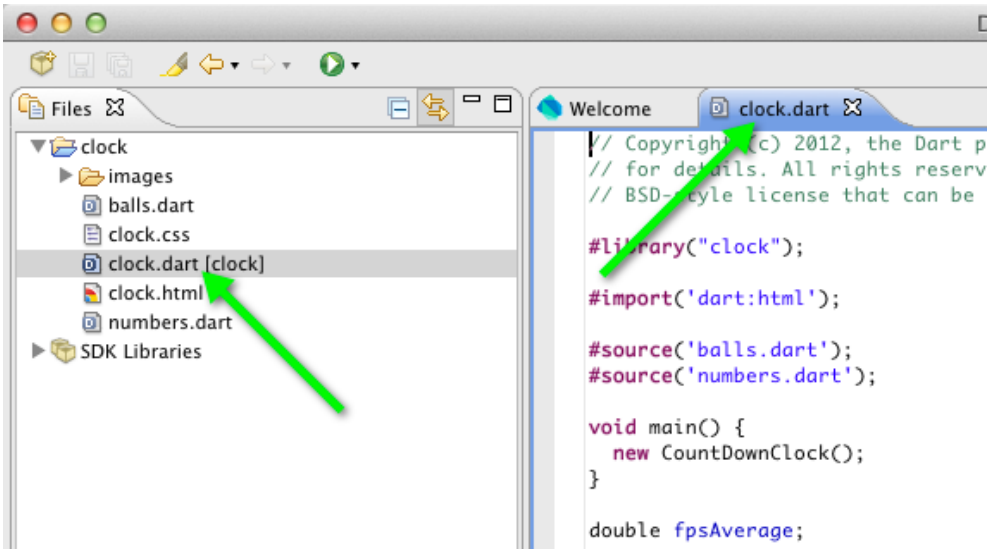
Click on the Clock sample from the Welcome window, which copies the `Clock` sample into Dart Editor and sets up a new project for you.



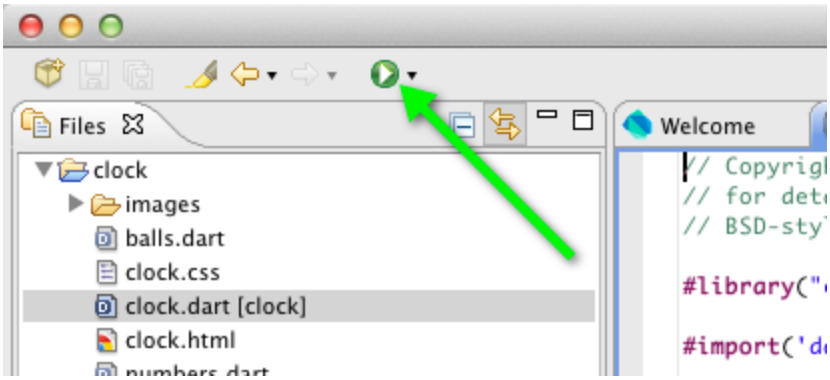
Tip: Can't find the Welcome view? You can go to Tools, Welcome Page to display it again.



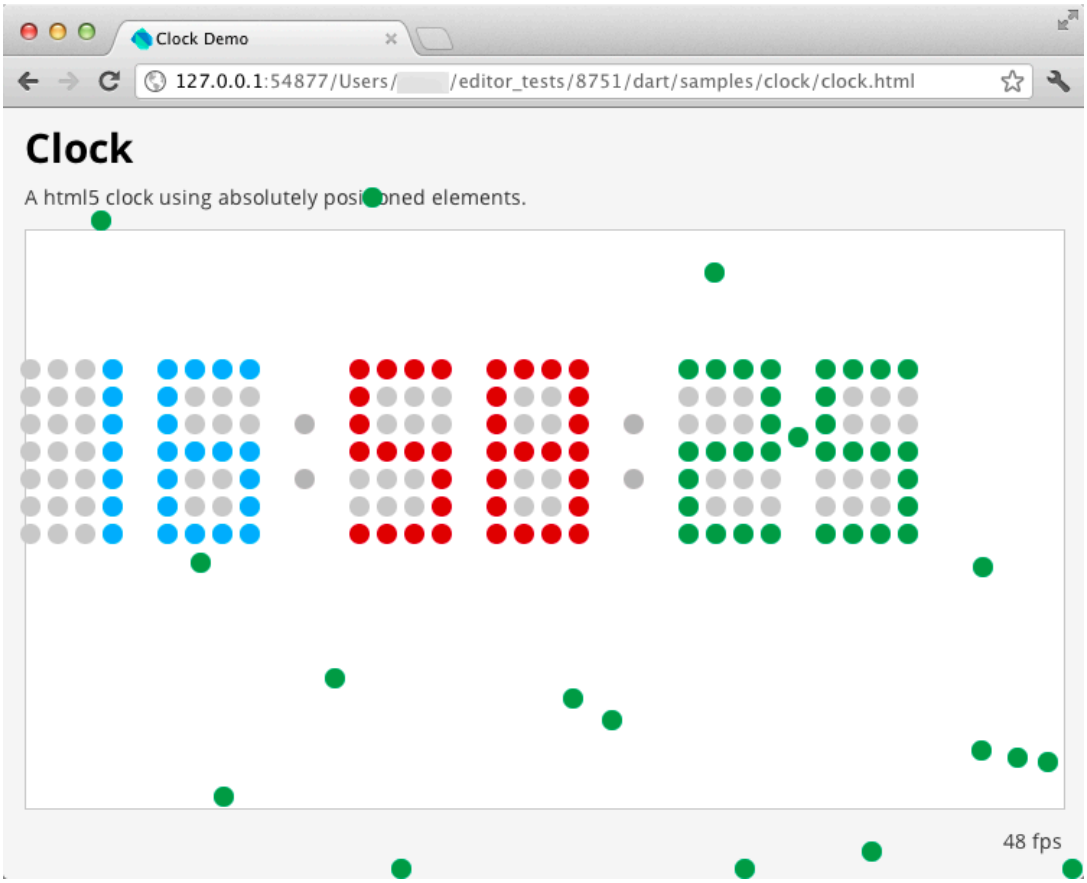
The Files view shows all of the files in the Clock sample, including all Dart files, the HTML file that hosts the app, as well as all of the images and CSS files. `clock.dart` is a Dart file that defines the "clock" library, and includes the `main()` method for the sample. The `clock.dart` file is automatically opened in the editor.



Ensure the `clock.dart` file is selected and highlighted. Click the Run button in Dart Editor, which launches the application by loading Dartium and pointing it to the `clock.html` file.



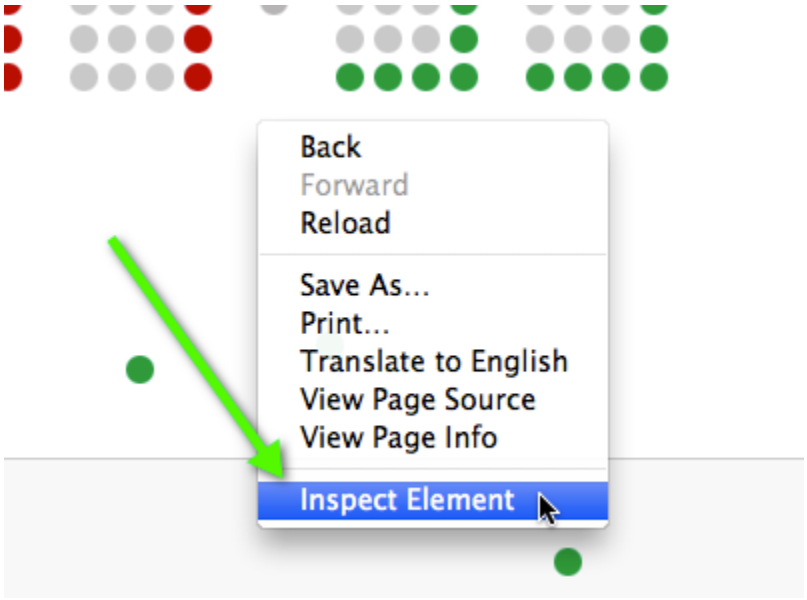
Here is the Clock sample app running inside Dartium. Congratulations, you are running your first Dart app!



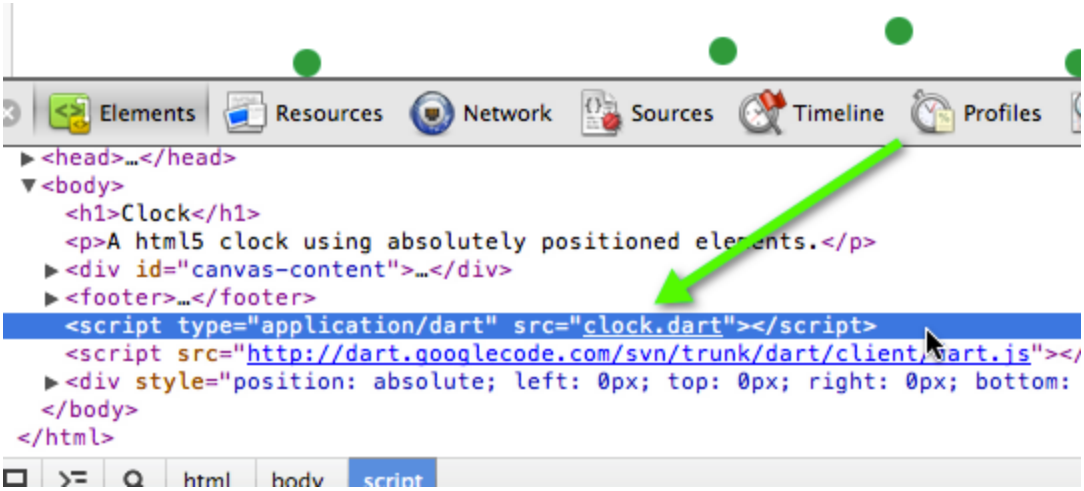
Dartium

Dartium is Chromium with an embedded Dart virtual machine (VM). Even though Dart compiles to JavaScript, you can speed up the "edit, reload" development cycle by running Dart apps directly in the browser.

To verify that Clock is running in Dartium, right-click in the browser and select `Inspect Element`.

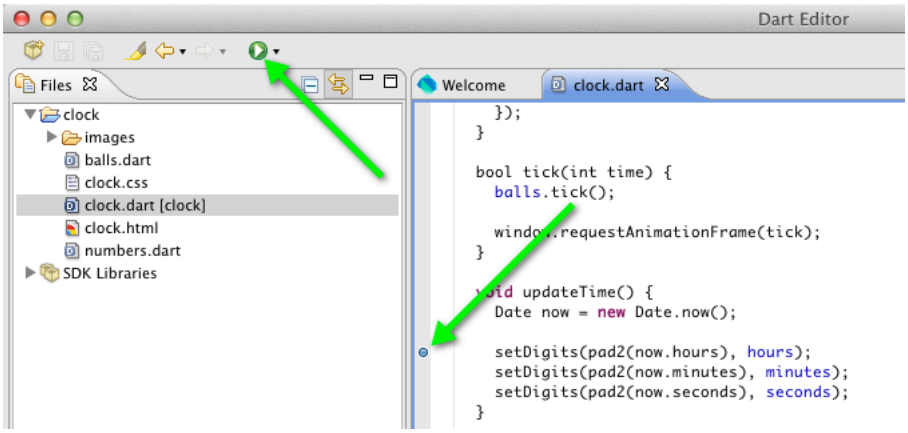


The Elements tab should be selected by default, and `clock.dart` should be listed as the script that is being run.

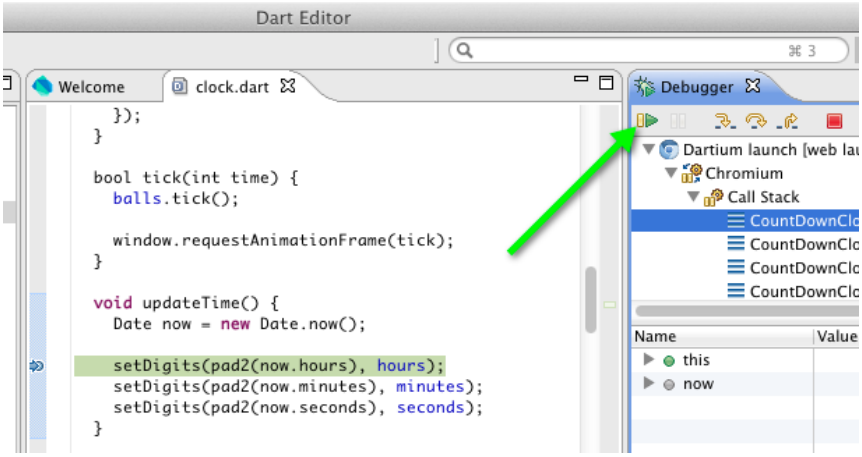


Debug with the Dart Editor

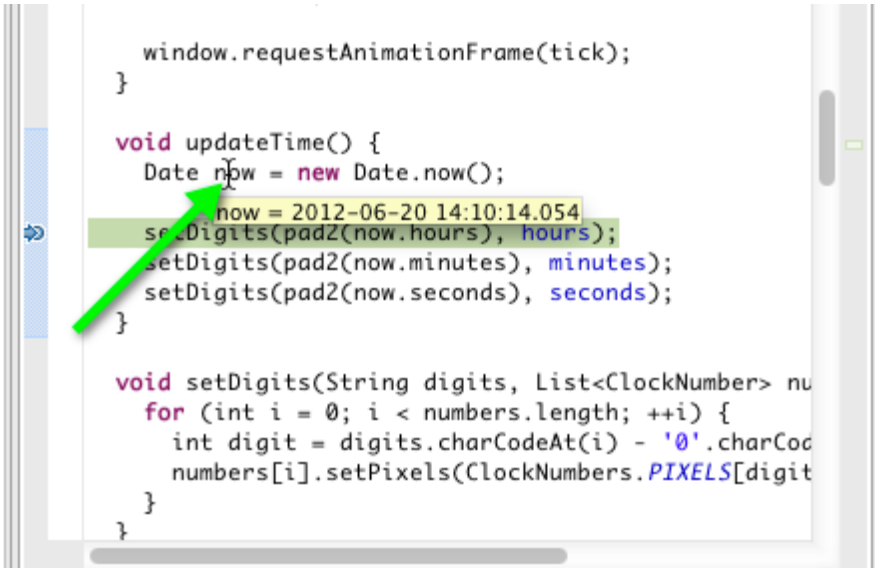
With Dart running directly in Dartium, the editor has debugging support for Dart applications. Set a breakpoint by double clicking in the left hand gutter in Dart Editor on first call to setDigits() inside the `updateTime()` method in `clock.dart`, line 66. With the breakpoint set (you will see a little blue dot in the gutter), click the Run button again.



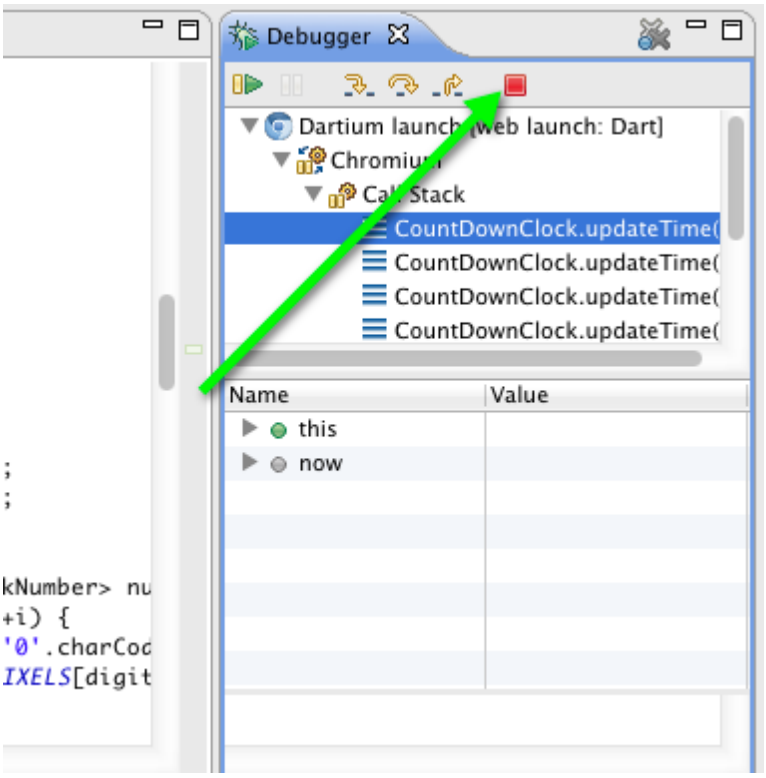
Notice how the program stop, and the Debugger view opens in the Dart Editor. To continue the program without leaving the debugger, click the Resume button (the green arrow in the Debugger view). The `updateTime()` method is called every 1000ms, therefore the breakpoint will be hit again within a second.



The Debugger view on the right hand side allows you to see which processes are running and what values are in scope at the breakpoint. Hover over the `now` field, the value will be displayed in a tooltip.



To terminate the debugger, click on the red square in the Debugger view. This will also stop the application.

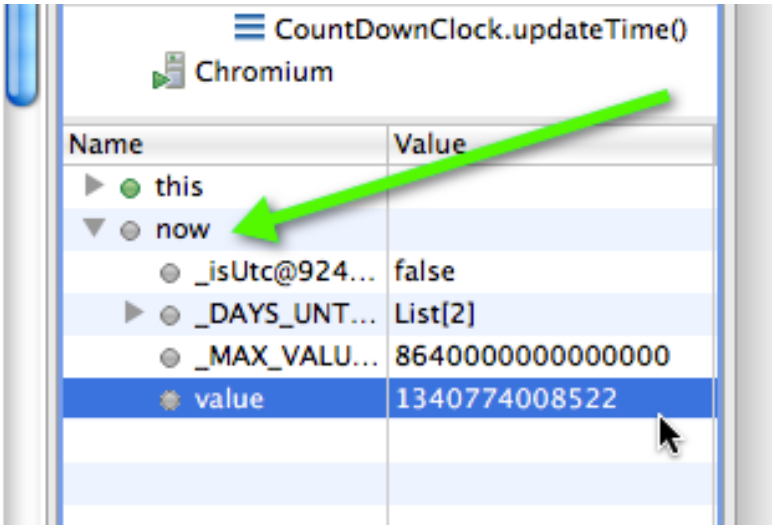


Advanced

Load and launch the other samples.

In the Clock sample, try changing the ball velocity or the gravity. Start with lines 12 and 117 in `balls.dart`.

Set more breakpoints and inspect the values by opening up the variables.



Step 2: Import and run the chat app

This codelab walks you through a custom chat application. You will now load this chat app into the editor and learn how to run both client and server Dart apps.

Objectives

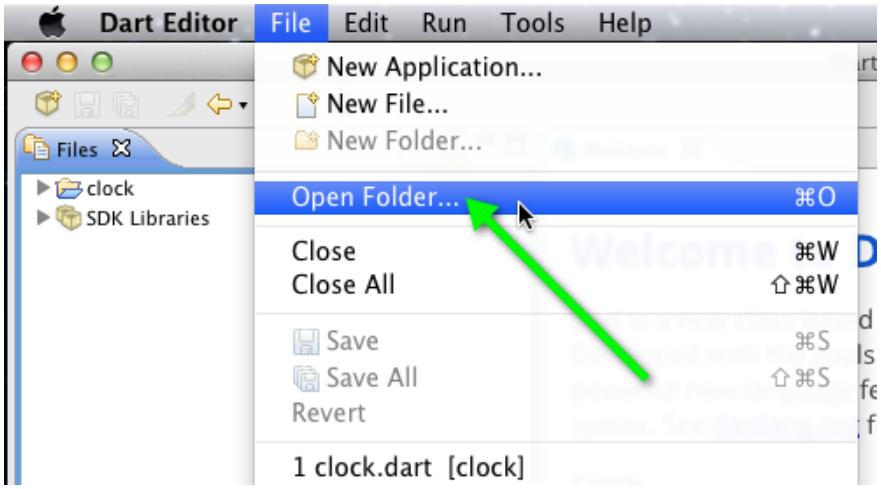
1. Load existing code into Dart Editor
2. Run the finished Dart Chat app
    a. Run a command-line app
    b. Run a Dartium app
3. View running processes
4. Read and clear console output
5. Close a running process

Walkthrough

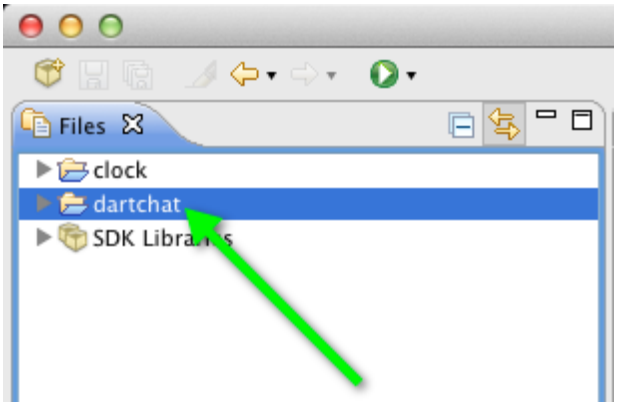
Load Dart Chat into Dart Editor

Copy over the `dartchat/` directory from the USB thumb drive to your computer.

Load the sample project for this codelab into the editor. Select `File` > `Open Folder...` in the editor. Find the `dartchat/` directory that you copied from the USB, select it, and click `Open`.



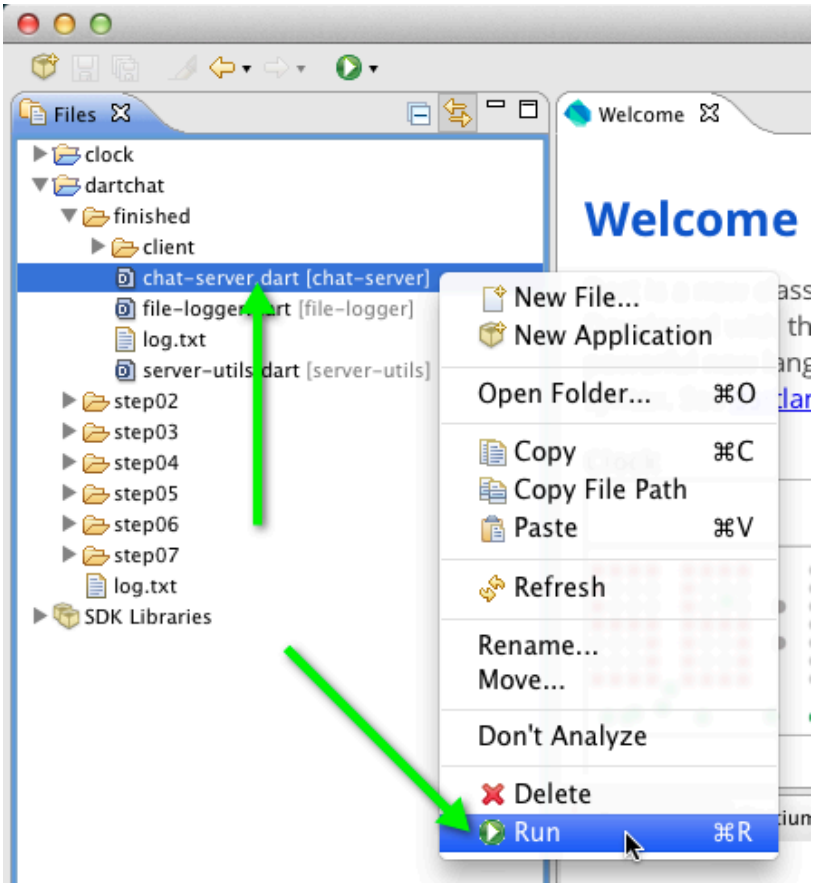
You will see a new dartchat project in the editor.



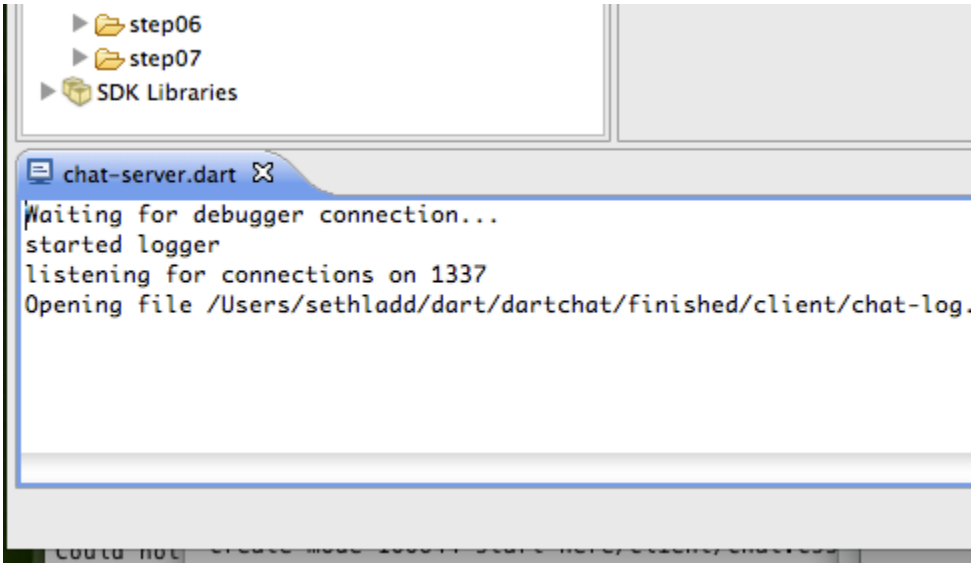
Launch the completed version of the Dart Chat sample

The sample chat app has both a client and a server component.

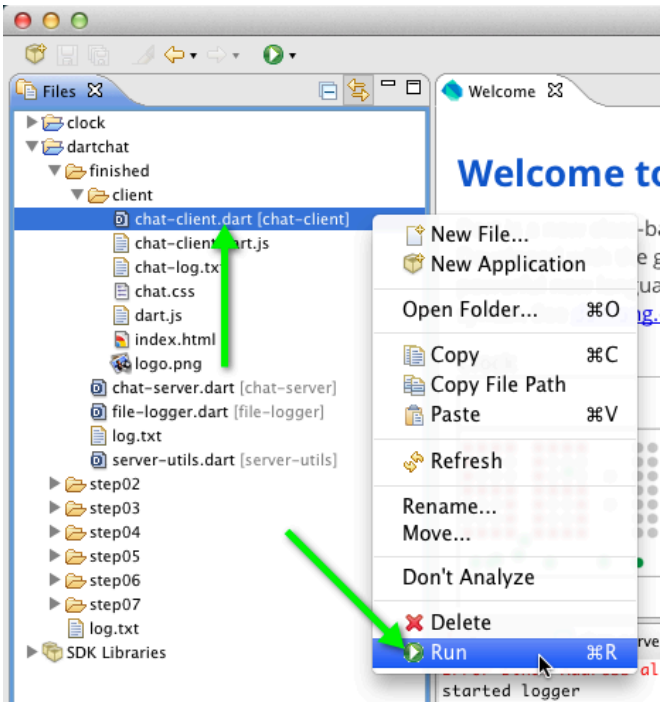
Run the server first. In the Files view on the left hand side of Dart Editor, navigate into the `dartchat` directory, `dartchat` > `finished` > `chat-server.dart`. Right click `chat-server.dart` and select `Run`.



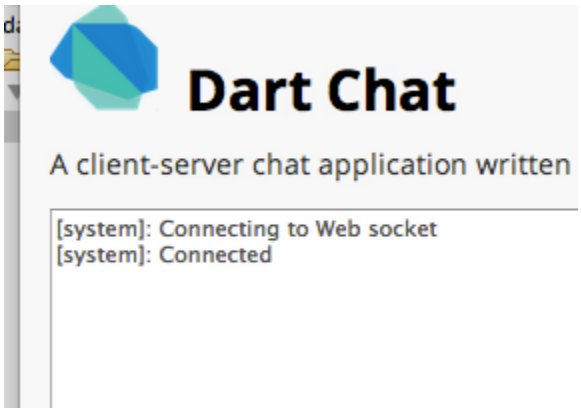
Verify the server is running by checking the `chat-server.dart` console output window at the bottom of your editor. You should see a message: "listening for connections on 1337".



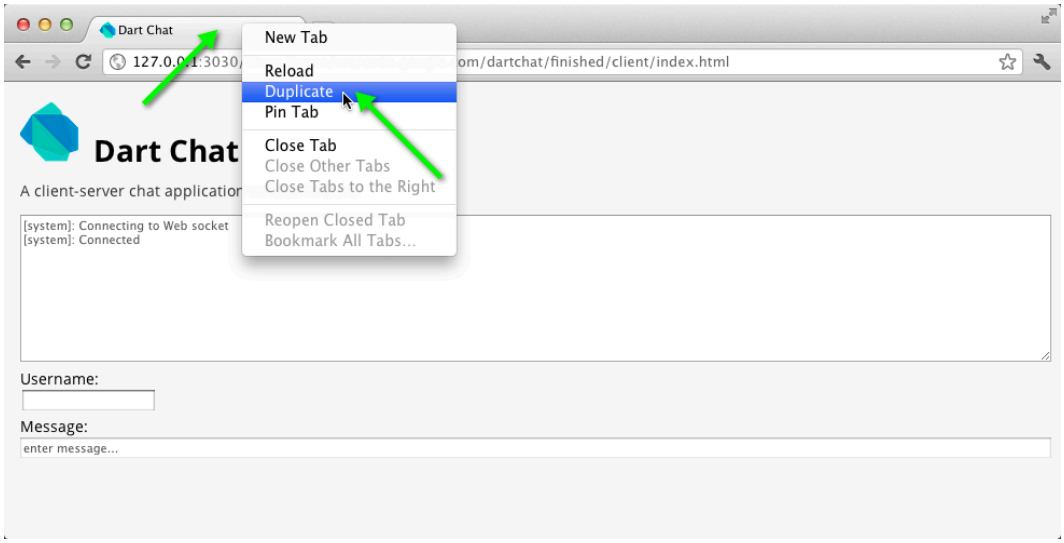
Next, run the client. Navigate into the `client` directory, and run the `chat-client.dart` file in the same way.



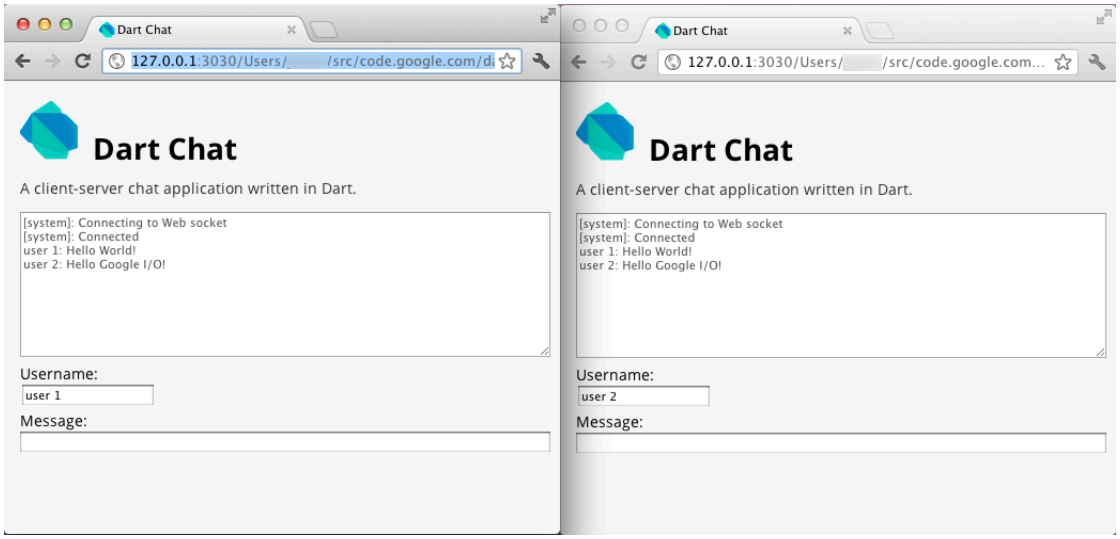
Notice how Dartium opens the chat app. Verify the chat client is connected to the server by looking for a "[system]: Connected" message in the chat window.



Open up another tab to have a proper chat. Right click the “Dart Chat” tab and select “Duplicate”.

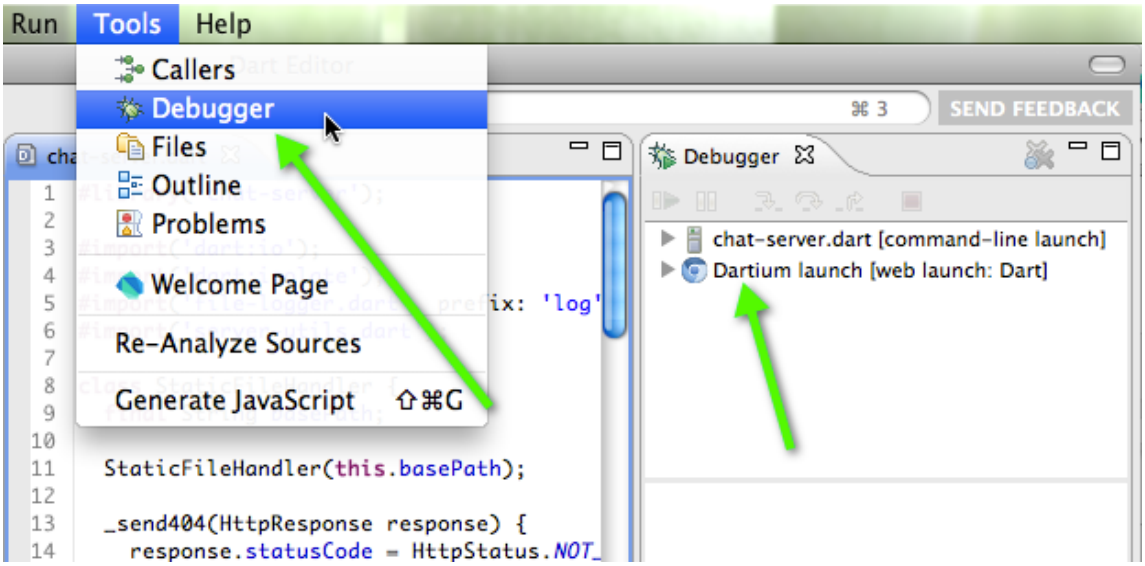


Drag the new tab into its own window. Experiment by sending messages between the two tabs. For more fun, add more clients.

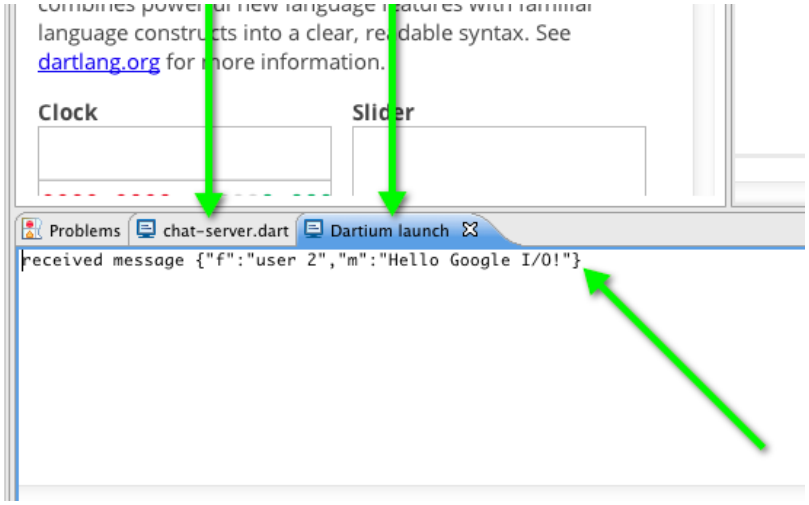


Debugger view and console output

Switch back to Dart Editor and select the `Tools` > `Debugger` in the top level menu. This lists the two processes that you started, the server and the client.

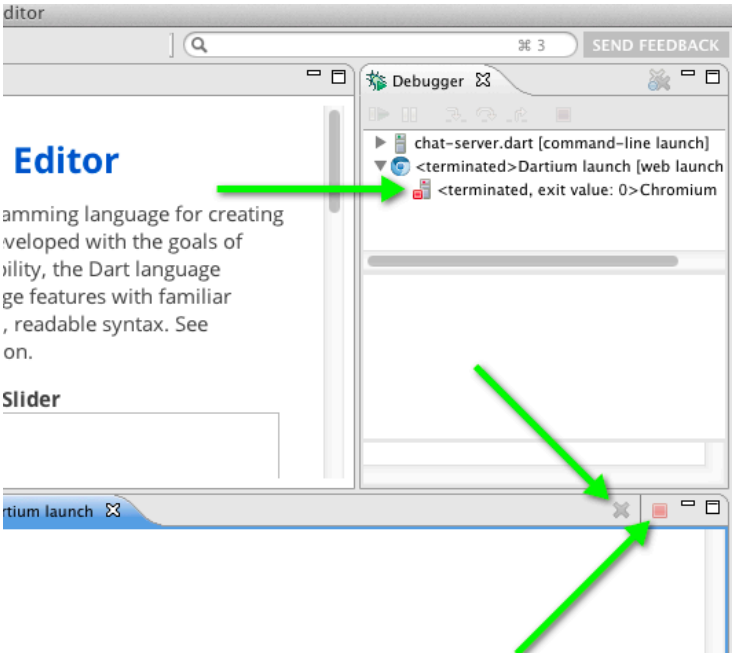


On the bottom of Dart Editor are two views, `chat-server.dart` and `Dartium launch`. Each view has the output from the respective process.



To clear a console output, click on the gray X icon in upper right of the console output view. To kill the process, click on the red box in the upper right of the console output view. After clicking on the red box, you will notice that the Debugger is updated to show that the process was killed.

Stop both processes now, first for the `Dartium launch`, and then for the `chat-server.dart`.



Advanced

If you finish early, explore more of the chat sample app code. Specifically, investigate the chat server code. The server logs to a file via an isolate, you can learn more about isolates in the Dart Library Tour[4].

[4] http://www.dartlang.org/docs/library-tour/#dartisolate---concurrency-with-isolates

Step 3: Navigate the chat app, and learn basic Dart language features

The Dart language is familiar to a wide range of developers. It is a class-based object oriented language with single inheritance, curly braces, and semicolons. The syntax is instantly recognizable and the concepts are comfortable.

As you tour the chat code, you will learn the basics of the Dart language[5]. You can use Dart Editor's code navigation features, such as search, outlines, and jump to definition to come up to speed more quickly with unfamiliar code.

Objectives

1. Search for code using Dart Editor
2. Jump to definition of class or method
3. Use the Outline view to see file structure
4. Learn about classes, superclass, generics, functions, methods, variables, libraries, and more
5. Learn about optional static types
6. Understand basic layout of sample app
7. Understand warnings in Dart Editor
8. Understand errors in Dart Editor

Before You Begin - Code

Begin your coding journey in the `start-here` directory of dartchat. If you need to catch up, or if you need to start over, you can copy `step03` into `start-here` for this portion of the codelab.

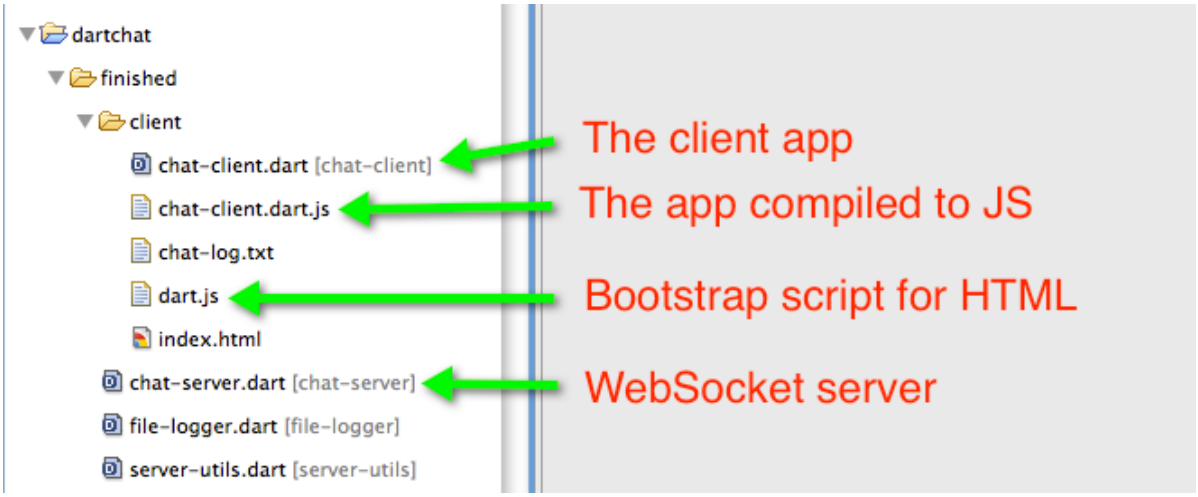
Walkthrough

In order to demonstrate Dart language and editor features, you will create a class to represent the chat window on the HTML page. This process will teach you more about the Dart Editor and the Dart language.

Project layout

Before you begin coding, it's helpful to understand the layout of the project. Open the `finished` directory, inside of the dartchat project in Dart Editor. Now, open the `client` directory to get a full layout of the finished app.

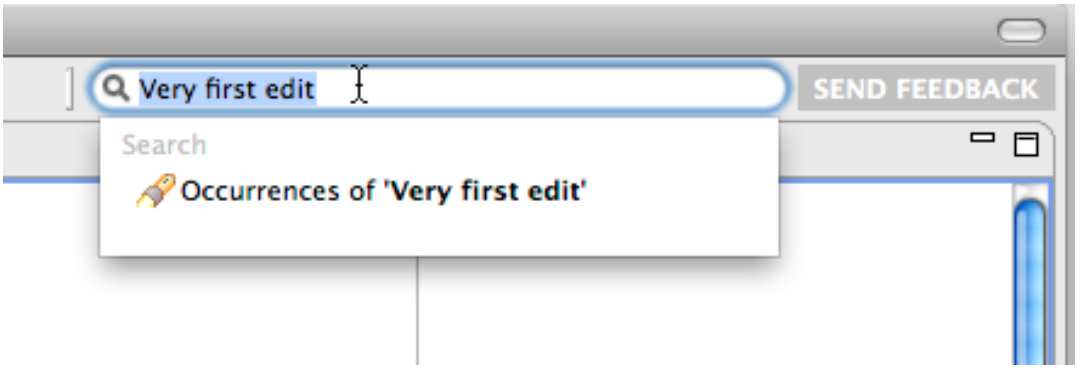
[5] http://www.dartlang.org/docs/language-tour/



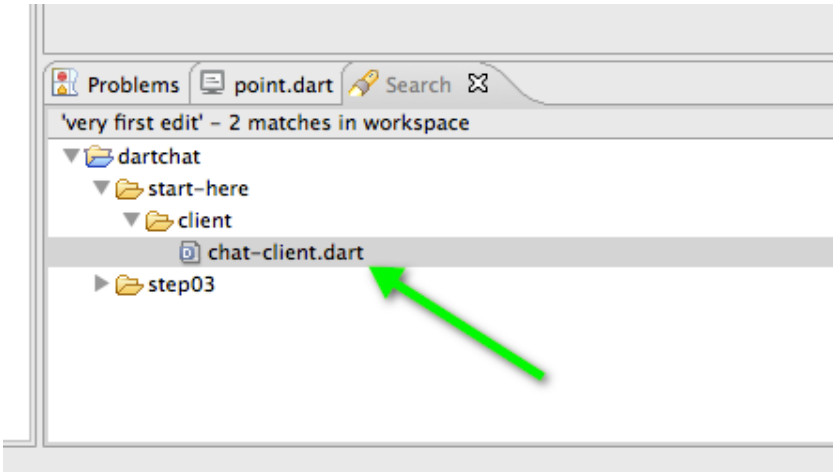
You will spend your time inside the `client` directory, mostly working on `chat-client.dart`. The project includes the code for both the client (`chat-client.dart`) and the server (`chat-server.dart`).

Search for code

Use the Search feature of the editor, in the upper right of the tool, to find "Very first edit" (a helpful comment that the authors included for you :), so you can start editing.



The search results are displayed in a dedicated view at the bottom of the editor.



Double-click on the `chat-client.dart` from `start-here` to open the file and start writing some Dart!

Tip: Use `start-here` to make all of your edits. Be sure not to edit `step03`, as this is intended to give you a fallback if you need to restart your tasks.

Add a top-level variable

Ensure `client/chat-client.dart` is open and scrolled to the top. Add a top-level variable for an object to represent the the chat window. Use `chatWindow` as the variable name and `ChatWindow` as the type annotation.

```
// client/chat-client.dart
…

UsernameInput usernameInput;

// Step 3: Very first edit
// Step 3: Add variable for chat window.
ChatWindow chatWindow;

class ChatConnection {
…
```

> *Tip*: You can use top-level functions and variables instead of wrapping everything in a class.

Save the file. Notice that after you add the new variable with a type annotation, the editor gives you a *warning* (with a yellow underline) indicating that it doesn't know what `ChatWindow` is. That makes sense, as we haven't defined it yet.

Advanced Topic: Dart is an optionally typed language, which means you can use type annotations when you want them, and leave them off when you don't. If the runtime can't find the type you are referring to, it will ignore that static type annotation and treat the variable as Dynamic (the stand-in type when no other type annotation is provided). A missing type mentioned in a type annotation does not prevent the program from compiling and running (because, again, Dart is optionally typed). It is only a *warning* that the editor can't find `ChatWindow`.

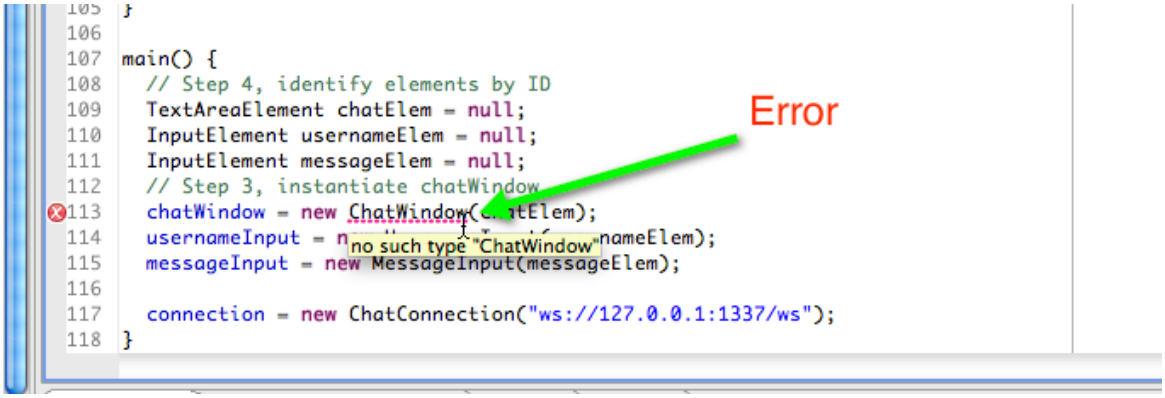
Instantiate an object

Scroll down to `main()` to create a new instance of `ChatWindow`. Much like other languages, use the `new` keyword to construct a new object.

```
// client/chat-client.dart

main() {
  // Step 4: Identify elements by ID.
  TextAreaElement chatElem = null;
  InputElement usernameElem = null;
  InputElement messageElem = null;
  // Step 3: Instantiate ChatWindow.
  chatWindow = new ChatWindow(chatElem);
  usernameInput = new UsernameInput(usernameElem);
…
```

Notice how the editor is reporting an *error* now, with a red underline. Specifically, the error is “no such type: “ChatWindow””.



An error will stop a program from compiling and running, so this is a problem that must be fixed before continuing. Dart strives to minimize the number of situations that result in a true error, but Dart does need to know what kind of object you want to construct.

Define a class

To silence the error, add the `ChatWindow` class. Find the “Define the ChatWindow class” comment and add the following class:

```
// client/chat-client.dart

…
// Step 3: Define the ChatWindow class.
class ChatWindow extends View<TextAreaElement> {

}
…
```

The editor will give you an error about a missing constructor. Don't worry, we'll fill this in next.

The `ChatWindow` class extends the `View` class (already defined for you in the application). The `View` class uses generics to further specify the type of the HTML element the view encapsulates. The above code says “ChatWindow is a specialized View of a TextAreaElement”.

> *Advanced Topic*: Dart support parameterized types, also known as generics. As with Dart’s optional types, you don’t need use generics. However, they are a powerful tool to help add more expressiveness to your code. Dart’s generics[6] are probably simpler than those found in other mainstream languages.

Add a constructor to `ChatWindow` that delegates to the constructor from View:

[6] http://www.dartlang.org/docs/language-tour/#generics

```
// client/chat-client.dart

…

// Step 3: Define the ChatWindow class.
class ChatWindow extends View<TextAreaElement> {
  ChatWindow(TextAreaElement elem) : super(elem);

…
```

What follows the `:` is the initializer list[7] for a class, used to initialize final variables and the super constructor.

Next, add two public methods and one private method to `ChatWindow`. These methods will display messages to the <textarea> element.

```
// client/chat-client.dart

…

class ChatWindow extends View<TextAreaElement> {
  ChatWindow(TextAreaElement elem) : super(elem);

  displayMessage(String msg, String from) {
    _display("$from: $msg\n");
  }

  displayNotice(String notice) {
    _display("[system]: $notice\n");
  }

  _display(String str) {
    elem.text = "${elem.text}$str";
  }
}

…
```

In the above code, the `_display()` method is *library private*. Names that are prefixed with an underscore are private to the library they are defined within. The ChatWindow class is defined in a file that is marked as `#library('chat-client')`, therefore `_display()` is only visible to code also in the 'chat-client' library. The other two methods are public.

The `text` property of `elem` accessing the contents of the <textarea> tag.

One of Dart's handy features is string interpolation, used in all three methods above. You can compose strings like "`$from: $msg\n`" by directly referencing variables with a `$` prefix.

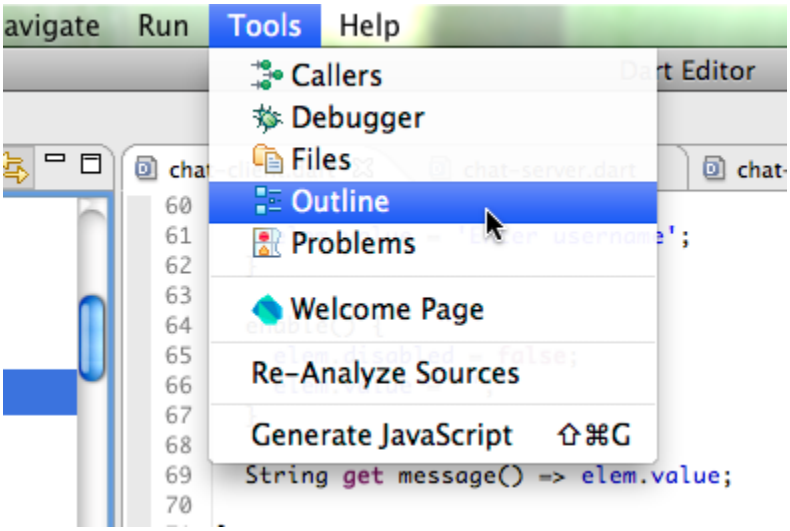
Note that `elem`, used inside of `_display()`, is found in the `View` superclass. `ChatWindow` extends `View`, thus gaining access to `View`'s instance variables.

[7] http://www.dartlang.org/docs/language-tour/#initializer-list

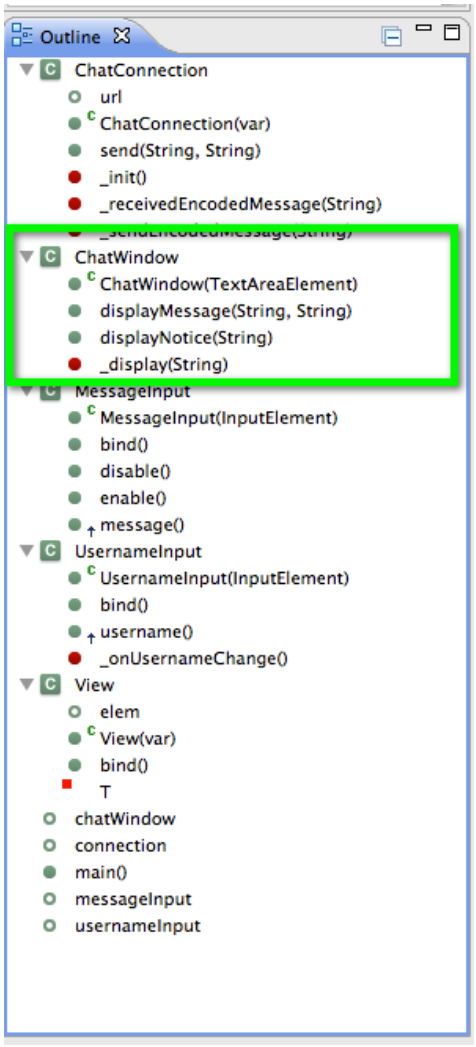
View the code outline

With the `ChatWindow` class now added, you can use the editor to easily browse the outline of the chat-client.dart file.

Select the `chat-client.dart` file and select the menu, Tools, Outline.



The Outline view will now appear, with a tree view of the classes, methods, and variables found in `chat-client.dart`.

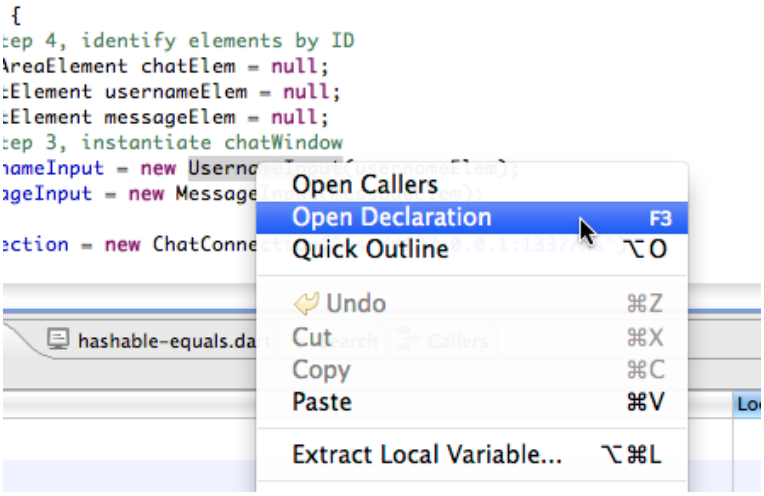


Advanced

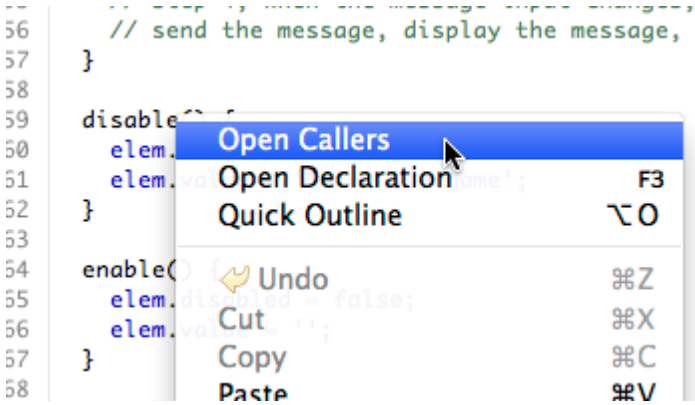
Learn more about the Dart language, such as strings[8] and classes[9].

[8] http://www.dartlang.org/docs/language-tour/#strings
[9] http://www.dartlang.org/docs/language-tour/#classes

Right-click a class name and select *Open Declaration*. Try this with classes defined in chat-client.dart as well as classes from Dart.



Right-click a method and select *Open Callers*. You'll see a list of all the locations that call this method.



Step 4: Respond to changes in input fields

Web developers are familiar with the DOM (Document Object Model, a set of APIs to access browser features), but most use jQuery to smooth over the browser interfaces and inconsistencies. Similar to how jQuery makes the DOM feel like JavaScript, Dart offers an *HTML library* to make programming the browser feel like Dart.

The HTML library[10] provides access to the elements on the web page— , <p>, and so on. You can find elements on the page, create new elements, change text, and lots more. You can also access new HTML5 features such as IndexedDB, Application Cache, and WebGL with the HTML library.

Web applications run in a single UI thread on the page[11]. To minimize visible pauses or delays, responses to events like mouse clicks and button presses happen inside of *callbacks*. Once you have a handle to an element on the page, such as an input field, you can register a function to be run on certain events (for example the “on change” event).

Objectives

1. Import a library
2. Find elements on an HTML page with CSS selector
3. Bind to events from HTML elements
4. Use lexical closures, inline callbacks, and one-line functions
5. Enable and disable input fields
6. Get the contents of an input element
7. See a Dart getter in action
8. Use code completion in Dart Editor to discover which methods you can use

Code

If you need to catch up, you can copy `step04` onto `start-here` for this portion of the codelab.

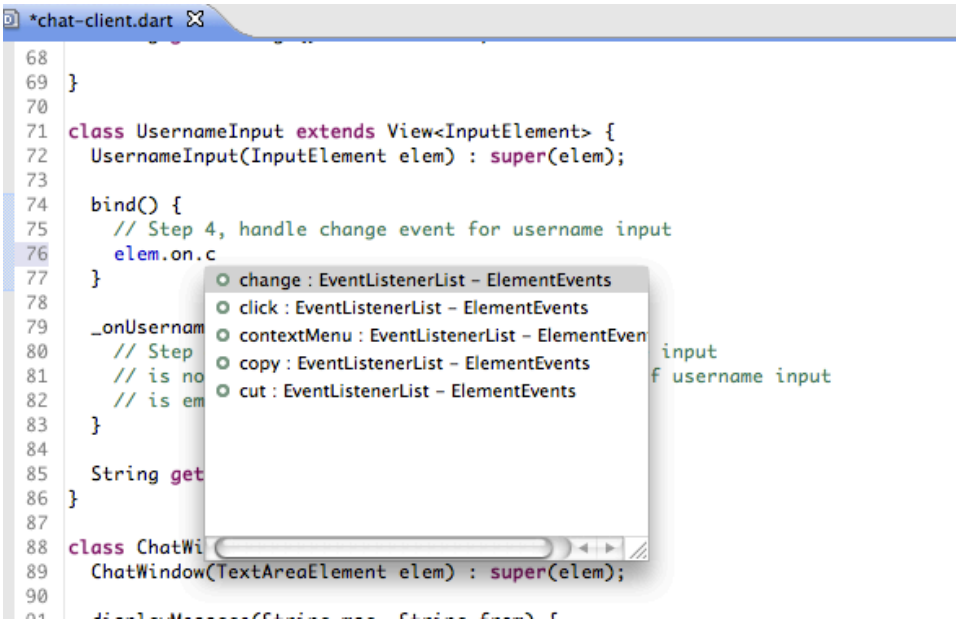
Walkthrough

Before you start on the code, we will introduce you to one of the more productive and helpful features of Dart Editor: code completion.

With enough type information (from direct type annotations, simple inferencing, or type propagation), Dart Editor can provide a list of available methods or fields on an object. On a Mac, Control-Space triggers code completion.

[10] http://api.dartlang.org/html.html

[11] Except for Web workers, an advanced topic not covered here.



As you are writing the code for this and future steps, try code completion. If Dart Editor doesn't give you reasonable options, try adding a type annotation to the object that you are working with.

(**Tip**: If you are trying code completion, and it's not giving you the right suggestions, use the Send Feedback button to let the Dart team know. Please include the code snippet that you are trying.)

Identify the elements in HTML

You will now add the HTML library, find existing elements on the page, register for events, and manipulate DOM elements.

Open `client/index.html`. Identify the IDs for the two `<input>` elements and the `<textarea>` element. For example:

```
// client/index.html

<textarea id="chat-display" rows="10" cols="100" disabled></textarea>
…
<input id="chat-username" name="chat-username" type="text">
…
<input id="chat-message" name="chat-message" type="text" disabled
value="enter username...">
```

HTML element IDs are unique in a document, making it easy to find a specific element.

Verify the dart:html library is imported

With the three IDs identified, you can use Dart's HTML library to get a handle on those elements. Open `client/chat-client.dart` and scroll to the top of the file.

Notice the dart:html import is already added for you. We left this in to eliminate errors, but normally you would add this import yourself.

```
// client/chat-client.dart

#library('chat-client');

#import('dart:html');  // This is already here, just pointing it out. :)
…
```

Find HTML elements

Use the `query(String selector)`[12] top-level function from dart:html to find elements in an HTML page. The `query()` function takes a CSS selector and returns a single matching element. (A corresponding `queryAll()`[13] function returns all matching elements.)

CSS selectors[14] are powerful pointers to elements on an HTML page. You can use them to find elements by IDs, class names, attribute values, position in the DOM, and more. For this step, you will use the element's ID as the selector. For example, the CSS selector `#the-id` will search for an HTML element with an ID of `the-id`.

[12] http://api.dartlang.org/html.html#query
[13] http://api.dartlang.org/html.html#queryAll
[14] https://developer.mozilla.org/en/CSS/Getting_Started/Selectors

Get the three element IDs that you identified from `client/index.html` ready. Next, scroll down to the bottom of `client/chat-client.dart` and find the `main()` method. Use what you know about IDs, CSS selectors, and the query() function to associate each HTML element with its equivalent Dart object from your app.

```
// client/chat-client.dart

…

main() {
  // Step 4: Identify elements by ID.
  TextAreaElement chatElem = query('#chat-display');
  InputElement usernameElem = query('#chat-username');
  InputElement messageElem = query('#chat-message');
  chatWindow = new ChatWindow(chatElem);
  usernameInput = new UsernameInput(usernameElem);
  messageInput = new MessageInput(messageElem);

…
```

> *Advanced Topic*: The astute reader might notice that the `query()` methods return an Element[15], yet the returned variable is annotated with more specific types (`TextAreaElement`[16] and `InputElement`[17]). Dart allows an "downcast" because it is by nature an optionally typed language, and it promotes an "innocent until proven guilty" programming experience. As long as the type annotation is in the type hierarchy of the expected return type, Dart's tools do not create a warning.

Notice how the constructors for `ChatWindow`, `UsernameInput`, and `MessageInput` all take a DOM element. Here's an example:

```
// client/chat-client.dart

…

class MessageInput extends View<InputElement> {
  MessageInput(InputElement elem) : super(elem);

…
```

Bind to events using lexical closures

You will now learn how to bind a function to an HTML element event. Specifically, you will bind event handlers to the "on change" events for both the username input field and the message input field.

For a simple one-line callback function, use the one-line style:

```
element.on.event.add((event) => handleTheEvent())
```

[15] http://api.dartlang.org/html/Element.html
[16] http://api.dartlang.org/html/TextAreaElement.html
[17] http://api.dartlang.org/html/InputElement.html

The *event* can be a mouse click, a button press, a value change, or any number of interesting actions or state changes.

> *Tip*: Pull up the API docs for ElementEvents[18] to see all the possible events from an Element.

The one-line function syntax (=>) used above is shorthand sugar for:

```
element.on.event.add((event) {
  return handleTheEvent();
});
```

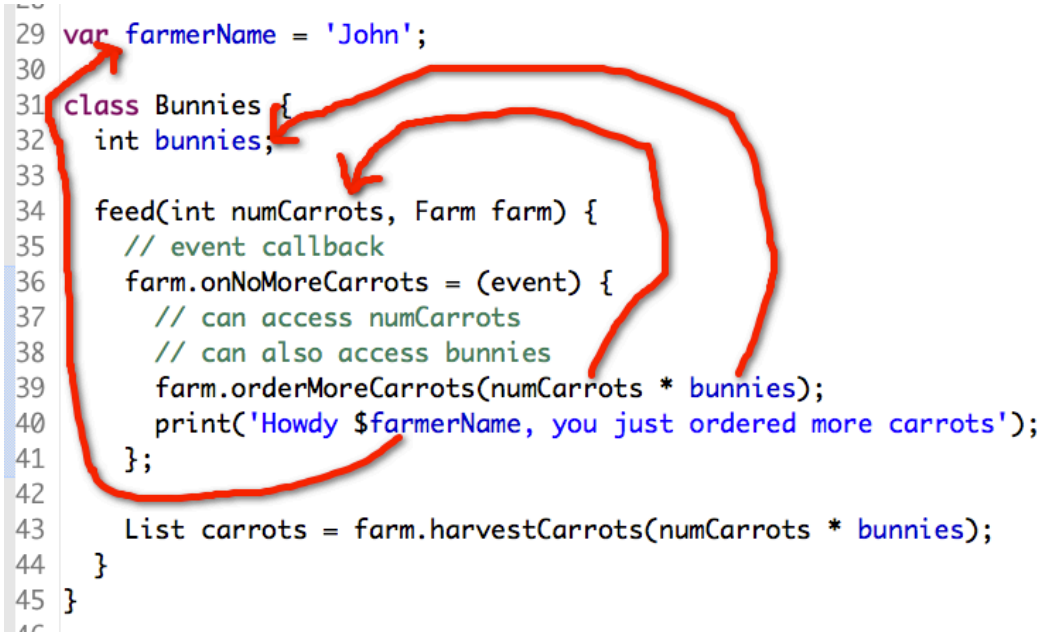
There's nothing special about the => being used for callbacks, you can use it anywhere a one-line function is valid. For example, here's a simple class with a simple one-line function:

```
// for illustrative purposes only, not part of the codelab
class Person {
  String firstName, lastName;
  Person(this.firstName, this.lastName);
  String toString() => '$firstName $lastName';
}
```

Dart's lexical closures make writing nested functions and callbacks easy. You can access variables in lexical scope from within callback functions, and even the "this" object is defined by its lexical scope.

> *Advanced Topic*: Lexical scope defines scope through the program structure. You can "follow the curly braces" to see which objects or variables are in scope, and that scope does not change based on program behavior. The following code snippet displays a few examples of lexical scope:

[18] http://api.dartlang.org/html/ElementEvents.html



```
29 var farmerName = 'John';
30
31 class Bunnies {
32   int bunnies;
33
34   feed(int numCarrots, Farm farm) {
35     // event callback
36     farm.onNoMoreCarrots = (event) {
37       // can access numCarrots
38       // can also access bunnies
39       farm.orderMoreCarrots(numCarrots * bunnies);
40       print('Howdy $farmerName, you just ordered more carrots');
41     };
42
43     List carrots = farm.harvestCarrots(numCarrots * bunnies);
44   }
45 }
```

Notice how the `onNoMoreCarrots` callback handler can access `numCarrots`, `bunnies`, and `farmerName`.

You will add the code to listen for the username input field's change event. When the username field is empty, the message input field should be disabled. When the username input field has a value, the message input field should be enabled.

Handle username field changes

Open `client/chat-client.dart` and find the `UsernameInput` class. Add the event handler inside the `bind()` method of `UsernameInput`:

```
// client/chat-client.dart

class UsernameInput extends View<InputElement> {
  UsernameInput(InputElement elem) : super(elem);

  bind() {
    // Step 4: Handle change event for username input.
    elem.on.change.add((e) => _onUsernameChange());
  }
…
```

Next, inside of `_onUsernameChange()`, you should enable the message input element if the username input element is not empty.

```
// client/chat-client.dart

  _onUsernameChange() {
    // Step 4: Enable the message input if username input
```

```
    // is not empty, or disable the message input if username input
    // is empty.
    if (!elem.value.isEmpty()) {
      messageInput.enable();
    } else {
      messageInput.disable();
    }
  }
…
```

Notice that InputElement's value field[19] is used to get the contents of the input field. The `isEmpty()`[20] is available to strings, and returns true if the string contains zero characters.

The `enable()` and `disable()` methods are called on the instance of `MessageInput`, a class we created for this chat application. Go check out the implementations for these methods.

Handle message field changes

The mechanics of handling the message input field are similar. When the message input field changes, you should do three things:

1. Send the message via the chat connection
2. Display the message in the chat window
3. Erase the message input field

[19] http://api.dartlang.org/html/InputElement.html#value
[20] http://api.dartlang.org/dart_core/String.html#isEmpty

Find the `MessageInput` class and add the following code:

```
// client/chat-client.dart

class MessageInput extends View<InputElement> {
  MessageInput(InputElement elem) : super(elem);

  bind() {
    // Step 4: When the message input changes,
    // send the message, display the message, and clear the message input.
    elem.on.change.add((e) {
      connection.send(usernameInput.username, message);
      chatWindow.displayMessage(message, usernameInput.username);
      elem.value = '';
    });
  }
```

Notice that `connection.send()` is passed *`message`*, which is a *getter method* inside of `MessageInput`:

```
class MessageInput extends View<InputElement> {
  …

  String get message() => elem.value;

  …
}
```

> *Advanced Topic*: A getter is a method that looks like a field and does not require `()` when called. Getters and setters[21] are intended for advanced API design, usually to help with refactoring from simple public fields to encapsulated properties.

Advanced

Instead of sending the message when the input field changes, send it when a Send button is clicked. Add a button, label it, and respond to click events.

Add a timestamp to messages displayed in the `ChatWindow`. Hint: check out the Date[22] class from dart:core.

Step 5: Encode and decode data with JSON

JSON, or JavaScript object notation, is a handy text format for encoding structured data like arrays, numbers, strings, booleans, and maps. JSON is well supported across programming

[21] http://www.dartlang.org/docs/language-tour/#classes-getters-and-setters
[22] http://api.dartlang.org/dart_core/Date.html

languages and libraries, making it easy to perform data interop across systems.

Here's what an arbitrary data object looks like when encoded into a JSON string:

```
// A Dart map
var data = {'scores': [12,54,99]};
assert(data is Map);
assert(data['scores'] is List);
assert(data['scores'][0] == 12);

// Convert to JSON string
var json = JSON.stringify(data);

// A Dart map encoded as a JSON string
assert(json == '{"scores": [12, 54, 99]}');
```

Looks pretty similar to Dart's map and list literals!

Objectives

1. Learn about JSON
2. Encode objects with into a JSON string
3. Decode a JSON string into an object

Code

If you need to catch up, you can copy `step05` onto `start-here` for this portion of the codelab.

Walkthrough

Open `client/chat-client.dart` and scroll to the top of the file. You will now add the JSON library, encode and decode JSON, and try some new Dart Editor features.

Import dart:json

To use the JSON functionality bundled with Dart, import the dart:json into the chat-client library, found in `client/chat-client.dart`.

```
// client/chat-client.dart

#library('chat-client');

#import('dart:html');

// Step 5: Import the JSON library.
#import('dart:json');

ChatConnection chatConnection;
```

Encode a message for sending

Find `send()` in `ChatConnection` and add the code to encode, or "stringify", both the username and message into a single JSON string.

```
// client/chat-client.dart

  send(String from, String message) {
    // Step 5. Encode from and message into one JSON string.
    var encoded = JSON.stringify({'f': from, 'm': message});
    _sendEncodedMessage(encoded);
  }
```

The above code creates a map literal using the `{ }` syntax. Both the `from` username and the `message` are placed into a single map, effectively saying "this message was sent by this username". The map is then encoded into a JSON string with `JSON.stringify()`.

Once the message is encoded into a string, it is passed to `_sendEncodedMessage()` to be sent by the WebSocket (which you'll code up in the next step).

Decode a received message

Use `JSON.parse()` to decode a JSON string into a Dart object. Find `_receivedEncodedMessage()` inside `ChatConnection` and add the code to decode a message and display it in the chat window.

```
// client/chat-client.dart

  _receivedEncodedMessage(String encodedMessage) {
    // Step 5: Decode a JSON string and display it in the chat window.
    Map message = JSON.parse(encodedMessage);
    if (message['f'] != null) {
      chatWindow.displayMessage(message['m'], message['f']);
```

```
    }
}
```

There is a bit of error checking code, ensuring the message has a sender’s username (f is short for from).

Advanced

Handle an incorrectly formatted message. Display a helpful message if a client receives a message it can't parse or understand.

Step 6: Keep code clean with Dart Editor's refactoring tools

Refactoring is the art of changing the structure of code without changing the behavior of code. It's important to clean your code as you evolve the system. Mature coding environments can help automate refactorings, reducing the friction to code maintenance.

A common refactoring technique is renaming methods, variables, classes, etc as more clear names become apparent. Keeping names understandable and clear is important for code health. Dart Editor can perform many automated rename refactorings for you.

Objectives

1. Automatically rename methods and variables

Code

If you need to catch up, you can copy `step06` onto `start-here` for this portion of the codelab.

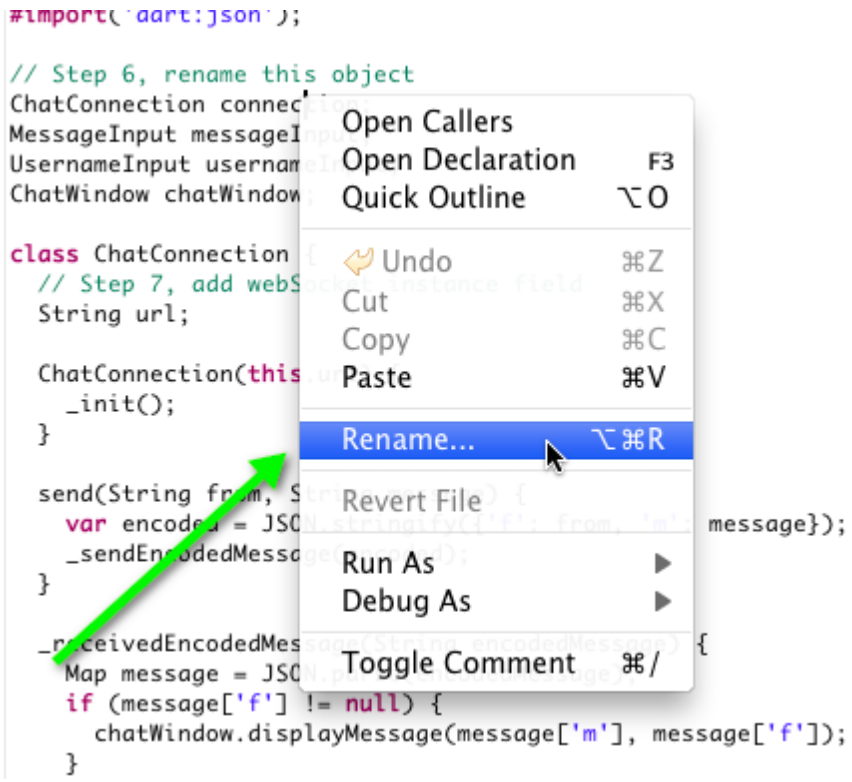
Walkthrough

Open `client/chat-client.dart` and scroll to the top. You will use Dart Editor to automatically rename variable names.

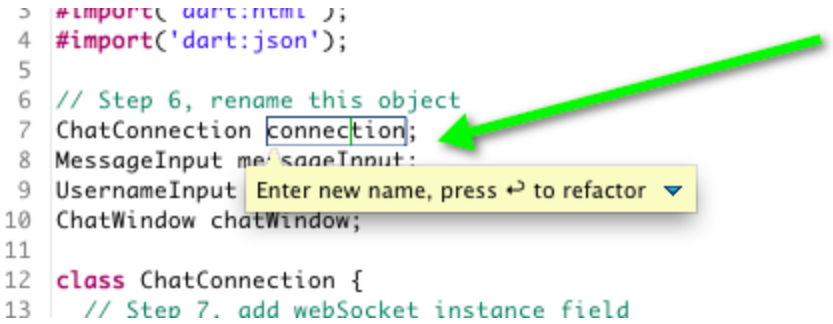
Rename a variable

Find the `ChatConnection connection` object, near the top. Rename this variable to something a bit more clear, as `connection` is vague. A quick glance at `connection` and we're not sure if it's the actual WebSocket connection. Of course your tools can tell you, but you want readable code.

Right click on the word connection, and select Rename...



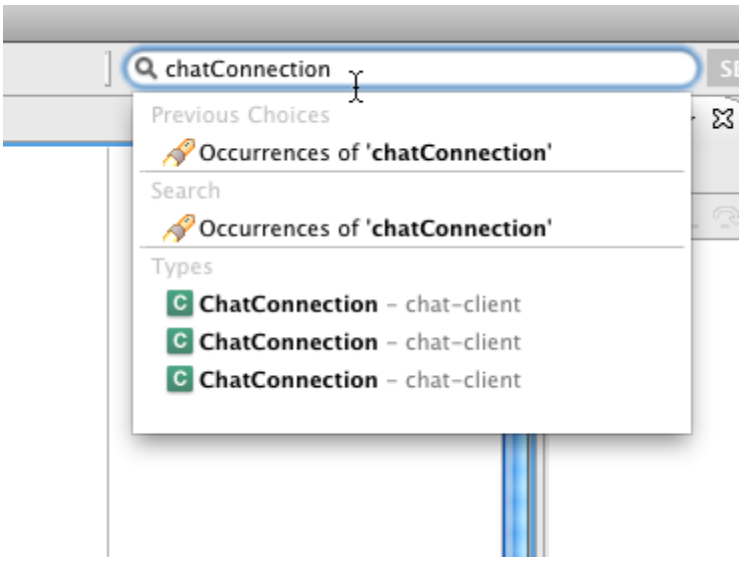
The variable name `connection` is highlighted, and you are instructed to enter a new name. Change the name to `chatConnection`. Press *enter* after you change the name.



Review the changes

The name `connection` has been changed to `chatConnection` throughout the file. Scroll to the bottom of the file. You'll see the name was changed inside `main()`. The name was also changed inside the `bind()` method of `MessageInput`.

You can search for `chatConnection` using Dart Editor search.



Advanced

Rename other methods and variables. Try renaming variables that are statically typed, and variables that are dynamically typed with var.

Tip: You might want to undo any other changes after you play around, because future steps might be a bit confusing if you change too many names.

Step 7: Send and receive data with WebSockets

WebSockets[23] are part of the HTML5 family of technologies. They enable bi-directional full-duplex communication between a modern web browser and a WebSocket server. WebSockets can stream discrete messages, either text or binary data.

WebSockets are more efficient than long polling or constant querying over HTTP. They are ideal for chat applications! :)

Objectives

1. Learn about WebSockets
2. Connect to a WebSocket server
3. Receive data from a WebSocket
4. Send data to a WebSocket

Code

If you need to catch up, you can copy `step07` onto `start-here` for this portion of the codelab.

Walkthrough

Open `client/chat-client.dart` and find the `ChatConnection` class. You will now add a WebSocket connection, listen for messages, and send messages.

Tip: Open the API docs for the WebSocket class at http://api.dartlang.org/html/WebSocket.html.

Connect to the WebSocket server

Add a new instance field for the WebSocket to the ChatConnection class.

```
// client/chat-client.dart

class ChatConnection {
  // Step 7. Add webSocket instance field.
  WebSocket webSocket;
  String url;

  ChatConnection(this.url) {
```

[23] http://en.wikipedia.org/wiki/WebSocket

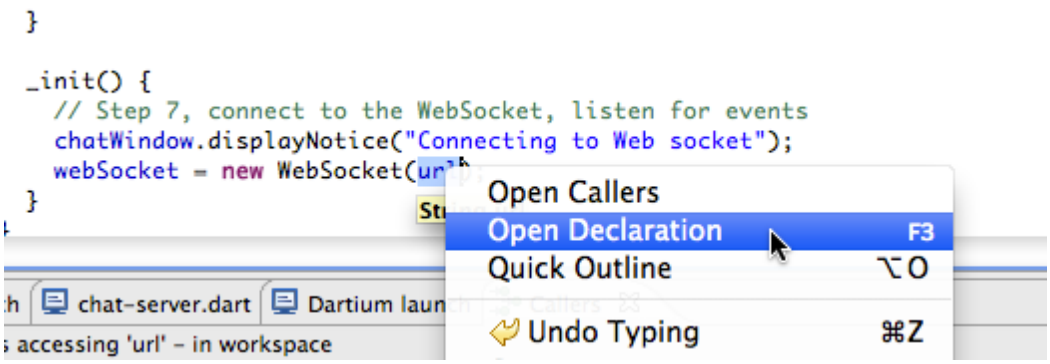
When the WebSocket instance is instantiated, it attempts to connect to the server at the URL. Find `_init` inside of `ChatConnection` and add the following code.

```
// client/chat-client.dart

  _init() {
    // Step 7. Connect to the WebSocket, listen for events.
    chatWindow.displayNotice("Connecting to Web socket");
    webSocket = new WebSocket(url);
```

First, the code displays a message to the chat window indicating that it's attempting a connection to the WebSocket server. Second, it instantiates a new WebSocket object, which attempts to connect to the WebSocket server.

Use the Open Declaration feature of the editor to determine where the `url` variable comes from.



Notice how the url variable is an instance variable of ChatConnection, originally set in the constructor.

Handle WebSocket events

WebSocket objects respond to four events: *open*, *close*, *error*, and *message*. The `message` event is fired when a new message is received on the WebSocket. The content of the message is found in `event.data`.

Tip: Open the API docs for the WebSocket events at http://api.dartlang.org/html/WebSocketEvents.html.

Handle the four WebSocket events inside of ChatConnection's `_init`.

```
// client/chat-client.dart

  _init() {
    // Step 7. Connect to the WebSocket, listen for events.
    chatWindow.displayNotice("Connecting to Web socket");
    webSocket = new WebSocket(url);

    webSocket.on.open.add((e) {
      chatWindow.displayNotice('Connected');
    });

    webSocket.on.close.add((e) {
      chatWindow.displayNotice('web socket closed');
    });

    webSocket.on.error.add((e) {
      chatWindow.displayNotice("Error connecting to ws");
    });

    webSocket.on.message.add((e) {
      print('received message ${e.data}');
    });
  }
```

Now that the event listeners for WebSocket events are configured, you are ready to use the `_receivedEncodedMessage()` method you created from Step 5.

```
// client/chat-client.dart

    webSocket.on.message.add((e) {
      print('received message ${e.data}');
      _receivedEncodedMessage(e.data);
    });
```

Send messages to the WebSocket server

Finally, with the `webSocket` instance connected, you can send messages to the server. Find `_sendEncodedMessage()` in the `ChatConnection` class and send the encoded message to the WebSocket server. Only send the message if the `webSocket` object exists and is actually connected to the server.

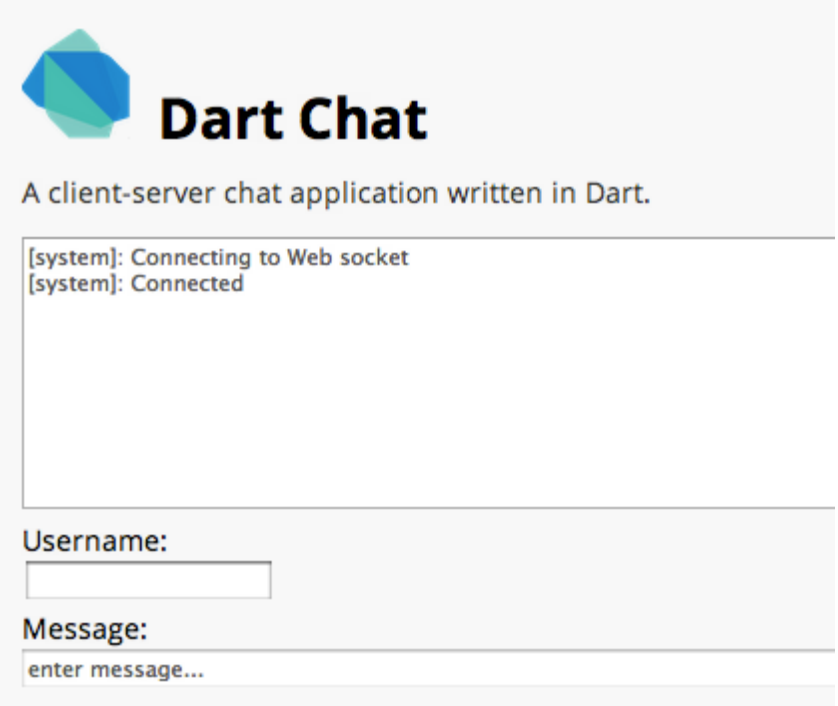
```
// client/chat-client.dart

  _sendEncodedMessage(String encodedMessage) {
    // Step 7. Send the message over the WebSocket.
    if (webSocket != null && webSocket.readyState == WebSocket.OPEN) {
      webSocket.send(encodedMessage);
    } else {
      print('WebSocket not connected, message $encodedMessage not sent');
    }
  }
```

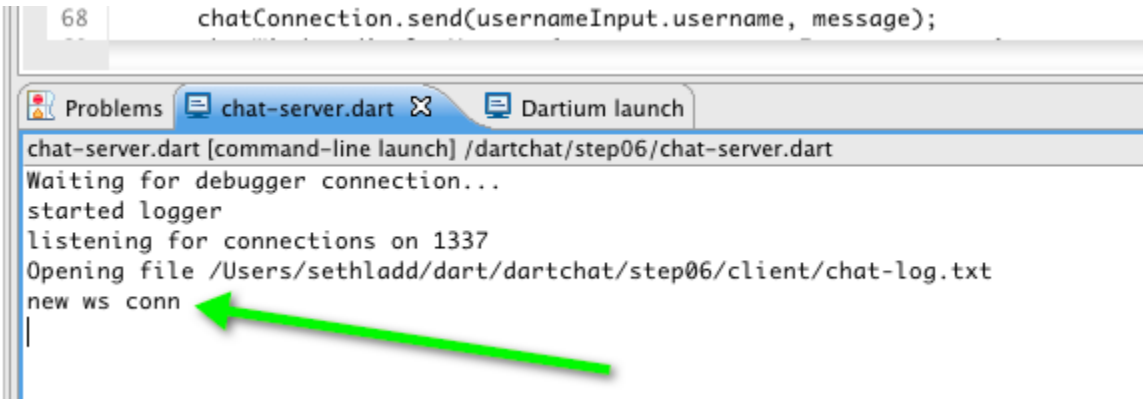
Test the app

Everything should be wired up, time to test your app!

Run the `chat-server.dart`, if not already running (see Step 2 for instructions). Next, run the `client/chat-client.dart` application, which launches Dartium. The chat window will print `[system]: Connected` if the client connects to the WebSocket server.

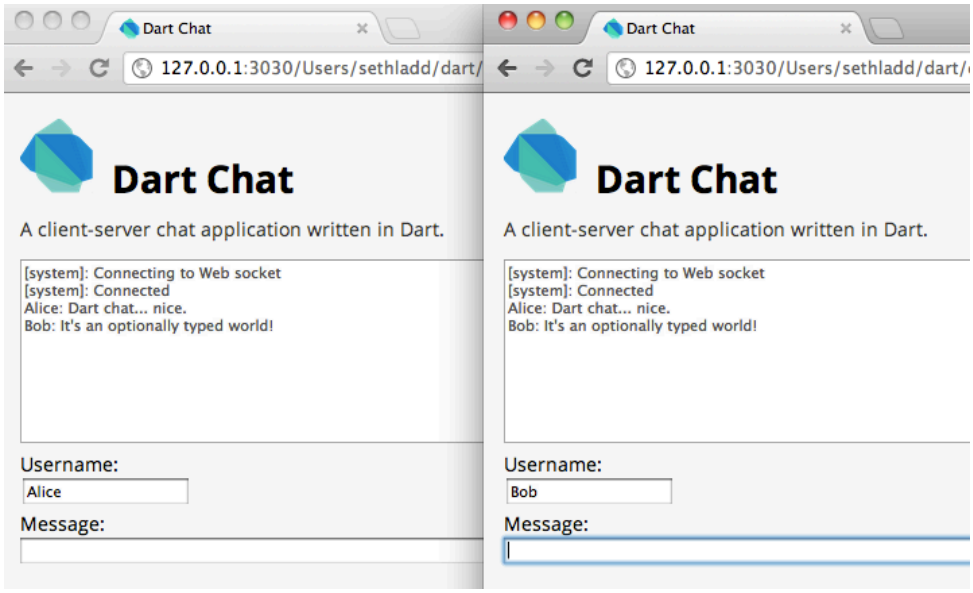


Check the `chat-server.dart` output in the Dart Editor to see "new ws conn message", which confirms your client has connected to the WebSocket server.



Not connecting to the server? Make sure the chat-server.dart is running. Sometimes, completely shutting down and restarting Dartium helps.

Send some messages! Copy the URL in Dartium, open a new Dartium tab, and paste the URL in. Enter a different username, and start chatting between the two windows.



Advanced

Handle connection failures by retrying the connection attempt. Add increasing back-off retry delays.

Update the `_init()` method to take a number of seconds to wait before retrying the connection. Add a boolean to track if an error was encountered (you will use this shortly).

```
// client/chat-client.dart

  _init([int retrySeconds = 2]) {
    // Step 6
    bool encounteredError = false;
```

In both the close and error event handlers, add a retry to reconnect after a delay. Set the `encounteredError` boolean to true so the retry timer is scheduled only once, in the event that both close and error events are fired.

```
// client/chat-client.dart

    webSocket.on.close.add((e) {
      chatWindow.displayNotice('web socket closed, retrying in $retrySeconds
seconds');
      if (!encounteredError) {
        window.setTimeout(() => _init(retrySeconds*2), 1000*retrySeconds);
      }
      encounteredError = true;
    });

    webSocket.on.error.add((e) {
      chatWindow.displayNotice("Error connecting to ws");
      if (!encounteredError) {
        window.setTimeout(() => _init(retrySeconds*2), 1000*retrySeconds);
      }
      encounteredError = true;
    });
```

The `setTimeout()` method takes two arguments: a function and when it should be called (in milliseconds from now). In this case, the `_init()` method runs after `retrySeconds`. This is a good example of Dart's lexical closures, because the inline function for `setTimeout()` easily references both `_init()` and `retrySeconds`.

Step 8: Compile to JavaScript, run in other browsers

One of Dart's core features is that it compiles to modern JavaScript to run across the modern

web.

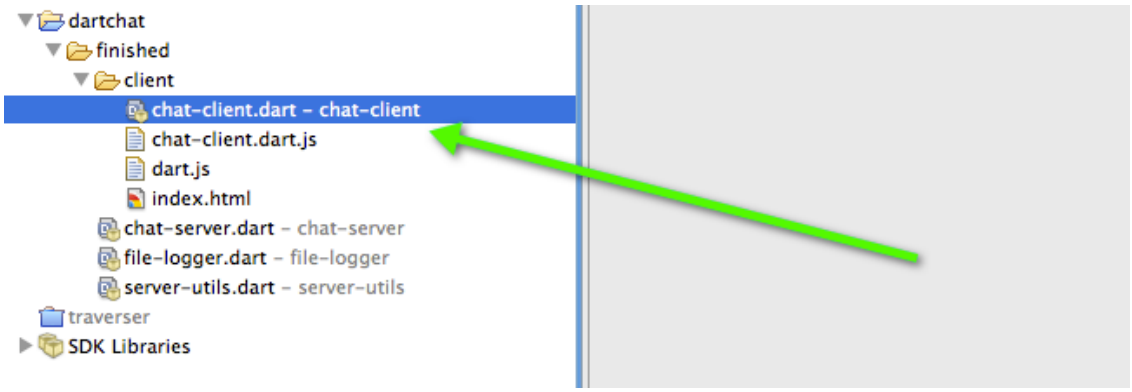
Objectives

1. Compile client app to JavaScript
2. Run Dart app in production browser
3. Understand how the same code runs in Dartium and production browsers

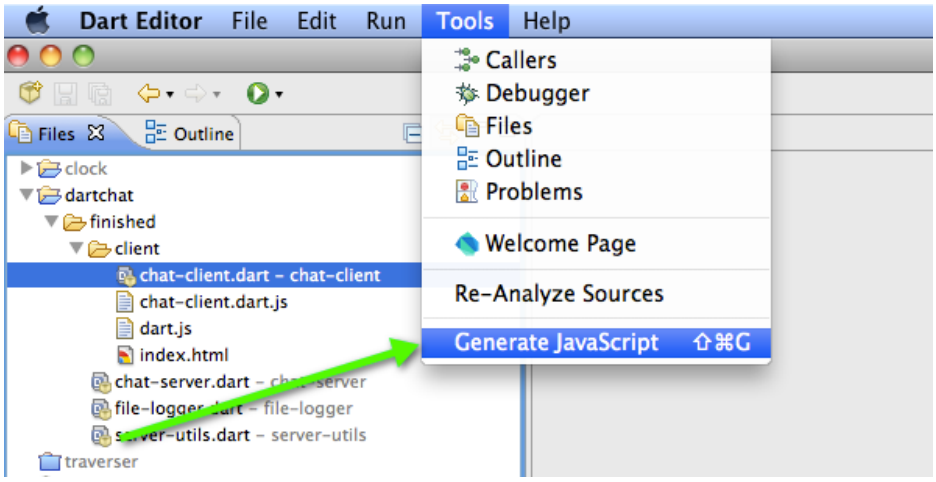
Walkthrough

Open and show the `finished` directory. This ensures you are compiling a working application. You can do this step in `start-here` if you are all caught up and it works successfully in Dartium (see previous step).

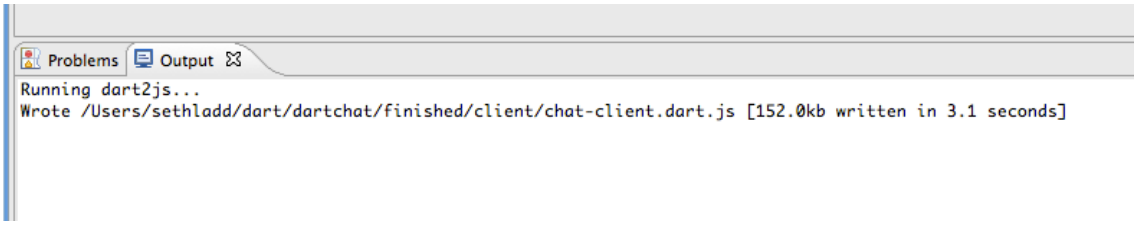
Select the `finished/client/chat-client.dart` file.



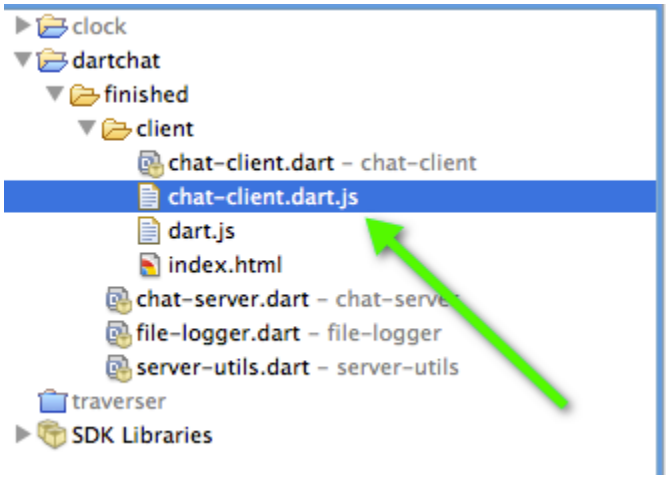
Choose **Tools** from the top menu and then select **Generate JavaScript**. This uses the dart2js[24] compiler to convert the Dart client app into modern JavaScript.



Notice the output in the console view at the bottom of Dart Editor, confirming compilation has succeeded.



A `finished/client/chat-client.dart.js` file is generated.

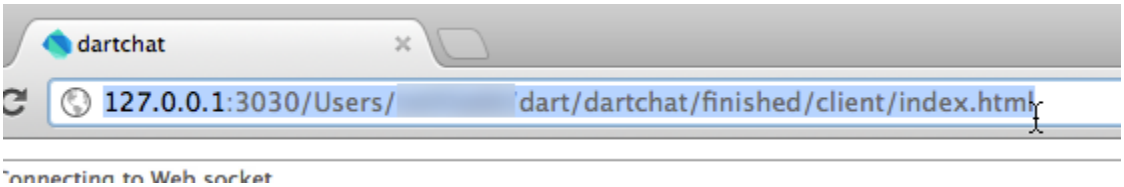


Ensure the chat-server.dart application is running. See a previous step for instructions.

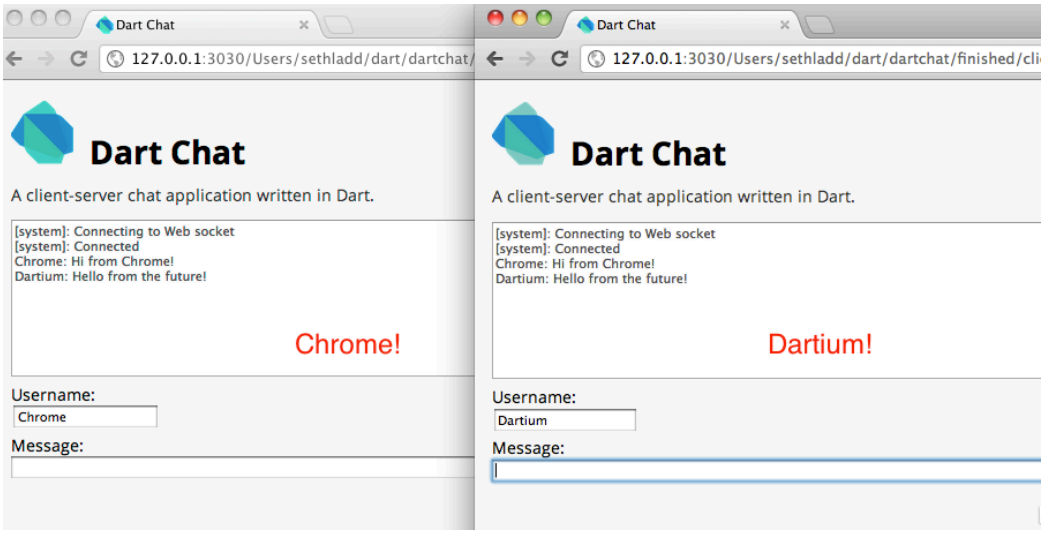
Run the `finished/client/chat-client.dart` client application. This will start Dartium. See a previous step for instructions.

[24] http://www.dartlang.org/docs/dart2js/

The application is now running Dartium. Grab the URL and copy it to the clipboard.



Open up Chrome (not Dartium!) and paste in the URL to load up the app. Chat between Dartium, running Dart code on the VM, and Chrome, running Dart compiled to JavaScript.



- **Note**: As of 2012-06-26, FireFox wouldn't connect. We don't think this is a Dart issue, but a WebSocket issue. Suggestions most welcome! If you have Firefox, please test WebSockets at http://www.websocket.org/echo.html and let us know.
- **Note**: As of 2012-06-26, Safari does not speak the latest version of the WebSockets protocol. Here's the bug to track: http://code.google.com/p/dart/issues/detail?id=3631

How does the same URL work in both Dartium and non-Dartium browsers? Open `finished/client/index.html` and notice the two scripts:

```
<script type="application/dart" src="chat-client.dart"></script>
<script src="dart.js"></script>
```

The `dart.js` file detects whether your browser has a Dart VM. If not, it removes the `application/dart` script and replaces it with a `text/javascript` script tag that points to `chat-client.dart.js`.

Advanced

Rebuild Gmail, but in Dart. Or, open up `dart.js` to learn more.

If you have time, you can add support for a command that lists all the people in the chat room.

Or, send a list of all people in a chat room when a new person joins the room.

Step 9: Super duper advanced ideas

Got this far? Congrats! Here are a few ideas to sharpen your Dart point.

- Add simple formatting when displaying the message. Parse messages, look for words wrapped with asterix's, and bold those words when displayed in the chat window. You might find Dart's Regular Expressions[25] helpful here.
    - For example, if a user types "*Dart is rad!!*" you should display "**Dart is rad!!**"
- Store the username into the browser's Local Storage and load it up when the client app starts. The Local Storage interface[26] will be useful.

[25] http://api.dartlang.org/dart_core/RegExp.html
[26] http://api.dartlang.org/html/Storage.html